Санкт-Петербургский Государственный Университет
Математико-механический факультет
Кафедра системного программирования

Выбор инструкций и локальное распределение регистров в настраиваемом генераторе кода

Дипломная работа студента 58 группы
Вигдорчика Евгения Николаевича

Научный руководитель:	.... (Булычев Д.Ю.)
Рецензент:	...... (к.ф-м.н. Лаврова Ю.К.)
"Допустить к защите", заведующий кафедрой, профессор,д.ф-м.н.	.... (Терехов А.Н.)

Санкт-Петербург
2001

Оглавление

Глава 1

Введение

Технология совместной разработки программы на языках высокого уровня и модели вычислительного устройства, оптимально подходящего для исполнения данной программы, устоялась как самостоятельное направление в начале 90-х годов. С тех пор эта технология постоянно развивается, появляются все новые предложения по автоматизации процесса кодизайна, однако ни один из прелагавшихся вариантов пока не нашел применения у разработчиков встроенных систем и систем реального времени. Данная работа ставит своей целью прояснить слабые места существующих разработок в области генерации кода и делает попытку устранить указанные недостатки. В качестве основы для построения переносимого компилятора **Uburg** выбрана широко известная и применяемая технология BURS.

1.1 Постановка задачи

С самого своего зарождения кодизайн ставил как одну из основных задачу комплексной разработки всех традиционно необходимых разработчикам встроенных систем инструментов, в список которых входят симулятор разрабатываемого устройства, ассемблер, дизассемблер, а также компилятор и отладчик. Как и ожидалось, основной трудностью на пути реализации предложенной парадигмы стало несоответствие уровня детализации процессора, требуемого для построения точного симулятора и оптимизирующего компилятора. Авторы некоторых технологий, осознав трудность задачи сведения знаний о процессоре в единую форму с разной степенью детализации, ограничились автоматическим построением симуляторов, либо оставив задачу кодогенерации полностью открытой, либо предлагая описывать целевую программу на том же низкоуровневом языке, на котором описывается архитектура. Однако, как думается, построение точного симулятора в большинстве задач не является первостепенной задачей, достаточно ограничиться симулятором, выдающим метрики программы, оптимальные относительно специфицированного набора инструкций, но в то же время лишь аппроксимирующие реальное число тактов исполнения программы. Такой путь реально выводит RTL реализацию вычислительного устройства из сферы кодизайна, что представляется вполне допустимым решением, тем более, что автоматическая генерация RTL модели из поведенческой до сих

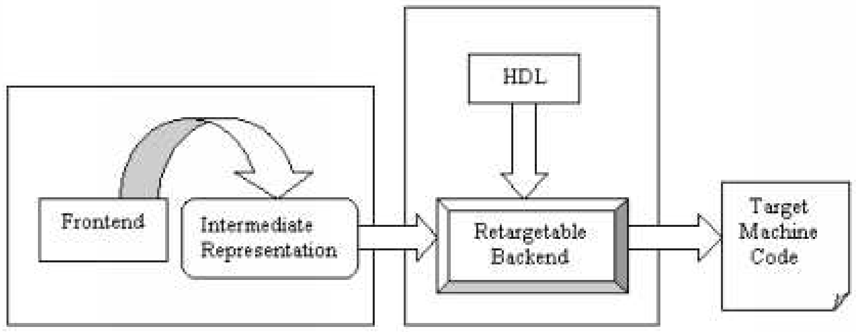

Рис. 1.1: Общая схема переносимой генерации кода при помощи HDL

пор не имеет эффективного решения.

В этих предположениях первоочередной задачей становится автоматическая генерация компилятора по некоему относитело высокоуровневому описанию вычислительного устройства, построенному при помощи некоторого формального языка описания архитектуры (*Hardware description language, HDL*). В качестве исходного языка удобно выбрать язык С, хотя и лишенный средств описания параллелизма, однако являющийся стандартом *de facto* в индустрии встроенных систем.

Основными требованием к компиляторам, применяемым в области кодизайна является его простота настраиваемости на новую машинную архитектуру (*retargetability*). Для процесса кодизайна это означает возможность сделать локальные изменения в описании архитектуры, такие как вставка новых инструкций, призванных ускорить выполнение целевой программы, введение сложных режимов адресации, усложнение структуры регистровых файлов, и получить транслятор, в максимальной мере использующий сделанные изменения. Кроме того, желательно было бы свести к минимуму обьем кода, не генерируемого напрямую из HDL. К таким компонентам традиционно относятся интерфейсы компоновки и модуль распределения регистров.

Алгоритм восходящего переписывания деревьев (*Bottom-Up Rewrite System*) является классическим двухпроходным методом кодогенерации, основанным на сопоставлении с образцом. (Подробнее алгоритм описан в главе 2.4) Данный алгоритм эффективно решает первые из поставленных задач (использование сложных инструкций и режимов адресаций) за счет введения сложных образцов. Однако другие упомянутые задачи в системах, основанных на BURS, в настоящее время либо не решаются автоматически, либо не решаются вовсе.

Данная работа ставит свой целью построить ориентированый на BURS кодогенератор, максимально интегрирующий распределение регистров в BURS-просмотры. Поскольку конечной целью является применение разрабатываемого транслятора в процессе кодизайна, основной упор сделан на генерацию кода для DSP процессоров и встроенных систем. К особенностям таких систем, не упоминавшимся ранее, следует еще отнести сложную структуру физического адресного пространства процессора, с доступом по разным шинам, и, следовательно, с различными временными задержками. Также характерно наличие нескольких регистровых файлов, вводимых специально для хранения некоторого подмножества данных целевого алгоритма. Задача тем самым включает в себя использование введенных регистров, управляемое описанием на HDL.

Ориентация данного транслятора на генерацию кода в кодизайне, однако, позволяет сделать и упрощающие предположения, использованные в данной работе. А именно, нет необходимости при порождении ассемблера учитывать его различные синтаксические особенности, как это происходит в переносимых трансляторах общего назначения (к примеру код для семейства процессоров **x86** для операционных систем Unix и Windows различается вплоть до полной неузнаваемости).

