Санкт–Петербургский Государственный Университет
Институт Информационных Технологий

Современные средства генерации эффективного кода

Д.Ю.Булычев
Е.Н.Вигдорчик
Д.С.Ломов
М.Н.Смирнов

Санкт–Петербург
2001

Содержание

1 Введение

В данном отчете описываются четыре свободно распространяемые системы:

- GCC (GNU Compiler Collection), основное средство разработки в среде GNU, один из наиболее авторитетных оптимизирующих компиляторов [30].
- lcc — ANSI C компилятор, разработанный в Принстонском университете, США [31].
- vpolcc — компилятор, построенный на базе VPO (Very Portable Optimizer) в Университете Вирджинии, США [34].
- SUIF — компилятор, разработанный в рамках проекта SUIF (Stanford Unified Intermediate Format) в Стэнфордском университете, США [33].

Можно сказать, что эти системы представляют собой последние достижения в области построения настраиваемых оптимизирующих компиляторов. Их сравнительное исследование проводилось с целью выяснения пригодности применения использованных в них подходов к задаче построения технологии кодизайна.

Под технологией кодизайна здесь понимается набор программных средств вместе с методологией их использования. Актуальность создания такой технологии обусловлена развитием направления программно-аппаратной соразработки *(hardware-software codesign)*, характерным свойством которого является необходимость быстрого и автоматического построения набора средств для получения эффективного машинного кода, как-то:

- компилятора;
- оптимизатора;
- отладчика;
- профилятора;
- симулятора;
- ассемблера и дизассемблера;

Следует отметить, что работы в этом направлении интенсивно ведутся во многих ведущих исследовательских центрах мира; тем не менее, данная задача еще далека от решения.

Ниже даны краткие характеристики каждой из исследованных систем, приведены результаты сравнительного тестирования и изложены основные соображения относительно применимости этих средств в области кодизайна.

2 Семейство компиляторов GCC

GCC (первоначальное название — GNU C Compiler) разработан в среде GNU в качестве основного компилятора. Первой его версией считается версия 1.17, выпущенная 9 января 1989 года. По мере разработки этот компилятор подвергался существенным усовершенствованиям и расширениям (об интенсивности работ говорит хотя бы то, что за это время было выпущено более семидесяти версий).

В 1997 году появилась дополнительная ветка развития GCC — EGCS — который отличался от GCC более полной тестированностью и небольшими усовершенствованиями. С 1999 года EGCS целиком заменил собой GCC, а сама аббревиатура стала означать GNU Compilers Collection в связи с тем, что открытая архитектура компилятора позволила создать над общим оптимизирующим ядром многоязыковую систему программирования. В настоящий момент среда GCC поддерживает, кроме C и C++, языки Objective C, Chill, Fortran и Java.

2.1 Парадигма GCC

Основной целью при разработке GCC являлось создание оптимизирующего переносимого компилятора. При этом его настраиваемость понималась как способность за сравнительно небольшое время (значительно меньшее, чем время написания свежего компилятора) произвести его *раскрутку*. В настоящее время усилия, необходимые для ракрутки GCC на новую платформу, составляют порядка нескольких человеко-месяцев; полученный в результате этого компилятор поддерживает полные стандарты языков и все оптимизирующие преобразования.

За время существования GCC был раскручен на большое число архитектур (несколько десятков) и операционных систем, а сама технология раскрутки отработана до мелочей. Сейчас при ракрутке на новую платформу иногда бывает достаточно взять уже раскрученную версию для похожей системы команд и модифицировать ее. Таким образом время раскрутки может быть существенно сокращено.

Важнейшим свойством GCC является наличие широкого набора машинно-зависимых оптимизаций. При этом основной упор делается на достижение эффекта в типичных ситуациях, а не на теоретическую стройность и полноту. В связи с этим оптимизирующая часть в целом представляет собой набор эвристически обоснованных преобразований. С точки зрения оптимизирующей компиляции GCC представляет из себя не полигон для опробования новых способов оптимизации (каковым, например, является SUIF), а коллекцией сравнительно простых трансформаций, рассчитаных на покрытие не экзотических, а общих случаев поведения программ. Это позволило GCC стать практичным инструменетом повседневной разработки программного обеспечения.

GCC реализован на C и хорошо документирован. Подробное описание содержится в [26].

2.2 Внутренняя архитектура

Главной частью внутренней архитектуры GCC является *язык регистровых передач* RTL *(Register Transfer Lists)* — LISP-подобный язык, описывающий программу на обобщенном языке низкого уровня. Сам GCC всегда оперирует с машинно-зависимым RTL, при этом все оптимизирующие преобразования поддерживают такой формат представления программы, который удовлетворяет машинно-зависимым ограничениям.

Таким образом, общая последовательность преобразования входной программы следующая:

- Входной текст синтаксическим анализатором преобразуется в дерево.
- По данному дереву генерируется машинно-зависимый RTL.
- Полученный RTL подвергается оптимизирующим преобразованиям.
- По финальному RTL генерируется ассемблерное представление порожденной программы.

2.2.1 RTL-представление

Как было указано выше, RTL является LISP-подобным языком. Основными конструкциями RTL являются выражения, векторы, целые, длинные целые и строки.

Выражения описывают операнды (регистры, память, внешние ссылки) и операции (пересылки, арифметико-логические операции и т.д.) Кроме этого, все выражения снабжены указаниями *машинных видов (machine modes)*. Каждый машинный вид определяет способ хранения и размер значения. Всего определено 24 машинных вида.

Приведем пример RTL-выражения:

```
(set (reg:SI 22)
       (plus:SI (reg:SI 23)
           (reg:SI 24))) -1 (nil)
   (expr_list:REG_EQUAL (plus:SI (mem/f:SI (reg:SI 17) 0)
           (mem/f:SI (plus:SI (reg:SI 17)
                   (const_int 4 [0x4])) 0))
       (nil))
```

Здесь:

- `set` — присваивание.
- `(reg:SI 22)` — регистр номер 22, имеющий машинный вид `SI` (single integer).
- `plus:SI` — сложение значений вида `SI`.
- `mem/f:SI` — операнд вида `SI` в памяти.
- `const_int` — целая константа.

Программа в RTL-форме представляет собой двусвязный список выражений специального вида — INSN (instruction). Каждый INSN соответствует одной или нескольким машинным инструкциям.

Бывают следующие виды инструкций:

- insn — инструкция, не осуществляющая передачи управления.
- jump_insn — инструкция, осуществляющая условный или безусловный переход.
- call_insn — инструкция вызова.
- code_label — метка.
- barrier — специальная инструкция, помещаемая непосредственно после безусловного перехода или вызова функции, из которой не бывает возврата (например, `exit` или `abort`) (для оптимизации переходов).
- note — комментарий или отладочная информация.

Важнейшим свойством RTL-формы является то, что она представляет собой код с *псевдорегистрами*. Только несколько регистров с начальными номерами соответствуют физическим машинным регистрам. Остальные должны быть разложены на физические регистры или вытеснены в специально отведенные позиции на стеке. Специальные номера регистров также отведены для указателей вершины стека, стековой рамки, регистра указания начала области параметров и т.д.

Например, программа

```
# include <stdio.h>

int f (int a, int b)
{
    return a+b;
}

main ()
{
    printf ("Hello, world!\n");
    return f (10, 2);
}
```

получит в RTL-представлении следующий вид:

```
...

 -- код для функции f:

(insn 8 6 10 (set (reg:SI 23)             -- взять в регистр 23 значение
     (mem/f:SI (reg:SI 17) 0)) -1 (nil)   -- с вершины стека
 (nil))

(insn 10 8 11 (set (reg:SI 24)            -- взять в регистр 24 значение
```

```
      (mem/f:SI (plus:SI (reg:SI 17)       -- из-под вершины стека
              (const_int 4 [0x4])) 0)) -1 (nil)
 (nil))

(insn 11 10 13 (set (reg:SI 22)            -- сложить регистр 23
      (plus:SI (reg:SI 23)                 -- с регистром 24 и результат
          (reg:SI 24))) -1 (nil)           -- записать в регистр 22
 (expr_list:REG_EQUAL (plus:SI (mem/f:SI (reg:SI 17) 0)
          (mem/f:SI (plus:SI (reg:SI 17)
                  (const_int 4 [0x4])) 0))
      (nil)))

(insn 13 11 14 (set (reg/i:SI 0 %eax)      -- записать результат в
      (reg:SI 22)) -1 (nil)                -- машинный регистр 0.
 (nil))
...
```

Здесь `(reg:SI 0)` соответствует физическому регистру `%eax`, а `(reg:SI 17)` — регистру `%esp` архитектуры `x86`.

