К вопросу о кодогенерации для архитектур с ILP *

Борис Муратшин (zzeng@mail.ru),
Александр Артюшин (alexnikart@mail.ru)

Декабрь 7, 2005

Содержание

1 Введение

Сначала о том, о чем, собственно, мы собираемся говорить. О VLIW. VLIW [1] является альтернативой суперскалярной обработке. Использование этого метода предполагает задание в командном слове совокупности параллельно выполняемых команд [5]. Основными достоинствами данной схемы являются:

*Instruction Level Parallelism

[1] Very Long Instruction Word

- Компилятор поставлен в менее жесткие временные рамки по сравнению с суперскалярным ядром и располагает большими ресурсами. Следовательно, потенциально он может более эффективно исследовать зависимости между командами и создавать параллельно исполняемый код.
- VLIW процессор более прост.

Основные недостатки:

- Компилятор не в состоянии предсказывать переходы, зависящие от вычисляемых данных
- Задача генерации кода для VLIW существенно сложнее ее же для суперскалярной архитектуры и на данный момент хорошего ее решения не известно.

В данный момент существует целый ряд процессоров, которые можно отнести к этой архитектуре. Видимо, существуют две причины этого. Во-первых, удобно (по разным причинам, например, чтобы облегчить дизайн) иметь VLIW-ядро процессора и "обертку"из микрокода. Во-вторых, элементы VLIW могут использоваться высокопроизводительными суперскалярными процессорами, для предоставления пользователю возможности управлять параллелизмом.

Чем нас привлекает данная архитектура? Ее потенциалом. Что нам предлагает текущий уровень развития технологии? Анонсированный Texas Instruments VLIW процессор TMS320C641x работает на частоте 600Mhz и выполняет до 6 инструкций за такт. Достижимая частота измеряется уже гигагерцами, причем надо отметить, что VLIW за счет своей простоты по сравнению с суперскалярным процессором переносит потенциально большие частоты. Если бы мы имели компилятор, способный загрузить его в среднем наполовину, получился бы процессор с реальной производительностью 1800MIPS, стоимостью $10 и не требующий охлаждения ввиду низкого энергопотребления. Вопросом является не изготовление такого процессора, а как раз создание компилятора.

Данная работа является попыткой наметить подходы к построению архитектуры, полноценно использующей внутренний параллелизм и при этом допускающей недорогую генерацию качественного кода.

2 Цели

Общей целью для нас (во всяком случае, в данном контексте) является возможность построения эффективной конечной системы. Подцелями будут:

- эффективный процессор
- эффективный компилятор
- подходящая аппаратная поддержка процессора

3 Тезисы

1. Распределение ресурсов при генерации кода для VLIW процессоров является NP-полной задачей.

2. Для суперскалярных процессоров распределение ресурсов тоже NP-полная задача, но для них существует множество отработанных дешевых эвристик, например, существующие языки программированию.

3. Было бы неплохо показать, почему именно старые добрые эвристики не работают в случае VLIW, например, из-за нелокальности изменений кода и динамической стоимости инструкций.

4. Вопрос - с чего (и для какой архитектуры) надо компилировать чтобы не возникала NP-полная задача.

5. Глобальным будем называть наименьший из всех локальных минимумов некоторой функции F(x1, x2, ...x(n)) в определенной области. В нашем случае глобально-оптимальным мы назовем тот код, который для определенных алгоритма и набора входных данных исполнится за наикратчайшее время.

6. Мы отказываемся от поиска глобального минимума, но не путем поиска ближайшего локального, а пытаемся переформулировать задачу и упростить ее таким образом, чтобы новый глобальный минимум находился за полиномиальное время. Тем более что понятие алгоритма не формализуемо, и мы всегда имеем дело с конкретной записью алгоритма, что уже является эвристикой. Поиск нужной формы записи алгоритма также является нашей задачей.

7. Мы пытаемся представить алгоритм в виде сети потоков данных и решить задачу самого быстрого пути в некоем образованном от этой сети графе, что легко делается жадными эвристиками.

8. При правильном выборе весов ребер, это не потребует бесконечных ресурсов для хранения промежуточных результатов. Т.е. хотя такой подход к решению задачи в чистом виде и не является "серебряной пулей"против экспоненты и по существу эквивалентен полному перебору вариантов, все же он переводит задачу из разряда "не поймешь, как подступиться"в категорию "понятно, где искать эвристики".

4 Проблемы

Основной проблемой является то, что задача оптимизации кода всегда рассматривалась в непригодном для VLIW контексте. Оптимизатор для традиционного

микропроцессора подразумевает локальность производимых изменений. Простой пример: переставив местами две независимые инструкции, мы, скорее всего, просто ничего не заметим, так как в случае суперскалярной архитектуры ядро перепланирует наш код к внутреннему представлению, иначе, эти инструкции просто выполнятся в обратном порядке. Т.е. существует возможность сгенерировать набросок кода в надежде улучшить его в дальнейшем. Следует отметить, что оптимизирующие компиляторы вообще появились относительно недавно в связи с возросшей мощью процессоров. Так, к примеру, язык С изначально не был языком высокого уровня т.к. программист имел множество возможностей давать компилятору подсказки относительно распределения регистров, быстрой арифметики и т.д. Впрочем, даже если задача оптимизации не стоит, мы должны каким-то образом объяснить компилятору что именно должен делать код, который ему сейчас предстоит выдать. Тут очень кстати пришлась концепция "переменной простая, понятная и обкатанная поколениями любителей считать вручную. Для того чтобы объяснить компилятору задачу, мы натягиваем ее на каркас переменных и действий с ними, разбиваем на дерево подзадач, кодирование листьев которого задача не слишком сложная.

С ростом мощности процессоров возникла физическая возможность оптимизировать код. Машинно-зависимая оптимизация нас мало интересует ввиду упоминавшейся нелокальности изменений кода. Машинно - независимая оптимизация заключается в преобразованиях, улучшающих структуру некоторого псевдокода, например из трехадресных инструкций, полученных путем добавления новых временных переменных вдобавок к уже определенным пользователем. Источниками оптимизации могут быть устранение общих подвыражений, размножение копий, удаление недоступного кода, дублирование констант...[4]. Используя разные методики, после нескольких проходов мы получаем вполне приличный (без тени сарказма) код.

