Использование особенностей ЦПОС в компиляторе языка "C"

В.В. Рубанов, А.И. Гриневич, Д.А. Марковцев

Аннотация. Данный обзор содержит описание характерных особенностей ЦПОС и связанных с ними оптимизаций, которые могут быть реализованы в компиляторе языка '"C". Рассматриваются как сами алгоритмы оптимизаций, так и взаимное влияние различных оптимизаций друг на друга.

1. Введение

Целью данной работы является аналитический обзор методов построения оптимизирующих компиляторов для цифровых процессоров обработки сигнала (ЦПОС). Рассматриваются основные особенности ЦПОС, оказывающие влияние на задачу генерации эффективного кода. При этом мы попытались провести сравнительный анализ имеющихся решений и проанализировать перспективы создания новых оптимизирующих преобразований. Материал статьи базируется на изучении публикаций об оптимизациях в компиляторах для рассматриваемого класса процессоров, а также на опыте авторов, полученном при реализации компиляторов для двух ЦПОС.

В последние годы за счет развития технологий и удешевления производства появляется огромное количество типов специализированных процессоров и их модификаций. К таким узкоспециализированным процессорам, в частности, относят и ЦПОС. При этом существует огромное количество готовых прикладных программ на языке C, что делает построение эффективных компиляторов для специализированных процессоров важной задачей, существенно влияющей на качество и время выхода на рынок новых чипов и решений на их основе. Появление утвержденного ISO стандарта расширений языка C для ЦПОС (Embedded C) подчеркивает актуальность задачи построения эффективных компиляторов для данного класса процессоров.

Особо отметим один из источников базового материала для нашей статьи – работу [20], содержащую обобщённый обзор наиболее актуального материала на 2000 год. Наша работа дополняет эти сведения описанием алгоритмов частичного дублирования данных, программной конвейеризации и рассмотрением новых подходов к задаче сопряжения фаз генерации кода.

Статья состоит из введения, четырех глав и заключения. В первой главе приводится описание ключевых особенностей ЦПОС, которые нужно

учитывать при построении компилятора. Во второй главе мы рассматриваем обобщённую схему работы компилятора. Третья глава содержит описание характерных алгоритмов оптимизаций, учитывающих приведённые в первой главе особенности ЦПОС. Четвертая глава рассматривает общие вопросы построения back-end'а компилятора и методы совмещения фаз. В заключении делаются выводы и описываются перспективы продолжения работы.

2. Рассматриваемые особенности ЦПОС

В данной главе приводится краткая сводка характерных особенностей ЦПОС, которые необходимо учитывать для эффективной генерации кода компилятором.

2.1. Расширенная гарвардская архитектура

Процессоры общего назначения обычно ориентированы на быстрое выполнение инструкций типа регистр-регистр и имеют ограниченную пропускную способность памяти. Однако большинство алгоритмов ЦПОС требуют как раз интенсивного использования операций с памятью, поэтому для эффективной поддержки цифровой обработки сигналов память системы должна иметь высокую пропускную способность. ЦПОС достигают требуемой пропускной способности памяти за счет использования нескольких устройств памяти с независимыми шинами доступа; например, некоторые ЦПОС могут осуществлять до шести выборок из памяти параллельно (см. [1]). При этом характерный объем памяти в системах ЦОС существенно меньше, чем в системах общего назначения, что позволяет использовать дорогую синхронную память с простой иерархией.

Рассмотрим фильтр с конечной импульсной характеристикой (КИХ-фильтр, или FIR-filter), вычислительная схема которого показана на рисунке 1.

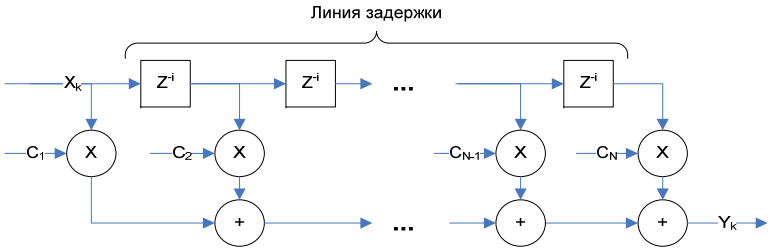


Рис. 1.

Основными действиями при работе фильтра являются:

1. Выборка инструкций из программной памяти.
2. Выборка двух операндов из памяти данных.
3. Умножение.
4. Накопление.

5. Сдвиг в линии задержки.

Все современные ЦПОС могут выдавать результат данной последовательности каждый такт. Это достигается за счет конвейерной организации специализированных действий (умножение, накопление, параллельная выборка из банков памяти).

Таким образом, для ЦПОС характерна *расширенная гарвардская архитектура*:

- Раздельные памяти для программ и данных;
- Несколько банков памяти данных (например, памяти X и Y) с параллельным доступом.

