Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Системное Программирование

Андреев Илья Алексеевич

Оптимизирующий просмотр

Бакалаврская работа

Научный руководитель:
д. ф.-м. н., профессор А. Н. Терехов

Рецензент:
Исполнительный директор ООО "СофтКом" В.В. Оносовский

Санкт-Петербург
2021

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
Software Engineering

Andreev Ilya

Optimisation pass

Bachelor's Thesis

Scientific supervisor:
prof. chair SE, D. Sc., professor Andrey Terekhov

Reviewer:
Executive director of SoftCom Valentin Onossovski

Saint-Petersburg
2021

Оглавление

Введение

В наше время кроме активно развивающихся высокоуровневых языков программирования, таких как Python или Swift, продолжают свое развитие и языки низкого уровня. Примером таких языков служит язык C, созданный в 1972 году и в настоящее время являющийся фактическим стандартом для программирования систем мониторинга, контроля и управления благодаря своей простоте и эффективности [1].

Профессором А. Н. Тереховым на кафедре системного программирования ведется разработка языка RuC, продолжающего идеи языка C в сторону повышения безопасности и надежности программирования. Изначально этот проект был создан с учебными целями: сообщения об ошибках на русском языке и ограничение ненадежных конструкций (например, арифметика указателей или отсутствие контроля границ массивов) упрощали обучение начинающих программистов. Наряду с этим, язык использовался в промышленных проектах для программирования роботов, а на данный момент RuC встраивается в большой проект по созданию среды для разработки высоконадёжного программного обеспечения [2].

Работа компилятора RuC состоит из трех этапов: синтаксического анализа, оптимизирующего просмотра и генерации кодов целевой машины. Результатом работы первого этапа служит синтаксическое дерево – промежуточное представление исходной программы. Оно строится таким образом, что операторы исходного кода сопоставляются с внутренними узлами дерева, а операнды - с листьями. По результатам работы синтаксического анализатора можно генерировать код, в том числе и с некоторыми оптимизациями. В таком случае сгенерированный код будет локально-оптимальным, то есть в рамках одной конструкции языка будет создан наиболее производительный код. Оптимизирующий просмотр нужен для того, чтобы можно было генерировать глобально-оптимальный код. По дереву, полученному от анализатора, этот просмотр собирает необходимую информацию и строит новое дерево, в котором будут реализованы глобальные оптимизации. Он мо-

жет добавлять новые узлы, переставлять уже имеющиеся или удалять ненужные, например те, которые соответствуют неиспользуемому коду. В результате производительность кода, сгенерированного по новому дереву, будет выше. Такой код называется глобально-оптимальным.

На данный момент существует генерация из RuC в коды специальной виртуальной машины RuC-VM, разрабатываются генерации для платформы MIPS [3] и в коды LLVM [4], планируется разработка генерации для ARM [5]. Задачей данной работы является реализация ряда важнейших оптимизаций в рамках компилятора RuC для платформы MIPS.

1. Цели и задачи

1.1. Цель работы

Реализация оптимизирующего просмотра транслятора RuC, поддерживающего ряд оптимизаций, улучшающих производительность кода, для платформы MIPS.

1.2. Поставленные задачи

Для выполнения обозначенной цели необходимо выполнить следующие задачи:

- изучить популярные трансляторы языка C;
- составить список наиболее важных опитимизаций;
- реализовать оптимизации в оптимизирующем просмотре;
- оценить результаты и сравнить с аналогами;
- определить направления дальнейшей работы.

2. Обзор

Задача оптимизации генерируемого кода появилась еще во времена разработки первого компилятора FORTRAN, когда его разработчики стремились генерировать код лучше, чем средний код на языке ассемблера, написанный пользователем. В дальнейшем и другие компиляторы стали оптимизировать код, в частности, после серии IBM System/360 IBM предоставила два отдельных компилятора – быстро исполняющий разбор кода и более медленный оптимизирующий вариант.

Оптимизирующие компиляторы необходимы программистам, так как позволяют писать высокоуровневые программы, перекладывая задачу производства оптимального кода на транслятор. Благодаря этому программисты могут создавать хорошо читаемые и модульные программы, что значительно ускоряет ход разработки.

