Санкт-Петербургский государственный университет

Кафедра Системного программирования

Милосердов Владимир Михайлович

Автоматизированный анализ производительности и регрессии компилятора для встроенных процессоров в системе LLVM LNT

Курсовая работа

Научный руководитель:
ст. преп. Немешев М. Х.

Санкт-Петербург
2018

Оглавление

Введение

Современные инновации в науке и промышленности требуют постоянного увелечения вычислительных мощностей, в то же время устанавливая жесткие ограничения, накладываемые на программно-аппаратные системы, включающие в себя в том числе процессоры, компиляторы, эмуляторы, отладчики. Требования устанавливаются на эффективность по времени исполнения программ, энергопотреблению, размерам, надёжности и портативности оборудования. Немаловажным требованием становится также время разработки таких систем. Особенно жёсткие требования предъявляются к встраиваемым решениям, процессоры в которых сильно ограничены в ресурсах. Сложность аппаратного и программного обеспечения огромна и постоянно растёт[4][5], а процесс их разработки зачастую включает в себя труд множества команд из разных организаций, стран, часовых поясов.

В процессе разработки большую роль играет тестирование и анализ производительности. При этом постоянно измененииются и доработываются как инструменты, так и аппаратура. Из-за огромной сложности составляющих набор функциональных тестов и бенчмарков достаточно велик так же как и количество различных инструменов тестирования, применяемое в разных командах.

Это порождает большой объём усилий, которые тратятся на такие задачи, как запуск тестов в попытках воспроизвести нужный результат, локализацию ошибок. Множество регрессий как функциональных, так и по производительности обнаруживается слишком поздно, что усложняет отладку и устранение ошибок. Зачастую сложно даже понять, какое изменение - в аппаратной части, каком-либо инструменте (например компиляторе или симуляторе) или тестовом покрытии вызвало регрессию.

В таких условиях единая система тестирования (как функционального, так и производительности), способная запускать различные виды тестов и предоставлять удобные инструменты для работы с их результатами помогла бы значительно облегчить процесс тестирования и раз-

работки.

В рамках данной работы рассматривается задача совершенствования одной из таких систем – LLVM LNT[3]: добавление новых и улучшение текущих представлений данных тестирования, применение диагностических сообщений компилятора для выявления его регрессий.

1. Постановка задачи

Целью работы является улучшение представления данных в системе автоматизации тестирования LLVM LNT, а также добавление в данную систему возможности выявления регрессий с использованием диагностических сообщений оптимизационных проходов компилятора.

Для достижения этой цели были выделены следующие задачи, разделенные на 2 части: задачи касательно диагностических сообщений и задачи по улучшению системы LLVM LNT.

Первая часть включает следующее:

1. Сделать обзор существующих решений, использующих диагностическую информацию оптимизационных проходов компилятора для выявления и исправления регрессий; улучшения производительности

2. Улучшить генерацию диагностических сообщений компилятора, построенного на базе LLVM, с целью их дальнейшей визуализации и использования в системе автоматизированного тестирования

3. Провести эксперимент, показывающий эффективность выявления регрессий с использованиям диагностических сообщений проходов компилятора

4. Добавить возможность визуализации диагностических сообщений проходов компилятора в системе LLVM LNT

Вторая часть разбивается так:

1. Сделать обзор существующих систем автоматизированного тестирования, применимых для разработки компилятора

2. Определить, какие из текущих представлений данных нуждаются в доработке

3. Улушить представления данных тестирования в LLVM LNT

2. Введение в предметную область

2.1. Тестирование

Тестирование программного обеспечения (Тестирование) – процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определенным образом.[2]

В рамках этой работы нас интересует функциональное тестирование и тестирование производительности.

Функциональное тестирование – это тестирование ПО с целью проверки корректной реализации функциональных требований к данному ПО

Тестирование производительности – это тестирование ПО с целью проверки его соответствия различным требованиям к производительности, а также с целью оценки показателей производительности данного ПО *(напр. время работы, потребляемая ОЗУ, т.д.)*

2.1.1. Тестирование компилятора

Разрабатываемый компилятор должен, как минимум, генерировать правильный код, в современных реалиях большое значение имеет производительность генерируемого кода и время компиляции.

На практике необходим систематический подход для проверки и оценки вышеперечисленного. Для проверки корректности обычно используются наборы функциональных тестов, включающих в себя различные случаи программ согласно спецификации компилятора. При этом важно проверить как можно больше таких случаев, т.е. обеспечить максимально большое тестовое покрытие.[1]

Для тестирования производительности применяются специальные тестовые программы, замеряющие метрики (характеристики) при разных сценариях работы программы – бенчмарки. Метрики, применяемые при разработки компиляторов для встроенных процессоров могут

включать количество тактов процессора, затрачиваемых на исполнение бенчмарка, размер генерируемого кода, а также промахи кэша, простои конвейеров и другие.