Машинно-зависимая часть компилятора тем самым должна состоять из единственного файла с описанием интерфейсов компоновки в достаточно обобщенном виде и собственно BURG-таблицы, описывающей набор инструкций целевого процессора. Дальнейшая интеграция в процесс кодизайна подразумевает конвертацию этого файла из HDL описания.

Глава 2

Близкие работы

2.1 Алгоритм Ахо-Джонсона. Оптимальный код.

В 1975 году Ахо и Джонсоном была доказана теорема об оптимальном распределении регистров для выражения, вычисляющего дерево. А именно, оптимальный код для дерева получается в результате применения алгоритма динамического программирования, в котором для каждого узла дерева вычисляется стоимость его вычисления в k регистрах, где k изменяется от 0 (выгрузка в память) до числа регистров в рассматриваемой архитектуре, а также оптимальная перестановка вычисления поддеревьев, на которой реализуется эта минимальная стоимость.

Данный результат, однако, подразумевает два существенных ограничения. Во-первых, структура программы считается деревом, что не позволяет, например, переиспользовать результат вычисления общих подвыражений. Во-вторых, все регистры в данной теореме считаются равноправными, то есть принадлежащими одному регистровому классу. Для DSP процессоров, как уже упоминалось, это предположение практически заведомо неверно.

В рамках исследования возможности применения идей, изложенных в этой теореме, была предпринята попытка обобщить результат на случай нескольких регистровых классов и дэгов. Появление дэгов в качестве промежуточного представления, как и ожидалось, привело к алгоритму, выдающему субоптимальное решение. Однако наличие нескольких неравноправных регистровых классов свело решение задачи к практически полному перебору, и от него пришлось отказаться.

2.2 Другие возможные подходы. Генератор кода CBC.

В первую очередь необходимо отметить неприменимость традиционных трансляторов, таких как gcc, для решения поставленных задач. В первую очередь это обьясняется их привязкой к фиксированному набору широко используемых процессоров и трудностью получить код для архитектуры, не входящей в этот ряд.

В настоящее время существует множество подходов к генерации кода. Некоторые методы приводят к трудно решаемым математическим задачам и требуют построения эффективных эвристик [Binate Covering]. Другие предлагаемые алгоритмы хотя и не приводят к экспоненциальному перебору, но тем не менее достаточно трудоемки для применения их на практике [Simulated Annealing]. Применение таких методов для переносимого транслятора осложняется тем, что выбор эвристик должен быть универсальным и одновременно достаточно полным, чтобы ограничить перебор.

Среди работ, посвященных кодогенерации, управляемой описанием машинной архитектуры следует выделить транслятор CBC, созданный в Техническом университете г.Берлина[9].

Как и в данной работе, в CBC сделан акцент на использование сложных потоков данных, практически всегда в изобилии присутствующих в архитектуре DSP вычислительного устройства. Кроме того, разработчики этого инструмента попытались учесть наличие нескольких функциональных блоков в процессоре, работающих параллельно, а также возможность разной ширины шин, и, соответственно, необходимость отображения базовых типов языка высокого уровня в типы, определяемые процессором. Первая из этих задач является классической задачей планирования, решаемой во VLIW архитектурах, поэтому ее рассмотрение не вошло в данную работу. Решение второй задачи, предлагаемое в CBC, а именно переписывание графа программы по стандартному набору правил и входной информации о длинах используемых операндов с последующей генерацией кода по перестроенному графу, может быть легко перенесено и в рассматриваемую модель.

Коренное отличие CBC от Uburg кроется во входном языке, на котором описывается целевой алгоритм. Разработчики CBC (вместе с компанией Siemens, поддерживавшей этот проект) сконцентрировали свои усилия на генерации кода для языка AlDisp. Описание алгоритма на этом языке подразумевает графическое или текстуальное задание графа потока сигналов. Фактически это означает задание потока данных, с дополнительными конструкциями управления. Таким образом генерация в гораздо большей степени ориентированна на поток данных, а не управления.

Исходя из вышесказанного разработчики CBC решили отказаться от традиционного в компиляторах промежуточного представления программы в виде дерева или ациклического направленного графа (DAG). Вместо этого ими использован смешанный граф потока данных/управления ($CDFG$). В этом графе графе вершины представляют операции программы, а дуги либо зависимость по данным, либо дополнительную зависимость по управлению. При этом второй тип дуг является динамическим, то есть такие дуги вставляются собственно генератором кода во время работы, и являются лишь ограничениями, необходимыми для правильной кодогенерации.

Именно чрезмерная ориентация CBC на поток данных, выражающаяся и в языке описания программы, сужает область его применения до небольшого числа классических алгоритмов цифровой обработки данных, в которых поток управления действительно играет второстепенную роль.

2.3 Система построения кодогенераторов BEG

В задачу инструмента BEG(*BackEnd Generator*), как следует из его названия, входит автоматическое построение кодогенератора из формального описания. Это, пожалуй, единственная на сегодняшний день попытка получить полностью управляемый внешней спецификацией процессора компилятор[4].

Механизм распределения регистров в BEG, как и в данной работе, основывается на связи нетерминала с классом размещения. Такая связь делает возможным распределение регистров на лету, то есть во время выбора инструкций, который осуществляется отличным от BURS, более медленным алгоритмом сопоставления с образцом. Для повышения качества генерируемого кода предусмотрен также другой вариант распределения регистров, выполняемого в отдельном просмотре дерева.

2.4 Описание алгоритма BURS

В общей постановке BURS представляет собой двухпроходный алгоритм сопоставления с образцом, применимый как к деревьям, так и к дэгам. На вход алгоритму поступает грамматика (N, Φ, Θ, C, R), где N - алфавит нетерминалов; Θ - набор образцов-поддеревьев, называемых также правилами (высоты 1 в случае приведенных грамматик), в котором каждый образец θ_k снабжен стоимостью $c_k \in C$; $\Phi = \{i \to j, c_{ij}; i, j \in N, c_{ij} \in C\}$ - набор цепных правил, также снабженных стоимостями; $R = \{r_k\}$ - множество абстрактных операций, применяемых во время редукции дерева. $r_k : R \times T \times S^k \to S$, где T - множество типов узлов дерева, k - степень дерева, а S - абстрактное множество результатов редукции.