2.2.2 Оптимизирующие преобразования

В настоящий момент GCC реализует следующие оптимизирующие преобразования:

- Оптимизацию переходов *(jump optimizations).*
- Локальное и глобальное сокращение общих подвыражений *(common subexpression elimination).*
- Оптимизацию циклов — чистку вверх *(invariant code motion)* и индуктивные вычисления *(strength reduction).*
- Локальное распределение регистров *register allocation.*
- Комбинирование *(combining)* — замену последовательности нескольких инструкций на одну инструкцию.
- Покадровую оптимизацию *(peephole optimization).*
- Планирование инструкций *(instruction scheduling).*

GCC позволяет ограниченно управлять набором и последовательностью преобразований. Следует отметить, однако, что они не являются полностью ортогональными. В связи с этим невозможно задавать порядок их применения произвольно. Мало того, исходный вид порожденного после синтаксического анализа RTL существенно зависит от того, будут ли в дальнейшем применяться оптимизирующие преобразования.

2.3 Настраиваемость

Для настройки на систему команд и архитектуру `GCC` использует описание машины *(machine description)*. Это описание содержится в двух файлах — один из них описывает систему команд с точки зрения порождения машинно-зависимого RTL, другой — различные особености архитектуры.

Описание системы команд не является строгим формализмом, а предоставляет возможность параметрически задавать правила покрытия исходного дерева разбора машинными инструкциями. Несмотря на то, что данный способ вполне оправдал себя для использования в целях раскрутки, такое положение дел привело к тому, что семантика описания системы команд туманна и запутана, содержит большое число недокументированных особенностей, связанных со спецификой использованных алгоритмов его обработки, и совершенно не годится для автоматического порождения.

Кроме того, при раскрутке `GCC` большое время занимает отладка описания системы команд, так как там содержится информация, влияющая на ход и корректность оптимизирующих преобразований. Для того, чтобы поддержать все имеющиеся преобразования, часто бывает необходимо знать специфику их проведения и добавлять соответствующую им детализацию в описание машины. Однако иногда и этого недостаточно — в тексте самих оптимизирующих просмотров содержатся машинно-зависимые вставки.

Вследствие всего вышеизложенного эффективность порожденного кода напрямую зависит от качества предоставленного описания системы команд. Размеры такого описания могут быть поистине огромны — так, для архитектуры `x86-linux` оно занимает более восьми тысяч строк.

2.3.1 Описание системы команд (machine description)

Описание системы команд представляет собой набор RTL-выражений особого вида. При раскрутке компилятора описание системы команд обрабатывается специальным образом, после чего порождаются функции, которые и используются при генерации машинно-зависимого RTL и работе с ним.

Ключевую роль в описании системы команд играют *шаблоны инструкций (instruction patterns)*, наиболее важными видами которых являются `define_insn` и `define_expand`.

RTL-выражения шаблонов инструкций состоят из следующих полей:

- Необязательного имени. Это имя может использоваться при порождении машинно-зависимого RTL, при этом оно должно совпасть с именем одного из узлов дерева промежуточного представления. Ассортимент этих узлов фиксирован раз и навсегда; если имя, указанное в шаблоне инструкции, не совпадает ни с каким именем узла дерева, данный шаблон не будет использован при генерации RTL. Если задано несколько шаблонов с одним и тем же именем, неоднозначность разрешается применением первого подходящего в порядке их следования в файле описания архитектуры (возможность применения шаблона определяется не только именем, но и рядом дополнительных условий, см. ниже). Наконец, если шаблон не имеет имени, то он не используется при генерации начального RTL, однако далее компилятор может им воспользоваться для генерации более эффективного кода (иначе говоря, наличие такого шаблона говорит о существовании машинной команды, но ничего не говорит о том, какому узлу дерева она соответствует).

- RTL-образца. Этот образец представляет из себя незаконченное RTL-выражение, в котором использованы специальные выражения, обозначающие вхождения операндов (`match_operand`, `match_operator` или `match_dup`).
- Условия — строки, которая содержит имя функции, определяющей возможность применения данного шаблона.
- Выходного шаблона — форматной строки, используемой при генерации ассемблерного текста.
- Необязательного списка различных атрибутов (их можно определять самим).

Пример шаблона инструкции дан ниже:

```
(define_insn "tstsi"
  [(set (cc0)                          -- RTL-образец

    (match_operand:SI                          -- операнд
       0                                       -- нет RTL-образца
       "general_operand"                       -- условие
       "rm"                                    -- атрибуты (регистр/память)
    ))]
  ""                                   -- нет условия
 "{ if (TARGET_68020 || !              -- выходной шаблон
      ADDRESS_REG_P (operands[0]))
    return \"tstl %0\";
  return \"cmpl #0,%0\"; }"
                                       -- нет атрибутов
)
```

Шаблоны инструкций используются в трех случаях:

1. При генерации машинно-зависимого RTL (используются только именованные шаблоны). В процессе этого проверяются условия, заданные вторыми элементами шаблонов. Если для данного узла дерева не найден ни один шаблон, то возбуждается фатальная ошибка. В противном случае в RTL-образец подставляются результаты генерации для его операндов и полученная RTL-инструкция добавляется в порожденных код.
2. При проведении оптимизаций компилятор пытается каким-либо образом сгенерировать более простые инструкции, используя неименованные шаблоны.
3. При генерации финального ассемблерного кода. RTL-инструкции финальной программы сопоставляются с RTL-образцами всех шаблонов и при совпадении производится выдача ассемблерного текста в соответствии с выходым форматом.

2.3.2 Описание архитектуры

Описание архитектуры представляет собой заголовочный *(header)* файл на языке C, в котором определены символы препроцессора, описывающие те или иные особенности архитектуры. Далее эти символы используются в различных местах компилятора, управляя видами преобразований. Практически все оптимизации проверяют состояние препроцессорных переменных, таким образом осуществляется дополнительная машинно-зависимая настройка.

В числе прочих описание особенностей архитектуры определяет:

- Размеры и выравнивание машинных данных.
- Размеры и свойства основных типов данных исходного языка.
- Имена и свойства физических регистров.
- Классы физических регистров (т.е. совокупности регистров со сходными свойствами).
- Направление роста и размер стека.
- Режимы адресации.
- Относительные стоимости различных операндов.
- Формат вывода ассемблерного текста.

3 Компилятор lcc

3.1 Общая информация

Компилятор `lcc` [31] разрабатывается в рамках проекта NCI (National Compler Infrastructure) в Принстонском университете США с 1991 года.

`lcc` представляет собой переносимый транслятор с языка C, полностью совместимый со стандартом ANSI. Именно переносимость является ключевой чертой транслятора, обусловившей его возрастающую популярность в исследовательских кругах. Действительно, в настоящее время `lcc` является одним из немногих (если не единственным) свободно распространяемым в исходных кодах транслятором, для которого написание кодогенератора для новой архитектуры процессора не является неподъемной задачей и доступно даже одиночному исследователю.

3.2 Структура компилятора

Компилятор `lcc` имеет традиционную архитектуру:

- Frontend. Frontend `lcc` определяется всего в нескольких файлах и легко может быть отделен от видозависимого анализатора и кодогенератора и переиспользован в других программных продуктах. Примером такого использования может служить использование font-end'а `lcc` при построении компилятора на базе переносимого оптимизатора `VPO` [4, 8].

В качестве промежуточного представления frontend `lcc` использует DAGs (Directed Acyclic Graphs), что делает возможным однократную генерацию кода для общих подвыражений и эффективное использование комплексных инструкций, обычно в изобилии определяемых DSP процессорами.

- Генератор кода. Для выбора инструкций `lcc` использует метод BURS (Bottom-Up Rewrite System) [2, 7, 13, 18], позволяющий (при некоторых ограничениях на целевую архитектуру) генерировать оптимальный код в случае, когда в качестве промежуточного представления используются деревья. В состав `lcc` входит специальное средство — `lburg` — которое осуществляет построение BURS-системы по входной грамматике, описывающей систему команд.