2.2. Режимы адресации

Использование параллельных банков памяти увеличивает пропускную способность памяти, однако для чтения нескольких ячеек памяти одной командой процессора необходимо знать адреса всех этих ячеек. Использование абсолютных адресов (прямая адресация) приводит к увеличению длины командного слова, что неприемлемо. Универсальным решением является использование косвенных регистровых режимов адресации. Такие режимы адресации являются основными для ЦПОС, так как они позволяют использовать несколько адресов в инструкции длиной в одно короткое слово. Для этой цели в ЦПОС обычно существует набор специальных адресных регистров. Как правило, банк адресных регистров небольшой, поэтому для их кодирования требуется всего несколько бит.

Кроме того, ЦПОС поддерживают аппаратную модификацию адресных регистров при косвенной адресации, что позволяет минимизировать число команд, необходимых для загрузки этих регистров. Одним из видов такой модификации является автоувеличение/уменьшение содержимого адресного регистра (++/--). При этом, шаг увеличения/уменьшения может определяться как константой, так и динамическим значением специальных шаговых регистров. Еще одним видом аппаратной модификации адресных регистров является циклическая адресация, когда автоувеличение адресного регистра приводит к зацикливанию его значения в пределах области, задаваемой парой специальных регистров (TOP/BOTTOM или START/LENGTH).

Характерным типом адресации служит также *косвенная адресация со смещением*, однако диапазон смещения невелик ввиду ограничений на длину командного слова.

В качестве примера рассмотрим разумную реализацию упоминавшегося выше КИХ-фильтра. В такой реализации один адресный регистр используется в качестве указателя на конец линии задержки, а другой регистр в качестве указателя на последний коэффициент фильтра. Тогда сдвиг в линии задержки (перемещение указателей к следующей паре «коэффициент, данные») будет осуществляться за счет автоуменьшения этих адресных регистров.

Таким образом, можно выделить следующие основные особенности ЦПОС, связанные с адресацией:

- Малое количество адресных регистров;
- Активное использование аппаратной модификации адресных регистров, в частности автоувеличения/уменьшения при косвенной адресации;
- Малый диапазон смещения при косвенной адресации со смещением.

2.3. Система команд

Для процессоров ЦОС характерно наличие множества специализированных функциональных устройств для быстрого выполнения определенных операций. Наиболее распространенными операциями являются умножение, умножение с накоплением, сложение с округлением, а также операции над числами с фиксированной точкой с поддержкой арифметики насыщения (saturation). Многие процессоры используют отдельные функциональные модули для поддержки конкретных алгоритмов, например быстрого преобразования Фурье или декодера Витерби.

При этом многие операции могут выполняться параллельно, то есть существуют комбинированные команды, выполняющие несколько операций параллельно (ограниченный параллелизм на уровне команд). Однако, в отличие от VLIW, возможные комбинации параллельно выполняющихся операций ограничены и нерегулярны. Ввиду относительно короткого командного слова, удается кодировать только некоторые комбинации, при этом система команд получается неортогональной. Под ортогональностью системы команд понимается возможность использования в командах любого регистра, доступного в регистровом файле. В рассматриваемом же классе ЦПОС операнды имеют уникальные для каждой команды ограничения на подмножество допустимых регистров, вплоть до того, что в некоторых командах разрешается использовать в качестве операнда только фиксированный регистр (из-за того, что необходимо уместить все необходимые операнды комбинированной команды в одно программное слово). В частности, при использовании косвенной адресации для разных команд допустимыми могут быть различные подмножества адресных регистров.

Среди характерных особенностей системы команд стоит отдельно упомянуть поддержку аппаратных циклов. В отличие от традиционной организации циклов с помощью команд условного ветвления, аппаратные циклы позволяют свести затраты на организацию цикла к нулю. Однако при этом существует ряд особенностей, таких как ограниченная вложенность циклов и ограничения на использование команд внутри тела цикла. Компилятор обязан учитывать все такие особенности при генерации аппаратных циклов.

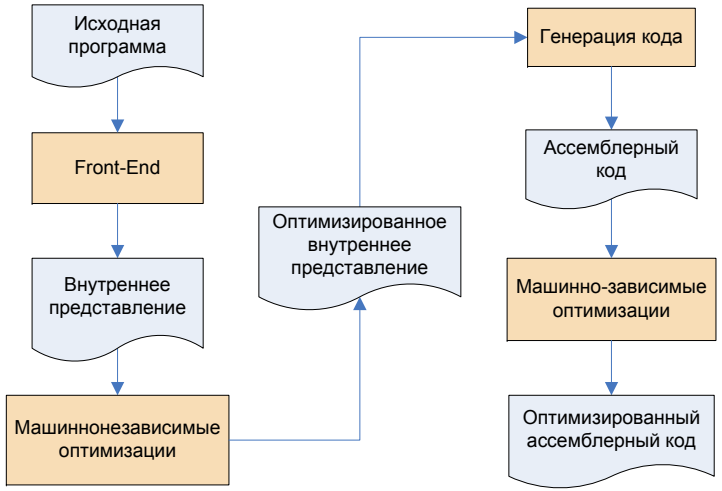
Таким образом, мы выделяем следующие особенности системы команд рассматриваемого класса ЦПОС, которые необходимо учитывать в компиляторе:

- Наличие отдельных функциональных устройств для вычисления специфических операций;

- Неортогональность системы команд;
- Поддержка аппаратных циклов.