С развитием технологии появилась возможность делать процессоры сложными. Механическое наращивание числа функциональных устройств (то, что сейчас и называется VLIW) оказалось невостребованным жизнью, и процессоры стали не только сложными, но еще и умными. Оставаясь внешне теми же, они обзавелись внутренней жизнью, стали сами оптимизировать код, устраивать конвейеризацию, прокатились CISC - RISC войны - одним словом произошло все то, что мы наблюдали последнее время с все возрастающим удивлением.

Все то время пока расцветала эта полупроводниковая революция, оптимизирующие компиляторы продолжали медленно эволюционировать. Появлялись все новые шаблоны, поддержка новых архитектур, расширялась область перебора вариантов: жизнь продолжала идти медленно и неторопливо.

А тем временем все возрастал (и продолжает расти) разрыв между номинальной мощностью систем и их реальной производительностью. Поддержание внешнего интерфейса для обратной совместимости обходится все дороже. Появляющиеся технологические возможности расходуются не непосредственно на рост производительности, а большей частью на непропорциональную изощренность внутренней логики. Выхода из сложившейся ситуации не видно и можно объявить о наличии логического тупика. Можно сколько угодно повышать тактовую частоту и плотность элементов на кристалле, проблема лежит не в техно-

логической области.

Вычислительная система есть единый организм. Ее разделение на подсистемы, взаимодействующие через интерфейсы - мера, безусловно, необходимая, можно даже сказать вынужденная. Так уж сложилось, что подсистемы определялись исходя из физической организации системы и эта схема близка к тому чтобы себя исчерпать. Более перспективным представляется разделение на логические подсистемы, например, "эффективная работа с памятью"подразумевает полный контроль и взаимопроникновение компилятора, операционной системы, процессора и т.д.. Иными словами, вдобавок к существующим аналитическим методикам организации вычислительных систем пора придумывать синтетические.

5 Глобальная оптимизация кода.

В предыдущем разделе, как может показаться, существует противоречие: мы отвергаем то единственное, что должно нас волновать (машинно - зависимую оптимизацию) и пытаемся заняться тем, что и без нас работает практически идеально (машинно-независимая оптимизация). Увы, но машинно - независимый псевдокод, предоставленный существующими оптимизационными техниками, для нас непригоден. Рассмотрим существующие линейные техники распределения ресурсов:

1. Распределение регистров путем раскраски графа. Метод требует двух проходов. При первом проходе инструкции целевой машины выбираются так, как будто у нас есть бесконечное множество символических регистров, по сути, имена, используемые в промежуточном представлении, становятся именами регистров, а трехадресные инструкции - инструкциями машинного языка. [4]. При втором проходе строится граф взаимодействия регистров, узлы которого представляют собой символические регистры, а дуги соединяют два узла в том случае, если в момент определения одного из регистров, второй оказывается жив. Далее предпринимается попытка раскрасить граф k цветами, где k - фактически доступное число регистров. Хотя эта задача является NP-полной, существует как минимум одна дешевая эвристика. Итеративно удаляются вершины графа, имеющие менее k ребер до тех пор, пока мы не получим пустой граф (в противном случае раскраска считается невозможной и производится перезагрузка оставшихся вершин с числом ребер >= k). В случае VLIW процессора мы должны строить и раскрашивать граф сразу по нескольким шкалам - по шкале на каждый набор устройств. К примеру, шкала сумматоров, шкала компараторов, шкала путей загрузки из памяти ... Очевидно, мощности этих новых шкал невелики, а число их значительно, в результате может оказаться, что мы не сумеем раскрасить практически никакой граф. А при наличии зависимости между номерами регистров и устройств, данная эвристика становится абсолютно неприменимой.

2. Генерация кода на основе помеченного дерева (дага (Directed Acyclic Graph)).

Даги являются очень удобным представлением для того, чтобы определить оптимальное переупорядочение окончательное последовательности вычислений. Причем, для некоторых моделей процессоров существуют простые алгоритмы вычисления оптимального переупорядочения. Например, для машин, в которых все вычисления производятся в регистрах и инструкции которых представляют собой операторы, примененные к двум регистрам или регистру и ячейке памяти, существует следующий линейный от размера дага алгоритм: все узлы дерева метятся значением, которое фактически означает необходимое для его вычисления количество регистров. После этого, рекурсивно обходя дерево, мы создаем код, исходя из соотношения меток детей каждого узла, его собственной метки и их отношения к фактически располагаемому количеству регистров. Данный алгоритм действительно дает оптимальный с точки зрения хранения промежуточных результатов в регистрах код. В случае VLIW вклад в стоимость кода именно этой составляющей не столь велик. Куда важнее отслеживание конфликтов между функциональными устройствами. Т.е. применение этого алгоритма и его вариантов (обобщений) к VLIW машине не дает нам никаких гарантий оптимальности кода, либо требует ветвлений (перебора) в некоторых узлах, что по сути та же экспонента.

3. Динамическое программирование. Алгоритм динамического программирования разделяет задачу генерации оптимального кода для выражения на подзадачи генерации оптимального кода для подвыражений. Алгоритм состоит из трех фаз. На первой стадии мы присваиваем каждой вершине дерева вектор стоимости длиной r + 1, где r - число регистров на целевой машине. Нулевой элемент вектора мы используем для вычисления данного узла в память, 1-й есть стоимость при одном доступном регистре, 2-й - при двух .... Во второй фазе алгоритма векторы стоимости используются для определения, какие узлы должны вычисляться в память. Во время третьей фазы происходит собственно генерация кода. Данный алгоритм нетрудно обобщить к VLIW архитектуре, но, к сожалению, в данном случае не верна исходная посылка об оптимальности кода выражения, полученного из оптимального кода подвыражений.

Исходя из вышесказанного, можно заключить, что линейный алгоритм для распределения ресурсов для сколько-нибудь практически интересной VLIW архитектуры на данный момент неизвестен. И, следовательно, размер дерева псевдокода должен быть как можно меньше, для чего можно имеет смысл в его узлах допускать куски локально-оптимального кода.