3. Обзор существующих решений

3.1. Обзор систем для автоматизированного тестирования

3.1.1. Jenkins со сценариями прогона тестов

Jenkins – серверное приложение непрерывной интеграции (CI). Решение имеет открытый исходный код.

В качестве средства автоматизированного тестирования Jenkins позволяет осуществлять прогоны тестов по сценариям при каждом изменение кодовой базы (коммите). Можно также автоматически создавать отчёты по результатам сборки и прогонам тестов.

Такой способ тестирования требует разработки сценариев (скриптов) запуска сборки, тестирования, анализа результатов и т.д.

Очевидно, функциональные возможности системы будут зависеть от конкретных реализаций сценариев.

3.1.2. LLVM LNT

LLVM LNT – клиент-серверное инфраструктурное решение для автоматизации тестирования LLVM. Часть открытого проекта LLVM.

С помощью LNT можно определить несколько различных конфигураций (напр. платформ или различных опций сборки) и работать с каждой конфигурацией отдельно.
Система предназначена для работы с разными версиями тестируемого приложения (компилятора), сравнения их производительности. Есть встроенные возможности профилирования приложения на отдельном тесте, визуализации результатов.

Стоит отметить, что приложение предназначается для тестирования компилятора на функциональных тестах и бенчмарках, но может использоваться для тестирования любых других приложений.

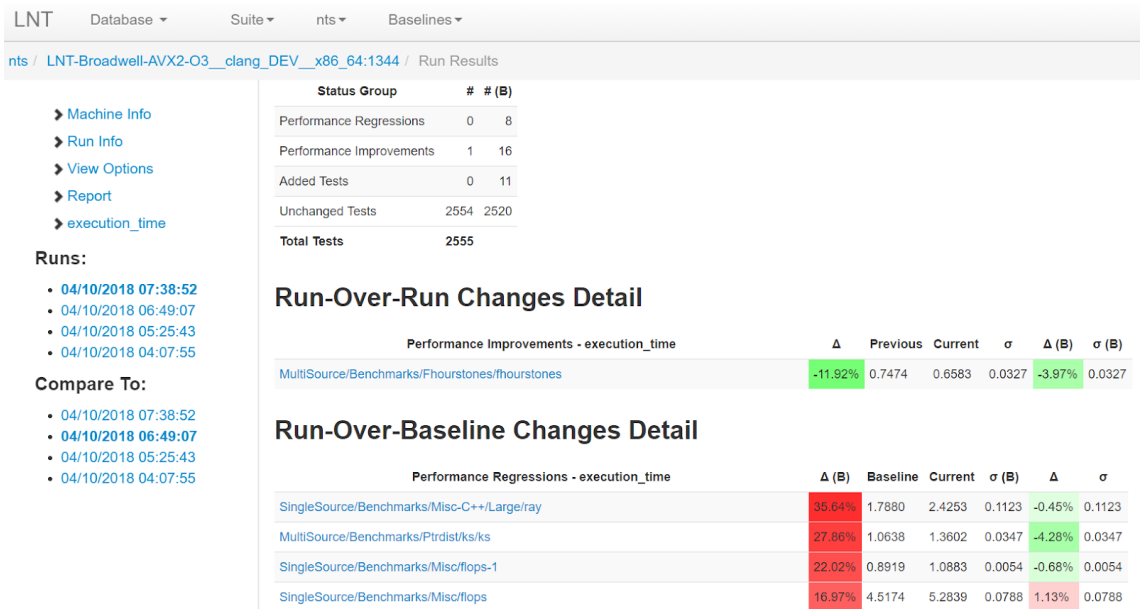

Рис. 1: Пользовательский интерфейс LLVM LNT

3.2. Обзор решений, использующих диагностическую информацию компилятора

Intel vectorization adviser[6] – средство анализа эффективности векторизации циклов компилятором. Программа показывает, что препятствует векторизации цикла, измеряет эффективность выполнения программы в циклах и даёт советы, как можно исправить исходный код анализируемой программы чтобы больше циклов было векторизовано и производительность улучшилась. Стоит отметить, что данное средство отличается продуманным пользовательским интерфейсом, дружественным к пользователю.

На рис. 1 показан интерфейс программы. Жёлтым выделены интересующие пользователя метрики, красным показана колонка в таблице, содержащая общую информацию по векторизации, полученную из диагностической информации компилятора.