Во время первого обхода дерева снизу вверх (разметка) происходит сопоставление с образцом и выбор оптимального покрытия дерева/дэга исходя из стоимости вывода поддеревьев и стоимости применения данного правила. Линейность алгоритма достигается за счет применения динамического программирования, а именно: в каждом узле дерева хранится набор правил r_j, выводящих нетерминал $j \in N$ с минимальной стоимостью. На каждом шаге алгоритма сначала просматривается множество нормальных правил Θ и выбираются правила, оптимально выводящие некоторое подмножество нетерминалов, а затем происходит замыкание этого подмножества применением цепных правил.

Второй проход сверху-вниз (редукция) начинается с выбора минимума стоимостей вывода корня дерева по всем нетерминалам. Далее последовательно применяются операции r_k, в качестве которых может, например, выступать печать ассемблерного кода инструкции, соответствующей правилу.

Общая теория генерации кода с помощью сопоставления с образцом на деревьях подробно изложена в [1]. Модификация алгоритма для дэгов может быть найдена в [2], и вкратце ее изложение присутствует далее в 3.1. Анализ алгоритма в применении к генерации кода с распределением регистров приведен ниже в 3.6.

2.5 Транслятор lcc. Frontend.

Транслятор lcc разрабатывается в рамках проекта NCI в Принстонском университете США с 1991 года и представляет собой переносимый транслятор с языка C, полностью совместимый со стандартом ANSI. lcc отличает высокая степень независимости модулей и компактность описания. Кроме того, разработчиками был предоставлен удобный интерфейс для усовершенствования возможностей компилятора через описание структуры программы на языке ASDL (*Abstract Syntax Definition Language*). Этот язык позволяет описывать структуры данных и получать их проекции на различных языках программирования, таких как C, C++, Java, Pascal, Ocaml (в AsdlGen версии 2.0). [6] Также генерируются процедуры сериализации описанных структур. Данный механизм заимствования синтаксического и видозависимого анализа был применен в данной работе.

Кодогенератор lcc использует алгоритм BURS для выбора инструкций. Для более эффективной работы для генерации кодогенератора применяется метод построения автоматно распознаваемой BURS грамматики [3]. После выбора инструкций выполняется распределение регистров при помощи алгоритма bin-packing. Этот алгоритм отличается малыми временными затратами, но получаемый код проигрывает по числу выгрузок в память традиционным методам раскраски.

Вторым, более существенным в контексте кодизайна, недостатком распределения регистров в lcc, является оторванность описания структуры регистровых файлов от описания набора инструкций и трудность автоматического получения этого описания из модели на HDL.

2.6 Peterburg

Peterburg представляет из себя попытку реализации BURS алгоритма, основанной на формальном определении BURS. Используемый в Peterburg формализм, позволяя задавать правила с неконстантной стоимостью, в остальном повторяет приведенный выше.

Данный инструмент, получая на вход дерево, множество нетерминалов, определение атрибутов разметки, набор правил для переписывания и определение операций для пересчета атрибутов и результатов редукции, производит разметку и выдает в качестве результата своей работы результат редукции дерева, производимой на основании полученной разметки.

Первоначально разработанный Д.Ю.Булычевым как инструмент для переписывания деревьев, Peterburg с небольшими изменениями был применен в данной работе как модуль, реализующий BURS алгоритм для дэгов.

Глава 3

Комплексное решение задачи при помощи BURS

В данном разделе мы опишем реализацию алгоритма BURS, интегрирующую в себя часть задач, связанных с распределением регистров. Кроме того предлагаемый алгоритм позволяет обьединить описание структуры регистровых файлов с описанием собственно системы команд, что значительно облегчает задачу автоматической генерации всей машинно-зависимой части компилятора из описания на HDL.

3.1 Алгоритм восходящего переписывания дэгов

Отличие приведенной формализации для деревьев от используемого алгоритма состоит в определении операций редукции. Мы полагаем $r_k : N \times (T \times (\{null\} \bigcup S) \times S^k \to S$. В этом определении предполагается, что информация о предыдущем результате редукции сохраняется и доступна при следующем посещении (разделяемого) узла дэга.

Неформально модификация состоит в том, что допускается заметить, что данный нетерминал уже был выведен раньше и переиспользовать результат. Данная формализация, очевидно, (задача получения оптимального кода для дэга *NP* трудна), не дает возможности получить оптимальный код, что иллюстрирует 3.1. Причина неоптимальности кроется в том, что решение о переиспользовании результата предыдущей редукции зависит от совпадения нетерминалов, к которым производится вывод. (Возможное эвристическое применение цепных правил на этапе редукции ведет к необходимости заново разметить дэг со стянутым в вершину подграфом с корнем в текущем узле. Таким образом, мы получаем квадратичный относительно размеров дэга алгоритм с сомнительным выигрышем в качестве порождаемого кода) Однако подобные ситуации являются относительно редкими и зависящими от особенностей архитектуры процессора.

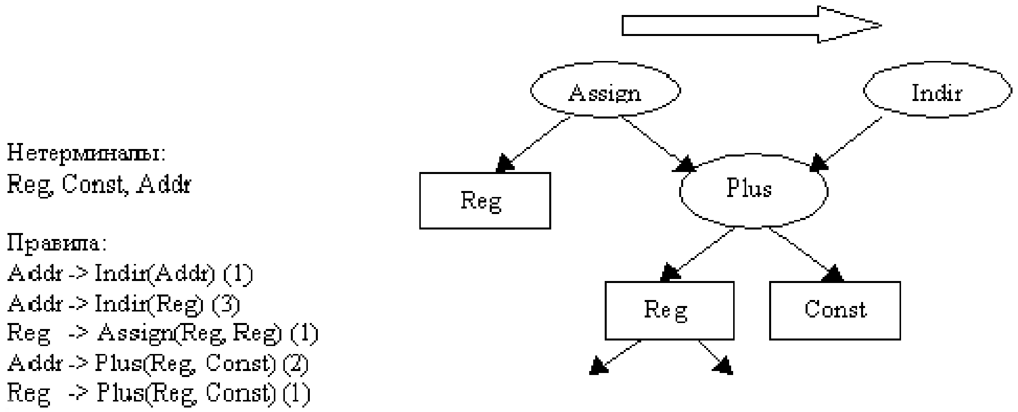

Рис. 3.1: Случай генерации неоптимального кода

3.2 Предлагаемая общая схема распределения регистров

Для начала определим виртуальный регистр как упорядоченной пару (r, i), где r - номер регистрового файла, а i - натуральное число, обозначающее номер регистра. При этом не предполагается, что в данном файле действительно должен быть регистр с таким номером.