Оптимизации подразделяются на платформозависимые и платформонезависимые. Оптимизации последней группы в рамках данной работы являются наиболее предпочтительными, так как они могут быть использованы в будущем для генерации оптимального кода на другие платформы. Платформа MIPS, выбор которой был продиктован интересами заказчика, не поддерживает векторные операции, что накладывает ограничение на выбор оптимизаций для реализации. Поэтому были выбраны такие платформонезависимые оптимизации, которые дают наибольший прирост производительности, не учитывая тех, которые используют векторизацию.

Исходя из опыта разработки трансляторов научным руководителем, среди подходящих оптимизаций были выбраны на наш взгляд наиболее часто употребляемые и эффективные. Для реализации в первую очередь были выбраны следующие оптимизации:

- индуцированные переменные в циклах – оптимизация последовательных вырезок элементов массива;

- вычисление числа повторов цикла перед телом цикла - подсчет условия окончания цикла можно вынести из цикла, если оно не

меняется при исполнении инструкций цикла;

- редукция неиспользуемой индуктивной переменной цикла - устранение инструкций поддержки значения индуктивной переменной, если ее значение не используется в цикле;

- размыкание цикла - вынесение условного оператора из цикла, если его значение условия не меняется при итерациях по циклу.

Важно упомянуть, что существует несколько видов промежуточного представления программы, среди которых: абстрактное синтаксическое дерево, SSA [6] и sea-of-nodes [7]. SSA-форма используется во многих современных трансляторах, но для выбранных в данной работе оптимизаций достаточно и уже имеющегося в проекте RuC представления в виде синтаксического дерева.

Для сравнения и анализа были выбраны компиляторы языка C Clang [8] и GCC [9]. Такой выбор был сделан из-за их известности, распространенности и большого списка реализуемых оптимизаций. Оба этих компилятора могут транслировать код на платформу MIPS, что позволит в том числе и сравнить их с результатами данной работы.

2.1. Анализ Clang

Ниже приведен пример работы компилятора Clang на программе умножения матриц.

Исходный код:	Код внутреннего цикла:

`for(i = 0; i < 200; ++i) {`
`  for(j = 0; j < 200; ++j) {`
`  register int cij = 0;`
`  for(k = 0; k < 200; ++k)`
`  cij += a[i][k] * b[k][j];`
`  c[i][j] = cij;`
`  }`
`}`
	`addu $1, $9, $24`
	`lw $1, 0($1)`
	`lw $25, 0($14)`
	`mul $1, $25, $1`
	`addu $15, $15, $1`
	`addiu $24, $24, 4`
	`bne $24, $6, $BB0`
	`addiu $14, $14, 800`

Таблица 1: Пример кода, генерируемого Clang

В данном коде можно заметить, что компилятор ввел индуцированные переменные, произвел вычисление количества повторов цикла перед телом цикла, удалил ненужные индуктивные переменные, оптимизировал последовательные переходы, и использовал слот задержки. Однако этот код все же не является оптимальным: вместо того, чтобы создать индуцированную переменную для вырезки `a[i][k]`, Clang создал индуцированную переменную для смещения относительно `a[0][0]` по переменным `i` и `k`. Это увеличивает результирующий код на одну команду, тем самым увеличивая время исполнения программы. Данный подход хорошо сработал бы, если желаемое количество индуцированных переменных превосходило бы количество доступных регистров. Но в данном случае это решение неоптимально.

В рамках данной работы индуцированные переменные объявляются для каждой вырезки, а задачей распределения регистров для них будет заниматься следующий просмотр – кодогенератор.

2.2. Анализ GCC

GCC также реализует выше упомянутые оптимизации. Кроме того, код, генерируемый компилятором GCC, лишен недостатка оптимизации индуцированных переменных, получаемого в результате работы Clang, и для каждой вырезки в цикле объявляется отдельная индуцированная переменная. Однако, время исполнения данного кода превышает время исполнения кода, генерируемого компилятором Clang, что объясняется использованием неоптимальных инструкций целевой машины при кодогенерации. Впрочем, проблема выбора оптимальных инструкций при генерации кодов выходит за рамки данной работы.

3. Реализация

Выбранные для реализации оптимизации используют понятия переменных, не меняющих своего значения в цикле, и выражений, значение которых постоянно в рамках цикла.

Считаем, что переменная не меняет своего значения в цикле, если верно, что переменная не объявлена глобальной, а в цикле нет операций инкремента/декремента с этой переменной, присваивания ей значения и взятия адреса этой переменной. Так как в RuC запрещена арифметика указателей, никаким другим способом невозможно изменить значение переменной, поэтому других проверок производить не требуется.

