

Министерство образования и науки Российской Федерации
Московский физико-технический институт
(национальный исследовательский университет)

Физтех-школа радиотехники и компьютерных технологий
Кафедра микропроцессорных технологий в телекоммуникационных сетях и
вычислительных системах

Выпускная квалификационная работа бакалавра по направлению 03.03.01
«Прикладные математика и физика»

Исследование и разработка системы анализа кода
для повышения производительности программ на
языке программирования высокого уровня

Студентка Б01-008 группы
Лирисман К. С.

Научный руководитель
Гаврин Е. А.

Долгопрудный
2024

Содержание

Аннотация	v
1. Введение	1
2. Постановка задачи	5
2.1. Цели работы:	5
2.2. Задачи работы:	5
3. Обзор существующих решений	7
4. Теоретическая часть	11
4.1. Целевой язык программирования	11
4.2. Бенчмаркинг	12
5. Практическая часть	15
6. Заключение	17
Литература	19

Аннотация

В настоящий момент активно развивается статически типизированный управляемый язык программирования, являющийся расширенной и более быстрой версией языка TypeScript (далее TS). Основная идея разработки спецификации и компилятора этого языка программирования — сделать его максимально похожим на TS для упрощения перехода будущих разработчиков между TS и выбранным для исследования языком, а также для ускорения переписывания существующих на TS приложений. Таким образом, между целевым языком и TS формируется общая часть, - корректная с точки зрения TS. Оставшуюся часть называют не поддерживаемой TS.

В данной работе предлагается выделить из целевого ЯП подмножество, наиболее выгодное с точки зрения производительности, в том числе за счёт отсекающего удобного для разработчиков, но медленного функционала. Таким образом, пока сам язык не развивается в сторону общности с TS'ом ради легкости перехода, предложенная система помогает держать фокус на производительности.

Анализ происходит в момент компиляции исходного кода и предлагает включение желаемых проверок группами или по отдельности путем добавления флагов компиляции. Реализовано 8 типов проверки, опирающихся на спецификацию выбранного языка программирования высокого уровня и реализацию его компилятора. К общим с TS проверкам относятся: неявная упаковка и распаковка, ускорение проверок равенства, запрет инструкций верхнего уровня, исключая классы и функции, предложение установки модификатора для класса или метода как финального и другие. К несовместимым с TS относятся: использование корутин вместо асинхронных функций, предложение использовать модификатор, ограничивающий наследование классов и методов в случае, если на момент проверок у них нет потомков. Предлагается использование двух режимов работы проверяющей системы — в состоянии предупреждений, а именно, предложений, не обязывающих разработчика к исправлению замечаний и не влияющих на результат работы программы, и в состоянии, приводящему к ошибке при ненулевом количестве предложений, ожидающих от пользователя последующих исправлений, и считающегося за ошибку компиляции. Предложенная система формулирует рекомендации для повышения производительности. Однако существуют сценарии использования, когда разработчикам необходимо отключение этих проверок. С помощью произведенной системы регулирования режимов проверок пользователи целевого языка могут активировать или деактивировать любую опцию по одной или включать группами. Также можно

снять с проверки определенные строки или целые частей кода - всё это реализуемо с помощью предложенного аналога системы точечного отключения проверок Clang Tidy непосредственно в исходном коде программы в виде многострочных или однострочных комментариев. Таким образом, данная система значительно повышает скорость работы и время запуска приложения на выбранном статически типизированном языке высокого уровня, позволяет разработчикам простыми методами улучшить их код и потенциально избежать некоторых ошибок.

Глава 1

Введение

В современном мире существует огромное множество языков программирования. Часть из этих языков программирования одной из своей целей при разработке доступность использования для широкого круга людей, которые не глубоко знакомы с устройством компьютеров, операционных систем.

Дополнительно, каждый язык программирования оптимизирован для своего типа задач, которые предположительно будут создаваться с его использованием.