  В качестве алгоритма распределения регистров `lcc` использует так называемый bin-packing алгоритм [16], превосходящий классические алгоритмы раскраски в случае линейного участка кода, однако в целом проигрывающий им. В настоящий момент выбор именно этого алгоритма непонятен и кажется целесообразным заимствовать смешанный алгоритм распределения регистров из уже упоминавшегося транслятора `VPO`, что не должно вызвать больших трудностей.

Следует отметить, что `lcc` не является оптимизирующим компилятором — он не содержит просмотров, реализующих отдельные оптимизирующие преобразования.

3.3 Способ генерации кода

Как было указано выше, `lcc` использует механизм восходящего переписывания деревьев (BURS) для реализации выбора инструкций.

BURS представляет собой попытку применить классические методы сопоставления с образцом к задаче кодогенерации. Именно, имея в качестве промежуточного представления набор деревьев, каждое из которых соответствует выражению исходной программы, BURS позволяет за два просмотра этого дерева сгенерировать оптимальный код. Оптимальность определяется как минимум сумм весов инструкций, задаваемых во входном файле описания архитектуры.

3.3.1 BURS (Bottom-Up Rewrite System)

Для задания системы команд используется так называемая *древесная грамматика (tree grammar)*, которая может быть описана как обычная КС-грамматика, у которой в правой части стоит не слово (линейная последовательность символов), а дерево. При этом внутренние узлы дерева помечены только терминалами.

Пример BURS-грамматики приведен ниже (терминалы взяты в кавычки, нетерминалы даны прописными буквами):

```
START: EXPR
EXPR : REG
EXPR : MEM
```

```
REG  : 'fetch' (REG)
REG  : '+' (REG, REG)
REG  : MEM
MEM  : REG
REG  : '=' (REG, REG)
REG  : '=' (REG, MEM)
MEM  : '=' (MEM, REG)
MEM  : 'const'
MEM  : 'var'
```

Аналогично КС-грамматикам можно определить понятие вывода в данной BURS-грамматике и понятие языка. Так, язык, определяемый приведенной выше грамматикой, состоит из всех деревьев степени 2, внутренние вершины которых помечены символами '+', '=' и 'fetch', а листья — символами 'const' и 'var'.

Как можно видеть, если дерево принадлежит языку, порождаемому данной BURS-грамматикой, то каждой его вершине можно сопоставить множество нетерминалов, из каждого из которых выводится поддерево с корнем в данной вершине (поскольку грамматика, вообще говоря, может оказаться неоднозначной). Алгоритм, строящий такую разметку, вполне очевиден — достаточно применять снизу-вверх всевозможные правила, пользуясь терминальными пометками вершин текущего поддерева и нетерминальными пометками, полученными на предыдущих шагах. Если по окончании этого процесса среди пометок корня дерева есть стартовый нетерминал, то это дерево принадлежит языку, а пометки вершин описывают всевозможные выводы этого дерева в данной граматике. Очевидно, что построение такого разбора при фиксированной грамматике линейно относительно числа вершин дерева.

Данный подход используется для выбора команд следующим образом. В качестве входного языка принимается промежуточное представление исходной программы в виде деревьев выражений. Нетерминалы обозначают классы размещения значений в данной системе команд (например регистры, память, стек), правила грамматики описывают машинные инструкции, используя терминальные пометки узлов дерева и размещения поддеревьев, представленные нетерминальными пометками.

Приведенная выше грамматика описывает систему команд вида

```
st  reg,mem    -- выгрузка
ld  mem,reg    -- загрузка
add reg,reg    -- регистровое сложение
mov reg,reg    -- регистровая пересылка
mov reg,[reg]  -- косвенная адресация
```

где `reg` и `mem` — нетерминалы, описывающие соответственно регистры и память. Как видно, правила позволяют гибко описывать разрешенные комбинации операндов и режимов адресации. В данном случае то, что система команд разрешает только регистровое сложение, отражено наличием единственного правила, с помощью которого может быть выведена терминальная пометка `'+'`, при этом ее сыновьями могут быть только поддеревья, выведенные из нетерминала `reg`.

Наконец, особую роль в BURS-грамматиках играют цепные правила — именно, они описывают пересылки значений. Например, правило

```
REG : MEM
```

трактуется как возможность преобразовать размещение в памяти в размещение в регистре (т.е. фактически описывает команду загрузки).

Как было указано выше, BURS-грамматики могут быть неоднозначными. Мало того, такое положение дел представляет собой стандартный случай в области выбора инструкций, поскольку такой выбор обычно неоднозначен.

Конечно, для достижения корректности порожденного кода достаточно выбрать любой разбор, полученный применением алгоритма BURS, однако такой выбор не обеспечивает оптимальности. Для ее достижения правила снабжаются стоимостями и используется принцип динамического программирования, а именно — оптимальный разбор вершины получается из оптимальных разборов ее потомков и ее собственной терминальной пометки применением правил грамматики. В качестве задачи оптимизации выступает минимизация функции стоимости, вычисляемой как сумма стоимостей вхождения правил в разбор данного дерева. Поэтому кроме собственно нетерминальной разметки вершины дерева помечаются правилами, которые были использованы для вывода того или иного нетерминала в данной вершине.

Таким образом, BURS-алгоритм для генерации кода состоит из двух просмотров:

1. *Разметка (labeling).* Заключается в построении нетерминальной разметки снизу-вверх, при этом среди нескольких правил, выводящих один и тот же нетерминал, используется правило с минимальной стоимостью. Стоимость применяемого правила определяется как сумма собственного веса правила и весов всех выведенных нетерминалов, входящих в него. Кроме собственно нетерминала, в разметку вершины входит правило, которым этот нетерминал был выведен.

2. *Свертка (reduce).* Представляет из себя проход по дереву сверху-вниз, при котором каждой вершине “спускается” предзаданный нетерминал. Каждый шаг прохода выбирает правило, которое вывело в данной вершине предзаданный нетерминал и на основании этого правила определяет среди потомков данной вершины очередные вершины, к которым будут применены следующие шаги свертки и их предзаданные нетерминалы. Для начальной вершины предзаданным нетерминалом считается стартовый. Заметим, что свертка вообще говоря не определяет порядок применения свертки к сыновьям данной вершины.

На практике обычно с каждым правилом связан некоторый программный текст (семантика), которая исполняется при свертке с использованием данного правила. Этот текст осуществляет выдачу ассемблерного кода и может осуществлять вызов дополнительных функций, необходимых для генерации кода (например, вызов распределителя регистров). Таким образом, корректность полученного кода складывается из достоверно верифицируемой корректности BURS-грамматики и эвристически обосновываемой корректности семантик.

Более подробно технология BURS описана в [2, 7, 13, 18, 6].

Кажущимся ограничением BURS метода является статичность весов инструкций, однако, как показывает практика, большинство стандартных процессоров допускают статическую параметризацию. Кроме того, BURS легко может быть расширен до варианта с нестатическими стоимостями.

Далее, несмотря на то, что формально BURS-система описывает язык, состоящий из деревьев конечной степени (а именно, максимальной из степеней правых частей правил), это ограничение не является существенным. Для его преодоления достаточно преобразовать исходное дерево в дерево конечной степени с помощью вставки дополнительных узлов и после разбора извлечь нужную информацию с помощью обратного преобразования [24]. Общий механизм таких преобразований, сохраняющий выводимость, известный как *линейный древесный гомоморфизм (linear tree homotorphism)*, описан в [5].

Действительно существенным ограничением является то, что код для DAG'ов, генерирующийся BURS, получается в результате расклеивания разделяемых узлов и вычисления поддеревьев отдельно. Единственное, что допускается — это заметить, что данный нетерминал уже был выведен раньше и переиспользовать результат. Очевидно, такой метод не дает возможности учитывать комплексные инструкции, покрывающие поддерево высоты больше 1, таким образом получаемый код субоптимален.

Наконец, узким местом BURS-подхода является то, что полученный в результате разметки код определяет только типы размещений, но не их экземпляры. Таким образом, этот код является кодом с псевдорегистрами, нуждающимися в дальнейшем распределении. Хуже того, оптимальное их распределение не может быть получено локальной модификацией кода, поскольку может потребоваться изменение порядка вычисления поддеревьев, который, вообще говоря, зависит исключительно от реализации свертки и не оперирует с качественными показателями.