Распределение регистров предлагается проводить в два этапа: первый этап, интегрированный в BURS, назначает виртуальный регистр всем узлам дэга, оптимальное размещение которых находится в регистре; второй этап, традиционно осуществляемый раскраской графа, присваивает виртуальным регистрам машинные, в случае же невозможности такого присваивания осуществляя выгрузку в память.

3.3 Нетерминалы

Нетерминалы в Uburg имеют помимо описанного в 2.4 значения дополнительную смысловую нагрузку, связанную с распределением регистров. Все нетерминалы делятся на две категории: с ограниченным числом экземпляров, представляющие различные регистровые файлы, и доступные в произвольном количестве. Вторая группа состоит, вообще говоря, из нетерминалов, представляющих размещение в памяти, и вспомогательных нетерминалов (вводимых, например, для учета сложных режимов адресации), однако с точки зрения алгоритма распределения регистров эти подклассы неразличимы. Данное разделение позволяет отличить, в каком случае для нетерминала-регистра необходимо вычислить его размещение, и в каком случае это делать необязательно и может даже привести к неправильному коду (как это происходит с нетерминалами, обозначающими размещение на стеке).

3.4 Ограничения на размещения нетерминалов

Как уже было отмечено, DSP процессоры отличаются достаточно сложной структурой регистровых файлов. Например, определение инструкции может требовать, чтобы ее операнды выбирались из двух различных регистровых файлов, а результат помещался бы в некоторой фиксированный регистр первого файла. Данная ситуация, конечно, может быть разрешена разделением файла на два подфайла, состоящих из выделенного регистра и всех оставшихся, однако такой путь не всегда представляется возможным. Примером ситуации, где такое решение неприменимо, является архитектура процессора Intel x86: как известно, в данной архитектуре предполагается вложение байтовых регистров в полусловные (длина машинного слова предполагается равной 32 битам), а тех в свою очередь в 32-битные регистры. Рассмотрение регистров разной длины независимо, таким образом, не представляется возможным, так как например значение, хранимое в cx, может быть использовано в инструкции, оперирующей с cl.

Другим примером необходимости явного задания ограничений на размещения нетерминалов может служить двухадресная система команд (Двухадресной называется инструкция с двумя операндами, результат которой должен быть помещен в один из регистров операндов). В этом случае требуется явное указание, в какой именно из регистров операндов должен попасть результат.

В качестве третьего примера рассмотрим присваивание, в котором и левая, и правая части присваивания находятся в регистрах одного регистрового класса. В этом случае единственное, что требуется указать в правиле для присваивания - это ограничение на равенство регистров операндов. Данное ограничение приведет к тому, что на этапе разметки будут выравнены номера виртуальных регистров (именно, операнд в левой части присваивания получит виртуальный регистр, присвоенный ранее операнду в правой части). Следует отметить, что данная модификация не вполне вписывается в классическую схему BURS алгоритма, не подразумевающего изменение атрибутов дочерних узлов.

Приведенные ограничения можно разбить на две группы:

- управляющие выделением виртуальных регистров; эти ограничения определяют работу фазы разметки, определяя области действия назначаемых виртуальных регистров.

- контролирующие назначение виртуальным регистрам машинных. Данные ограничения, в свою очередь, подразделяются на парные и собственные, первые из которых задаются для пар виртуальных регистров, а вторые для одного регистра.

3.5 Определения

Определение: Топологической сортировкой дэга называется такая нумерация его узлов M, при которой $n_1 \to^* n_2 \Rightarrow M(n_1) < M(n_2)$

В случае одного регистрового файла можно дать также следующее определение: Оптимальной будем называть топологическую сортировку, индуцирующую вычисление дэга в минимальном числе регистров.

Для определения оптимальной сортировки в случае множества регистровых файлов нетерминалы дополнительно снабжаются весами, под которыми понимается стоимость размещения значения в данном регистровом файле. К примеру в процессоре x86 размещение значения, участвующего в арифметическом выражении, в регистре %si менее желательно, чем в %ax, что естественно выразить большей стоимостью.

Определение: Областью действия wol в дэге D с началом в t, называется такой непустой подграф $G \subseteq D$, что $r \to^* t \forall r \in G$, то есть t является постдоминатором всех вершин подграфа.

Областью действия виртуального регистра называется область действия, все вершины которой помечены данным виртуальным регистром, и ни одна другая вершина дэга не помечена данным виртуальным регистром.

3.6 Разметка дэга

Фаза разметки предполагает следующие модификации классического алгоритма, связанные с распределением регистров и индуцированной топологической сортировкой:

#1 Производится назначение виртуальных регистров. Данное назначение предлагается делать оптимистически, продолжая по возможности область действия какого-либо из виртуальных регистров операндов. Такое продолжение возможно при применении данного правила, если выводимый по этому правилу нетерминал совпадает с одним из нетерминалов, покрываемых образцом.

При необходимости программист может указать в определении правила ограничение, в результате которого в текущем узле будет порожден новый виртуальный регистр. Однако такое ограничение не гарантирует, что результат вычисления данного узла действительно не попадет в тот же *машинный* регистр, что и соответствующий операнд: чтобы обеспечить такое условие, программист должен явно указать еще и ограничение в виде предиката, передаваемого раскрасчику.

#2 В текущем узле сохраняются все ограничения, диктуемые применением правила. При этом в узле не сохраняются уже накопленные ограничения.