3. Этапы компиляции

В этой главе мы приводим обобщённое представление о процессе компиляции. Обобщённая диаграмма работы компилятора выглядит следующим образом:



В этом процессе оптимизации производятся дважды:

- Машиннонезависимые производятся над внутренним представлением, полученным от front-end;
- Машиннозависимые оптимизации производятся над предварительным ассемблерным кодом, полученным от генератора кода.

Хотелось бы отметить, что, несмотря на своё название, машиннонезависимые оптимизации нельзя считать полностью оторванными от конкретной архитектуры. Многие из них разрабатывались с учётом общих представлений о свойствах некоторого класса машин (как правило, это машина с большим количеством регистров, фон-неймановской архитектуры, с относительно большой по размерам и медленной памятью). В современных компиляторах они, как правило, нацелены на достижение предельных скоростных характеристик программы, в то время как для встраиваемых систем к критическим параметрам также относится и размер получаемого кода. Некоторые оптимизирующие преобразования (такие, например, как раскрутка циклов) могут ухудшить программу ввиду резкого разрастания размеров и утери информации, полезной для последующих стадий оптимизации.

Ещё одна особенность компилятора для встраиваемой системы вытекает из свойства большинства таких систем: ограниченность памяти и отсутствие динамических библиотек. Программа компилируется единожды целиком и в

последствии не может изменяться по частям. Это порождает следующие преимущества при создании компилятора:

1. Исходные программы достаточно компактны, следовательно, время, необходимое на их компиляцию, меньше. Поэтому можно использовать более тяжеловесные алгоритмы и эвристики, чем те, что применяются в компиляторах общего назначения.
2. Есть возможность получить информацию обо всей программе целиком, не ограничиваясь только рамками собираемого модуля или библиотеки. Это свойство создаёт предпосылки для применения межпроцедурного анализа и глобальных оптимизаций.

Практически каждый компилятор производит определённый набор машинно-независимых оптимизаций. Вот примеры таких оптимизаций:

Исключение общих подвыражений: если внутреннее представление генерируется последовательно для каждого подвыражения, оно обычно содержит большое количество избыточных вычислений. Вычисление избыточно в данной точке программы, если оно уже было выполнено ранее. Такие избыточности могут быть исключены путём сохранения вычисленного значения на регистре и последующего использования этого значения вместо повторного вычисления.

Удаление мёртвого кода: любое вычисление, производящее результат, который в дальнейшем более не будет использован, может быть удалено без последствий для семантики программы.

Вынесение инвариантов циклов: вычисления в цикле, результаты которых не зависят от других вычислений цикла, могут быть вынесены за пределы цикла как инварианты с целью увеличения скорости.

Вычисление константных подвыражений: вычисления, которые гарантированно дают константу, могут быть произведены уже в процессе компиляции.

Несмотря на то, что эти преобразования называются машиннонезависимыми, иногда их стоит использовать с осторожностью. Вычисление общих подвыражений, к примеру, сокращая количество вычисляемых выражений, увеличивает потребность в регистрах. Для процессора с большим количеством функциональных устройств и небольшим количеством регистров повторение вычисления может оказаться более эффективным. Аналогичным образом вычисление константных подвыражений может отразиться на семантике программы, когда типы машин, производящих компиляцию и выполняющих программу, различны.

Все описанные ниже алгоритмы, за исключением межпроцедурного анализа, работают на этапе машиннозависимых оптимизаций. Здесь следует особо отметить, что их эффективность зачастую напрямую зависит от наличия той или иной дополнительной информации об исходной программе, а именно:

1. «Программная конвейеризация» и сворачивание в «аппаратные циклы» напрямую зависят от информации о циклах исходной программы и их

свойствах (цикл for, цикл while, фиксированное или нет число шагов, ветвление внутри цикла, степень вложенности). Таким образом, необходимо, во-первых, исключить машиннонезависимые преобразования, разрушающие структуру цикла, и, во-вторых, обеспечить передачу информации об этих циклах через кодогенератор.

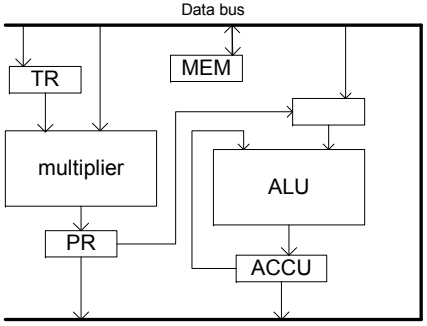
2. Алгоритмы «SOA» и «GOA» (см. ниже) используют информацию о локальных переменных программы. Требуется информация о том, что данное обращение к памяти суть обращение к переменной и эта переменная локальная.
3. Алгоритмы раскладки локальных переменных по банкам памяти требуют информации об обращениях к локальным переменным, плюс важно знать, используется ли где-либо адрес каждой из локальных переменных (т.е. возможно ли обращение по указателю).
4. Алгоритм «Array Index Allocation» и «частичное дублирование данных» должен иметь на входе информацию о массивах и обращениях к ним.

