Санкт-Петербургский государственный университет

Направление «Математическое обеспечение и администрирование информационных систем»
Профиль «Системное программирование»

Иван Сергеевич Архипов

Генерация оптимального объектного кода

Бакалаврская работа

Научный руководитель:
заведующий кафедрой системного программирования,
д.ф.-м.н., профессор А.Н. Терехов

Рецензент:
Исполнительный директор ООО «СофтКом»
В.В. Оносовский

Санкт-Петербург
2021

SAINT-PETERSBURG STATE UNIVERSITY

Mathematical Software and Information Systems Administration
System Programming

Arkhipov Ivan

Generation of optimal object code

Bachelor's Thesis

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

Reviewer:
Executive director of «SoftCom»
Valentin Onossovski

Saint-Petersburg
2021

Содержание

Введение

Несмотря на то, что C является фактическим стандартом для низкоуровневого программирования и программирования встроенных систем, он имеет существенные недостатки, такие как арифметика указателей и отсутствие контроля за границами массива при доступе к элементу массива. Примечательно, что это недостатки самого языка, а не отдельных компиляторов. Эти недостатки негативно сказываются на безопасности языка C.

На кафедре системного программирования Санкт-Петербургского государственного университета разрабатывается транслятор языка RuC, представляющий собой усовершенствованную версию языка C [1]. На эту тему были сделаны научные публикации [2, 3]. На данный момент проект RuC стал промышленным, что свидетельствует об актуальности и практической значимости данного проекта.

С развитием RuC возникла необходимость в реализации оптимизаций. Без оптимизаций транслятор не сможет конкурировать с популярными компиляторами с языка C, такими, как GCC или Clang.

Генерация оптимального объектного кода является одной из основных и наиболее сложных задач в области разработки компиляторов. Данная работа посвящена оптимизации генерации кода для процессоров с архитектурой MIPS [4], что усложняет задачу. MIPS – это RISC-архитектура, допускающая огромную вариативность в генерации кода и его оптимизации.

Разработчик генератора кода сталкивается со сложными вопросами. Какие оптимизации должны быть реализованы? Какие оптимизации дадут значительный выигрыш в скорости работы программы? Какие оптимизации часто ускоряют работу программы, а какие срабатывают очень редко? Какие особенности архитектуры следует учитывать при оптимизации кода? Все это требует анализа других трансляторов, подходов к оптимизации и большой работы с машиной, в коды которой переводится программа.

Некоторые оптимизации не могут быть реализованы только в рамках генератора кода. Для сложных оптимизаций может потребоваться изменить абстрактное синтаксическое дерево или добавить дополнительную информацию в таблицы транслятора до начала кодогенерации, что требует включения в компилятор дополнительного оптимизирующего просмотра.

После реализации необходимо повторно проанализировать другие трансляторы, чтобы выяснить, где и как еще можно оптимизировать код, насколько и в каких случаях компилятор RuC проигрывает своим аналогам, а в каких случаях и на сколько он стал лучше.

Работа состоит из пяти глав. В первой главе ставится цель работы и описываются конкретные задачи для достижения цели. Во второй главе представлен обзор оптимизированного кода, сгенерированного GCC и Clang, а также обзор подходов к оптимизации. Третья глава содержит общий подход к реализации оптимизаций и детали реализации конкретных оптимизаций. В четвёртой главе представлены измерения, их оценка и повторное сравнение с GCC и Clang. В пятой главе описаны результаты работы и возможности для дальнейших исследований.

1 Цели и задачи

Целью работы является реализация оптимизирующей компоненты в рамках кодогенератора кодов MIPS для компилятора RuC.

Для достижения обозначенной цели были поставлены следующие задачи:

- Анализ генерации кодов MIPS в двух наиболее распространенных компиляторах с C: GCC и Clang.
- Составление списка оптимизаций и подготовка требований для оптимизирующего просмотра.
- Реализация оптимизаций в кодогенераторе RuC.

• Оценка результатов и сравнение с аналогами: GCC и Clang.

2 Обзор

В качестве аналогов для сравнения и анализа были взяты трансляторы с языка C в коды MIPS: GCC [5] и Clang [6]. Такой выбор обусловлен их популярностью, распространённостью и большим списком оптимизаций, реализованных в этих трансляторах.

Для тестирования скорости была выбрана программа для умножения матриц 200x200 1000 раз. Код тестовой программы приведен в Приложении 1. Размер матриц и количество повторов были взяты такими с целью разумного времени выполнения программы, чтобы не ждать долго окончания теста и погрешность измерения времени незначительно влияла на результат.