Тем не менее, как метод выбора инструкции BURS является наилучшим среди существующих на сегодняшний день.

3.3.2 Генерация кода для целевой машины

Входной файл инструмента `lburg` аналогичен по своей структуре файлам описания грамматик, обрабатываемым `yacc`'ом.

В первой его части описываются терминалы. В качестве терминалов выступают пометки вершин DAG'а, например `CNST`, `CVI` или `ADD`. Приведенные имена терминалов являются родовыми и дополняются суффиксами и размерами для указания типа результата и его размера в определяемой архитектуре. К примеру `CVIF4` означает операцию приведения типа `signed int` к `float`, причем результат занимает 4 байта. `lburg` аналогичным образом воспринимает также `CVIF8`, `CVIF12` и `CVIF16`, таким образом достигается максимальная гибкость в описании архитектуры.

Во второй части burg-файла описываются покрытия узлов дерева и соответствующий каждому образцу ассемблерный код.

Фрагмент реального файла описания архитектуры процессора x86 приведен ниже:

```
-- преамбула:

%{

-- описания типов данных

enum { EAX=0, ECX=1, EDX=2, EBX=3, ESI=6, EDI=7 };
...

-- описания семантических функций

static void address(Symbol, Symbol, long);
static void blkfetch(int, int, int, int);
static void blkloop(int, int, int, int, int, int[]);
...
%}

-- описание стартового нетерминала

%start stmt

-- описание терминалов

%term CNSTF4=4113    -- 4-х байтовая целая константа
%term CNSTF8=8209    -- 8-ми байтовая целая константа
...

%term ARGB=41        -- однобайтовый аргумент функции
%term ARGF4=4129     -- плавающий четырехбайтовый аргумент функции
...

%term ASGNB=57       -- однобайтовое присваивание
%term ASGNF4=4145    -- плавающее четырехбайтовое присваивание
...

%term INDIRB=73      -- однобайтовый косвенный операнд
...

%term ADDF4=4401     -- плавающее четырехбайтовое сложение

-- описание правил

%%
reg:  INDIRI1(VREGP)     "# read register\n"
...

stmt: ASGNI1(VREGP,reg)  "# write register\n"
...

con: CNSTI1  "%a"
```

```
...
addr: ADDI4(index,base)  "%1[%0]"
addr: ADDP4(index,base)  "%1[%0]"
addr: ADDU4(index,base)  "%1[%0]"
addr: index  "[%0]"
...

mem: INDIRI1(addr)  "byte ptr %0"
mem: INDIRI2(addr)  "word ptr %0"
...

%%
```

Использование `lburg` позволяет существенно сократить размер архитектурно-зависимой части компилятора (так, описание системы команд x86-linux занимает всего 999 строк). Кроме того, BURS-грамматика является строгим формализмом, допускающим автоматические манипуляции и порождение.

После описания образцов для генерации кода необходимо явным образом указать параметры архитектуры. Для этого служит структура `Interface`, в которой описываются:

- размеры стандартных типов данных
- функции, вызываемые для генерации кода на глобальном уровне (пролог-эпилог программы, пролог функции, генерация отладочной информации и т.п.)
- специальные флаги (`endianess`, в каком порядке аргументы функции размещаются на стеке и т.д.)

На основании BURG описания `lburg`'ом генерируется код на C, основную часть которого составляет определение автомата для данной грамматики. Этот автомат в своих состояниях и переходах содержит всю информацию о стоимостях инструкций, и таким образом переносит все решения динамического программирования на фазу compile-compile [19].

3.3.3 Распределение регистров

Как было указано выше, `lcc` использует bin-packing алгоритм распределения регистров [16, 12]. Данный алгоритм подразумевает, что стадия выбора инструкций уже успешно завершена. Bin-packing, следовательно, является алгоритмом вытеснения регистровых значений в память, причем отличается особенной простотой по сравнению с традиционными методами раскраски графов несовместимости.

В упрощенном понимании данный алгоритм можно представить так: инструкции в DAG'е прошиты в соответствии с полученным `lburg`'ом кодом. Этот список просматривается с головы (в порядке исполнения инструкций), причем в каждом узле (которому, вообще говоря, могут соответствовать несколько инструкций) отслеживается информация об используемых регистрах и регистрах, где сохраняются результаты операций. Кроме того, заводится глобальная структура (битовая шкала), показывающая какие регистры в данный момент доступны. Занятый регистр становится свободным, когда счетчик ссылок из родителей, уменьшающийся при просмотре очередного родителя достигает нуля. Данная конструкция позволяет переиспользовать значения, вычисляемые в разделяемых узлах DAG'а.

При отсутствии достаточного количества свободных регистров производится выгрузка в память. Для выгрузки выбирается максимально долго не используемое в дальнейшем значение, что приводит к вложенному просмотру кода и квадратичности алгоритма. Последнее действие шага алгоритма — это вставка инструкции, соответствующие выгрузке и загрузке и линеаризация полученного кода.

Например, рассмотрим следующий фрагмент программы:

```
double a[10], b[10];

int i;

int f()
{
   int j = i<<3;

   i = (a[i]+b[i])*(a[i]-b[i]);

   return i;
}
```

Соответствующий функции f линеаризованный DAG вместе с выделяемыми регистрами будет выглядеть так:

```
                             использует       сохраняет

1.  ADDRGP i                    r1
2.  INDIR  1                    r1               r2 (r1 нужен)
3.  CNSTI  3                                     r3 (r2 нужен)
4.  LSHI   2   3                r2 r3            r2
5.  ADDRGP a                                     r3
6.  ADDP   4   5                r2 r3            r3
7.  INDIRD 6                    r3               r3 r4
8.  ADDRGP b                                     r5
9.  ADDP   4   8                r2 r5            r2
10. INDIRD 9                    r2
11. ADDD   7  10
12. SUBD   7  10
13. MULD   11 12
14. CVID   13
15. ASGNI  1  14
```

Пусть в архитектуре 5 регистров значения типа double занимают два (смежных) регистра, тогда в узле 10 потребуется выгрузка, так как `r1` занят узлом 1 (переиспользуется в узле 15), пара `r3-r4` занята под `a[i]`, значит свободны лишь `r2` и `r5`, но они не смежны, следовательно не могут хранить значение типа double. В результате будет вытеснено значение из `r1`, так как его ближайшее использование только в узле 15, в то время как `r4` используется уже в следующей инструкции 11.

3.4 Интерфейс с ASDL

ASDL (Abstract Syntax Definition Language) [3], как следует из названия — это язык для абстрактного описания структур. Компилятор ASDL получает на вход данное описание и генерирует соответствующие структуры и функции их сериализации на языках высокого уровня C, C++, Java, SML-NJ и Haskell. ASDL не является частью `lcc`, но также разрабатывается в рамках проекта Zephyr [3, 20, 21].

`lcc` определяет ASDL-описание для своего промежуточного представления и опцию компилятора, позволяющую получить на выходе файл, сгенерированный ASDL примитивами. Возможным использованием этого файла может служить его чтение программой на любом из перечисленных языков, какое-либо оптимизирующее преобразование кода и снова запись в файл. Именно таким образом можно легко вставлять оптимизации в `lcc`, не читая ни строчки исходных текстов компилятора. Утилита `pass2` вызывает собственно генерирующую часть компилятора.

Кроме того, в состав `lcc` входит специальная утилита, позволяющая получать из бинарного дампа ASDL соответствующий ему HTML-документ.

3.5 Исправленные ошибки

В интерфейсе с ASDL была допущена неточность, приведшая к неправильной генерации определения констант типа `float`. В ASDL числа с плавающей точкой отсутствуют, в результате чего они представляются двумя целыми: `lsb` и `msb`. `lcc` декларирует константы как `long double`, подразумавая при этом размер 8 байтов, однако в `GCC` данный тип занимает 12 байтов; в результате деление на два 32-битных слова приводит к неправильным результатам для `lcc`, построенного при помощи `GCC`. Корректное решение — привести величину к типу `double` (коль скоро размеры их в `lcc` совпадают).