Таким образом, нельзя полностью абстрагироваться от начальных стадий работы компилятора, рассматривая их как производящий ассемблерный код «чёрный ящик». Предварительные стадии компиляции должны учитывать особенности класса целевой машины, чтобы создавать предпосылки к максимальной эффективности алгоритмов машиннозависимых оптимизаций.

4. Алгоритмы оптимизаций

4.1. Распределение регистров

Практически в каждом ЦПОС имеются так называемые специальные регистры. Их наличие обязательно должно учитываться при распределении регистров, что усложняет эту задачу. В качестве простейшего примера возьмём архитектуру тракта данных процессора Texas Instruments C25:



TR – первый операнд для мультипликатора

PR – результат умножения

ACCU – регистр сумматора обратным путём к ALU

С точки зрения разработчика компилятора, регистры специального назначения – скорее неудобство, поскольку задача распределения регистров не может быть полностью разделена с задачей выбора инструкций. Генерация кода без учёта конкретной архитектуры регистров может привести к необоснованному разрастанию кода в результате возникновения дополнительных пересылок между регистрами и памятью с целью поместить входные данные в подходящие регистры. Таким образом, концепция гомогенного неограниченного набора псевдорегистров вместе со стандартными подходами (например, раскраска графов) оказывается неэффективной в случае ЦПОС.

Более того, необходимо учитывать ограниченность количества регистров специального назначения. В большинстве случаев эти регистры (или регистровые файлы) вмещают одно или два значения. Таким образом, необходимо позаботиться, чтобы регистр не был перезаписан, пока его значение не используется в последний раз. Иначе возникнут дополнительные, дорогостоящие пересылки в память и обратно. В некоторых же случаях операция сохранения в памяти усложнена, как, например, на приведённом выше рисунке. Сохранение регистра TR возможно только через регистр PR (который, в свою очередь, может содержать полезное значение), а загрузка PR обратно из памяти ещё потребует умножения на 1.

Эффективность распределения регистров напрямую зависит как от результата выбора инструкций, так и от алгоритма распределения регистров. Причём желательно, чтобы алгоритм выбора инструкций учитывал особенности архитектуры, т.е. не был полностью независимым. Наиболее очевидное решение здесь видится в композиции этих фаз компиляции. Эффективные алгоритмы композиции фаз рассматриваются ниже.

4.2. Программная конвейеризация

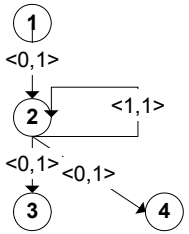
Software Pipelining (loop pipelining) – конвейеризация циклов. Это преобразование, призванное использовать способность процессора к параллельному вычислению для операций, находящихся на различных итерациях цикла.

Здесь мы расскажем об алгоритме итеративной модульной планировки (Iterative Modulo Scheduling), впервые описанном в [17].

В качестве примера рассмотрим вот такой цикл:

```
for i
    O1: a[i]=i*i
    O2: b[i]=a[i]*b[i-1]
    O3: c[i]=b[i]/n
    O4: d[i]=b[i]%n
```

Для удобства цикл представлен с помощью графа зависимости по данным (DDG) следующего вида:



В кружках стоят номера инструкций, а каждое ребро между инструкциями i и j помечено парой чисел <d,p>. Ребро означает:

- j зависит от значения, вычисленного i инструкцией p итераций цикла назад;
- как минимум d тактов должно пройти после выполнения соответствующего экземпляра i, чтобы можно было выполнить j.

После программной конвейеризации данный цикл может иметь следующий вид:

		Итерации		
Время (такты)	Пролог	1		
		2	1	
	Ядро (n-3 раз)	**3,4**	**2**	**1**
	Эпилог		3,4	2
				3,4

Как видно из таблицы, после преобразования цикл разбивается на три стадии: пролог, эпилог и ядро. В данном случае время между началом последовательных итераций ядра – один такт. Вообще задержка между началом последовательных итераций называется основным интервалом (initiation interval, размер ядра получаемого цикла) и обычно обозначается как II.

Описываемый здесь алгоритм – итеративный. Он пытается построить план (schedule) для значения основного интервала II, в случае неудачи увеличивая II на 1. Таким образом, очевидно, что скорость работы алгоритма непосредственно зависит от того, насколько удачно было выбрано начальное значение основного интервала. На выборе этого значения остановимся подробнее.

Для начала построим процедуру оценки минимального значения основного интервала II, обозначив её как MII.