Почему был выбран данный тест? Во-первых, он содержит важные конструкции языка, такие как циклы и вырезки массива. Во-вторых, умножение матриц является широко распространенной задачей в различных вычислительных программах.

Прежде всего, необходимо измерить скорость кода, полученного с использованием компиляторов GCC и Clang. Все тесты в работе проводились на процессоре Байкал-T1 [7]. Время измерялось с помощью утилиты time [8] 10 раз, а затем рассчитывался доверительный интервал с уровнем доверия 95%. Результаты измерений представлены в Таблице 1.

Транслятор	**Время**
Clang	31,466 s ± 0,02 s
GCC	36,037 s ± 0,005 s

Таблица 1: Сравнение Clang и GCC

Больше всего на время работы программы влияет оптимизация самого внутреннего цикла, так как команды в нём исполняются больше всего раз. Анализ и сравнение внуренних циклов требуют

особого внимания.

2.1 Анализ Clang

Код внутреннего цикла программы из Приложения 1 (строки 28-31), полученный после трансляции Clang, приведён во вставке Algorithm 1. Здесь можно увидеть несколько важных оптимизаций: индуцированные переменные, которые создаются для вырезки элементов массива, оптимизация последовательных переходов, вычисление количества повторов цикла перед телом цикла, использования слота задержки, удаление ненужных индуктивных переменных, то есть переменных, по которым совершаются итерации цикла.

```
$BB0_12:
    addu $1, $9, $24      # вычисление адреса a[i][k]
    lw $1, 0($1)          # вырезка a[i][k]
    lw $25, 0($14)        # вырезка b[k][j]
    mul $1, $25, $1       # умножение
    addu $15, $15, $1     # сложение
    addiu $24, $24, 4     # инкремент инд переменной
    bne $24, $6, $BB0_12  # переход
    addiu $14, $14, 800   # инкремент инд переменной
```

Algorithm 1: Внутренний цикл Clang

Однако даже такой хороший код ещё не до конца оптимизирован. Во время трансляции была создана индуцированная переменная не для a[i][k], а для смещения относительно a[0][0] по переменным i и k. Из-за этого в начале цикла необходимо вычислять адрес a[i][k], что добавляет одну лишнюю команду во внутреннем цикле. Такой подход хорошо работает, когда во внутреннем цикле имеется много вырезок вида array[i][k] из большого количества массивов. Из-за этого для вырезки из каждого массива невозможно завести индуцированную переменную, так как не хватит регистров.

Поэтому создаётся индуцированная переменная для смещения по переменным i и k, а адреса считаются отдельно. Но в данном тесте этого не нужно, более того, для b[k][j] была создана отдельная индуцированная переменная. Такое решение со стороны Clang неоптимально.

2.2 Анализ GCC

Код внутреннего цикла программы из Приложения 1 (строки 28-31), полученный после трансляции GCC, приведён во вставке Algorithm 2. Как и в Clang, здесь тоже реализованы основные и самые важные оптимизации. GCC лишён упомянутого выше недостатка Clang. Для данной программы индуцированные переменные реализованы наилучшим образом, для каждой вырезки создана своя индуцированная переменная. Кроме того, GCC заменил две команды сложения и умножения на одну команду madd. Эта команда перемножает значения из двух регистров и добавляет результат умножения к специальным регистрам hi и lo [9].

```
$L8:
   addiu $2,$2,800  # инкремент инд переменной
   lw $5,0($3)      # вырезка a[i][k]
   lw $4,-800($2)   # вырезка b[k][j]
   addiu $3,$3,4    # инкремент инд переменной
   bne $6,$2,$L8    # переход
   madd $5,$4       # умножение и сложение
```

Algorithm 2: Внутренний цикл GCC

Однако время исполнения программы, полученной в ходе трансляции GCC, больше, чем полученной в ходе трансляции Clang. Как оказалось, причина заключается в использовании команды madd. Был поставлен эксперимент. В объектном коде, сгенерированном GCC, команда madd была вручную заменена на две команды mul и addu, и было замерено время исполнения программы на плате

Байкал-T1. Результат представлен в Таблице 2.

	Время
С использованием команды madd	36.12 s ± 0,012 s
Без использования команды madd	27.26 s ± 0,007 s

Таблица 2: Код GCC с madd и без madd

Чтобы убедиться в результатах, дополнительно был проведён ещё один эксперимент. В объектном коде программы, сгенерированном Clang, вручную были заменены две команды mul и addu на одну команду madd, и было измерено время исполнения кода программы. Результат представлен в Таблице 3.