4 Компилятор на основе SUIF

Система SUIF (Stanford University Intermediate Format), разрабатывается, как следует из ее названия, в Стенфордском университете, США [33]. Система MachSUIF разрабатывается в Гарвардском университете. Разработка обеих систем ведется в рамках проекта NCI (National Compiler Infrastructure).

Система SIUF предназначена для создания унифицированного внутреннего представления программы внутри компилятора и представляет собой набор C++ классов. По идеологии SUIF-а, компилятор строится как набор просмотров (passes), каждый из которых работает над общим представлением программы. Упор сделан на расширяемость и удобство использования системы. Как внутреннее представление, так и набор просмотров можно полностью заменить (что и сделано в системе MachSUIF). Внутренне представление, уже существующее в SUIF, предоставляет несколько уровней абстракции исходной программы.

Система MachSUIF перднaзначена для построения back-endов компилятора. Она построена в соответствии с SUIF-идеологией и представляет собой набор SUIF-просмотров и расширений внутреннего представления SUIF.

Работающий компилятор на базе данных систем выглядит так:

- Входной файл на языке программирования передается на вход какому-либо frontend'у, умеющему генерировать SUIF-представление. В настоящей работе использовался свободно распостраняемый в бинарном виде EDG Frontend[29].
- Полученный файл представляет собой языконезависимое SUIF-представление исходной программы. С помощью SUIFa к нему можно применить машинно-независимые оптимизации, а также "понизить" уровень абстракции. При этом происходит "расшивание" сложных конструкций языка, таких например, как цикл "for".
- Пониженное SUIF-представление передается системе MachSUIF, которая генерирует по нему код и применяет машинно-зависимые оптимизации.

4.1 SUIF

Основные принципы, из которых исходили авторы SUIF, таковы [15]:

- Максимальная модульность системы, позволяющая легко объединять различные просмотры, написанные разными людьми.
- Максимальная расширяемость, позволяющая подгонять внутреннее представление под конкретные языки или просмотры, и при этом использовать уже имеющиеся просмотры, максимально поощряя переиспользование кода.

4.1.1 Принципы организации

SUIF базируется на понятии *модуля*. Модуль представляет собой класс C++ – наследник класса `Module`. Модули объединяются в разделяемые библиотеки.

Ядро системы также представляет собой набор модулей.

Модули могут быть двух видов:

1. Набор элементов (узлов) внутреннего представления. Пользователь может расширять или полностью заменять узлы внутренего представления, используя описания на hoof (специально разработанном для этой цели языке) или на C++.
2. Просмотр компилятора.

Если пользователь хочет, чтобы существующий просмотр работал на новых определенных им узлах внутреннего представления, он должен определить стандартные итераторы по подузлам новых узлов. Hoof позволяет определять нужные итераторы почти автоматически.

В свою очередь, просмотры должны работать на сколь возможно высоком уровне абстракции для обеспечения расширяемости.

При соблюдении этих условий обеспечивается высокая расширяемость внутреннего представления.

Следует отметить, что все исходные тексты хорошо документированы, так как написаны с использованием средств самодокументирования *(literate programming)*.

В системе имеется также драйвер `suifdriver`, позволяющий в интерактивном режие или с помощью простого скриптового языка загружать внутреннее представление из файла, применять к нему различные просмотры и сохранять результат в файле.

4.1.2 Обзор иерархии классов

Внутреннее представление Полная иерархия классов приведена в [15]. Здесь перечислим только характерные черты.

Типы и переменные программы являются объектами подклассов класса `SymbolTableObject`. Операторы и выражения программы являются объектами подклассов классов `Statement` и `Expression`, которые в свою очередь, являются подклассами класса `ExecutionObject`.

Есть несколько уровней представления программы. К примеру, имеется как класс `JumpStatement`, так и класс `ForStatement`.

Чтобы абстрагироваться от конретных узлов внутреннего представления, просмотры могут (и должны) использовать итераторы, такие как:

- `source_ops`, `source_vars`, `defined_labels` — итераторы для всех корней поддеревьев, всех непосредственно используемых переменных и определяемых меток соответственно.

- `child_statements`, `destination_vars` — все вложенные операторы и непосредственно присаиваемые переменные.

Просмотры Все просмотры должны быть подклассами класса `Pass`. Есть глобальные и локальные просмотры. Последние могут быть применены к отдельной процедуре, а не только к любой программе. Локальные просмотры должны быть подклассами класса `PipelineablePass`.

4.1.3 Обзор существующих просмотров

В текущей версии SUIF имеются в основном вспомогательные просмотры, а также различные dismantl'еры, т.е. просмотры, понижающие конструкции высокого уровня до более низких. Все они находятся в модуле `transforms`. Имеются (правда, в зачаточном состоянии) и средства межпроцедурного анализа.

4.2 MachSUIF

Система Machine SUIF основана на системе SUIF. В качестве промежуточного представления используются обобщенные машинные инструкции.

Авторы ставят перед собой следующие цели:

- Обеспечение возможности создания машинно-зависимых оптимизаций, легко переносимых с машины на машину.
- Обеспечение простоты добавления новой поддерживаемой архитектуры.
- Поддержку различных внутренних представлений.

Также как и SUIF, MachSUIF написан с использованием “литературного программирования”.

4.2.1 Принципы организации

В качестве основного обобщенного внутреннего представления машинного кода в MachSUIF используется понятие инструкций некоторой виртуальной машины под названием SUIFvm. На первом этапе кодогенерации SUIF-представление программы переводится в код SUIFvm с помощью просмотра `s2m`. Код SUIFvm может быть параметризован для соответсвующей целевой архитектуры. Это делает просмотр `gen`, которому в качестве параметра передается библиотека, описывающая целевую архитектуру. Фактически, при этом каждой инструкции SUIFvm ставится в соответствие последовательность инструкций целевой архитектуры. На выходе просмотра `gen` получается код в инструкциях целевой архитектуры, но с псевдорегистрами.

Код может быть представлен как в виде последовательности инструкций, так и в виде уграфа (CFG, Control Flow Graph) [17, 25]. Преобразование между этими представлениям осуществляют просмотры `il2cfg` и `cfg2il`.

Инструкции представляют собой объекты подклассов класса `Instr` и предоставляют общий интерфейс, позволяющий писать общие просмотры для различных целевых архитектур. Примером такого просмотра является просмотр распределения регистров, `raga` (Register Allocation by George and Appel).

После того, как получен код целевой архитектуры с псевдорегистрами, можно запускать различные оптимизирующие просмотры, а также распределять регистры. По окнчании оптимизаций следует запустить просмотр `fin`, который расставляет прологи и эпилоги в процедурах, а затем просмотр `m2a`, непосредственно выдающий ассемблерный код.

4.2.2 Внутреннее представление

В целом, MachSUIF – это набор SUIF-модулей, определяющих различные узлы внутреннего представления и просмотры. Мы уже частично рассмотрели существующие просмотры в предыдущем разделе, так что сосредоточимся на узлах внутренного предсталения.

Все внутреннее представление вертится вокруг классов `Instr` и `Opnd` – инструкций и их операндов, соответственно.

Имеется несколько видов инструкций: активные (к ним относятся арифметико-логические (alm) и инструкции передачи управления (cti)) и пассивные (метки и псевдоиструкции целевого ассемблера). Имеется таже несколько видов операндов, именно:

- Элементарные операнды. К ним относятся:

– *пустые операнды* – “нет операнда”
– *переменные* – всевозможные переменные (временные или определенные в программе
– *регистры* – псевдорегистры или регистры целевой архитектуры
– *непосредственные адреса*

- Адресные выражения различных видов

4.2.3 Описания целевой архитектуры

Описание целевой архитектуры задается в *виде библиотеки описания архитектры (target library)*, которая представляет собой набор C++ классов.

Эти классы, во-первых, предоставляют, через унифицированный интерфейс, различные параметры целевой архитектуры (число регистров, их конфигурацию, требования к выравниванию etc.), а во-вторых, определяют отображение иструкций SUIFvm на инструкции целевой архитектуры. Последнее достигается с помощью подкласса класса `CodeGen`, в котором определены виртуальные функции, генерирующие код для каждой конкретной инструкции SUIFvm. Например, функция `void translate_cal(Instr * i)` определяет проекцию инструкции вызова поцедуры `CAL`.