$delay(c) = \sum_{i \in c} d_i$ – суммарная задержка по зависимостям данных для цикла c;

$distance(c) = \sum_{i \in c} p_i$ – суммарная дистанция в итерациях для цикла c;

$uses(r)$ – количество использований ресурса r за одну итерацию цикла;

$units(r)$ – число функциональных модулей типа r,

где c – естественный цикл на графе, являющемся транзитивным замыканием исходного графа зависимости по данным, G – множество всех таких циклов, а R – множество ресурсов.

Нижняя оценка величины основного интервала может быть получена двумя способами. Во-первых, можно оценить максимальную задержку по зависимостям по данным для каждого из естественных циклов на графе:

$$RecMII = \max_{c \in G} \frac{delay(c)}{distance(c)}.$$

Во-вторых, задержку можно оценить, исходя из предельной загруженности функциональных модулей по каждому из ресурсов:

$$ResMII = \max_{r \in R} \frac{uses(r)}{units(r)}.$$

Разумно взять наибольшую из полученных выше оценок за начальную величину основного интервала:

$$MII = \max(RecMII, ResMII).$$

Сам алгоритм выглядит следующим образом:

```
procedure modulo_schedule(budget_ratio)
   compute MII;
   II:=MII;
   budget:=budget_ratio*количество инструкций;
   while не найдено подходящего расписания do
         iterative_schedule(II, budget);
         II:=II+1;
```

где budget_ratio – количество возвратов в переборе перед тем, как переходить к следующему значению II. Непосредственно сам перебор с возвратом происходит в процедуре iterative_schedule:

```
procedure iterative_schedule(II, budget);
   compute height-based priorities;
   while есть неспланированные операции и budget > 0 do
         op:=операция с наивысшим приоритетом;
         min:=наиболее раннее время для операции op;
         max:=min+II-1;
         t:=find_slot(op,min,max);
         выставить операцию op в момент t
переместить все операции, имеющие конфликт с op, обратно в
список неспланированных команд;
         budget:=budget-1;
```

На каждом шаге выбирается инструкция с максимальным приоритетом. Возможны различные подходы к вычислению приоритета инструкции $H(i)$, например:

$H(i) = 0$, если i не имеет последователей, иначе

$$H(i) = \max_{k \in succ(i)} H(k) + latency(i,k) - II * distance(i,k).$$

Непосредственно временной слот, в который будет помещена конкретная инструкция j, ограничен снизу уже расставленными предшественниками k:

$$time(j) \geq time(k) + latency(k,j) - II * distance(k,j).$$

Процедура find_slot находит минимальный временной слот между min и max, такой, что операция op может быть размещена без конфликтов по ресурсам. Отметим, что выбранный слот может вызывать конфликты по ресурсам. Таким образом, размещение некоторых более ранних операций можно откатить и повторить заново.

```
procedure find_slot(op, min, max);
   for t:=min to max do
         if op не имеет конфликтов в моменте времени t
               return t;
   if op ещё не встречалась or
         min > предыдущее спланированное время для op
         return min;
   else
         return 1 + предыдущее спланированное время для op;
```

Для проверки конфликтов по ресурсам используется таблица резервирования. Высота этой таблицы равна II. Если ресурс r используется в момент времени t, то в таблице помечается ячейка [r, t mod II], поэтому эту таблицу ещё называют модульной таблицей резервирования (modulo reservation table, MRT).

Приведённый здесь алгоритм – лишь один из представителей целого класса. Он может учитывать множество дополнительных особенностей используемой архитектуры, такой как предикатное исполнение. Более подробно обзор различных подходов к программной конвейеризации циклов может быть найден в [16].

4.3. Использование характерных особенностей ЦПОС

В этом разделе приведены преобразования, несложные для реализации, но представляющие интерес, поскольку опираются на внутренние особенности ЦПОС.

Раскладка локальных переменных по банкам памяти. В основе данного оптимизирующего преобразования лежит тот факт, что гарвардская архитектура предоставляет возможность параллельного доступа к двум

ячейкам, при условии, что они находятся в различных памятях. Таким образом, появляется возможность ускорить доступ к данным при одновременном сокращении размера программного кода [11].

Раскладка данных в свою очередь приводит к трём взаимосвязанным проблемам:

1. Необходимо установить связи между различными элементами данных в контексте раскладки по разным памятям. Это включает в себя определение пар элементов данных, потенциально способных находиться в различных банках памяти.
2. Необходимо определить функцию стоимости для каждой связи.
3. Необходимо задать правила таким образом, чтобы удовлетворить как можно большему количеству разделяющих связей, тем самым достигнув наибольшей общей производительности.

Частичное дублирование данных [11] оказывается полезным для оптимизации работы с участками кода, подобными приведённому ниже:

```
for(n=1; n<r; n++)
{
R[n] +=signal[n] * signal[n+m];
}
```

Подобный код используется, в частности, при вычислении функции автокорреляции, что нередко встречается в алгоритмах для ЦПОС. Особенность этого фрагмента в том, что происходит одновременное обращение к двум элементам одного и того же массива.

