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

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 отсечения удобного для разработчиков, но медленного функционала. Таким образом, пока сам язык неминуемо разивается 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 исправлением 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

Языки программирования создаются с главной целью облегчить использование компьютеров для широкого круга людей, которые не обязаны иметь детальные знания об их внутреннем устройстве. Каждый язык программирования адаптирован под определенные типы приложений, которые планируется программировать с его использованием. Идеальным языком программирования был бы такой, который позволяет точно и однозначно выражать спецификации решаемой задачи, а затем преобразовывать их в последовательность инструкций для компьютера. Однако достичь этого идеала крайне сложно, поскольку зачастую отсутствует чёткая спецификация задачи. Более того, создание алгоритма на основе этих спецификаций требует глубоких предметных знаний и значительного опыта.

Существует множество языков программирования, их насчитывается более тысячи, и каждый из них предназначен для определённого класса приложений. Все современные языки программирования создаются таким образом, чтобы быть независимыми от конкретной машины, на которой они выполняются. Это означает, что структура языка программирования не привязана к внутренней структуре какого-либо конкретного компьютера. Программа, написанная на таком языке, должна выполняться на любом компьютере. Эти языки известны как машинно-независимые языки программирования высокого уровня.

Область языков программирования является очень динамичной и даже в некоторой степени хаотичной. Появляются всё новые языки и подходы, совершенствуются существующие, меняются предпочтения и практики разработчиков. Постоянно происходят изменения, которые влияют на то, как создаются и используются языки программирования. По мере появления на рынке более сложных аппаратных систем появляются новые компьютерные приложения. Эти приложения порождают новые языки для решения таких приложений. Другой тенденцией является постоянное увеличение сложности приложений по мере того, как аппаратное обеспечение становится более сложным и дешевым. Увеличение размера программ требует новых методов решения проблемы сложности, сохраняя при этом низкую стоимость разработки программ и обеспечивая их корректность.

В современном программировании особенно значимы языки программирования высокого уровня, которые позволяют разработчикам сфокусироваться на логике приложений, а не на деталях управления памятью или другими низкоуровневыми задачами. Однако, на таких языках возникают проблемы с производительностью, особенно в контексте больших и сложных проектов. Люди вынуждены создавать новые языки программирования высокого уровня, так как:

1. Новые языки могут быть спроектированы с целью оптимизации производительности и улучшения эффективности разработки программного обеспечения. Это может включать в себя более эффективное использование ресурсов компьютера, упрощение синтаксиса для повышения читаемости кода или введение новых функций и конструкций для облегчения работы разработчиков.

2. Новые языки могут быть созданы для решения конкретных проблем или задач, которые не могут быть эффективно решены с использованием существующих языков. Например, некоторые языки могут быть специально разработаны для работы с распределенными системами, большими данными или машинным обучением.

3. Некоторые языки могут быть разработаны для удовлетворения специфических потребностей определенных областей или индустрий. Например, языки для разработки игр, веб-приложений, научных вычислений или встроенных систем имеют свои особенности.

4. Разработка новых языков программирования является частью процесса постоянной эволюции и инновации в области информационных технологий. Новые языки могут предложить новые идеи, концепции и подходы, которые могут привнести свежий взгляд на программирование и стимулировать развитие отрасли в целом.

В этой работе мы обращаемся новому к языку программирования, который активно развивается как быстрая альтернатива уже широко используемому TypeScript. Основная цель разработки этого языка заключается в создании расширенной и более производительной версии TypeScript. При этом важно сохранить совместимость с TypeScript для облегчения перехода существующих проектов и обучения новых разработчиков.

Современная динамичная среда программирования требует от разработчиков не только функциональности, но и высокой производительности своих приложений. В этом контексте особенно актуальными становятся инструменты, способные автоматизированно анализировать и оптимизировать исходный код программ. В этом контексте, важную роль играют инструменты, способные анализировать исходный код программ, для повышения производительности программ на языке программирования высокого уровня

Следует уточнить, что подразумевается под повышением производительности в данной работе. В целом существует неколько параметров программы, которые определяют качество произвоительности. Это потребление энергии во время работы программы,

время исполнения программы, то есть ее performance, или, например, итоговый размер бинарного файла для исполнения. Эти и другие параметры вместе двигать к идеальному состоянию невзможно, так как он тесно связаны, и улучшение одной характеристики зачастую неминуемо ведет к ушудшению зависимой. Размер итогово файла после компиляции является статическим параметром, то есть зависящим только от компилятора, использованной архитектуры и примененных оптимизаций.

Производительность и потребление энергии работающего приложения являются динамическими параметрами программы. Это означает, что они значительно зависят от устройства, на которых им предстоит исполняться. Конечно, это делает замеры значительно более сложными и распределенными, но можно опираться на сравнение формата "до"и "после"в среднем в рамках выбранных устройств и оценить полученную динамику в среднем между разными девайсами уже после получения их локальных результатов.

Скорость выполнения программы и энергопотребление - это два важных, но различных аспекта, которые могут быть оптимизированы по-разному. Некоторые изменения в языке программирования или в реализации компилятора могут привести к увеличению скорости выполнения программы за счет более эффективного использования ресурсов процессора или памяти. Однако, эти же изменения могут привести к увеличению потребления энергии, например, за счет увеличения числа операций или увеличения нагрузки на процессор. Примеры таких изменений могут включать в себя:

1. Увеличение использования параллелизма: параллельные алгоритмы могут увеличить скорость выполнения программы, но при этом могут потреблять больше энергии из-за увеличенной нагрузки на процессор.

2. Изменение алгоритма или структуры данных: некоторые изменения в алгоритмах или структурах данных могут сделать программу более эффективной с точки зрения скорости выполнения, но могут также потреблять больше энергии из-за увеличенного числа операций или использования памяти.

3. Оптимизации компилятора: оптимизации компилятора могут улучшить скорость выполнения программы, но в некоторых случаях могут привести к увеличению потребления энергии из-за более сложных оптимизационных процессов.

Однако в данной работе не предлагается использование иных методов ускорения, отличных от связанных со спецификацией и реализацией его компилятора. Поэтому вышеупомянутые нерелевантны, а ипользование более высокоуровневых абстракций предполагается заранее считать вредным для быстродействия, поэтому будет предлагаться поиск менее дорогостоящих аналогов с точки зрения скорости исполнения.

Таким образом, в данном случае уместно считать, что меньшее время исполнения программы влечет к меньшему энергопотреблению. Далее под повышением производительности будет пониматься скорость выполнения программы 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 точечно, непосредственно в исходном коде.

2.2 Задачи работы:

1. Изучение существующих решений

2. Разработка системы анализа кода на этапе компиляции

3. Анализ текущей спецификация целевого языка программирования на предмет потенциально медленных языковых конструкций и функционала, сбор данных для дальнейшего тестирования.

4. Предоставить вариант исправления, ускорящий работу приложения, корректный с точки зрения выбранного языка программироания, 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.**

   (a) **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) **AOT-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.

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 н адежного 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 мощность. 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:

   (a) Классы с единичным наследованием

   (b) Интерфейсы как абстракции, которые реализуются классами

   (c) Виртуальные функции (члены класса) с механизмом динамического переопределения

   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 некого 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 наиважнейших 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

заключаем

Литература

Будет добавлена.