4.2.4 Существующие просмотры и генераторы

В текущей версии MachSUIF имеется поддержка для процессоров DEC Alpha и Intel x86 (последняя в недоработанном виде). Кроме этого, имеется библиотека для анализа потока данных.

Есть также простейший покадровый оптимизирующий просмотр, однако он не работает для x86.

4.3 Исправленные ошибки

В настроенной версии для x86 были обнаружены и исправлены следущие ошибки:

- *x86 target library*. Неправильный формат печати инструкции `cwde`.
- *x86 target library*. Неправильная передача байтовых и двухбайтовых параметров в функции (нет приведения к 4-байтовому значению).
- *s2m convertor*. Неправильный размер временных переменных для масштаба и смещения при генерации выборок из массивов с элементами длиннее 1 байта. Размер временных переменных полагался равным размеру индекса элемента в массиве, что приводило к неправильному коду, если индекс был байтовый или двухбайтовый.

5 Компилятор на базе VPO

Система `VPO` (Very Portable Optimizer) разрабатывается в рамках проекта Zephyr [35] в университете Вирджинии. Главная задача — создание такого оптимизирующего back-end'а для компиляторов, который имел бы машиннонезависимый интерфейс и легко настраивался бы на новую архитектуру.

Несмотря на то, что VPO написан в стиле “литературного программирования”, уровень его документированности представляется недостаточным.

5.1 Общее устройство

Как было указано выше, VPO не является самостоятельным компилятором, а представляет собой настраиваемый back-end. Компилятор, построенный с использованием VPO, выглядит следующим образом:

- Frontend. В настоящий момент существуют следующие frontend'ы:
  - K&R — не поддерживает ANSI C.
  - EDG C++ frontend компании Edison Design Group [29] — не является свободно распространяемым.
  - lcc frontend, построенный с использованием frontend'а компилятора lcc.
- Code-expander, преобразующий выходное представление frontend'а во входное представление VPO, а также вносящий машиннозависимую специфику.
- Собственно VPO. Существует в двух версиях: VPO1 и VPO2. Поскольку VPO1 поставляется только с K&R frontend'ом, на нем останавливаться не будем.

Исходный текст на ЯП подается на вход frontend'у, который генерирует некое выходное (обычно текстовое) представление программы. Например, fron-end lcc преобразует исходную программу

```
# include <stdio.h>

static void printchar(char c)
{
  putc (c,stdout);
}

int main(void)
{
  int i;

  printf("Hello, world!\n");

  for(i=0; i < 10; i++)
    printchar('*');

  printchar('\n');
}
```

в текстовое представление, которое является просто префиксной формой записи его порожденного дерева:

```
-0
-16 .p0.0_c 0 1 5 4 4 0 0 0 0 -- объявление и инициализация переменной
-11 printchar 0 0            -- тело функции
+17 7 4 6 0 0                -- представление конструкций
+4 5 4 0
+8 5 1 4 4
+17 7 4 6 0 0
+3 5 1 5 1 1
-14 "test.c"                 -- комментарий

...

-11 main 0 0                 -- тело функции
-15 10
-15 13
+16 7 4 1 L8 0
+2 7 4 0 5 4 4
...

-12 main                     -- объявление функции
...

-10 fprintf 7 0              -- объявление внешней ссылки
...
```

Этот текст далее поступает на вход code-expander'у, который генерирует текстовое представление машинно-зависимого RTL:

```
-.data
-.align 8
-.K_D0:
-.long 0
...

-.local L8
-.text
-.align 8
-printchar:
-.set .p0.0_c,8
+1                L[A[dr[14]]]=r[12];        NZ
+2                       r[12]=r[14];        NZ
+3                       b[0]=L[A[r[12]+.p0.0_c]];        NZ
+4               3        B[A[r[12]+.p0.0_c]]=b[0];        NZb[0]
# File = "test.c", Line = 5
# File = "test.c", Line = 6
+5                      r[0]=ME,B[A[r[12]+.p0.0_c]];        NZ
+6               5        L[A[dr[14]]]=r[0];        r[0]
+7                      L[A[dr[14]]]=L[A[stdout]];        NZ
+8                      ST=_IO_putc;        NZ
```

```
+9                         r[14]=r[14]+8;        NZ
# File = "test.c", Line = 7
+10                         NL=LV;
+11                         PC=RT;
*
-.align 8
-main:
-.set        .l1.0_i,-4
+1                         NL=EN,4;
# File = "test.c", Line = 10
# File = "test.c", Line = 13
+2                         L[A[dr[14]]]=L8;
+3                         ST=printf;        NZ
+4                         r[14]=r[14]+4;        NZ
# File = "test.c", Line = 14
+5                         L[A[r[12]+.l1.0_i]]=0;        NZ
        L9
# File = "test.c", Line = 15
+6                         L[A[dr[14]]]=42;        NZ
+7                         ST=printchar;        NZ
+8                         r[14]=r[14]+4;        NZ
# File = "test.c", Line = 14
+9                         L[A[r[12]+.l1.0_i]]=L[A[r[12]+.l1.0_i]]+1;        NZ
+10                9          NZ=10?L[A[r[12]+.l1.0_i]];
+11                10          PC=NZ<0,L9;
# File = "test.c", Line = 16
+12                         L[A[dr[14]]]=10;        NZ
+13                         ST=printchar;        NZ
+14                         r[14]=r[14]+4;        NZ
+15                         r[0]=0;        NZ
# File = "test.c", Line = 17
+16                         NL=LV;
+17                         PC=RT;
...
```

Здесь:

- `L[...]` — загрузка
- `A[...]` — адрес
- `PC` — счетчик команд и т.д.

Далее это представление разбирается `VPO`, который порождает финальный ассемблер:

```
.data
.align 8
.K_D0:
.long 0
...
```

```
.globl main
.local L8
.text
.align 8
printchar:
.set        .p0.0_c,8
        pushl        %ebp
        movl        %esp,%ebp
        movb        .p0.0_c(%ebp),%al
        movb        %al,.p0.0_c(%ebp)
# File = "test.c", Line = 5
# File = "test.c", Line = 6
        movsbl        .p0.0_c(%ebp),%eax
        pushl        %eax
        pushl        stdout
        call        _IO_putc
        addl        $8,%esp
# File = "test.c", Line = 7
        leave
        ret
.align 8
main:
.set        .l1.0_i,-4
        enter        $4,$0
# File = "test.c", Line = 10
# File = "test.c", Line = 13
        pushl        $L8
        call        printf
        addl        $4,%esp
# File = "test.c", Line = 14
        movl        $0,.l1.0_i(%ebp)
.L9:
# File = "test.c", Line = 15
        pushl        $42
        call        printchar
        addl        $4,%esp
# File = "test.c", Line = 14
        incl        .l1.0_i(%ebp)
        cmpl        $10,.l1.0_i(%ebp)
        jl        .L9
# File = "test.c", Line = 16
        pushl        $10
        call        printchar
        addl        $4,%esp
        xorl        %eax,%eax
# File = "test.c", Line = 17
        leave
        ret
...
```

5.2 Настройка на архитектуру

Настройка на новую архитектуру компилятора на базе `VPO` заключается в замене code expander'а, поскольку именно он занимается генерацией машинно-зависимого RTL.

`VPO` содержит специальную библиотеку — `VPOI` — которая предоставляет базовые функции работы с RTL. Сам code-expander должен быть написан с использованием функций этой библиотеки. При этом входное представление программы для code-expander'а является frontend-специфичным.

Теоретически, code-expander читает входное текстовое представление, полученное из frontend'а, и каждую конструкцию переводит в корректное с точки зрения системы команд RTL-выражение, используя интерфейс `VPOI`. На практике, однако, ситуация не так проста.

Сложности возникают из-за того, что в `VPO` полностью отсутствует как таковое описание системы команд. Поэтому каждая новая система команд вносит свою сецифику в `VPOI`. Фактически это означает, что настройка на новую систему команд должна проводиться вручную.

Наконец, машинно-зависимый RTL, вообще говоря, отличается от просто RTL, поэтому дополнительно с code-expander'ом должен быть изменен и сам `VPO`. В настоящий момент анализатор входного RTL написан на `yacc`'е, поэтому сам `VPO` содержит несколько файлов грамматик (по одному для каждой архитектуры). Несмотря на то, что эти файлы называются "machine description", они фактически таковыми не являются, поскольку не управляют процессом генерации машинного кода.