Теперь немного об оптимизации как таковой. Глобальная оптимизация дискретной функции при отсутствии априорной информации решается либо перебором, либо оптимизацией непрерывной функции, натянутой на данный дискретный каркас. Второй путь для нас мало-перспективен т.к. наличие недифференцируемых мест (разрывов, изломов) сильно затрудняет процесс оптимизации, а наша функция, похоже, будет состоять из одних недифференцируемостей плюс сильный шум - наличие огромного количества бесполезных локальных минимумов. По той же причине не помогут здесь и методики наподобие "simulated

annealing". С другой стороны, поиск оптимума по самой замечательно - оптимизируемой (O(1)) функции ничего не стоит превратить в задачу, решаемую только перебором. Достаточно просто перемешать входные (или выходные) значения некоторым однозначным, но внешне непредсказуемым образом. Отметим, что от наличия "перемешивателя"задача не перестала быть принципиально оптимизируемой за O(1), она лишь потеряла это качество для нас. В связи с этим, первое, что мы попробуем сделать, начиная решать фактически переборную задачу, это постараемся очистить ее от всего органически ей не присущего.

Еще один момент. Если нам по каким-то (пусть даже самым что ни на есть объективным) причинам не удается существенным образом упростить оптимизацию (т.е. найти полиномиальный алгоритм), мы можем предположить что для любой степени полинома N существует подкласс задачи, оптимизируемый за время O(n**N). Во всяком случае, всегда существует тривиальный подкласс таблиц ответов O(1). Вопрос в балансе времени/памяти и той цене, которую мы готовы заплатить за нужную степень, а также в том, насколько качественные результаты дает порождающая эвристика. Повторимся, но отметим, что никакая эвристика не поможет в ситуации, когда между нами и задачей вклинивается "перемешиваюший"посредник.

Теперь давайте попробуем поставить задачу. Вот, к примеру, классическая постановка словами Н.Ющенко[1]:

Существует некоторая задача пользователя Task. Для решения этой задачи на компьютере, надо предложить алгоритм ее решения. Для одной задачи, вообще говоря, может существовать сколько угодно алгоритмов.
Хотя имеются попытки определить понятие "алгоритм"формально, в данном случае это приведет к потере общности. Поэтому вместо принятия формализма "алгоритм"мы примем формализм "способ записи алгоритма". Способ записи алгоритма - это такое представление алгоритма, в котором процесс выполнения алгоритма становится определенным с математической точностью. Например, язык программирования является способом записи алгоритма.
Таки образом:

- задаче Task соответствует множество алгоритмов: Alg1, ..., AlgN, ...
- имеется множество способов записи алгоритмов: Lang1, ... LangM, ...
- т.е. конкретное решение задачи Task, которое может выписать пользователь, можно обозначить как Prog (AlgI, LangJ)

Естественно, конкретный способ записи LangJ может не позволить записать все AlgI. Например, если способ записи предполагает терминологию графов, то те Alg, которые выражены иначе, не могут быть записаны. Но разделить способ записи алгоритма и сам алгоритм надо, чтобы формализовать понятие трансляции.
Трансляция Comp - это отображение алгоритма, записанного при помощи способа записи LangI, на конкретную машинную программу. Именно для результата трансляции определено время его выполнения в тактах.
Итак, какие же получаются степени свободы, по которым можно пытаться "оптимизировать"преобразование Task -> Code ?

Code = Code (Alg, Lang, Comp).

Т.е. можно, независимо друг от друга:

- менять алгоритм
- менять язык, на котором записан один и тот же алгоритм,
- менять алгоритм трансляции конкретного языка.

С чем здесь хочется не согласиться? Действительно, понятие алгоритма не формализуемо. Однако, существование любого из (Alg1, ..., Algn), уже подразумевает некоторый его способ записи. Если я могу представить алгоритм, то

обязательно существует некоторая виртуальная машина, к которой я его мысленно применяю. Это может быть подобие процессора с бесконечным количеством регистров или стековый процессор, у кого-то (вдруг) даже машина Тьюринга - в любом случае это последовательная виртуальная машина. Тут ничего не поделаешь, при всем параллелизме организации нашего мозга, мыслим мы преимущественно последовательно (что само по себе и не плохо). Если не предпринимать специальных усилий, то (одномерный) процесс сериализации на (двумерную) бумагу в общем случае трехмерного образа алгоритма неизбежно приведет к искажению первоначального замысла, что для нас весьма нежелательно. Следовательно, должен существовать способ записи алгоритмов без потери внутреннего параллелизма и очевидным кандидатом на это (с некоторыми изменениями) является концепция базовых блоков (basic blocks).

Традиционно базовым блоком называется "последовательность смежных инструкций, в которые поток управления входит в их начале и покидает в конце, без останова программы или возможности ветвления (за исключением конца блока)"[4].

Базовые блоки используются как элементы построения графов потоков управления (Control Flow Graph). В задачу анализа потока управления (control flow analysis) входит определение свойств передачи управления между операторами программы. Проверка многих свойств этого вида необходима для решения задач оптимизации, преобразований программ и т.д. Граф потоков управления - ориентированный граф с двумя выделенными вершинами __input__ и __output__, такими, что

- в __input__ не заходит ни одна дуга
- из __output__ не выходит ни одна дуга
- произвольная вершина принадлежит хотя бы одному пути из __input__ в __output__

Под анализом потоков данных понимают совокупность задач, нацеленных на выяснение некоторых глобальных свойств программы, то есть извлечение информации о поведении тех или иных конструкций в некотором контексте. Такая постановка задачи возможна по той причине, что язык программирования и вычислительная среда определяют некоторую общую, "безопасную"семантику конструкций, которая годится "на все случаи жизни". Учет же контекстных условий позволяет делать более конкретные, частные заключения о поведении той или иной конструкции; при этом такие заключения, вообще говоря, перестают быть верными в другом контексте. Например, общая семантика присваивания заключается в вычислении выражения, стоящего в правой части, и присваивании полученного значения в переменную, стоящую в левой части. Однако в случае, когда выражение в правой части не имеет побочных эффектов, а переменная в левой части более нигде не используется, данный оператор становится эквивалентен пустому [3].