#3 Выбор топологической сортировки происходит следующим образом: для дэга, растущего из текущего узла, веса его поддэгов сравниваются и упорядочиваются по убыванию. В качестве веса самого дэга выбирается некоторая эвристически выбираемая функция, например: $W_t = w_{n_d} + max(W_i)$, где w_{n_d} - вес выводимого по данному правилу нетерминала, а W_i - веса поддэгов. Данная эвристика реализовывает принцип "первым вычисляется дерево с наибольшим регистровым аппетитом", при этом полагая все регистры вычисленного поддэга освободившимися.

Существенным недостатком данной эвристики является, однако, некорректное поведение для разделяемых узлов дэга: так как на стадии разметки информация о разделяемости узла недоступна, то веса нетерминалов из разделяемого поддэга учитываются многократно, потенциально порождая тем самым неоптимальную топологическую сортировку.

3.7 Редукция дэга. Построение локального графа несовместимостей.

При выполнении редукции мы считаем, что в каждом узле доступна информация о перенумерации сыновей, получаемая из топологической сортировки, номер i виртуального регистра и список ограничений, которым должна удовлетворять раскраска данного узла и его соседей. В соответствии с общей схемой BURS эта информация сохраняется как атрибут разметки, ассоциированный с выводом нетерминала (что вместе с номером i полностью определяет виртуальный регистр r) в данном узле.

По этой информации строится локальный граф несовместимостей следующим образом:

#1 Пусть степень текущего узла равна 2 и (r, r_2) - виртуальные регистры, которыми помечены текущий узел и вычисляющийся вторым поддэг соответственно. Тогда для всех виртуальных регистров r_i, таких что $wol(r_i) \cap first \neq \emptyset$, где $first$ - поддэг текущего дэга, вычисляющийся первым, добавить в граф ребро (r_2, r_i). Если r_2 вычисляется в память, то данный шаг опускается.

#2 Для всех парных ограничений раскраски, заданных для r и некоторого r_j, добавить в граф ребро (r, r_j), помеченную данным ограничением. Для собственных ограничений r добавить петлю (r, r).

3.8 Модификация классического алгоритма раскраски графов

Раскраску графа предлагается проводить оптимистически согласно описанию в [8] со следующими изменениями:

В результате построения графа несовместимостей данный граф содержит ребра двух типов, а именно отражающие собственно несовместимость(т.е. невозможность нахождения в одном регистре), а также ребра, помеченные предикатом, истинность которого должен обеспечить результат раскраски. Предлагается учитывать данные ограничения следующим образом:

- При наличии вершины с числом несовместных соседей меньше числа регистров в требуемом регистровом файле данная вершина кладется на стек и удаляется из графа со всеми инцидентными ей дугами.

- Если таких вершин в графе нет, то на стек кладется вершина, с минимальной взвешенной стоимостью, определяемой как $I_{node} + k * C_{node}$, где I_{node} и C_{node} - число инцидентных дуг первого и второго сорта соответственно, а k - эмпирически подбираемый коээфициент.

- После того, как все вершины графа будут переработаны, вершины одна за одной извлекаются со стека. Текущая извлекаемая вершина должна получить машинный регистр из требуемого файла, отличный от уже назначенных ее соседям первого рода (имеются в виду инцидентные вершины первого рода), и удовлетворяющий всем условиям, составляющим пометку петли в данной вершине, если таковая присутствует в графе. Далее все инцидентные дуги второго рода удаляются из графа, а вместо них добавляются петли в соответствующие инцидентные вершины.

- При невозможности назначить машинный регистр в соответствии с описанной процедурой производится выгрузка и загрузка соседа первого рода в память и присваивание занимаемого им регистра текущей вершине. В контексте локального распределения регистров граф строится для размещений внутри линейного участка, таким образом места выгрузки и загрузки локализованы в данном линейном участке, и их определение не представляет труда. Однако необходимо иметь в виду, что области действия двух интерферирующих виртуальных регистров могут пересекаться более чем в одном узле дэга программы, таким образом количество выгрузок и загрузок,производимых в результате выгрузки соседа в графе несовместимостей, может превосходить 1 и порождать неоптимальный код. Существует три способа борьбы с подобными нежелательными эффектами: первый из них предлагает разделить область действия виртуального регистра, перестроив граф несовместимостей, во втором случае предлагается подсчитывать во время редукции дополнительно стоимость вычисления программы без разделяемых узлов (и, следовательно, множественных точек интерференции) и при превышении основной стоимости вычисления над дополнительной вычислять программу действительно как дерево. Третий способ, наиболее подходящий для генерации кода в контексте кодизайна, состоит в том, что программист самостоятельно должен указать набор правил, в которых прерывается область действия виртуального регистра, ограничивая, таким образом степень вершин в графе несовместимостей.

Глава 4

Настраиваемый кодогенератор Uburg

4.1 Реализация предложенного алгоритма

Здесь будут приведены некоторые детали реализации описанного в предыдущей части алгоритма, а также на примере архитектуры процессора SPARC будет дано описание структуры машинно-зависимой части кодогенератора.

4.1.1 Структура uburg

Для решения поставленной задачи было решено использовать язык Ocaml[7] . Данное решение обьясняется следующими факторами:

- Ocaml, как и другие функциональные языки, позволяет легко реализовывать алгоритмы над ссылочными типами данных
- Ocaml обладает мощной модульной системой, необходимой для построения большой системы
- В отличие от большинства других функциональных языков ocaml обладает статической типизацией, что, в частности, значительно снижает процент ошибок, доживающих до времени исполнения программы.

В настоящий момент в состав uburg входят следующие модули:

Tree конвертирует промежуточное представление lcc во внутреннее представление uburg. Основное отличие получаемой структуры кода состоит в наличии физически (на уровне равенства указателей) разделяемых узлов. Также происходит отбрасывание ненужной для генерации ассемблера информации, такой как, например, тип идентификатора или его класс хранения.