	Время
С использованием команды madd	37.18 s ± 0,015 s
Без использования команды madd	31.46 s ± 0,008 s

Таблица 3: Код Clang с madd и без madd

Очевидно, что команда madd не должна использоваться для оптимизации.

2.3 Оптимизация на основе SSA-формы

Существует несколько подходов к реализации оптимизаций. Необходимо рассмотреть их и выбрать подходящий.

Многие современные трансляторы используют промежуточное представление в форме SSA для реализации оптимизаций [5, 6]. SSA-формой является такое промежуточное представление, в котором каждой переменной присвается значение только один раз. Также оптимизации на основе SSA продемонстрированы в книгах [10, 11]. Оптимизациям в SSA-форме посвящены множество научных работ [12, 13].

SSA-форма может быть и высокоуровневой, и низкоуровневой, что даёт такому представлению гибкость. Высокоуровневое пред-

ставление близко к исходному языку, а низкоуровневое – к целевому коду. Для трансляции с языков высокого уровня может быть создано несколько представлений в форме SSA, что позволяет реализовать оптимизации на разных уровнях абстракции. Низкоуровневое представление хорошо подходит для реализации машинно-зависимых оптимизаций.

Однако сложность такого подхода заключается в его трудоёмкости. Необходимо сначала разработать SSA форму промежуточного представления, затем реализовать кодогенератор в коды данного представления и кодогенератор из кодов данного представления в коды реальной машины, а затем уже реализовывать оптимизатор.

2.4 Оптимизация на основе AST-формы

Другой формой промежуточного представления является абстрактное синтаксическое дерево. Оно является представлением высокого уровня и отображает иерархическую структуру программ на транслируемом языке программирования. Данная форма плохо подходит для низкоуровневых оптимизаций, и, как следствие, её недостаточно для эффективной трансляции высокоуровневых языков программирования.

Однако достоинством такого подхода является его простота. Нет необходимости дополнительно разрабатывать SSA форму, писать кодогенератор в её коды и из её кодов в коды целевой машины. Благодаря простоте увеличивается скорость компиляции, это достоинство используется, например, в TCC [14].

3 Реализация

3.1 Общий подход

По результатам анализа решений, реализованных в компиляторах GCC и Clang, было принято решение о реализации следующих

оптимизаций:

- Оптимизация последовательных переходов
- Вычисление количества повторов цикла перед телом цикла
- Использования слота задержки
- Индуцированные переменные
- Удаление ненужных индуктивных переменных

В RuC в настоящее время нет необходимости в промежуточном представлении в форме SSA. RuC – это достаточно низкоуровневый язык программирования, поэтому нет необходимости создавать сложные формы промежуточного представления для анализа и оптимизации программных конструкций.

Некоторые оптимизации могут быть реализованы непосредственно в генераторе кода без изменения абстрактного синтаксического дерева. В процессе обхода дерева в зависимости от оптимизации устанавливаются флаги для генерации определенных команд, специфичных для каждой оптимизации. Например, в последующих разделах будет описываться оптимизация вычисления количества повторов цикла. Там нужно генерировать не команду сравнения с нулём, как было до оптимизации, а команду сравнения двух регистров. Для этого в процессе обработки узла TFor устанавливается флаг для генерации таких команд.

Для сложных оптимизаций необходима специфическая дополнительная информация. К таким оптимизациям относятся индуцированные переменные и устранение ненужных индуктивных переменных. Необходимую информацию в абстрактное синтаксическое дерево добавляет оптимизирующий просмотр, который работает с деревом перед фазой кодогенерации. Реализация оптимизирующего просмотра выходит за рамки данной работы.

3.2 Оптимизация последовательных переходов

В процессе генерации кода для циклов могут сформироваться ненужные команды ветвления, которые создают последовательность переходов. Рассмотрим код самого внутреннего цикла из Приложения 1 (строки 28-31), генерируемый транслятором RuC. Код на ассемблере приведён во вставке Algorithm 3.