Постановка задачи создания идеального языка программирования включает в себя точное и однозначное выражение спецификации задачи и способность превратить написанное в машинные инструкции. Очевидно, что такая задача едва ли достижима, так как тяжело довести до совершенства формулировки большого количества задач, и еще сложнее реализовать нужный алгоритм.

На данный момент в мире насчитывается больше сотни языков программирования, хотя большинство людей знакомы только с несколькими десятками, и это при условии глубоких знаний и опыта в сфере.

Такое большое количество языков программирования необходимо для качественной реализации определенного класса приложений. При этом большинство современных языков программирования создаются машинно-независимыми, то есть их структура не привязана к архитектуре. Отсюда их название, - машинно-независимые языки программирования высокого уровня.

Сфера компьютерных наук развивается довольно быстро, динамично и хаотично по причине своей популярности и обширности. Это значит, что постоянно на рынке появляются новые подходы, старые становятся либо неактуальными, либо дорабатываются, чтобы соответствовать современным стандартам.

По мере появления более сложных систем и программного обеспечения возникают все новые сценарии использования, которые требуют все новых языков. Сложность приложений и систем растет, что требует новых методов решения и подходов. При этом не менее важным параметром является поддержание доступной стоимости разработки.

Как было сказано ранее, на сегодняшний день на первый план вышли языки программирования высокого уровня, которые позволяют разработчику сфокусироваться

на логике работы приложения и не переживать об управлении памятью и иными ресурсами, относящимися к низкоуровневым задачам. Все это позволяет сформулировать несколько причин для создания новых языков программирования высокого уровня:

1. Проектирование новых языков с целью оптимизации производительности может включать в качестве результата более эффективное использование ресурсов компьютера, упрощение синтаксика в контексте целевой задачи или разработка новых синтаксических конструкций или функционала, облегчающих работу разработчиков конкретной сферы.
2. Проектирование новых языков под новые задачи, для которых на данный момент нет удобного или оптимального решения. К таким задачам можно отнести, например, задачи, связанные с узко направленной научной деятельностью.
3. Разработка нового языка может являться чисто стратегическим действием для продвижения коммерческих решений в открытый доступ или ради развития новых идей, видений и подходов в области компьютерных наук

В настоящее время активно развивается язык программирования, представляющий собой более быструю альтернативу существующему TypeScript (далее TS). Основной целью этого языка является создание расширенное и более быстрой статической версий TS при условии сохранения совместимости с самим TS ради упрощения перехода будущих разработчиков с одного языка на другой.

Современные среды программирования требуют высокую производительность приложений. В таком контексте принципиально важны дополнительные инструменты, позволяющие автоматически или с определенной персонализацией под конкретную задачу проанализировать исходный код и выдвинуть предложения по его оптимизации.

Под повышением производительности в целом могут стоять разные параметры, например, энергопотребление или размер полученного бинарного файла. Эти и другие параметры обычно тесно взаимосвязаны, и потому в попытках улучшить показатели одного параметра можно получить отрицательный результат для другого.

Выделяют динамические параметры программы, к которым традиционно относят производительность и энергопотребление. Смысл динамических параметров заключается в зависимости от конкретного устройства, на котором исполняется программа, то есть при проведении замеров результаты не будут повторяться. Однако такие параметры можно измерять группами: для одной группы устройств и для другой. Замеры формата "до" и "после" в рамках выбранной группы должны отвечать свойству повторяемости. Дополнительно можно сказать, что тенденция должна сохраняться при переходе из одной группы в другую. Например, если использование некоего алгоритма А в группе А приводит к ускорению приложения на 20% по сравнению с методом Б в рамках группы А, то при переходе в группу Б не должно наблюдаться противоположного результата, то есть, например, замедления на 15% при таком же сравнении метода А и метода Б.

Если такие результаты и удастся получить, это может означать только некорректность постановки задачи или использованных метрик.