Functors описывает реализацию интерфейсов, определяемых модулем **Peterburg**

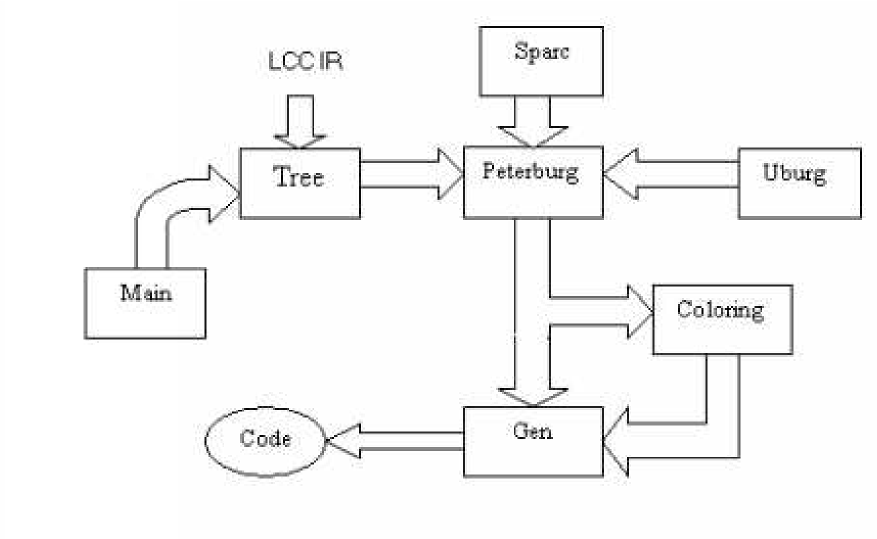

Рис. 4.1: Зависимость между модулями в Uburg

Peterburg Данный модуль содержит определения интерфейсов (*module type* в терминах языка ocaml) и реализацию алгоритма BURS для модулей, реализующих эти интерфейсы, в соответствии с описанием в 2.6.

Graph определяет интерфейс и реализацию абстрактного графа

Cfg дополняет **Graph** до графа потока управления введением выделенных вершин `start` и `stop`

Uburg содержит собственно операции, вызываемые **Peterburg** во время разметки и редукции

Coloring раскрашивает построенный при помощи BURS граф несовместимостей, распределяя, таким образом, машинные регистры

Gen,Main модули верхнего уровня

Sparc содержит описание подмножества набора инструкций процессора **SPARC V.9** для целочисленных операндов

4.1.2 Типы вершин графа управления и сохраняемая дополнительная информация. Модуль Tree.

Программа (точнее, код каждой функции) в Uburg может быть представлена двумя способами: либо линейной последовательностью узлов типа **node**, либо графом потока управления. Первое представление используется в качестве входных данных в BURS алгоритме. Граф потока управления (CFG) традиционно строится для проведения всевозможных оптимизаций, главной из которых в контексте данной работы можно назвать глобальное распределение регистров, предполагаемое в дальнейшем. В настоящий момент CFG-представление в Uburg не испольуется.

Тип **node** определяется в модуле Tree следующим образом:

```
type node =
   Binary of mode * size' * node * bin_op * node
|  Unary of mode * size' * node * unary_op
|  Jump of node
|  Local of uid * size' * align'
|  Global of string
|  Param of uid * param_dir * size' * align'
|  Const of int
|  Constf of float
|  Arg of node * size'
|  Call of mode * size' * node
|  Cmp of node * cmp_op * node * int
|  Branch of int
|  Convert of mode * node * size' * size'
|  Enter
|  Leave
|  Label of int
```

Большинство перечисленных типов узлов совпадает с описанием в lcc (файл rcc.asdl), однако некоторые из них были введены в Uburg.

- Узлы *Enter и Leave* служат индикаторами входа и выхода из функции соответственно. Покрытие такого узла может, например, состоять в переключении оконных регистров в SPARC или в сохранении указателя стека (однако стековый аппетит функции в момент покрытия неизвестен; для сдвига указателя верхушки стека предусмотрен другой механизм).

- Узлы *Global, Local и Param* представляют переменные программы. Именно эти узлы (в комбинации с операцией Unary(INDIR), разыменовывающей указатель) являются наиболее часто разделяемыми узлами. Глобальные переменные представлены только идентификатором, так как мы предполагаем, что конечный правильный код для перемещаемых (*relocatable*) обьектов будет сгенерирован линкером. Описание локальных переменных и параметров функций содержит атрибуты размера и выравнивания, необходимые для принятия решения о размещении в регистрах, в памяти или на стеке, а также для расчета стекового аппетита в последнем случае. Для параметров также добавлен атрибут **param_dir**, обозначающий передачу параметра от вызывающей функции к вызываемой или наоборот. INOUT параметр, таким образом, будет рассмотрен как два независимых параметра. Такое независимое рассмотрение позволяет реализовывать достаточно сложные соглашения о вызовах. Атрибут **uid** в описании локальных переменных и параметров служит ключом, позволяющим различать переменные с совпадающими размером и требованием по выравнванию, предотвращая разделение таких узлов.

Распределение ресурсов для локальных переменных и параметров производится при первом покрытии соответствующего узла. Так как извлекаемые параметры должны быть согласованы с аргументами в вызывающей функции, то список выражений, покрываемых Uburg дополняется узлами с конструктором *Param* для всех описанных параметров(добавление происходит в голову списка). Такая модификация внутренней структуры упорядочивает покрытие данных узлов в соответствии с описанием на С, сохраняя в то же время возможность распределения ресурсов для параметров внутри BURS4.1.7.

4.1.3 Модификация Peterburg для дэгов

В своей первоначальной реализации Peterburg подразумевал покрытие как деревьев, так и дэгов. Однако в процессе разработки Uburg выяснилось, что для корректной работы Peterburg с дэгами необходимо произвести изменения, существенно снижающие скорость работы Uburg и требующие корректировки Peterburg в дальнейшем.

Согласно основной концепции 2.6, на вход Peterburg должен поступать неразмеченный граф, что дает возможность максимально инкапсулировать реализацию BURS внутри Peterburg, производя разметку, невидимую вне данного модуля. Такое решение было вполне оправдано в случае покрытия деревьев, однако, в случае дэгов привело к следующей проблеме.