Сначала вычисляется условие цикла, затем оно проверяется, а в конце выполняется переход к вычислению и проверке условия. Этот цикл может быть оптимизирован путем установки условия и команды ветвления перед циклом и в конце цикла. Оптимизированный код представлен во вставке Algorithm 4.

```
BEGLOOP23:
   addi $t1, $s2, -200  # вычисление условия
   bgez $t1, ELSE22     # условный переход
   nop                  # заполнение слота задержки
   ...                  # тело цикла
CONT23:
   addi $s2, $s2, 1     # инкремент индуктивной
                        # переменной
   j BEGLOOP23          # безусловный переход
   nop                  # заполнение слота задержки
ELSE22:
   ...                  # код после цикла
```

Algorithm 3: Цикл до оптимизации

Эта оптимизация реализуется в генераторе кода в модуле обработки узла TFor и узлов условных выражений. В модуле обработки узла TFor структура цикла организуется соответствующим образом. В модуле обработки узлов условного выражения генерируются соответствующие команды ветвления.

Чтобы проверить корректность, были проведены тесты для различных условий выхода из цикла for [15]. В данной работе не рассматривается формальное доказательство правильности оптимизаций. Корректность выполненной оптимизации проверяется мето-

дом тестирования.

```
    addi $t1, $s2, -200   # вычисление условия
    bgez $t1, ELSE26      # условный переход
    nop                   # заполнение слота задержки
BEGLOOP27:
           ...            # тело цикла
CONT27:
    addi $s2, $s2, 1      # инкремент индуктивной
                          # переменной
    addi $t1, $s2, -200   # вычисление условия
    bltz $t1, BEGLOOP27   # условный переход
    nop                   # заполнение слота задержки
ELSE26:
    ...                   # код после цикла
```

Algorithm 4: Цикл после оптимизации

3.3 Вычисление количества повторов цикла перед телом цикла

При генерации кода для циклов лучше сгенерировать код условия выхода перед телом цикла. Условие будет вычисляться перед циклом и не будет вычисляться на каждой итерации цикла. Необходимо запомнить условие в регистре и в конце цикла сравнить значение индуктивной переменной с этим регистром. Код без этой оптимизации для самого внутреннего цикла программы в Приложении 1 (строки 28-31) показан во вставке Algorithm 4, а оптимизированный код показан во вставке Algorithm 5.

Эта оптимизация требует выделения регистра для хранения условия. Из-за ограниченного количества регистров оптимизация реализуется только для самых внутренних циклов. Эта оптимизация реализована в генераторе кода в модуле обработки узла TFor и уз-

```
   addi $s5, $0, 200        # вычисление и
                            # сохранение условия
   bge $s2, $s5, ELSE26     # переход
BEGLOOP27:
   ...                      # тело цикла
CONT27:
   addi $s2, $s2, 1         # инкремент
                            # индуктивной переменной
   bne $s2, $s5, BEGLOOP27  # переход
ELSE26:
   ...                      # код после цикла
```

Algorithm 5: Цикл после оптимизации

лов условных выражений. В модуле обработки узла TFor структура цикла организуется соответствующим образом. В модуле обработки узлов условного выражения генерируются соответствующие команды ветвления.

Могут возникнуть ситуации, когда условие выхода из цикла слишком сложное, поэтому применить эту оптимизацию невозможно. Возможность применимости будет рассмотрена на более ранних этапах трансляции, что выходит за рамки данной работы.

Чтобы проверить корректность, были проведены тесты для различных условий выхода из цикла for [16].

3.4 Использование слота задержки

Слот задержки – это слот команды, выполняемой без эффектов предыдущей команды. Например, инструкция, расположенная сразу после инструкции перехода, будет выполнена вне зависимости от того, будет выполнен переход или нет. В архитектуре MIPS слоты задержки есть у всех команд переходов [9]. Это особенность

конвейерных вычислений.

Директива архитектуры MIPS ".set reorder" [17] и опция ассемблера GCC "-mcompact-branches=optimal" [18] позволяют заполнить слот задержки предыдущей командой, если это возможно. Если это невозможно, то вставляется команда nop. Команда может быть вставлена в слот задержки только в том случае, если она не работает с регистрами, используемыми в команде ветвления.

```
   addi $s5, $0, 200        # вычисление и
                            # сохранение условия
   bge $s2, $s5, ELSE26     # переход
   addi $s2, $s2, -1        # декремент
                            # индуктивной переменной
   addi $s5, $s5, -1        # декремент условия
BEGLOOP27:
   addi $s2, $s2, 1         # инкремент
                            # индуктивной переменной
   ...                      # тело цикла
CONT27:
   bne $s2, $s5, BEGLOOP27  # переход
   addi $s5, $s5, 1         # инкремент условия
ELSE26:
   ...                      # код после цикла
```

Algorithm 6: Цикл после оптимизации

Суть оптимизации заключается в том, чтобы позволить GCC заполнить слот задержки. Необходимо переместить инкремент индуктивной переменной в начало цикла и реорганизовать код цикла. Код без этой оптимизации для самого внутреннего цикла программы в Приложении 1 (строки 28-31) показан во вставке Algorithm 5, а оптимизированный код показан во вставке Algorithm 6.