Говорим, что значение выражение постоянно в рамках цикла, если оно не содержит вырезок, вызовов функций, а любая переменная, входящая в его состав, не меняет своего значения при итерациях цикла.

Также важно отметить, что разработанный в рамках данной работы оптимизирующий просмотр применяет оптимизации только к минимальным по вложению циклам, то есть циклам, не содержащих других циклов в своем теле. Это ограничение связано с тем, что целевая платформа имеет ограниченное число регистров, а оптимизация самых вложенных циклов даст больший прирост производительности в сравнении с оптимизациями внешних циклов.

3.1. Индуцированные переменные в циклах

Вырезка элемента массива – это часто используемая операция, которая реализуется большим количеством машинных команд. Нередкий случай – использование вырезок в цикле для итерации по массиву. В случае, если индекс вырезаемого элемента между итерациями тела цикла меняется линейно, для сокращения количества используемых инструкций для вырезки в циклах могут создаваться индуцированные переменные.

Индуцированная переменная - специальная переменная, в которую записывается адрес элемента массива, к которому исполняемый код будет обращаться при следующей итерации цикла. Перед началом испол-

нения цикла в эту переменную записывается адрес начала массива и вычисляется шаг между последовательными обращениями к элементам массива. После каждого исполнения тела цикла к значению этой переменной будет прибавляться шаг. Тогда при каждом исполнении тела цикла вместо последовательных операций подсчета адреса элемента массива (который включает в себя операции сложения и умножения) будет исполняться одно обращение по адресу, записанному в индуцированной переменной. Кроме того, такая замена может выполняться сразу для нескольких массивов в одном цикле, что позволит существенно ускорить генерируемый код.

Для того, чтобы можно было применить данную оптимизацию, выражения продвижения по циклу и само выражение вырезки должны иметь особый вид в промежуточном представлении компилятора, который будет гарантировать, что адрес необходимого элемента массива будет меняться линейно.

Выражение продвижения по циклу должно иметь следующий вид:

- `identifier++`, `identifier--`, `++identifier` или `--identifier`;
- `identifier += constant` или `identifier -= constant`;

Кроме того, переменная, обозначенная выше как `identifier`, должна быть целочисленного типа, так как вырезки можно совершать только выражениями целочисленного типа, и не должна менять своего значения кроме как в этом выражении. В случае, если выражение продвижения по циклу имеет указанный выше вид, будем называть такую переменную *индуктивной*.

Выражение вырезки может быть многомерным, но каждое составляющее выражение для одномерной вырезки должно иметь следующий вид:

- `identifier ± constant`
- `identifier`
- `constant`

Кроме того, переменная, обозначенная выше как `identifier` может быть индуктивной переменной цикла не более одного раза в рамках многомерной вырезки. Значения остальных переменных, входящих в выражение каждой вырезки, должны быть постоянными в пределах цикла.

Если описанные выше условия выполняются, то в рамках данной оптимизации компилятор в начало тела цикла добавляет узлы объявления индуцированных переменных `TIndVal`. Единственным его аргументом является ее уникальный в рамках цикла номер, чтобы различать, какая переменная в данный момент описывается или какой переменной заменяется выражение вырезки. Это позволит объявлять несколько индуцированных переменных в одном цикле. Также узел объявления индуцированной переменной имеет двух потомков: само выражение вырезки и выражение для подсчета шага. Выражение вырезки копируется из дерева и используется для получения адреса элемента массива, к которому программа будет обращаться при первой итерации цикла. При кодогенерации это выражение будет подсчитано, и адрес будет помещен в специально выделенный регистр. Второй потомок формируется при добавлении узла:

- если вырезка производится в последнем измерении цикла, то на этом месте стоит шаг индуктивной переменной, умноженный на размер памяти, занимаемый одним элементом массива;
- если вырезка производится не в последнем измерении цикла, а массив статический, то здесь будет длина массива, умноженная на размер элемента и на шаг индуктивной переменной;
- иначе (т.е. если массив динамический) здесь будет специальный узел, который укажет, границу какого измерения и какого массива нужно использовать для подсчета шага. Так как размер массива может задаваться переменной, значение которой до исполнения цикла могло измениться, кодогенератор вместо той же переменной должен будет использовать размерность массива, которая в RuC записывается в массив перед первым элементом.