Итак, для раскрутки `VPO` на другую архитектуру необходимо:

- Внести машинно-специфические изменения в `VPOI`.
- Переписать code-expander с использованием этого `VPOI`.
- Описать текстовой формат машинно-зависимого RTL в виде `yacc`-грамматики и включить полученный анализатор в состав `VPO`.

5.3 Оптимизирующие преобразования VPO

Основной чертой `VPO` является наличие большого набора оптимизирующих преобразований.

В исследуемой версии `VPO` реализованы следующие преобразования:

- Удаление мертвых переменных *(variable elimination)* и бесполезных пересылок.
- Перемещение кода *(code motion)*.
- Индуктивные вычисления в циклах *(loop strength reduction)*.
- Удаление общих подвыражений *(common subexpression elimination)*.
- Планирование инструкций *(instruction scheduling)*.
- Оптимизация переходов *(control-flow optimization)*.
- Локальное распределение регистров *(local register coloring)*.

- Глобальное распределение регистров *(global register allocation).*
- Константные вычисления *(constant folding).*
- Раскрутка циклов *(loop unrolling).*

Кроме этого, `VPO` содержит набор оптимизаций, суть которых ясна не до конца.

5.4 Проблемы и исправленные ошибки

При использовании настроенной версии `VPO` вскрылись следующие проблемы:

- В используемой версии отсутствует часть back-end'а — `cfe` — участвующая в генерации кода. Пришлось добавить ее из версии `VPO1`.
- Для архитектуры `x86` отсутствует планирование инструкций *(instruction scheduling)* и слотов задержки *(delay slots filling).*
- Для оформления стековой рамки в архитектуре `x86` `VPO` использует команды `enter` и `leave`. Как оказалось, однако, многие операционные системы некорректно обрабатывают исключение "ошибка страницы", которое может возбуждать команда `enter`. Это относится, в частности, к Microsoft Windows NT 4.0 (Build 1381 SP5) и Linux 6.2 (в то же время Free BSD и Microsoft Windos 2000 обрабатывают данное исключение корректно). Данная проблема не является ошибкой `VPO`, но ее необходимо иметь в виду при использовании его в указанных системах. В настоящий момент порождение команды `enter` подавлено.

Кроме проблем общего порядка, были обнаружены следующие ошибки:

- Frontend `lcc` порождал промежуточное представление с неправильным порядком передачи параметров (исправлено).
- При отключенных оптимизациях `VPO` неправильно распределяет регистры (суть проблемы пока не ясна).

6 Результаты сравнительного анализа

В данном разделе приводятся результаты сравнения описываемых средств. В качестве критериев для такого сравнения использовались следующие:

- Полнота системы с точки зрения задач кодизайна.
- Характеристика с точки зрения настраиваемости.
- Простота модификации и добавления просмотров и оптимизирующих преобразований.
- Качество порождаемого кода.

Заметим, что достоверное заключение по первым трем критерием может быть сделано только после реализации самостоятельной настройки на свежую архитектуру и анализа затраченных усилий. Тем не менее априорный анализ рассматриваемых средств можно считать некоторой оценкой трудоемкости такой операции.

6.1 Полнота

К сожалению, ни одно из рассматриваемых средств не обладает полнотой с точки зрения задач кодизайна. Только окружение GCC позволяет в какой-то (но далеко не полной) степени порождать программную модель вычислителя и ассемблер. Автоматическое же порождение дизассемблера, симулятора и, тем более, генератора кода не реализовано ни в одной системе. Мало того, совершенно неясно, каким образом эти возможности могут быть добавлены без кардинальной смены идеологии и используемых подходов.

6.2 Настраиваемость

С точки зрения легкости настройки на машинную архитектуру представляется справедливым расположить рассматриваемые системы в следующем порядке (по убыванию настраиваемости):

- `lcc`. Применение BURS делает настройку `lcc` на другую архитектуру локальной (необходимо поменять лишь один файл) и сравнительно простой операцией. Интеграция порождения кода и распределения регистров имеет, конечно, негативные свойства с точки зрения качества кода, зато позволяет получить этот код значительно раньше, чем с помощью других средств и с меньшими усилиями.

- `GCC`. GCC также содержит формализм для описания системы команд и поддерживает автоматическое построение кодогенератора. Кроме того, за время существования компилятора он был раскручен на огромное количество архитектур и технология такой раскрутки отработана до деталей. Вместе с тем для описания системы команд использован нечеткий формализм с большим числом исключений и недокументированных особенностей. Поэтому процесс создания описания архитектуры фактически ведется методом проб и ошибок, что замедляет его. Наконец, в отличие от `lcc`, качество порождаемого кода напрямую зависит от качества описания системы команд. Например, огромный размер описания `x86` обусловлен не только свойствами самого формализма, но и необходимостью внесния большого числа уточнений и оптимизаций, необходимых для получения качественного кода.

- `VPO и SUIF`. Данные системы представляются примерно эквивалентными по качеству настраиваемости. Они не содержат никакого механизма для описания системы команд, вся настройка осуществляется переписыванием кодогенератора, хотя сам такой кодогенератор хорошо параметризован.

6.3 Простота модификации

С точки зрения легкости добавления просмотров и их ортогональности представляется справедливым расположить рассматриваемые системы в следующем порядке:

- `lcc` и `SUIF`. Обе системы обладают возможностью отчуждения отдельных просмотров и организации их взаимодействия (мало того, это являлось основной целью создания `SUIF`'а). Что же касается `lcc`, то это его свойство является следствием использования технологии ASDL. В сзязи с этим `lcc` получает еще одно преимущество — именно, дополнительные просмотры могут быть написаны на любом языке, поддерживаемом ASDL-генератором (для `SUIF`'а единственным языком реализации остается C++).

- `VPO` и `GCC`. Данные системы не содержат никаких специальных средств для создания дополнительных просмотров и управления ими. Следует отметить, что "дополнение" `GCC` представляется более хитроумной задачей, поскольку он значительно более громоздок и запутан.

6.4 Качество кода

Результаты тестирования по критерию качества кода приведены в Приложении. В соответствии с ними, настроенные на `x86` версии компиляторов расположились в следующем порядке:

- `GCC`. В целом продемонстрировал наилучшее качество кода, однако следует отметить, что версия для `x86` является предметом особой заботы его разработчиков. Возможно, что "свежераскрученный" `GCC` будет проигрывать по качеству кода "свежераскрученному" `VPO`.
- `VPO`. Код несколько уступает по качеству `GCC`, но в некоторых случаюх превосходит его.
- `lcc`. Несколько уступает `VPO` по качеству, однако не столь существенно, как этого можно было бы ожидать от неоптимизирующего компилятора. На одном тесте (не специально подобранном) `lcc` существенно выиграл даже у `GCC`.
- `SUIF`. Продемонстрировал наихудшие показатели, существенно проиграв все остальным системам. Такое положение дел, видимо, связано с двухуровневостью избранного способа генерации кода (сначала — в коды виртуальной машины, затем — в код целевого процессора).

7 Выводы

По результатам проведенного исследования можно сделать следующие выводы:

- Для архитектуры `x86` ни `SUIF`, ни `VPO` не являются на сегодняшний день полноценными компиляторами.
- Ни одно из средств не поддерживает построения полного спектра средств кодизайна. Мало того, не представляется возможным доработать их с этой целью.
- `GCC` остается наиболее эффективным компилятором, хотя при определенных усилиях `VPO` может достигнуть его уровня.
- В очередной раз продемонстрирована применимость метода BURS для построения эффективных компиляторов — одно только применение BURS для выбора инструкций приближает качество кода к оптимизированному.

8 Направление дальнейшей деятельности

В свете полученных выводов представляется целесообразным проведение следующих работ:

- Сравнение качества кода, порождаемого `VPO`, `GCC` и `lcc` в условиях RISC-системы команд (тестировать `SUIF` представляется нецелесообразным).

- Довести `VPO` и `SUIF` до состояния, в котором они будут способны правильно компилировать тесты SPEC и провести полное SPEC-тестирование.

- Построить более тонкую характеризацию оптимизирующих преобразование `VPO` с целью выяснения, что именно препятствует генерации эффективного кода.

- Создать программную модель простого вычислительного устройства и раскрутить на него все имеющиеся средства.