Эта оптимизация реализована в генераторе кода в модуле обработки узла TFor. В модуле обработки узла TFor структура цикла

организуется соответствующим образом.

Чтобы проверить корректность, были проведены тесты для различных условий выхода из цикла for [19].

3.5 Индуцированные переменные

```
    addi $s1, $0, 0          # j = 0
    addi $s2, $0, 5          # вычисление условия
    bge $s1, $s2, ELSE4      # переход
    addi $s1, $s1, -1        # декремент j
    addi $s2, $s2, -1        # декремент условия
BEGLOOP5:
    addi $s1, $s1, 1         # инкремент j
    lw $s4, 88($sp)          # команды вырезки a[i][j]
    sll $t1, $s0, 2
    add $s4, $s4, $t1
    lw $s4, 0($s4)
    sll $t1, $s1, 2
    add $s4, $s4, $t1
    move $s3, $s4
    add $s4, $s0, $s1       # i + j
    sw $s4, 0($s3)          # запись в память
    bne $s1, $s2, BEGLOOP5  # переход
    addi $s2, $s2, 1        # инкремент условия
ELSE4:                      # код после цикла
```

Algorithm 7: Цикл до оптимизации

Индуцированные переменные создаются перед циклом, чтобы не вычислять адрес элементов массива в теле цикла, когда индекс вырезки зависит от индуктивной переменной цикла. Перед телом

цикла вычисляется и записывается в регистр адрес нулевого элемента массива. В самом теле цикла во время каждой итерации к адресу прибавляется некоторое фиксированное значение. Это значение зависит от типа элементов, размера и размерности массива.

```
    addi $s1, $0, 0         # j = 0
    addi $s2, $0, 5         # вычисление условия
    bge $s1, $s2, ELSE4     # переход
    lw $s4, 88($sp)         # команды вырезки a[i][j]
    sll $t1, $s0, 2
    add $s4, $s4, $t1
    lw $s4, 0($s4)
    sll $t1, $s1, 2
    add $s4, $s4, $t1
    move $s3, $s4
    addi $s1, $s1, -1       # декремент j
    addi $s2, $s2, -1       # декремент условия
BEGLOOP5:
    addi $s1, $s1, 1        # инкремент j
    add $s5, $s0, $s1       # i + j
    sw $s5, 0($s3)          # запись в память
    addi $s3, $s3, 4        # инкремент
                            # индуцированной переменной
    bne $s1, $s2, BEGLOOP5  # переход
    addi $s2, $s2, 1        # инкремент условия
ELSE4:                      # код после цикла
```

Algorithm 8: Цикл после оптимизации

Было принято решение создавать индуцированные переменные только в циклах for наибольшей вложенности, так как оптимизация таких циклов даёт наибольший выигрыш по времени. Для каких именно вырезок из массивов будут создаваться индуцированные переменные, решает оптимизирующий просмотр дерева, реализа-