Мы же сделаем попытку объединить вышеупомянутые потоки и "синтезировать"единое видение проблемы. Попробуем расширить определение базового

блока следующим образом: каждый базовый блок за исключением тривиальных входного и выходного псевдоблоков, могут иметь не менее одного входа и произвольное число выходов, причем некоторые из последних могут ветвиться. В данный момент нам совершенно неинтересно, что находится внутри блоков, важно лишь что:

- мы считаем код блоков оптимальным
- мы считаем код блоков корректным
- мы можем попасть только в начало блока
- к моменту любого выхода из блока все результирующие значения готовы.

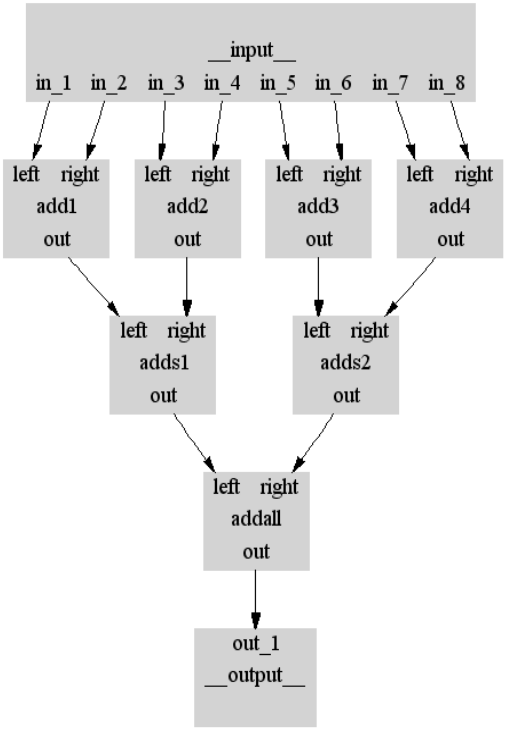
"Рассмотрим теперь отображение DU, которое каждому выходу сопоставляет некоторое подмножество множества входов. Тогда для каждого оператора"(у нас, но не в оригинале - базового блока w) "определяются множества def(w) и use(w), которые есть соответственно множества его выходов и множество тех выходов других операторов, образы которых при отображении DU содержат хотя бы один вход оператора w."[3].

Такое представление позволяет отследить потоки данных в программе без учета того, как именно эти данные преобразуются. Множества входов и выходов соответствуют интуитивному представлению об аргументах и результатах некоторых операторов, а отображение DU просто описывает, как используются выработанные операторами результаты.

Дуги, соединяющие блок (элементы DU) теперь передают управление одновременно с данными. Ветвление выхода блока означает передачу данных и управления по одному из возможных направлений. Очевидная аналогия здесь - сеть труб между некоторыми устройствами, тогда ветвление выхода - переключатель.

Кроме того, мы разреши тупиковые блоки, т.е. такие, которые можно достичь из __input__, но не имеющие выходов. Такой код не считается мертвым, но, очевидно, конец такого блока должен обрабатываться специальным образом.

В качестве примера рассмотрим задачу суммирования восьми чисел "пирамидкой". Входной псевдоблок __input__ содержит 8 выходов - числа, которые надо просуммировать. Выходной псевдоблок __output__ обладает одним входом - суммой. Семь блоков с разными названиями, но одним содержимым, каждый из которых имеет два входа (left и right), а также один выход - их сумму, связаны между собой таким образом, что образуют перевернутую пирамиду. Отметим, что полученный нами в результате код будет реализовывать именно эту схему и это является с одной стороны минусом т.к. в некоторых условиях эта схема может оказаться не самой эффективной, с другой стороны это явный плюс т.к. компилятор делает строго то, что ему сказали, т.е. предсказуем. А выбор наиболее подходящей схемы как мы попытаемся показать дальше проще решать другими методами.



Что дает нам такое представление алгоритма?

- оно интуитивно понятно
- оно полно, т.е. любой алгоритм может быть определен таким образом
- оно свободно от проблем лексического и синтаксического анализов
- обладает естественным параллелизмом - любые два независимых (с небольшими исключениями таких, что не существует пути из любого выхода любого из них к входу второго) блока считаются потенциально параллельно исполняющимися
- она наглядна - программист буквально видит граф своего алгоритма, и это невольно заставляет его искать более "красивые"решения, то, что раньше было определено неявно посредством языка программирования и построение чего стоило компилятору определенных трудов, теперь предоставляется непосредственно разработчиком алгоритма и, думается, по крайней мере, не менее эффективно
- запись алгоритма может быть легко проверена на корректность
- к ней применимы существующие алгоритмы машинно - независимой оптимизации, например, упрощение циклов

Каким образом мы собираемся создавать код для такого представления алгоритмов? Как мы уже видели, существующие алгоритмы распределения регистров действуют на очень ограниченных моделях целевой архитектуры. На прочих архитектурах нет никаких гарантий оптимальности кода. И если для суперскалярных архитектур такое приближение дает более-менее приемлемые результаты (тем более за линейное время), в случае VLIW такой метод не работает преимущественно потому, что стоимость большинства инструкций мы узнаем лишь в момент записи, у нас нет возможности тем или иным методом разметить дерево и затем по этой разметке выдать код - все должно быть сделано за один проход.

Итак, начнем. На основании имеющегося графа алгоритма G построим производный от него граф вариантов GX. Граф GX также как и исходный G - ориентированный граф с двумя выделенными вершинами __input__ и __output__, такими, что

- в __input__ не заходит ни одна дуга
- из __output__ не выходит ни одна дуга

Имем:

- blocks[...]- множество блоков описания алгоритма
- def[blocks[I], ...]- множество выходов блока I
- use[blocks[I], ...]- множество входов блока I
- DU[def[blocks[I], II], def[blocks[J],JJ]] - матрица связности описания алгоритма

```
1.  procedure explore_vertex ( текущая вершина , контекст - список
        всех готовых входов )
2.  begin
3.  while def[текущая вершина, I] != null
4.     найти блок, связанный с данным через DU
5.     известить этот блок о том что готов соответствующий вход,
         пометить вход в контексте
6.     if этот корреспондирующий блок имеет все входы готовыми
7.        добавить в GX вершину - этот блок;
8.        добавить в GX ребро - текущая вершина -> данный блок;
9.        call explore_vertex (корреспондирующий блок, контекст);
10. while существует блок со всеми готовыми входами
11.    добавить в GX вершину - этот блок;
12.    добавить в GX ребро - текущая вершина -> данный блок;
13.    Call explore_vertex (найденный блок, контекст );
14. end ;
15. добавить в GX вершину __input__;
16. call explore_vertex (__input__, пустой контекст);
```