При построении нумерации узлов дэга необходимо хранить все встреченные узлы, чтобы при следующем прохождении узла корректно присвоить уже вычисленный номер. Таким образом, даже при оптимальном хранении всех узлов, на поиск нужного узла нужно потратить время, пропорциональное логаримфу общего числа узлов в дэге. Ситуация усугубляется еще и тем, что каждое сравнение выполняется, вообще говоря, рекурсивно для всего поддэга, растущего из данного узла (В Ocaml нет возможности сравнивать указатели на равенство, так как указываемые объекты могут быть перемещены сборщиком мусора).

Частичное решение сформулированной проблемы, применяемое в Uburg, заключается в явном указании предиката, которому должен удовлетворять разделяемый узел. (В Uburg разделяются только узлы типа *Local, Global, Param, Binary и Unary*) Это позволяет свести трудность поиска нужного узла к $M * \log(N)$, где N - число потенциально разделяемых узлов, а M - размер макимального поддерева, растущего из этих узлов.

4.1.4 Задание BURS правил

Для построения таблицы правил в модуле **Peterburg** определены следующие функции:

```
        val rule : nonterm * Tree.t * nonterm list * sema * 'a
rema -> 'a core
        val chain : nonterm * nonterm * sema * 'a rema -> 'a core
```

В данном определении *sema* и *'a rema* - тип callback функций, вызываемых соответственно при разметке и редукции по данному правилу; *'a core* - общий тип всех правил.

В дополнение к обычным правилам Peterburg предоставляет условные, отличающиеся тем, что применение такого правила возможно только в случае выполнения в узле t заявленного предиката.

Пример построения таблицы правил в **Uburg** приводится далее в 4.1.7.

4.1.5 Дэги в Uburg

Как было описано выше [4.1.2], большинство операций в промежуточном представлении имеют атрибут размера результата. Это связано, очевидно, с тем, что архитектура процессора чаще всего предоставляет различные инструкции для выполнения операций над данными различной длины. Однако некоторые из операций зачастую выполняются одинако для всех базовых машинных типов. С другой стороны, присваивание отдельного типа всем операциям со всевозможными размерами результатов в несколько раз увеличило бы число типов узлов и тем самым входной файл с описанием архитектуры.

Предлагаемое решение опирается на применение условных правил: в качестве типа узла выбирается операция вне зависимости от размера ее результата. При необходимости использовать какое-либо правило для данных фиксированного размера данное правило оформляется как условное [4.1.7].

4.1.6 Ограничения на размещения нетерминалов

В **Uburg** программисту дана возможность указать желаемые ограничения на размещения в определении правила. Ограничения задаются с использованием типа ext_constraint, определенного следующим образом:

```
type bin_constr = int -> int -> bool
   and unar_constr = int -> bool
   and rule_pos = RES | LEFT | RIGHT
   and ext_constraint = Die |
         Pred of rule_pos |
         Constr2 of rule_pos * rule_pos * bin_constr |
         Constr1 of rule_pos * unar_constr
```

Ограничения *Die* служат явным указанием Uburg поместить результат операции в новый виртуальный регистр. Данное ограничение необходимо, в случае если результат физически не может быть помещен в регистр операнда, например из-за недоступности шины.

Ограничения *Pred*, наоборот, явно указывают позицию операнда, в чей регистр будет помещен результат. При этом предполагается, что нетерминалы результата правила и операнда в данной позиции совпадают и являются регистровыми.

Наконец последние два конструктора позволяют задать предикат на множестве натуральных чисел (*Constr1*) для собственных ограничений или пар натуральных чисел (*Constr2*) для парных ограничений, которым должны удовлетворять регистры для применения данной инструкции. При этом проверка непустоты характеристического множества предиката, а также однородность этого множества оставляются программисту, осуществляющему спецификацию процессора.

Данное определение, а также концепция карринга, поддерживаемая Ocaml, делает тривиальным перенос ограничений из дуг в вершины, необходимый во время раскраски: достаточно лишь записать в вершину частично примененную функцию, с подставленным значением вычисленного машинного регистра.

4.1.7 Размещение значений в памяти

Хотя в данной работе в основном рассматривается задача эффективного размещения значений в регистрах, однако, в некоторых случаях оптимальным, а иногда даже единственно приемлемым, как в случае структур, оказывается размещение в памяти.

Для вычисления смещения размещаемого значения в памяти в Uburg введен специальный класс объектов, представляющих область размещения, с единственным полем, отражающим смещение текущего значения, и методом, позволяющим по длине размещаемого объекта и его требованию по выравниванию получить требуемое смещение. Данное смещение является аналогом виртуального регистра и, будучи переведено в строку согласно определению нетерминала, отвечающего за данную область памяти, дает ассемблерный код адреса данного объекта. Следует отметить, что размещение требуемых значений происходит последовательно, что не дает возможности реализовать генератор кода для стековой машины.

4.2 Описание машинной архитектуры на примере SPARC

В этом разделе мы на примере целочисленного подмножества системы команд процессора SPARC V.9 покажем основные синтаксические конструкции файла описания архитектуры.

4.2.1 Определение нетерминалов

Как известно, в архитектуре SPARC предусмотрено несколько классов регистров:

- 8 глобальных регистров доступны в любой момент выполнения программы и служат тем самым для хранения глобальнных данных.
- 8 локальных регистров хранят локальные переменные процедуры, данные регистры имеют видимость, ограниченную данной процедурой (в разных процедурах регистровое окно содержит разные локальные регистры)
- Входные и выходные регистры служат для передачи аргументов и результатов между вызывающей и вызываемой функцией.

В рассматриваемой реализации, однако, все регистры представляются одним нетерминалом Reg, так как во-первых глобальные переменные хранятся в памяти (данная работа не подразумевает проведения глобального анализа графа вызова процедур для выяснения переменных, кандидатов на размещение в регистрах), а во-вторых входные и выходные регистры могут использоваться в вычислениях наравне с локальными. Приоритет использования регистров, задаваемый порядком их выделения раскрасчиком, позволяет указать первоочередное использование локальных регистров, затем глобальных, и лишь в крайнем случае занятие входных или выходных регистров (исключая регистр %%i0, используемый только для возвращения результата функции).