3.2. Вычисление числа повторов цикла перед телом цикла

Данная оптимизация помогает уменьшить количество операций, выполняемых в циклах. В свою очередь, сокращение количества операций в циклах может существенно ускорить программу, так как эти инструкции могут повторяться сотни и тысячи раз.

В случае, если в ходе компиляции можно подсчитать, какое количество раз будет повторяться цикл, что позволит не вычислять каждый раз условие его окончания. Вместо этого на регистре, выделенном для счетчика цикла, будет оставшееся число повторов, и проверка условия окончания превратится в одну операцию сравнения счетчика с нулем.

Для того, чтобы было возможно применить данную оптимизацию, условие окончания цикла должно быть выражением сравнения (relational-expression или equality-expression в грамматике языка C). Если с одной стороны оператора сравнения стоит индуктивная переменная цикла, а выражение, стоящее с другой стороны, не меняет своего значения в рамках цикла, то оптимизирующий просмотр "разворачивает" оператор сравнения, так чтобы в левой части была индуктивная переменная, и в узел, сопоставляемый с циклом for, ставится соответствующий флаг, являющийся сигналом для кодогенератора, что условие может быть подсчитано один раз перед началом цикла.

3.3. Редукция неиспользуемой индуктивной переменной цикла

После выполнения описанных выше оптимизаций может произойти такое, что индуктивная переменная больше нигде в цикле не используется по значению. В таком случае оптимизирующий просмотр ставит соответствующий флаг, являющийся сигналом для кодогенератора, что можно не генерировать инструкции для изменения значения индуктивной переменно, а условие выхода из цикла будет может быть пересчитано с помощью индуцированной переменной.

3.4. Размыкание цикла

Размыкание цикла - это оптимизация, которая заключается в вынесении условного оператора из тела цикла и дублировании цикла. Эта оптимизация позволяет не пересчитывать условие оператора if и не выполнять условные переходы при каждой итерации цикла.

Результат применения данной оптимизации схематично показан ниже:

До оптимизации:	После оптимизации:

`for (expr1; expr2; expr3)`
`  A`
`  if (condition)`
`  B`
`  C`
	`if (condition)`
	`  for (expr1; expr2; expr3`
	`  A; B; C`
	`else`
	`  for (expr1; expr2; expr3`
	`  A; B`

Таблица 2: Схематичный пример размыкания цикла

Данная оптимизация применяется только для первого условного оператора, условие постоянно в рамках цикла. Подстановка делается только для одного условного оператора, так как количество кода может расти экспоненциально.

Важной особенностью данной оптимизации является то, что кроме того, что она сама по себе ускоряет исполнение программы, в связи с произведёнными изменениями циклы теперь могут удовлетворять условиям для прочих оптимизаций. В приведенном ниже примере оптимизация размыкания цикла позволит использовать индуцированную переменную вместо вырезки, так как после размыкания во втором цикле значение переменной `i` не меняется нигде, кроме выражения продвижения по циклу.

Данная особенность означает, что размыкание цикла оптимальнее делать в первую очередь.

До оптимизации:	После оптимизации:

`for (i = 0; i < 10; i++) {`
`  somefunc();`
`  if (a <= 5) {`
`  i++;`
`  } else {`
`  c[i] = i;`
`  }`
`}`
	`if (a <= 5)`
	`  for (i = 0; i < 10; i++) {`
	`  somefunc();`
	`  i++;`
	`  }`
	`else`
	`  for (i = 0; i < 10; i++) {`
	`  somefunc();`
	`  c[i] = i;`
	`  }`

Таблица 3: Пример размыкания цикла

4. Тестирование

Для каждой оптимизации, реализуемых оптимизирующим просмотром, который был разработан в рамках данной работы, были написаны тесты корректности, чтобы показать, что преобразования дерева не меняли результатов, получаемых в ходе использования оптимизированной программы.

Для демонстрации эффективности был создан пул тестов, состоящий из математических задач с матрицами, в том числе, содержащий программу умножения матриц разных размеров. Данные тесты содержат циклы и вырезки, что позволяет наглядно показать прирост производительности, полученный благодаря оптимизирующему просмотру. Каждый сгенерировнный компилятором RuC файл с ассемблерным кодом с помощью изменённой версии GCC для платы Байкал-T1 был преобразован в исполняемый файл. В свою очередь, исполняемый файл запускался на плате Байкал-T1 с замером времени исполнения с помощью утилиты time. Для каждого теста был подсчитан 95%-ый доверительный интервал времени исполнения с каждой реализованной оптимизацией по отдельности, а также со всеми ими в совокупности, произведено его сравнение с временем исполнения программ, генерируемых GCC 10.2.0 и Clang 12.0.5. Оба компилятора запускались с флагами `-O3` и `--target=mipsel-linux-gnu`.