Фактически, мы, каждый раз добавляя вершину (блок) в граф GX, добавляем к ней рекурсивно все готовые в данный момент блоки. Число вершин в GX экспоненциально от размера исходного графа G. В таком графе любой путь от __input__ к __output__ представляет собой некоторый вариант генерации кода, но наша задача заключается в нахождении кратчайшего пути, что легко делается жадными алгоритмами, например, алгоритмом Дейкстры. Самой главной проблемой является выбор весов ребер графа GX - в простейшем варианте при весе ребра равном изменению текущей длины кода в тиках, поиск кратчайшего пути превратится в простой перебор большей части вариантов т.к. промежуточные длины путей распределены очень скученно, даже наличие ряда грубых конфликтов инструкций не приводит к выбросу варианта за пределы рассмотрения. При удачном же выборе системы весов можно дойти до финиша, перебрав лишь малую часть возможных вариантов. Предлагается к рассмотрению следующий вариант:

1. вес ребра (т.е. стоимость добавления кода некоторого блока к уже имеющемуся) определяется динамически - непосредственно в тот момент, когда мы пытаемся это сделать

2. длина пути не аддитивна, а мультипликативна - мы не складываем, а перемножаем длины ребер впрочем, если мы по-прежнему считаем, что реализуем алгоритм Дейкстры, длина ребра определяется умножением текущей длины пути на некоторую штрафную величину кодирования нового блока

Стоит упомянуть еще об одной проблеме, которая, к счастью, имеет простое решение. При поиске кратчайшего пути в GX мы можем столкнуться с ситуацией выбора между совершенно одинаковыми продолжениями. Например, в вышеупомянутом суммировании в самом начале мы имеем возможность закодировать любой из блоков add1, add2, add3, add4 - ввиду симметрии графа эти варианты абсолютно равноценны. Аналогично равноценны adds1 и adds2. Устранив тривиальные ветвления, мы могли бы в худшем случае перебрать грубо в 4*3*2*2 раз меньше вариантов. Решение представляется достаточно простым - каждой вершине графа алгоритма мы приписываем некоторую метрику, получаемую из описания путей, достижимых из данной вершины любым устойчивым образом (здесь мы имеем в виду устойчивость метрики вершины к изоморфным преобразованиям графа G). Тогда, исследуя соседей любой вершины, мы станем реализовывать только те дуги, которые ведут к вершинам с разными метриками.

До сих пор мы принимали базовые блоки как данные. Нас не интересовало их содержимое, мы лишь предполагали, что оно корректно и в некотором смысле оптимально. Но раз уж мы упомянули, что собираемся начислять штрафы за кодирование блоков, пора вплотную заняться их содержимым. Очевидно, блоки состоят из инструкций. Не менее очевидно, что эти инструкции не могут быть статичными т.к. на все случаи жизни блоков не напасешься, кроме того, никто не сможет заставить программиста оперировать миллионами оных. Стало быть, содержимое блока есть шаблон кода с рядом параметров. Параметры могут быть, например, использованы для задания конкретных номеров регистров. Таким образом, работа компилятора заключалась бы в стыковке блоков через

параметры. Данный вариант достаточно прост, но не слишком гибок, поэтому чтобы добавить разнообразия, мы разделим понятие блока на два

1. тип блока - в вышеприведенном примере сложения все блоки были одного типа, определение типа обязано включать
   - имя, используя именно эти имена, программист будет составлять граф алгоритма
   - список входов блока - просто имена для определения DU
   - список выходов блока - аналогично, просто имена для определения DU, некоторые из выходов могут ветвиться
2. имплементация блока - вариант кодирования, содержит
   - имя типа блока
   - список параметров (переменных), которые должен определить компилятор перед кодированием данной имплементации
   - список входов с теми же именами что и входы типа, но с определением возможно параметризованного места, где ожидается найти данные для этого входа. Например, если мы определили параметр x и имеем пул регистров A, таким местом может быть A[x]
   - аналогичный список выходов с именами из описания типа и определением мест, где окажутся результаты выполнения кода имплементации.
   - собственно код - список инструкций с возможно параметризованными аргументами. Все инструкции считаются последовательными и распараллеливаются компилятором, если существует такая возможность.

Иными словами, определяя граф алгоритма, программист оперирует типами блоков, тогда как компилятор при кодогенерации пытается подставить все имеющиеся имплементации блоков данного типа. К примеру, блок деления двух величин может иметь следующие имплементации (шутка):

- аппаратное деление (если есть)
- аппроксимация по Ньютону-Раффсону
- логарифмирование/вычитание/экспонирование

Отметим, что граф GX теперь требует переопределения. В обоих местах (строки 7 и 11) вместо

```
добавить в GX вершину - этот блок;
добавить в GX ребро - текущая вершина -> данный блок;
Call explore_vertex (найденный блок, контекст );
```

следует использовать

```
while имплементация блока != null
    добавить в GX вершину - эту имплементацию;
    добавить в GX ребро - текущая вершина -> данная имплементация;
    call explore_vertex (найденная имплементация, контекст );
```

Получившийся граф будем называть GXX.

6 Пробная реализация.

В начале нам потребуется создать по возможности универсальное описание архитектуры целевой машины.