Данный фрагмент можно существенно ускорить за счёт дублирования значений signal[n] в двух банках памяти. Найти массивы-кандидаты на дублирование несложно – области жизни двух обращений к одному и тому же массиву должны пересекаться.

Дублирование данных не всегда приводит к повышению производительности. В частности, если в последствии не удастся запаковать оба обращения в одну инструкцию. В этом случае дублирование данных только ухудшает программу.

Кроме того, дублирование может усложнить работу с прерываниями. Если записи различных двух копий одних и тех же данных окажутся разнесёнными по различным инструкциям, теоретически возникает возможность аппаратного прерывания между ними, а значит, данные в двух копиях могут оказаться в несогласованном состоянии.

Использование аппаратной поддержки циклов. В ЦПОС присутствует поддержка аппаратных циклов. Генерация кода для этого случая, в принципе, тривиальна: если имеется информация об исходном цикле, необходимо проверить, что этот цикл удовлетворяет некоторым условиям:

1. Внутри цикла нет ветвлений и вызовов процедур;
2. Границы цикла известны и фиксированы;
3. Тело цикла не зависит напрямую от переменной индукции;
4. Тело цикла не содержит запрещённых инструкций.

В данном случае важно, что компилятор сохраняет информацию о циклах до довольно поздней стадии: стадии, когда уже сгенерирован код. Например, в SPAM это достигается за счёт того факта, что SUIF, используемый в качестве front-end, позволяет «протаскивать» информацию об исходных циклах через весь слой машиннонезависимых оптимизаций и генерации кода.

4.4. Сжатие кода

Сжатие кода (Code Compaction) – стадия работы компилятора, направленная на формирование микроинструкций из команд, которые могут быть выполнены параллельно. При этом планирование команд должно проводиться с учетом зависимостей по данным и ограничений на использование ресурсов. Учет зависимостей по данным гарантирует, что никакая команда не будет выполнена до тех пор, пока все команды, от которых она зависит, не будут выполнены полностью. Ограничения на ресурсы гарантируют, что построенное расписание не затребует ресурсов (функциональных модулей) больше, чем позволяет архитектура. Целью сжатия кода является получение минимального количества микроинструкций.

Известно, что задача сжатия кода NP-полная. Поэтому был разработан целый ряд эвристических алгоритмов планирования. Перечислим такие, как first-come-first-served, critical path и list scheduling. Каждая из этих эвристик приводит к близким к оптимуму результатам в большинстве случаев, отличаясь в свою очередь скоростью и простотой. Детальное описание этих алгоритмов (сложности O(n2)) может быть найдено в [3]. В этом обзоре мы остановимся подробнее на алгоритме list scheduling.

Приоритетное списочное планирование (list scheduling) относится к разряду алгоритмов, решающих задачу локального сжатия кода, т.е. планирования команд внутри базового блока. Стоит отметить, что в этом случае зависимости по данным представляются в виде ориентированного ациклического направленного графа (DAG – direct acyclic graph). Алгоритм планирует команды с нулевого момента времени, начиная с входной команды базового блока. В каждый момент времени t поддерживается список готовых команд (ReadyList), т.е. команд, чьи предшественники уже были спланированы и произведут результат в выходном регистре к моменту времени t. List scheduling является жадной эвристикой и всегда планирует в текущий момент времени готовую инструкцию в случае отсутствия конфликтов по ресурсам. Требования по ресурсам для спланированных команд поддерживаются в глобальной таблице резервирования (GRT – global reservation table).

Если есть несколько готовых команд, то выбирается команда с наивысшим приоритетом. Приоритеты назначаются в соответствии с различными эвристиками. Различные виды алгоритма list scheduling отличаются способами задания приоритетов для команд. Отметим, что назначение приоритета может быть как статическим (приоритет назначается один раз и не меняется больше в течение планирования), так и динамическим (меняется в течение

планирования) и таким образом требовать пересчета приоритетов готовых команд после каждой спланированной команды. Мы рассмотрим вариант алгоритма со статическими приоритетами.

Когда все команды из ReadyList, не приводящие к конфликту по ресурсам, спланированы, то момент времени увеличивается на единицу, и ReadyList пополняется новыми готовыми командами. Затем ReadyList сортируется в порядке убывания приоритета. Процесс планирования продолжается до тех пор, пока все команды не будут спланированы. В общем виде алгоритм list scheduling можно записать следующим образом:

```
Вход:  DAG
Выход: Instruction Schedule
AssignPriority (DAG); /* каждой команде присваивается
приоритет в соответствии с выбранной эвристикой */
ReadyList = source node in the DAG;
timestep = 0;
while (DAG содержит неспланированные команды) do
{
Сортировать ReadyList в порядке убывания приоритета;
while (не испробованы все команды из ReadyList) do
      {
       Выбрать следующую команду i из ReadyList;
       Проверить на конфликт по ресурсам;
       if (команду можно спланировать)
       {
              update GRT (i, timestep);
              Удалить команду из ReadyList;
                    }
      }
      timestep++;
      Добавить команды, которые стали готовыми, в
ReadyList
}
```