Данное средство полноценно поддерживает лишь компилятор Intel, языки программирования C/C++, Fortran. Кроме того данное решение – проприетарное ПО.

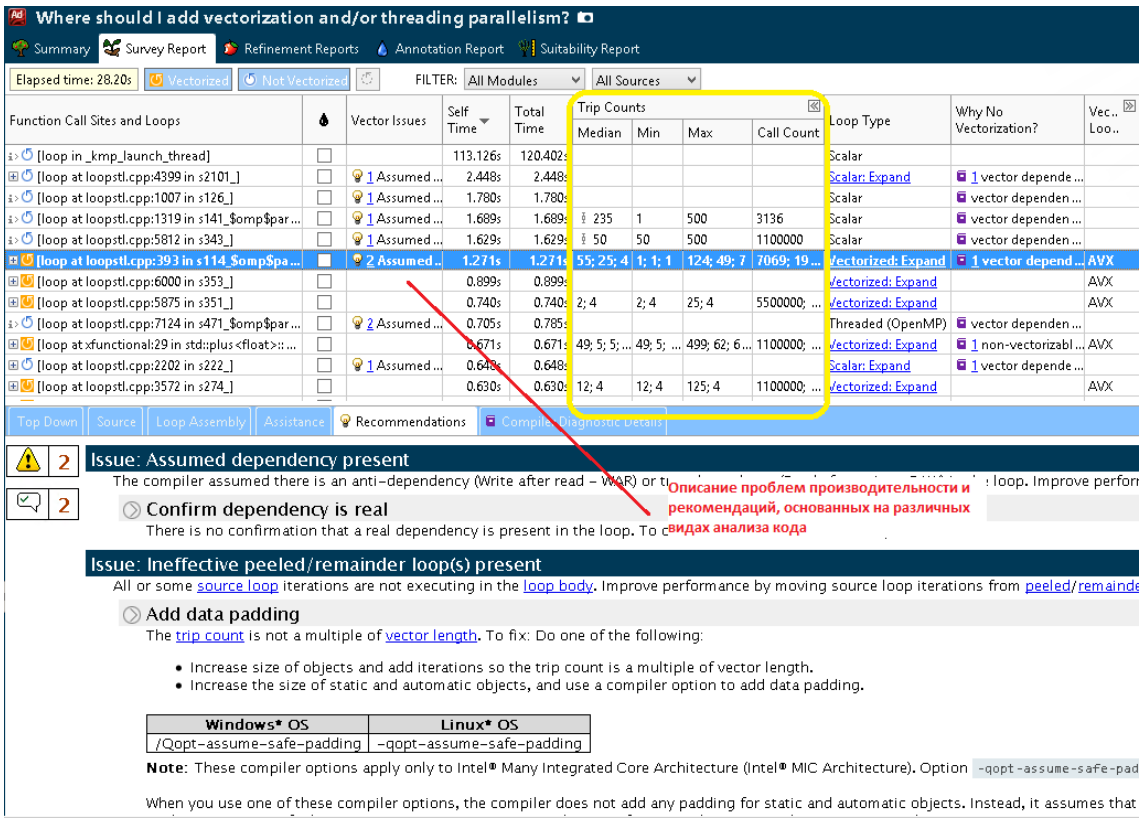

Рис. 2: Пользовательский интерфейс средства Intel vectorization adviser

4. Применение диагностических сообщений для выявления регрессий

Иногда данных обычных функциональных тестов и бенчмарков нехватает чтобы быстро обнаружить регрессию и проанализировать её происхождение. Рассмотрим следующий фрагмент кода.

```
void f1(int *x, int end) {
    end >>= 2;
    for (int i = 1; i < end; i++)
        x[i] += 10;
}
```

Допустим, наша целевая платформа поддерживает векторизацию, она получилась выгодна по cost-модели. LLVM умеет производить векторизацию циклов с неизвестным на этапе компиляции (но вычислимым) количеством итераций. Компилятор векторизует цикл и выдаст следующую диагностику.

```
test.cxx:3:1: remark: loop vectorized
```

Теперь если мы случайно ”испортим” достаточно сложный анализирующий проход ScalarEvolution, вычисляющий количество итераций циклов, векторизовать цикл не получится. Мы получим такое сообщение.

```
test.cxx:3:1: remark: loop not vectorized: uncountable iteration count
```

Без такой диагностики можно не заметить регрессию вообще (напр. если нет теста, в котором будет значительное снижение по какой-либо метрике). Кроме того, наличие сообщения упрощает процесс отладки: разработчик сразу поймёт, в каком проходе возможна проблема даже без необходимости использования средств отладки.

Таким образом, на базе таких сообщений компилятора можно улучшать тестовое покрытие. В рамках данной работы была улучшена диагностика проходов, связанных с векторизацией и других проходах, были разработаны новые тесты.