Сначала определим места, где могут находиться данные. Это могут быть:

- регистры
- блок (и) памяти, например, если память расщеплена
- устройства процессора - всевозможные сумматоры, умножители, тракты памяти ...
- специальная память, например порты периферии
- кэш-память всех уровней

Отметим, что такие места могут быть как скалярными (порты, устройства процессора) так и векторными (память).

```
pool {
    comment="Unit_L1 Первый шлюз памяти ";
    name="L1";
    range={0-0};
}; pool {
    comment="Unit_L2 Второй шлюз памяти ";
    name="L2";
    range={0-0};
}; pool {
    comment="Unit_S1 Первый сумматор ";
    name="S1";
    range={0-0};
}; pool {
    comment="Unit_S2 Второй сумматор ";
    name="S2";
    range={0-0};
}; pool {
    comment="Register_pool_A";
    name="A";
    range={0-15};
}; pool {
    comment="Register_pool_B";
    name="B";
```

```
    range={0-15};
}; pool {
    comment="Memory_pool";
    name="MEM";
    range={0-100000000};
};
```

Аттрибуты тега pool очевидны: name - идентификатор, comment - комментарий, range_from...range_to - границы допустимого интервала вектора, если равны - то скаляр

Теперь наша задача - описать атомарные операции процессора.

```
cop {   name="ADD";
    comment="ADD через S1";
    time=7;
    inargs={    A = "1111111"};
    outargs={A = "1111111"};
    units={S1[0] = "1111111"    };
}; cop {   name="ADD";
    comment="ADD через S2";
    time=7;
    inargs={    B = "1111111"};
    outargs={B = "1111111"};
    units={S2[0] = "1111111"};
}; cop {   name="ADD";
    comment="перекрестный ADD через S1";
    time=9;
    inargs={    B = "111111111" };
    outargs={A = "111111111"};
    units={S1[0] = "111111111"};
}; cop {   name="ADD";
    comment="перекрестный ADD через S2";
    time=9;
    inargs={    A = "111111111" };
    outargs={B = "111111111"};
    units={S2[0] = "111111111"};
}; cop {   name="LOAD";
    comment="загрузка через L1";
    time=9;
    inargs={    MEM = "111111111"};
    outargs={A =   "111111111"};
    units={L1[0] =  "111111111"};
}; cop {   name="LOAD";
    comment="загрузка через L2";
    time=9;
    inargs={    MEM = "111111111"};
    outargs={B =   "111111111"};
    units={L2[0] =  "111111111"};
}; cop {   name="STORE";
    comment="выгрузка через L1";
```

```
    time=9;
    inargs={    A = "111111111" };
    outargs={MEM =   "111111111"};
    units={L1[0] =  "111111111"};
}; cop {   name="STORE";
    comment="выгрузка через L2";
    time=9;
    inargs={    B = "111111111"};
    outargs={MEM =   "111111111"};
    units={L2[0] =  "111111111"};
}; cop {   name="MOV";
    comment="копирование регистра в пуле A";
    time=1;
    inargs={    A = "1"};
    outargs={A = "1"    };
}; cop {   name="MOV";
    comment="копирование регистра в пуле B";
    time=1;
    inargs={    B = "1"};
    outargs={B = "1"    };
}; cop {   name="MOV";
    comment="копирование регистра из A в B";
    time=2;
    inargs={    A = "11"};
    outargs={B = "11"};
}; cop {   name="MOV";
    comment="копирование регистра из B в A";
    time=2;
    inargs={    B = "11"    };
    outargs={A = "11"};
};
```

Атрибуты тега cop: name - идентификатор, comment - комментарий, time - формальное время в тактах, требуемое для выполнения. Теги inargs, outargs, units описывают аргумент инструкции, различия заключаются в том, что аргумент, описанный, как unit не требует упоминания в коде, а outargs в отличие от inargs оставляет ячейку занятой и она не может быть использована компилятором в текущем блоке без явного упоминания. Описание аргумента содержит пару имя=значение, где имя соответствует имени тега pool, значение - строка из 0 и 1 длиной в time, где 1 обозначает, что данное устройство в данный такт занято (пока для простоты будем считать, что все устройства заняты все время выполнения инструкции).

Теперь самое интересное - собственно базовые блоки:

типы

```
block {
    comment="plus";
    name="plus";
    input  {left, right};
    output {out};
```

```
}; block {
    comment="bind";
    name="bind";
    input  {in};
    output {out};
};
```

и имплементации:

```
implementation {
    comment="plus";
    name="plus";
    var=y,x,z;
    input  {    left=A(x);
        right=A(y);};
    output {    out=A(z);};
    code {  ADD     A(x)    A(y);
        MOV     A(y)    A(z);};
}; implementation {
    comment="plus";
    name="plus";
    var=y,x,z;
    input  {    left=B(x);
        right=B(y);};
    output {    out=B(z);};
    code {  ADD     B(x)    B(y);
        MOV     B(y)    B(z);};
}; implementation {
    comment="bind";
    name="bind";
    var=y,x;
    input  {    in=A(x);};
    output {    out=A(y);};
    code {  MOV  A(x) A(y); };
};

implementation {
    comment="bind";
    name="bind";
    var=y,x;
    input  {    in=B(x);};
    output {    out=B(y);};
    code {  MOV  B(x) B(y);};
}; implementation {
    comment="bind";
    name="bind";
    var=y,x;
    input  {    in=A(x);};
    output {    out=B(y);};
    code {  MOV  A(x) B(y);};
```

```
}; implementation {
    comment="bind";
    name="bind";
    var=y,x;
    input  {    in=B(x);    };
    output {    out=A(y);};
    code {  MOV  B(x) A(y); };
}; implementation {
    comment="bind";
    name="bind";
    var=y,x;
    input  {    in=MEM(x);};
    output {    out=A(y);};
    code {  LOAD  MEM(x) A(y);};
}; implementation {
    comment="bind";
    name="bind";
    var=y,x;
    input  {    in=MEM(x);};
    output {    out=B(y);};
    code {  LOAD  MEM(x) B(y);};
}; implementation {
    comment="bind";
    name="bind";
    var=y,x;
    input  {    in=B(x);};
    output {    out=MEM(y);};
    code {  STORE  B(x) MEM(y);};
}; implementation {
    comment="bind";
    name="bind";
    var=y,x;
    input  {    in=A(x);};
    output {    out=MEM(y);};
    code {  STORE  A(x) MEM(y);};
};
```

Атрибуты name и comment означают то же, что и раньше. Обратим внимание, что атрибут name больше не является уникальным, благодаря чему и появляется возможность ветвления. Блок внешне представляет собой черный ящик с набором входов и выходов. И входы (тег input), и выходы (тег output) являются ячейками данных, отсюда появляется потребность в наличии внутренних параметров. Параметр (тег var) является индексом конкретной ячейки в том или ином пуле и может соответствовать не только входам/выходам, но также использоваться для определения внутренней жизни блока. К моменту имплементации блока все параметры для входов должны быть определены. Однако, если определены параметры кроме входных, это является дополнительным местом ветвления.