В Таблице 4 показан средний по всем тестам прирост производительности для каждой оптимизации по отдельности, а в Таблице 5 приведено сравнение кода, генерируемого RuC со всеми оптимизациями, с кодами, полученными от GCC и Clang.

Оптимизация	Средний прирост, %
Вычисление числа повторов цикла	2,1 ± 0,3
Редукция индуктивной переменной	4,6 ± 1,5
Индуцированные переменные	55,3 ± 3,6

Таблица 4: Средний прирост производительности исполнения тестов

Компилятор	Время исполнения, сек
RuC	27,98 ± 8,24
Clang	27,77 ± 8,00
GCC	30,67 ± 7,99

Таблица 5: Среднее время исполнения тестов

В Таблице 6 отдельно показаны результаты замеров программы умножения матриц размером 199x199, так как код, генерируемый RuC на этом тесте, заметно обгоняет результаты GCC и Clang.

Компилятор	Время исполнения, сек
RuC	30,292 ± 0,074
Clang	44,583 ± 0,012
GCC	42,932 ± 0,014

Таблица 6: Время исполнения умножения матриц 199x199

Для демонстрации результата применения размыкания цикла был подсчитан 95%-ый доверительный интервал на основе 10 замеров при помощи утилиты time для вариантов RuC без оптимизаций, со всеми, кроме размыкания цикла, только с размыканием и со всеми оптимизациями на программе из Приложения 1. Результаты замеров приведены в Таблице 7.

Оптимизации	Время исполнения, сек
Без оптимизаций	133,484 ± 0,248
Все, кроме размыкания	125,020 ± 0,012
Только размыкание	131,063 ± 0,095
Все оптимизации	100,113 ± 0,187

Таблица 7: Демонстрация работы размыкания цикла

Заключение

В рамках данной работы были проанализированы оптимизации, выполняемые компиляторами GCC и Clang для целевой платформы MIPS. На основе проведенного анализа были выбраны оптимизации для реализации: вычисление количества повторов цикла перед телом цикла, индуцированные переменные, редукция неиспользуемой индуктивной переменной и размыкание цикла. Эффективность реализованных оптимизаций была показана на примере нескольких программ, решающих математические задачи с матрицами.

Направлением дальнейшей работы над оптимизирующим просмотром является реализация других оптимизаций, например: оптимизаций распределения регистров и вызовов функций, подстановок inline-функций, устранение недостижимого кода и неиспользуемых переменных. Кроме того, эти реализованные в рамках данной работы оптимизации будут использованы в других направлениях компилятора RuC, в частности, при генерации кода для платформы ARM.

Список литературы

[1] Barr Michael, Massa Anthony. Programming Embedded Systems, 2nd Edition, 2006.

[2] Терехов А.Н., Терехов М.А. Проект РуСи для обучения и создания высоконадежных программных систем. — Известия высших учебных заведений. Северо-Кавказский регион. Технические науки, 2017.

[3] MIPS32 Architecture – URL: `https://www.mips.com/products/architectures/mips32-3/` (accessed: 28.05.2021)

[4] The LLVM Compiler Infrastructure – URL: `https://llvm.org` (accessed: 28.05.2021)

[5] ARM Architecture – URL: `https://www.arm.com/why-arm/architecture` (accessed: 28.05.2021)

[6] Lots of authors. Static Single Assignment Book, 2018.

[7] Delphine Demange, Yon Fernández de Retana, David Pichardie. Semantic reasoning about the sea of nodes, 2018.

[8] LLVM's Analysis and Transform Passes – URL: `http://llvm.org/docs/Passes.html` (accessed: 28.05.2021)

[9] GCC optimization options – URL: `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html` (accessed: 28.05.2021)

Приложение 1

```
1  void main()
2  {
3      int a, i, b, c[10];
4      for (a = 0; a < 1000000000; a++)
5      {
6          for (i = 0; i < 10; i++)
7          {
8              b = 0;
9              if (a <= 5)
10             {
11                 i++;
12             }
13             else
14             {
15                 c[i] = i;
16             }
17             b = 1;
18         }
19     }
20 }
```