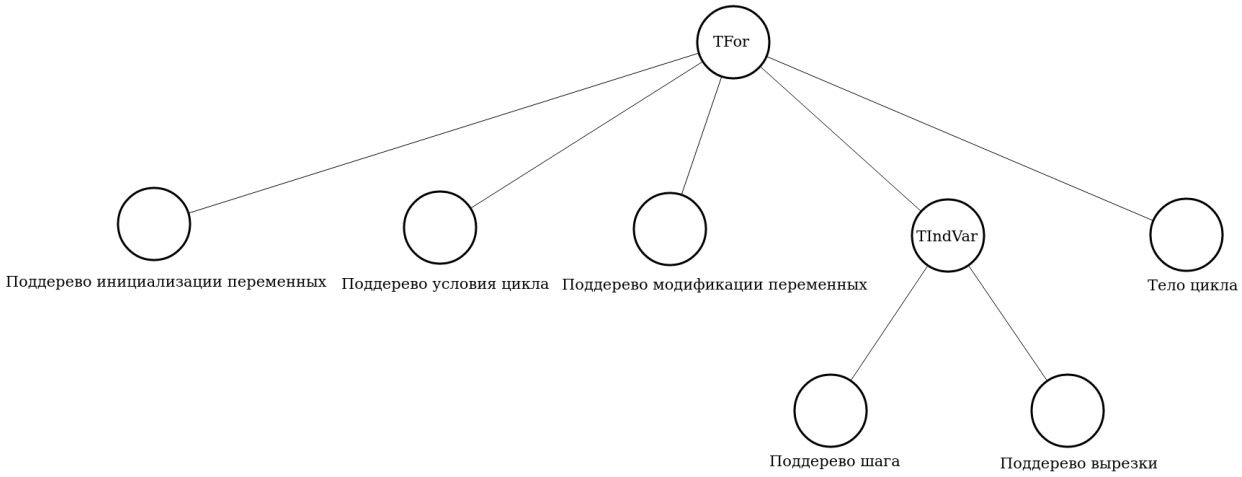

Рис. 1: Дерево для цикла For

ция которого не входит в рамки данной работы.

После оптимизирующего просмотра в дереве перед телом цикла for находятся узлы индуцированных переменных. В качестве атрибутов они содержат ссылку на вырезку нулевого элемента массива, прибавляемое к адресу каждую итерацию цикла значение и номер индуцированной переменной. Пример дерева представлен на Рис. 1. TIndVar обозначает узел индуцированной переменной.

В самом теле цикла оптимизирующий просмотр вместо вырезки из массива ставит номер индуцированной переменной.

В кодогенераторе, когда встречается узел индуцированной переменной, генерируются команды вырезки из массива, и адрес сохраняется в отдельном регистре. Когда в теле цикла кодогенератор встречает индуцированную переменную, он по её номеру определяет регистр, в котором хранится нужный адрес, и не вычисляет заново этот адрес. В конце тела цикла кодогенератор генерирует команды изменения адресов исходя из значений, предоставленных ему оптимизирующим просмотром.

Рассмотрим пример из Приложения 2. Код самого внутреннего цикла for (строки 8-11) после трансляции RuC без оптимизации

представлен во вставке Algorithm 7.

После оптимизации код вырезки вынесен за пределы цикла, а к адресу каждую итерацию прибавляется 4, так как массив имеет тип int, и вырезка идёт по последнему измерению. Оптимизированный код представлен во вставке Algorithm 8.

Чтобы проверить корректность, были проведены тесты для различных размерностей статических и динамических массивов [20].

3.6 Удаление ненужных индуктивных переменных

```
BEGLOOP6:
    addi $s0, $s0, 1        # инкремент i
    lw $s5, 0($s3)          # загрузка из памяти a[i]
    addi $s5, $s5, 1        # прибавление 1
    sw $s5, 0($s2)          # сохранение в b[i]
    addi $s2, $s2, 4        # инкремент инд переменной
    addi $s3, $s3, 4        # инкремент инд переменной
    bne $s0, $s1, BEGLOOP6  # переход
```

Algorithm 9: Цикл до оптимизации

Возможна ситуация, когда индуктивная переменная цикла используется только для вырезки из массива. Например, такая ситуция продемонстрирована в самом внутреннем цикле for в программе из Приложения 1 (строки 28-31). В таком случае можно не тратить команды на инкремент индуктивной переменной, а использовать индуцированные переменные для проверки выхода из цикла.

Рассмотрим пример из Приложения 3. Второй цикл (строки 11-14) использует переменную i только для вырезки из массива. После оптимизации индуцированных переменных прямых использований

i в цикле нет, поэтому от неё можно избавиться. Цикл до оптимизации представлен во вставке Algorithm 9, а после неё во вставке Algorithm 10.

Для данной оптимизации необходимо пересчитать условие выхода из цикла, так как проверка на выход осуществляется по индуцированной переменной. Для этого необходимо перед телом цикла умножить условие на шаг индуцированной переменной и прибавить его к адресу нулевого элемента.

```
    addi $t1, $0, 4
    mul $s1, $s1, $t1       # умножение условия на 4
    add $s1, $s1, $s2       # прибавление адреса
                            # индуцированной переменной
BEGLOOP6:
    lw $s5, 0($s3)          # загрузка из памяти a[i]
    addi $s5, $s5, 1        # прибавление 1
    sw $s5, 0($s2)          # сохранение в b[i]
    addi $s2, $s2, 4        # инкремент инд переменной
    addi $s3, $s3, 4        # инкремент инд переменной
    bne $s2, $s1, BEGLOOP6 # переход
```

Algorithm 10: Цикл после оптимизации

Чтобы проверить корректность, были проведены тесты для различных размерностей статических и динамических массивов и различного количества массивов [21].