Рассмотрим эвристики, используемые для назначения приоритета готовым командам. Обычно используется эвристика, основанная на максимальном расстоянии от узла до фиктивного узла стока. Максимальное расстояние узла стока до себя принимается за ноль. Максимальное расстояние узла u определяется как:

$$MaxDistance(u) = \max(MaxDistance(v_i) + weight(u, v_i))$$

где максимум берется по всем vi – потомкам узла u. MaxDistance вычисляется с помощью обратного прохода графа зависимости и является статическим приоритетом. Приоритет отдается узлам с наибольшим MaxDistance.

Часто более высокий приоритет отдается командам, которые имеют наименьшее значение Estart. Для входного узла графа Estart принимается равным нулю. Значение Estart команды v определяется как:

$$Estart(v) = max(Estart(u_i) + weight(u_i, v))$$

где максимум берется по всем ui – предкам v. Похожим образом можно отдать приоритет командам с наименьшим значением параметра Lstart, который определяется как:

$$Lstart(u) = min(Lstart(v_i) - weight(u, v_i))$$

где минимум берется по всем vi – потомкам узла u. Значение Lstart для узла стока принимается равным Estart. Разность между Lstart(u) и Estart(u) называется резервом времени или мобильностью и также может быть использована для назначения приоритета. Команды с меньшим резервом времени получают больший приоритет. Команды, лежащие на критическом пути, могут иметь нулевой резерв времени и поэтому иметь больший приоритет над командами, лежащими вне критического пути.

Мы уже упоминали, что задача сжатия кода является NP-полной. Для нахождения оптимального расписания команд можно воспользоваться методом целочисленного линейного программирования. Сформулируем задачу для случая планирования базового блока при наличии ограничений на ресурсы. Предполагается простая модель ресурсов с полностью конвейеризированными функциональными модулями. Рассмотрим архитектуру с функциональными модулями различного типа (например, АЛУ, устройство записи/чтения, умножитель/делитель), где задержка выполнения команд может быть разной в различных модулях. Также будем считать, что существует Rr экземпляров для типа r функционального модуля.

Пусть si – момент времени, для которого спланирована команда i, и d(i, j) – вес ребра (i, j). Чтобы не нарушить зависимостей по данным для каждого ребра (i, j), в DAG должно выполняться:

$$(1) \qquad s_j \geq s_i + d_{(i,\ j)}.$$

Рассмотрим матрицу K размера n x T, где n – количество команд или узлов в DAG, и T – оценка для наихудшего времени выполнения всего расписания. Обычно T – это сумма времен выполнения всех команд, входящих в базовый блок. Отметим, что T – константа и может быть получено из DAG. Элемент матрицы K[i,t] равен единице, если команда i спланирована для момента времени t, и нолю – в противном случае. Время планирования si для команды i можно выразить с помощью K в следующем виде:

$$s_i = k_{i,0} * 0 + k_{i,1} * 1 + \ldots + k_{i,T-1}.$$

Для всех si это может быть выражено в матричной форме:

$$(2) \qquad s = K * N.$$

где столбец s = {s0, s1, … , sn-1} и столбец N = {0, 1, …, T-1}.

Тот факт, что каждая команда может быть спланирована в точности один раз, выражается условием:

$$(3) \qquad \sum k_{i,t} = 1 \quad , \forall i.$$

Наконец, ограничения на ресурсы задаются неравенством:

$$(4) \qquad \sum k_{i,t} \leq R_r \quad , \forall t, \forall r, i \in F(r),$$

где F(r) представляет собой набор команд, которые могут выполняться на функциональном модуле типа r.

Целевой функцией является минимизация длины расписания, что можно выразить как:

$$minimize\ (max\ (s_i + d_{(i,j)})).$$

Чтобы выразить это в линейной форме, введем:

$$(5) \qquad z \geq s_i + d_{(i,\ j)}.$$

Таким образом, решение задачи состоит в минимизации z при условиях (1) – (5).

Стоит отметить, что важным фактором для реализации любого из алгоритмов сжатия кода является описание имеющихся в наличии параллельных функциональных модулей и подходящих для этого инструкций с учётом нерегулярности кодирования.

4.5. Оптимизации адресного кода

Данный набор оптимизирующих преобразований основывается на способности блока генерации адресов (AGU – address generation unit) при использовании индексного регистра для адресации производить его изменение (увеличение или уменьшение на фиксированный шаг). Данный набор преобразований в такой комбинации стал уже «джентльменским набором» оптимизаций для встраиваемых систем. Их описание можно встретить как один из пунктов в [12, 20, 6] или отдельно в [14, 13].