Очевидно, что скорость выполнения программы и энергопотребление являются двумя очень важными аспектами, что при разработке всегда хотелось бы выиграть в обоих. Тем не менее, они могут быть очень по-разному оптимизированы. Тогда некоторые изменения в языке программирования или в реализации его компилятора могут привести одновременно к увеличению потребления энергии из-за увеличения числа операций или увеличения нагрузки на процессор, чем мы не можем пренебрегать в случае устройств с ограниченными ресурсами, и к увеличению скорости выполнения программы за счет более эффективного распределения ресурсов машины. Некоторые из таких изменений могут быть следующими:

1. При использовании параллелизма алгоритмы увеличивают свою скорость выполнения, но при этом могут потреблять большее количество энергии, так как нагрузка на процессор стала значительно больше.
2. При изменении алгоритма или структуры данных некоторые операции могут оказаться более эффективными по времени, но менее эффективными по памяти, и тогда расход энергии увеличится.
3. При оптимизациях компилятора запускаются сложные оптимизационные процессы, которые могут больше расходовать ресурсы. Может быть частой проблемой при нерациональном или неграмотном подходе к оптимизациям.

В данной работе фокус идет на повышение производительности в контексте времени выполнения и запуска программы за счет гибкой работы со спецификацией целевого языка программирования и реализацией его компилятора. По данной причине вышеупомянутые оптимизации для работы не являются релевантными, а использование других, более высокоуровневых абстракций предполагается заранее считать вредными для быстрого действия. Таким образом, будет предлагаться поиск недорогостоящих аналогов с точки зрения скорости выполнения и спецификации выбранного языка программирования.

Итак, в данном случае уместно считать, что меньшее время исполнения программы влечет к меньшему энергопотреблению. Далее под повышением производительности будет пониматься скорость выполнения программы и время ее запуска в рамках выбранных устройств и опций компилятора.

Глава 2

Постановка задачи

2.1 Цели работы:

1. Разработка системы анализа и предупреждений исходного кода выбранного языка программирования высокого уровня для повышения его производительности и ускорения запуска написанных на нем приложений
2. Реализация системы-аналога Clang Tidy для выборочного отключения выбранных проверок точно, непосредственно в исходном коде.

2.2 Задачи работы:

1. Изучение существующих решений
2. Разработка системы анализа кода на этапе компиляции
3. Анализ текущей спецификации целевого языка программирования на предмет потенциально медленных языковых конструкций и функционала, сбор данных для дальнейшего тестирования.
4. Предоставить вариант исправления, ускоряющий работу приложения, корректный с точки зрения выбранного языка программирования, для каждой языковой единицы среди предложенных
5. Протестировать каждое предложение: замерить скорость работы приложения до и после предложенных исправлений
6. Реализовать соответствующую поверку в системе анализа после подтверждения положительных результатов тестирования
7. Разработать систему точечного и группового отключения выбранных разработчиком проверок, поддержать наиболее популярные сценарии использования, опираясь на данные, полученные при изучении существующих решений

8. Поддержать возможность анализа в системе многофайловой сборки

Цели работы разумно считать достигнутыми при выявлении и реализации не менее пяти предложений, ускоряющих работу приложения в среднем не менее, чем на 5 %, а также разработке системы отключения проверок, успешной поддержке проектной сборки и прохождении тестирования, составленного из некоторых потенциальных сценариев применения предложенных решений.

Глава 3

Обзор существующих решений

Сегодня представлено огромное число средств, позволяющих на разных этапах проанализировать код программы на качество, причем понимание о качестве могут подразумевать быстродействие, а могут и нет. По данной причине есть смысл рассмотреть основные классы данных инструментов. Это позволит правильно классифицировать и применить результат данной работы на практике другим пользователям и читателям.

Итак, разные системы могут включать как статический, так и динамический анализ, и они используются для различных языков программирования, таких как C, C++, Java, Python и другие. Существует несколько основных подходов, применяемых для анализа и оптимизации кода.

1. Статический анализ кода

Цель: Анализ кода без его выполнения.

Примеры инструментов: Lint для C, SonarQube для многих языков.