4 Измерения и оценка

Для демонстрации эффективности оптимизаций и замера времени была выбрана программа умножения матриц 200x200 1000 раз, код которой приведен в Приложении 1. Размер матриц и количество повторов были взяты из соображений разумного времени работы программы и незначительности влияния погрешности измерения.

Почему был выбран именно данный тест? Во-первых, он содержит важные конструкции языка, такие как циклы и вырезки массива. Во-вторых, умножение матриц является широко распространенной задачей в различных вычислительных программах.

Все тесты проводились на процессоре Байкал-T1 [7]. Время измерялось с помощью утилиты time [8] 10 раз, а затем рассчитывался доверительный интервал с уровнем доверия 95%. Результаты измерений представлены в Таблице 4.

	Время
Без оптимизаций	97,81 s ± 0,0459 s
Оптимизация последовательных переходов	94,559 s ± 0,0102 s
Предыдущее + Оптимизация вычисления условия	92,037 s ± 0,0382 s
Предыдущее + Использование слота задержки	89,63 s ± 0,0073 s
Предыдущее + Индуцированные переменные	47,211 s ± 0,0219 s
Предыдущее + Устранение индуктивных переменных	43,861 s ± 0,0019 s

Таблица 4: Оптимизации RuC

Время работы программы умножения матриц стало значительно меньше. Однако оно всё ещё больше, чем у аналогов RuC (Таблица 5).

Транслятор	**Время**
Clang	31,466 s ± 0,02 s
GCC	36,037 s ± 0,0056 s
RuC	43,861 s ± 0,0019 s

Таблица 5: Время умножения матриц 200x200

Данный результат является неожиданным, так как код внутреннего цикла программы из Приложения 1 (строки 28-31), полученный после трансляции RuC (вкладка Algorithm 11), мало чем отличается от кодов, полученных после трансляции Clang и GCC (вкладки Algorithm 1 и Algorithm 2).

```
BEGLOOP27:
    lw $t8, 0($s6)          # вырезка a[i][k]
    lw $t0, 0($s7)          # вырезка b[k][j]
    mul $t8, $t8, $t0       # умножение
    add $s4, $s4, $t8       # сложение
    addi $s6, $s6, 4        # инкремент инд переменной
    addi $s7, $s7, -804     # инкремент инд переменной
    bne $s6, $s5, BEGLOOP27 # переход
```

Algorithm 11: Цикл после оптимизаций

Были устранены недостатки аналогов, которые описаны в обзоре, однако программа всё равно работает медленнее. Для выяснения причин были проведены дополнительные тесты с умножением матриц других размеров. Время работы программы умножения матриц 50x50 16000 раз представлено в Таблице 6, а время работы программы умножения матриц 100x100 8000 раз представлено в Таблице 7.

Есть предположение, что подобный эффект с разницой во времени происходит из-за размещения элементов массива в кэше. В RuC для хранения строки требуется несколько больше ячеек па-

Транслятор	**Время**
Clang	7,451 s ± 0,002 s
RuC	7,791 s ± 0,002 s
GCC	8,989 s ± 0,002 s

Таблица 6: Время умножения матриц 50x50

Транслятор	**Время**
Clang	28,501 s ± 0,002 s
RuC	29,152 s ± 0,002 s
GCC	34,751 s ± 0,002 s

Таблица 7: Время умножения матриц 100x100

мяти, чем элементов массива в строке. Это связано с хранением дополнительной информации о массиве в RuC. Из-за этого для программы, сгенерированной Clang и GCC, строка массива может целиком помещаться в кэш, а для программы, сгенерированной RuC, нет. Это предположение в дальнейшем требует дополнительного рассмотрения.

Но почему Clang всё равно работает лучше RuC? Предполагается, что такое происходит из-за порядка команд и работы конвейера. Эта гипотеза требует дальнейшей проверки уже за рамками данной работы.

5 Заключение

В данной работе решается задача оптимизации генерации ассемблерного кода архитектуры MIPS. Были проанализированы компиляторы GCC и Clang. На основе проведенного анализа были выбраны оптимизации для реализации: оптимизация последовательных переходов, вычисление количества повторов цикла перед телом цикла, использования слота задержки, индуцированные переменные и удаление ненужных индуктивных переменных. Эффектив-

ность оптимизаций была показана на примере умножения матриц. Таким образом, генерация кода для циклов и вырезок была оптимизирована в трансляторе RuC.

Дальнейшим направлением работ по оптимизации генерации RuC в архитектуру MIPS могут быть: оптимизация вызовов функций, арифметических операций, распределение регистров, устранение неиспользуемых переменных.