4.2.2 Примеры правил

Для начала рассмотрим примеры простых правил, так как все правила в Uburg строятся похожим образом:

```
rule (Con, cnst, [], (laba 0 Con []), (reda ("%a", false) ))
rule (Reg, global, [], (laba 1 Reg []), (reda ("set .L%a,
%%%c\n", true) ))
```

Первое из этх правил определяет константу как непосредственное (immediate) значение, второе загружает значение глобальной переменной в регистр.

В этих определениях *rule* - абстрактная функция для построения правила, определяемая в Peterburg; Con и Reg - соответствующие нетерминалы; cnst и global - образцы, покрываемые правилом; laba и reda - функции Uburg для разметки и редукции дэгов; 0 и 1 - стоимости соответствующих правил; строковые выражения - определения ассемблерного представления для данного правила; логические константы указывают на необходимость распечатать данное ассемблерное представление в результирующем коде; [] на верхнем уровне задает пустой список сыновей данного образца; [] в функции разметки обозначает пустой список ограничений.

```
rule_p (Stk13, local, [], frame13, (laba 0 Stk13 []),
reduce_stk )
```

В данном условном правиле, размещающем локальные переменные на стеке для редукции используется стандартная функция разметки стека, получающая в качестве параметров размер локальной переменной и требования к выравниванию и возвращающая смещение переменной относительно начала стековой рамки текущей процедуры. В качестве условия для применения этого правила фигурирует предикат frame13, истинный в случае, если смещение переменной на стеке по модулю меньше 2^12.

```
chain (Reg, Stk13, (laba 2 Reg []), (reda ("add %0, %%fp,
%%%c\n", true) ))
```

При помощи *chain* задаются цепные правила. В данном случае значение, хранимое на стеке со смещением, не превосходящим 4092, загружается в регистр.

```
rule (Base, addi, [Reg;Con13], (laba 0 Base []), (reda
("%%%0+%1", false) ))
```

В этом примере базовое значение формируется как сумма регистрового и короткой константы. Этот пример иллюстрирует типичное вычисление арифметического выражения.

В качестве последнего примера рассмотрим правило, определяющее присваивание двух регистровых значений. Как уже упоминалось, в этом случае стратегия **Uburg** предполагает не копирование значений между регистрами, а сращение регистров, выражающееся в присваивании левой и правой частям присваивания одного виртуального регистра.

Синтаксически такое правило для присваивания выглядит следующим образом:

```
rule (Stmt, asgni, [Reg;Reg], (laba 0 Stmt [Equalize]),
(reda (, false) ))
```

Глава 5

Заключение

5.1 Заключение

В предыдущих разделах мы представили модификацию классического алгоритма выбора инструкций при помощи восходящего переписывания дэгов, максимально включающую в себя распределение регистров, а также явное планирование инструкций. Предложенный подход позволяет описывать широкий класс машинных архитектур, включая DSP процессоры с помощью определения формальных правили регистровых нетерминалов. Порождаемый код для процессора SPARC по качеству сравним с кодом, генерящимся транслятором lcc, однако простота описания структуры регистровых файлов 4.2 значительно расширяет применимость Uburg по сравнению с lcc.

5.2 Дальнейшие направления исследования

В рамках данной работы был рассмотрен вопрос об интеграции локального распределения регистров в BURS просмотры. Очевидно, что для более эффективной генерации кода требуется распределять регистры глобально. Также очевидно, что такое глобальное назначение регистров невозможно без анализа графа потока данных (или его аналога), и, следовательно, напрямую не выражается на языке BURS просмотров. Однако, как видится, существует возможность использования полученных результатов и в контексте глобального распределения регистров. Именно, вместо того, чтобы сразу после получения локального графа несовместимостей пытаться раскрасить его (вставляя на границе линейного участка выгрузки в память и загрузки обратно в регистры), можно использовать полученные графы в качестве элементов полурешетки для решения задачи склеивания локальных графов, результатом которой должен стать общий граф несовместимостей в вершине $STOP$ графа управления. Данное предложение нуждается в дальнейшей формализации и проработке.

Также нуждаются в переработке некоторые аспекты реализации, в частности внутреннее представление программы было слишком упрощено, что создало определенные проблемы при раскрутке Uburg на SPARC. Исходя из полученного опыта, представляется целесообразным определить класс архитектур, на которые Uburg должен легко раскручиваться, и максимально оптимизировать промежуточное представление для этого класса.

5.3 Приложение. Примеры порождаемого кода

Ниже приведена простая программа на C, вычисляющая факториал, а также ассемлерный код, сгенерированный uburg и lcc соответственно.

```
unsigned fact(unsigned n) {
  unsigned res = 1, i;
  for (i = 1; i <= n; i++)
  res = res * i;
  return res;
}
```

% This is Uburg output	% This is lcc's output
.global fact	.global fact
.section ".text"	.section ".text"
.align 4	.align 4
fact:	fact:
save %sp,-112,%sp	set 1,%o4
set 1, %l0	set 1,%o5
set 1, %l1	ba .L5; nop
ba .L5; nop	.L2:
.L2:	umul %o5,%o4,%o4
umul %l1,%l0,%l0	.L3:
.L3:	set 1,%o3
add %l1,1,%l1	add %o5,%o3,%o5
.L5:	.L5:
cmp %l1,%i0; bl .L2; nop	cmp %o5,%o0; blu .L2; nop
mov %l0,%i0	mov %o4,%o0
ret; restore	.L1:
.L1:	retl; nop
ret; restore	.type fact,#function
.type fact, #function	.size fact,.-fact
.size fact ,.-fact	

Из приведенного ассемблерного листинга видно, что lcc потребовалось для 3 регистра, включая дополнительный регистр для константы 1, в то время как Uburg использовал лишь 2.

Литература

[1] 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.

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

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

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

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

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

[7] Objective Caml:
http://cristal.inria.fr/

[8] G.J.Chaitin. *Register allocation and spilling via graph coloring.* ACM SIGPLAN Simp. on Compiler Construction, June 1982.

[9] Andreas Fauth, Günter Hommel, Alois Knoll, Carsten Müller. *Global Code Selection for Directed Acyclic Graphs*