В нашей простой задаче используется единственный блок - сумматор, прав-

да, существует ряд блоков с предопределенным именем (например "BIND"), предназначенных для пересылки данных с выхода одного блока на вход другого, а также "JUMP", возникающий, когда возникает потребность слить ветки коде, образовавшиеся после ветвления выхода блока (в данном примере отсутствуют):

Наш случай - простейший, но даже в нем много вариантов пересылок. В действительности их должно быть намного больше - в том числе двухходовые и т.д. Количество вариантов не скажется существенно на скорости поиска т.к. в первую очередь будут учитываться короткие варианты. Длинные - же варианты пойдут в ход только в случае вынужденного простоя ресурсов, если по какой - либо причине такой путь окажется более дешевым. Особенно стоит отметить случай неоднородной памяти. Мы уже говорили о том, что иногда процессоры имеют расщепленную память. Иногда пулы памяти имеют разные времена доступа из-за их физической организации - ПЗУ, FLASH, ..., иногда, позволяется присвоить адреса кэш-памяти - все эти ситуации мы можем с легкостью описать в наших терминах. Кеш - память в случае VLIW процессора заслуживает отдельных слов, но об этом позже.

Теперь время описать собственно алгоритм.

```
item {  name="add1";    block=plus;};
item {  name="add2";    block=plus;};
item {  name="add3";    block=plus;};
item {  name="add4";    block=plus;};
item {  name="adds1";   block=plus;};
item {  name="adds2";   block=plus;};
item {  name="addall";  block=plus;};
```

Имеется семь блоков: первые четыре попарно суммируют наши восемь чисел, остальные - промежуточные результаты. Отметим, что атрибут name опять является уникальным идентификатором, а block ссылается на групповой идентификатор бока.

Теперь очередь за межблочными связями.

```
link {  from=in_1;  to=add1(left);};
link {  from=in_2;  to=add1(right);};
link {  from=in_3;  to=add2(left);};
link {  from=in_4;  to=add2(right);};
link {  from=add1(out); to=adds1(left);};
link {  from=add2(out); to=adds1(right);};
link {  from=adds1(out);    to=addall(left);};
link {  from=in_5;  to=add3(left);};
link {  from=in_6;  to=add3(right);};
link {  from=in_7;  to=add4(left);};
link {  from=in_8;  to=add4(right);};
link {  from=add3(out); to=adds2(left);};
link {  from=add4(out); to=adds2(right);};
link {  from=adds2(out);    to=addall(right);};
link {  from=addall(out);   to=out_1;};
```

где тэг from ссылается на конкретный выход определенного типа блока, а тэг to - на конкретный вход определенного типа блока. Некоторые из этих связей ссылаются на входы алгоритма:

```
input  {
    in_1=MEM(0);
    in_2=MEM(1);
    in_3=MEM(2);
    in_4=MEM(3);
    in_5=MEM(4);
    in_6=MEM(5);
    in_7=MEM(6);
    in_8=MEM(7);
};
```

и выходы:

```
output  {
    out_1=MEM(1);
};
```

Результат применения алгоритма Дейкстры к графу GXX для нашей задачи приводит к следующему результату [2] :

[2] http://www.geocities.com/creta_ru/samples/test.html

N такта	Инструкция	Аргумент1	Аргумент2
0	LOAD	MEM[6]	A[0]
0	LOAD	MEM[0]	B[0]
9	LOAD	MEM[1]	B[1]
9	LOAD	MEM[7]	A[1]
18	ADD	B[0]	B[1]
18	ADD	A[0]	A[1]
18	LOAD	MEM[4]	B[3]
18	LOAD	MEM[2]	A[3]
25	MOV	A[1]	A[2]
25	MOV	B[1]	B[2]
26	MOV	A[2]	B[5]
26	MOV	B[2]	A[5]
27	LOAD	MEM[5]	B[1]
27	LOAD	MEM[3]	A[1]
36	ADD	A[3]	A[1]
36	ADD	B[3]	B[1]
43	MOV	B[1]	B[4]
43	MOV	A[1]	A[4]
44	ADD	B[4]	B[5]
44	ADD	A[5]	A[4]
51	MOV	A[4]	A[1]
51	MOV	B[5]	B[1]
52	MOV	B[1]	A[4]
54	ADD	A[1]	A[4]
61	MOV	A[4]	A[2]
62	STORE	A[2]	MEM[1]

Без избежания идентичных ветвлений и применения эвристик было создано 7615 промежуточных путей.

7 Выводы

Какие выводы можно сделать из этого примера?

- мы получили код, который вычисляет то, что мы хотели.
- этот код действительно самый быстрый в контексте поставленной задачи
- по абсолютному значению этот код не самый быстрый, в данной ситуации мы потеряли на второй инструкции блока SUM - можно было бы обойтись и без нее, но раз уж так определен блок - ничего не поделаешь
- очень много тривиальных ветвлений, впрочем, с этим можно бороться, например, введением сигнатур состояний или объединить параллельные устройства в пулы: вместо L1 & L2 иметь пул L[0...1] тем самым, сократив число декларируемых типов блоков и, следовательно, ветвлений

- потенциально большей проблемой являются приблизительно-равнозначные ветвления, например, наличие большего колличества примерно равных по потребляемым ресурсам подвыражений приведет к интенсивному перебору с очень малой эффективностью ускорения. Данную проблему можно решать двумя путями - вычислением эвристических весов подвыражений и рекоммендацией конечному программисту избегать таких конструкций, например, погружая их в "суперблоки"

- наш компилятор не привязан к конкретной архитектуре, его можно внешне настроить (хотелось бы верить) на любую из них.

- пользователь может самостоятельно дополнять и расширять библиотеку блоков и, даже, делать тонкую настройку на свою систему (вспомним здесь про расщепленную память).