9 Приложение: результаты тестов

Здесь приводятся предварительные результаты тестирования рассмотренных компиляторов. К сожалению, не удалось пропустить компиляторы через систему SPEC2000 [32], так как большинство компиляторов не смогло оттранслировать ее тесты целиком.

9.1 Тесты

В качестве тестов использовались следующие программы:

ranking Реализация алгоритма Symbol Ranking Text Encoding, автор Д.С.Ломов.

bzip2 `bzip2` Архиватор, основанный на преобразовании Барроуза-Виллера (BWT, Burrows-Wheeler Tranform) , автор Julian Seward.

gzip `gzip` Стандартный архиватор в системе GNU, основанный на LZW-сжатии.

9.2 Окружение

Тесты запускались на процессоре Intel Pentium III 850MHz, ОЗУ 384 Мб, под управлением Linux 2.2.14-5 (RedHat Zoot). В системе работал обычный набор демонов (`inet.d`, `httpd` etc.), слабо загруженных. Кроме пользователя, запускающего тесты, других пользователей на машине не было.

9.3 Результаты

Результаты тестирования сведены в нижеследующую таблицу.

Указано время в секундах, среднее по трем запускам.

Benchmark	Results					
	`GCC`	`GCC -O3`	`lcc`	`VPO noopt` [1]	`VPO`	`suif`
Ranking (44K file)	3.76	2.32	3.78	4.01	1.87	8.34
bzip2 (660K file)	8.64	3.63	7.09	8.02	5.85	9.52
gzip (2M file)	1.80	1.80	2.74	?	2.50	4.34

9.4 Интерпретация

Таким образом были получены следующие результаты: в среднем `lcc` на 30 процентов уступает `GCC` по времени исполнения кода, но в то же время значительно выигрывает у `SUIF`. Основной причиной невыгодного для `lcc` сравнения с `GCC` является то, что в `lcc` отсутствуют какие-либо машинно-зависимые оптимизации, в изобилии присутствующие в `GCC`.

Интересен косвенный результат, полученный сравнением эффективности кода, порожденного компиляторами для байт-код интерпретатора языка OCaml [28]. Для него `lcc` значительно выиграл не только у `GCC`, но и у `VPO`. Такой результат был достигнут благодаря эффективности bin-packing алгоритма в сборщике мусора.

На сегодняшний день, как думается, `lcc` вряд ли может соперничать с другими рассматривавшимися компиляторами (за исключением чрезвычайно неэффективного `SUIF`) по качеству получаемого кода для архитектуры `x86`. В то же время, исходя из относительной редкости вытеснений значений в память, являющейся характеристическим свойством представления front end'а lcc, можно предположить, что на процессорах с большим числом регистров (SPARC, R6000 ...) результаты `lcc` будут более впечатляющими.

Что касается эффективности кода, порождаемого `VPO`, то он, к сожалению, все еще в целом уступает коду, порождаемому `GCC`. Это связано, видимо, с меньшим набором оптимизирующих преобразований, неполной раскруткой на архитектуру `x86` и не в последнюю очередь с неэффективным способом порождения первичного машинно-зависимого RTL.

Список литературы

[1] Alfred V.Aho, Ravi Sethi. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Pub Co., November 1985.

[2] Alfred V.Aho, Steven W.K.Tjiang. *Code Generation Using Tree Matching and Dynamic Programming.* ACM Transactions on Programming Languages and Systems, Vol. 11, No. 4, Oct. 1989, pp. 491-516.

[3] Andrew W.Appel, Jack W.Davidson, Norman Ramsey. *The Zephyr Compiler Infrastructure.* November, 1998. *http://www.cs.virginia.edu/zephyr.*

[4] Manuel E. Benitez, Jack W. Davidson. *Target-specific Global Code Improvement: Principles and applications.*

Department of Computer Sciences University of Virginia, *ftp://ftp.cs.virginia.edu/pub/techreports/cs-9442.ps.Z.*

[5] Hubert Comon, Max Dauchet *et al.* *Tree Automata Techniques and Applications.* *http://l3ux02.univ-lille3.fr/ tommasi/TATAHTML/main.html*

[6] Helmut Emmelmann. *BEG — A Back-End Generator, Version 1.* November 1989.

[7] M. Anton Ertl. *Optimal Code Selection in DAGs.* Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, 1999, pp. 242-249.

[8] Jack W. Davidson, Steve G. Losen, Norman Bensey. *Zephyr Code Generation Interfaces.* Department of Computer Sciences University of Virginia 1998, http://www.cs.virginia.edu/zephyr/vpoi.

[9] Christopher W.Fraser, David R.Hanson. *A Retargetable C compiler: Design and Implementation.* Benjamin/Cummings Publishing, 1995.

[10] Christopher W.Fraser, David R.Hanson. *A Retargetable Compiler for ANSI C.* ACM SIGPLAN Notices, Vol. 26, No. 10, October 1991, pp. 29-43.

[11] Christopher W.Fraser, David R.Hanson. *A Code Generation Interface for ANSI C.* Software-Practice and Experience Vol. 21, №9, September 1991, pp. 963-988.

[12] Christopher W.Fraser, David R.Hanson. *Simple regigster spilling in retargetable compiler.* Software — Practice & Experience.

[13] Christopher W.Fraser, David R.Hanson, Todd A.Proebsting. *Engeneering a simple, efficient code generator generator.* ACM Letters on Programming Languages and Systems, 1993.

[14] David R. Hanson. *Early Experience with ASDL in lcc.* Software — Practice and Experience (to appear), *http://www.cs.princeton.edu/software/lcc/doc/.*

[15] Monica Lam *et al.* *An Overview of the SUIF2 Compiler Infrastructure.* Computer Systems Laboratory, Stanford University, 2000

[16] Robert Morgan. *Building an Optimizing Compiler.* February 1998,Digital Press; ISBN: 155558179X

[17] Steven Muchnik. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers; July 1997.

[18] Eduardo Pelegri-Llopart, Susan L.Graham. *Optimal Code Generation for Expression Trees: An Application of BURS Theory.* Proceedings of the conference on Principles of programming languages, 1988, pp. 294-308.

[19] Todd A.Proebsting. *BURS Automata Generation.* ACM Transactions on Programming Languages and Systems, Vol. 17, No. 3, May 1995, pp. 461-486.

[20] Norman Ramsey, Mary F.Fernandez. *Specifying Representations of Machine Instructions.* ACM Transactions on programming Languages and Systems, Vol. 19, №3, May 1997, pp. 492-524.

[21] Norman Ramsey, Jack W.Davidson. *Specifying Instructions' Semantics Using λ-RTL (Interim Report).* University of Virginia, July 11, 1999.

[22] Michael D. Smith, Glenn Holloway. *An Introduction to Machine SUIF and Its Portable Libraries and Optimizations.* Division of Engineering and Applied Sciences, Harvard University, 2000

[23] Michael D. Smith, Glenn Holloway. *A User's Guide to the Optimization Programming Interfaces.* Division of Engineering and Applied Sciences, Harvard University, 2000

[24] Булычев Д.Ю. *Сопоставление с образцом и идентификация структур в языке PL/1.* Автоматиизированный реинжиниринг программ, Изд-во СПбГУ, 2000г.

[25] Касьянов В.Н.*Оптимизирующие преобразования программ.* М., "Наука", 1988.

[26] Using and Porting GNU Compiler Collection (GCC). *http://gcc.gnu.org/onlinedocs/gcc_toc.html*

[27] ASDL (Abstract Syntax Definition Language): *http://www.cs.virginia.edu/zephyr/asdl.html*

[28] Objective Caml: *http://cristal.inria.fr/*

[29] EDG (Edison Design Group): *http://nci.pgroup.com*

[30] GCC (GNU Compiler Collection): *http://www.gnu.org/software/GCC/GCC.html*

[31] LCC: *http://www.cs.princeton.edu/software/lcc/*

[32] SPEC (Standart Performance Evaluation Corporation): *http://www.spec.org*

[33] SUIF (Stanford Unified Intermediate Format): *http://suif.stanford.edu*

[34] VPO (Very Portable Optimizer): *http://www.cs.virginia.edu/zephyr/vpo/*

[35] Zephyr: *http://www.cs.virginia.edu/zephyr/*