5. Улучшение представлений данных тестирования LNT

Ввиду большого количества различных тестов в системе, представления результатов тестирования имеет важное значение для эффективности работы команды разработчиков. Быстрая навигация по различным конфигурациям, сравнение различных прогонов тестов, визуализация метрик уменьшает время, необходимое на анализ результатов тестирования, количество возможных ошибок при их интерпретации.

В рамках работы были сделаны небольшие улучшения фронтенда LLVM LNT. С использованием фреймворка Bootstrap и плагинов для него со свободным исходным кодом была добавлена возможность генерации сводной таблицы тестирования с удобной сортировкой.

Test Data

Summary data table

Name	compile_time						execution_time						Log
Test	Prev	Current	%	Δ	σ	MAD	Prev	Current	%	Δ	σ	MAD	
MultiSource/Benchmarks/Prolangs-C/allroots/allroots-link	0.0918	0.0900	-1.96%	-0.0018	-	-	-	-	-	0.0000	-	-	Log
MultiSource/Benchmarks/Prolangs-C/unix-tbl/unix-tbl-link	0.2811	0.3015	7.26%	0.0204	-	-	-	-	-	0.0000	-	-	Log
MultiSource/Benchmarks/Prolangs-C/gnugo/gnugo	3.3230	3.6131	8.73%	0.2901	-	-	82.4819	91.9274	11.45%	9.4455	-	-	Log
MultiSource/Benchmarks/Prolangs-C/football/football	5.1747	5.7475	11.07%	0.5728	-	-	0.2140	0.2491	16.40%	0.0351	-	-	Log
MultiSource/Benchmarks/Prolangs-C/football/football-link	0.1516	0.1695	11.81%	0.0179	-	-	-	-	-	0.0000	-	-	Log
MultiSource/Benchmarks/Prolangs-C/fixoutput/fixoutput	0.3263	0.3802	16.52%	0.0539	-	-	0.2288	0.2659	16.22%	0.0371	-	-	Log
MultiSource/Benchmarks/Prolangs-C/gnugo/gnugo-link	0.3573	0.4187	17.18%	0.0614	-	-	-	-	-	0.0000	-	-	Log
MultiSource/Benchmarks/Prolangs-C/allroots/allroots	0.4740	0.5563	17.36%	0.0823	-	-	0.3613	0.4227	16.99%	0.0614	-	-	Log
MultiSource/Benchmarks/Prolangs-C/fixoutput/fixoutput-link	0.1167	0.1387	18.85%	0.0220	-	-	-	-	-	0.0000	-	-	Log
MultiSource/Benchmarks/Prolangs-C/unix-tbl/unix-tbl	5.5225	7.9669	44.26%	2.4444	-	-	0.1820	0.2106	15.71%	0.0286	-	-	Log

Рис. 3: Сводная таблица по тестам, часть GUI LLVM LNT

Заключение

В рамках работы были выполнены следующие задачи.

1. Сделан обзор решений, использующих диагностическую информацию компилятора с целью выявления регрессий и улучшения производительности
2. Сделан обзор систем автоматизированного тестирования, применимых для разработки компилятора
3. В нескольких проходах LLVM улучшена генерация диагностических сообщений. Для одного из таких проходов, код улучшений включён в проект LLVM[7]
4. Проведен эксперимент, показывающий эффективность использования оптимизационных сообщений компилятора с целью выявления регрессий
5. Улучшены некоторые представления информации в LLVM LNT

Список литературы

[1] Alfred V. Aho Monica S. Lam Ravi Sethi Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. — 1986.

[2] Bourque P., Fairley R.E. Guide to the Software Engineering Body of Knowledge, Version 3.0. — IEEE Computer Society, 2014. — URL: `www.swebok.org`.

[3] Dunbar Daniel. LNT Overview // LLVM Documentation. — 2018. — URL: `http://llvm.org/docs/lnt/intro.html` (дата обращения: 27.05.2018).

[4] Progress In Digital Integrated Electronics : Rep. / Intel Corporation ; Executor: Gordon E. Moore : 1975.

[5] Rogier Wester John Koster. The Software behind Moore's Law // IEEE. — 2015. — March.

[6] corp. Intel. Intel vectorization assistant overview // Vectorization assistant FAQ. — 2016. — URL: `https://software.intel.com/en-us/articles/vectorization-advisor-faq` (дата обращения: 27.05.2018).

[7] Обзор изменений по улучшению диагностики прохода SLP // Система code review LLVM. — 2018. — URL: `https://reviews.llvm.org/D38367` (дата обращения: 27.05.2018).