Методы: Проверка стиля кода, поиск потенциальных ошибок, анализ контрольных потоков, оптимизация использования памяти.

2. Динамический анализ кода

Цель: Анализ кода во время его выполнения.

Примеры инструментов: Valgrind для C/C++, JProfiler для Java.

Методы: Профилирование времени выполнения, анализ использования памяти, поиск утечек памяти.

3. Оптимизации компиляции.

(а) JIT-компиляция (Just-In-Time)

Цель: Компиляция кода во время его выполнения для оптимизации под конкретное окружение.

Примеры: JVM для Java, CLR для .NET.

Преимущества: Улучшенная производительность благодаря адаптации кода к текущим условиям выполнения.

(b) **АОТ-компиляция (Ahead-Of-Time)**

Цель: Компиляция кода до его выполнения.

Примеры: GraalVM для Java, LLVM для C/C++.

Преимущества: Быстрая загрузка и запуск программ, уменьшение накладных расходов на JIT-компиляцию.

4. **Оптимизация кода**

(a) **Инлайн-функции**

Цель: Уменьшение накладных расходов на вызов функций.

Методы: Замена вызова функции ее телом в местах вызова.

(b) **Разворачивание циклов**

Цель: Уменьшение количества итераций цикла.

Методы: Объединение нескольких итераций цикла в одну.

(c) **Удаление мертвого кода**

Цель: Удаление кода, который никогда не будет выполнен.

Методы: Статический анализ для обнаружения и удаления ненужных блоков кода.

Основное, что объединяет вышеперечисленные методы в рамках данной работы, это ограниченность их анализа спецификацией языка. То есть анализ идет на основе того, как конкретный пользователь языка использует его конструкции в своей программе, не затрагивая идеи, на которых строился и писался сам язык программирования.

Теперь рассмотрим инструменты, по сути своей выделяющие из конкретных языков программирования некоторые подмножества с фокусом, например, на производительность, безопасность или надежность кода.

1. **Ada**

Ada имеет концепцию *restriction pragmas* (ограничительных прагм), которые позволяют программистам ограничивать использование определенных функций языка для повышения производительности и надежности. Эти ограничения могут включать запрет на использование динамической памяти, запрет на использование исключений.

2. **SPARK (подмножество Ada)**

SPARK — это строгий подмножество языка Ada, предназначенное для разработки систем с высокими требованиями к надежности и безопасности. SPARK исключает некоторые языковые конструкции Ada, чтобы позволить формальную верификацию кода и обеспечить высокую производительность и надежность.

КОНЕЦ БЕЗУМИЯ АНТИПЛАГИАТА

3. Ada Ravenscar

Ada Ravenscar — это подмножество языка программирования Ada, предназначенное для разработки высоконадёжных и безопасных систем реального времени. Ravenscar ограничивает использование некоторых возможностей Ada, чтобы обеспечить однозначность и предсказуемость выполнения программ. Ravenscar активно используется в аэрокосмической и оборонной промышленности.

4. Misra C/C++

MISRA (Motor Industry Software Reliability Association) C и C++ — это набор рекомендаций и правил для написания безопасного и надежного кода на языках C и C++. Эти правила часто используются в автомобильной и других критически важных отраслях, где важна высокая производительность и безопасность.

5. CERT C

CERT C — это набор руководящих принципов для написания безопасного и надежного кода на C, разработанный Software Engineering Institute (SEI) при Carnegie Mellon University. Стандарт фокусируется на безопасности: включает рекомендации по предотвращению распространенных уязвимостей, таких как переполнения буфера, ошибки с указателями.

6. Standard ECMA-327

Данная версия сосредотачивается на минимизации использования памяти и процессорного времени для устройств с ограниченными ресурсами.

7. Java ME (Micro Edition)

Для Java существует подмножество под названием Java ME (Micro Edition), специально разработанное для устройств с ограниченными ресурсами, таких как встроенные системы и мобильные устройства. Java ME предоставляет сокращенный набор библиотек и API, подходящих для встраиваемых сред, и исключает более ресурсоемкие функции, типичные для Java SE (Standard Edition).