Также работа оставляет после себя несколько открытых проблем для исследований. Например, в обзоре было указано, что команда madd сильно замедляет программу. Но почему так происходит? Это хорошая тема для исследований свойств архитектуры процессора. Сам результат данной работы может стать объектом для исследования. Почему оптимизированная программа работает всё ещё медленнее аналогов в отдельных случаях? Вопрос остаётся открытым.

Список литературы

[1] Github проекта RuC – URL: https://github.com/andrey-terekhov/RuC (accessed: 04.04.2021)

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

[3] Code generation for floating-point arithmetic in architecture MIPS. Arkhipov I.S. Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS). 2020;32(3):49-56

[4] SYSTEM V APPLICATION BINARY INTERFACE MIPS RISC Processor, 3rd Edition

[5] GCC official site – URL: https://GCC.gnu.org/ (accessed: 04.04.2021)

[6] LLVM official site – URL: https://llvm.org/ (accessed: 04.04.2021)

[7] Baikal-T1 specifications – URL: http://www.baikalelectronics.ru/products/35/ (accessed: 04.04.2021)

[8] time(1) Linux manual page – URL: https://man7.org/linux/man-pages/man1/time.1.html (accessed: 04.04.2021)

[9] MIPS Architecture for Programmers Volume II-A: The MIPS32 Instruction Set Manual

[10] Aho, Alfred Vaino; Lam, Monica Sin-Ling; Sethi, Ravi; Ullman, Jeffrey David (2006). Compilers: Principles, Techniques, and Tools (2 ed.)

[11] S. Muchnick. Advanced Compiler Design and Implementation, 1997

[12] Kathleen Knobe, Vivek Sarkar. Array SSA form and its use in parallelization. POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, January 1998

[13] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, Steve Arthur Zdancewic. Formal verification of SSA-based optimizations for LLVM. PLDI '13: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2013

[14] Tiny C Compiler Reference Documentation – URL: https://bellard.org/tcc/tcc-doc.html (accessed: 04.04.2021)

[15] Тесты для оптимизации последовательных переходов – URL: https://github.com/IvanArkhipov1999/RuC/tree/mips/tests/mips/optimizations/cycle_jump_reduce (accessed: 04.04.2021)

[16] Тесты для оптимизации вычисления условия – URL: https://github.com/IvanArkhipov1999/RuC/tree/mips/tests/mips/optimizations/cycle_condition_calculation (accessed: 04.04.2021)

[17] MIPS Assembly Language Programmer's Guide

[18] GCC MIPS options – URL: https://gcc.gnu.org/onlinedocs/gcc/MIPS-Options.html (accessed: 04.04.2021)

[19] Тесты для оптимизации слота задержки – URL: https://github.com/IvanArkhipov1999/RuC/tree/mips/tests/mips/optimizations/delay_slot (accessed: 04.04.2021)

[20] Тесты для оптимизации индуцированных переменных – URL: https://github.com/IvanArkhipov1999/RuC/tree/mips/tests/mips/optimizations/ind_var (accessed: 04.04.2021)

[21] Тесты для оптимизации редукции ненужной индуктивной переменной – URL: https://github.com/IvanArkhipov1999/RuC/

tree/mips/tests/mips/optimizations/ind_var_reduction (accessed: 04.04.2021)

Приложение 1

```
1  void main() {
2    int a[200][200], b[200][200], c[200][200];
3    register int i, j, k, v;
4
5    for (i = 0; i < 200; ++i)
6    {
7      for (j = 0; j < 200; ++j)
8      {
9        a[i][j] = i * j;
10     }
11   }
12
13   for (i = 0; i < 200; ++i)
14   {
15     for (j = 0; j < 200; ++j) {
16
17       b[i][j] = i + j;
18     }
19   }
20
21   for (v = 0; v < 1000; ++v)
22   {
23     for(i = 0; i < 200; ++i)
24     {
25         for(j = 0; j < 200; ++j)
26         {
27           register int cij = 0;
28           for(k = 0; k < 200; ++k)
29           {
30               cij += a[i][k] * b[k][j];
31           }
32           c[i][j] = cij;
33         }
34     }
35   }
36   printf("%i\n", c[0][0]);
37 }
```

Приложение 2

```
1  void main()
2  {
3    register int i, j;
4    int a[5][5];
5
6    for (i = 0; i < 5; ++i)
7    {
8      for (j = 0; j < 5; ++j)
9      {
10       a[i][j] = i + j;
11     }
12   }
13   printid(a);
14 }
```

Приложение 3

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