- нигде не говорится, что можно создавать код только для VLIW

- можно генерировать код для конгломерата из нескольких процессоров при условии их жесткой синхронизации. Ключевое слово здесь - предсказуемость, мы должны быть уверены, что все события происходят именно тогда, когда мы этого от них ждем

- вопрос, что делать в случае нехватки ресурсов, по-прежнему остается частично открытым, естественный путь - просто отрезать такие веточки в надежде, что найдется путь, где имеющегося количества недостающих ресурсов, скажем, регистров, хватит для реализации схемы оставляет много вопросов. Такой подход представляется стратегически правильным, хотя и является объектом для размышлений.

- в этом смысле кэш - память для нас по меньшей мере бесполезна, никакого ускорения мы не получим т.к. всегда будем ждать худшего случая. Многоуровневая кэш-память в ее нынешнем виде вообще представляется воплощением бессилия и неспособности распорядиться имеющимися ресурсами. Трудно себе представить алгоритм, которому действительно нужны сотни килобайтов быстрой памяти. Для реализации алгоритма, который может быть написан и отлажен человеком, думается, достаточно нескольких десятков слов. Если же требуется постоянно обрабатывать значительный по размеру массив, гораздо эффективнее организовать параллельный транспорт данных. Поэтому в нашем случае более эффективно иметь небольшую, но очень быструю адресуемую память, которая может быть и многоуровневой, т.е. существует несколько пулов памяти каждый со своим временем доступа. Какие нужны предпосылки, чтобы данная схема заработала?

    1. Нужен ряд независимых устройств, способных независимо работать с данными. Во всей этой затее нет никакого смысла, если вся работа по пересылке будет осуществляться теми же L1 & L2. Процессор может заранее и не знать будущей конфигурации быстрой памяти. Не

знает этого заранее и компилятор, впрочем, мы имеем возможность сделать ту самую его "тонкую настройку"на конкретные условия. Но в целом необходим тройственный союз между компилятором, процессором и контроллером памяти (периферией). Процессор должен быть в состоянии различать пересылки данных и отдавать их различным устройствам, часть из которых может находиться вне самого процессора. Также придется разрешить пересылки из памяти в память.

2. Компилятор должен быть абсолютно уверен, что данные, помещенные им в быструю память, никуда оттуда не денутся. Иными словами, при переключении контекста на другую задачу, данные из быстрой памяти должны сохраняться с последующим восстановлением при переключении обратно на этот поток. Возможны варианты со страничной организацией быстрой памяти и спасением только в случае отсутствия свободных страниц, спасение/восстановление можно осуществлять параллельно и это в любом случае необходимо т.к., повторимся, компилятор должен быть уверен в данных.

- количество вычислений, необходимых для генерации кода представляется большим, но не чрезмерно

- количество вычислений можно ограничить, если строить большие алгоритмы из маленьких подалгоритмиков, используя их в качестве своеобразных блоков. Иными словами, написав и скомпилировав некоторую схему, можно запомнить несколько наиболее удачных вариантов и сохранить в параметризованном виде в библиотеке блоков/имплементаций в качества блока высокого уровня, таким образом можно строить иерархически сложные алгоритмы, существенно ограничив мощность перебора. Отметим, что такие суперблоки при отсутствии внутренних переходов попрежнему могут перекрываться и выполняться параллельно.

- естественно реализуется техника "расщепления регистров когда один регистр (любое устройство процессора) может быть использован одновременно в нескольких инструкциях, если, например, число после умножения должно быть помещено в регистр, но произойдет это лишь через несколько тактов после начала инструкции есть соблазн использовать простаивающий регистр где-то еще. Для этой цели у нас предназначался атрибут mask тега cop, но мы не демонстрировали эту возможность ввиду трудностей восприятия такого кода.

- в случае использования этой техники обычная кэш-память не просто бесполезна, она смертельна для компилятора т.к. никто не имеет права вернуть результат быстрее, чем его об этом попросили, а вдруг его место еще занято?

- интересный момент - переходы. Для их реализации придется ввести синхронизацию процессора - место, где возможно прерывание без потери дан-

ных. В этом месте должно отсутствовать перекрытие инструкций и компилятор начинает работу "с чистого листа".

8 Что дальше

Реализованный на данный момент опытный компилятор способен кодировать статические выражения равно как и простые условные ветвления. При этом поиск кратчайшего пути происходит без применения каких бы то ни было эвристических отсечений простым перебором. Первоочередными задачами являются:

- отладка работы с циклами т.к.в частности, эффективная укладка коротких циклов это то, для чего изначально все это задумывалось
- отсечение тривиальных ветвлений
- отладка технологии применения эвристик отсечения вариантов при поиске кратчайшего пути
- обкатка различных эвристик на разных типах задач
- создание минимально-визуальной среды для составления графов алгоритмов

Не исключено что к моменту, когда Вы читаете эти слова что-то из этого уже реализовано, при этом каждый, добравшийся до этого места может чувствовать себя вправе обратиться к авторам и получить исчерпывающие разъяснения по любому связанному с темой вопросу.

9 Благодарности

Искренней признательности заслуживает Василий Туран как за помощь в программной реализации этих идей и бескомпромисную борьбу с моими и своими ошибками так и за проявленное в процессе работы отменное чувство здравого смысла.

Нельзя не упомянуть с благодарностью и Никиту Ющенко за долготерпение и непоколебимый скептицизм, столь способствовавших ускоренному избавлению от чрезмерного оптимизма и наивных иллюзий.

А если Вы читаете английский вариант этого текста, то это целиком является заслугой Андрея Тананакина, сберегшего немало столь драгоценного времени и непоседевших волос.

Список литературы

[1] Никита Ющенко (Nikita V. Youshchenko, youshcs.msu.su), личные контакты

[2] Jean Noel and Charles Trullemans "MOMO : Scheduling Method for Large Data Flow Graph". [www.dice.ucl.ac.be/ anmarie/patmos/papers/S2/2_3.pdf]

[3] Терехов А.А. и др. "Разработка компиляторов для .NET" [se.math.spbu.ru/courses/dotNET Compiler Engineering/Lectures/11 Optimization.doc], [research.microsoft.com/programs/europe/Curriculum/Compiler Engineering/Lectures/11. Optimization.doc]

[4] А. Ахо, Р. Сети, Дж. Ульман "Компиляторы: принципы, технологии и инструменты", М.: "Вильямс", 2001.

[5] В.В.Корнеев А.В.Киселев "Современные микропроцессоры"М.:"Нолидж", 1998