Данные варианты максимально близки идейно с тем, что будет далее представлено в работе. Теперь, когда проведен обзор разного рода решений, направленных на анализ с целью ускорения работы программы, ясны методы и инструменты, необходимые для реализации поставленных целей.

Глава 4

Теоретическая часть

4.1 Целевой язык программирования

Для успешной реализации системы анализа кода для повышения производительности программ на языке программирования высокого уровня требуются некоторые знания о целевом языке программирования.

Выбранный язык сочетает и поддерживает функции, которые используются во многих известных языках программирования, где эти инструменты уже доказали свою полезность и мощь. Он поддерживает императивные, объектно-ориентированные, функциональные и обобщенные парадигмы программирования, объединяя их безопасно и последовательно. В то же время целевой ЯП не поддерживает функции, позволяющие разработчикам программного обеспечения писать опасный, небезопасный или неэффективный код. В частности, язык использует принцип строгой статической типизации. Он не допускает динамических изменений типов, так как типы объектов определяются их объявлениями. Их семантическая корректность проверяется на этапе компиляции.

Основные аспекты, характеризующие данный язык в целом:

1. Объектная ориентированность

Поддержка традиционного подхода к программированию на основе классов и объектно-ориентированного программирования (ООП). Основные понятия этого подхода следующие:

- (а) Классы с единичным наследованием
- (b) Интерфейсы как абстракции, которые реализуются классами
- (с) Виртуальные функции (члены класса) с механизмом динамического перепределения

Объектная ориентированность, общая для многих (если не всех) современных языков программирования, обеспечивает мощный, гибкий, безопасный, понятный и адекватный дизайн программного обеспечения.

2. Модульность

Язык поддерживает компонентный подход к программированию. Предполагается, что программное обеспечение разрабатывается и реализуется как композиция единиц компиляции. Единица компиляции обычно представлена в виде модуля или пакета.

3. Статическая типизация

Связь с типом, например, возвращаемого значения функции или переменной, происходит сразу, в момент объявления, и не может быть изменен позже.

4. Управляемость

Язык работает в управляемой среде выполнения, и управление ресурсами осуществляется не напрямую операционной системой, а через промежуточное программное обеспечение, которое обеспечивает ряд услуг, таких как управление памятью и сборка мусора (garbage collection).

4.2 Бенчмаркинг

В общем случае бенчмарком называется программа, которая измеряет некоторые характеристики производительности приложения или фрагментов кода. В данной работе предлагается считать бенчмарк экспериментом, так как с его помощью должны получаться результаты, позволяющие узнать более подробно поведение приложения.

Получив значения метрик, нужно объяснить их и быть уверенным в том, что предложенное объяснение является верным. В ходе работы часто будет использоваться бенчмарк для сравнения некоего метода А и метода Б в рамках конкретного функционала языка.

Тривиального сравнения скорости методов с последующим предпочтением более быстрого не является корректным решением ни одной из задач данной работы, так как является в корне не верным подходом. Подобного рода вывод носит строго локальный характер, исключает анализ данных, на которых проводились замеры, состояние системы, причины увиденной разницы в скорости.

Зачастую аномальные значения в результатах замеров производительности связаны с ошибками в методологии измерений. Таким образом, фиксация прироста скорости метода А по сравнению с методом Б не является решенной задачей. Кроме того, бенчмаркинг в целом не является универсальным подходом, полезным при любом исследовании производительности.

Одним из важнейших требований к качественному бенчмаркингу является повторяемость полученных результатов. Предполагается, что между измерениями

допустима только незначительная разница, не влияющая на выводы из эксперимента и не дающая качественных различий в полученных результатах, относящихся к разным запускам.

Глава 5

Практическая часть

практикуем.

Глава 6

Заключение

закключаем

Литература

Будет добавлена.