Попытаемся на простых примерах пояснить смысл каждого из этих преобразований.

SOA (Simple Offset Assignment)

Допустим, имеется последовательность доступов к переменным (a, b, c, d – имена переменных):

b c a d b c a d

Обозначим через & операцию вычисления адреса, A – операцию доступа, I – индексный регистр, ++ или -- операцию постизменения индексного регистра.

В общем случае каждое обращение к переменной имеет вид:

```
I = &var
A(I)
```

т.е. для каждого обращения необходимо произвести две операции. Допустим теперь, что эти переменные в памяти были расположены в лексикографическом порядке, т.е:

```
a b c d
```

Для работы с рассмотренной выше последовательностью обращений будет необходимо произвести следующие операции:

```
I=&b
A(I++)
A(I)
I=&a
A(I)
I=d
A(I)
I=&b
A(I++)
A(I)
I=&a
A(I)
I=d
A(I)
```

В итоге, для работы с данной последовательностью обращений при имеющемся расположении переменных в памяти нам необходимо затратить 14 операций. Допустим теперь, что мы разложили упомянутые выше переменные следующим образом:

```
b c a d
```

Теперь последовательность операций будет выглядеть так:

```
I(&b)
A(I++)
A(I++)
A(I++)
A(I)
I(&b)
A(I++)
A(I++)
A(I++)
A(I)
```

Всего мы затратили 10 операций. Таким образом, правильное распределение в памяти может дать существенную экономию на операциях загрузки адреса при использовании возможностей автоувеличения или автоуменьшения.

Для вычисления оптимального расположения переменных в памяти используются различные эвристики (эвристики для решения задачи SOA).

GOA (General Offset Assignment)

Задача GOA аналогична SOA, с той разницей, что мы учитываем наличие более чем одного индексного регистра, что иногда позволяет получить более экономный код.

Array Index Allocation

И, наконец, алгоритм Array Index Allocation позволяет оптимизировать циклические доступы к массиву. К примеру, данный цикл:

```
for(int i=0;i<10;i++)
{
 a[i];
 a[i+1];
}
```

Может быть приведен к следующему виду:

```
ptr = a;
for(int i=0;i<10;i++)
{
 ptr++;
 ptr;
}
```

Таким образом, мы перешли от операции косвенной адресации a[i+1] к индексному регистру с автоувеличением ptr++, сэкономив на операции косвенной адресации. Для того, чтобы эта оптимизация была возможна, необходимо «протаскивать» информацию о доступах к массиву и о циклах.

Вообще говоря, в некоторых архитектурах бывает возможен доступ с использованием абсолютной адресации или смещения, но и то и другое, как правило, требует каких-либо дополнительных затрат, например, дополнительного командного слова для кодирования смещения и, соответственно, задержки в выполнении. Кроме того, такие операции в дальнейшем не могут быть подвергнуты сжатию кода, поэтому обращение с использованием индексных регистров с автоувеличением/автоуменьшением оказывается предпочтительнее с точки зрения конечного качества кода.

Подробнее о реализации приведённых здесь преобразований в SPAM см.[6].

5. Взаимное влияние оптимизаций

Ввиду большой вычислительной сложности задачи компиляции, компилятор, как правило, разбивается на ряд различных фаз. Этот очевидный сам по себе подход подразумевает, что фазы выполняются в определённом порядке. Существует нетривиальная проблема определения наиболее оптимального порядка следования фаз. Многие фазы могут влиять на потенциальную

эффективность выполнения последующих фаз. Машиннонезависимая оптимизация «подстановка констант», к примеру, обычно повышает эффективность «вычисления константных выражений», что может привести к появлению последующих констант для подстановки. Таким образом, ясно, что эти две оптимизации должны производиться итеративно.

Что касается back-end'a, то задача определения порядка фаз здесь усложняется, поскольку некоторые фазы сокращают применимость последующих фаз. Более того, зависимость между фазами обычно циклична. Мы проиллюстрируем эту проблему следующими примерами:

Выбор инструкций и распределение регистров: алгоритм выбора инструкций преобразует элементы внутреннего представления в машинные команды. Обычно эта фаза также определяет, какие значения будут оставаться на виртуальных регистрах. Более того, при наличии более одного регистрового файла алгоритм выбора инструкций также должен определять типы регистров для каждой операции, поскольку, как правило, имеется оптимальный вариант с точки зрения стоимости инструкции. Тем не менее, только при распределении регистров можно понять, был ли тот или иной выбор удачным, если принять во внимание стоимость сохранения временного значения в памяти.

Распределение регистров и планирование команд: распределение регистров сопоставляет виртуальным регистрам физические. Обычно при этом один физический регистр используется для нескольких виртуальных, что приводит к дополнительным антизависимостям, мешающим планировщику. С другой стороны, последовательное планирование команд явно влияет на области жизни виртуальных регистров, тем самым непосредственно влияя на результаты распределения регистров.

Планирование команд и оптимизации адресного кода: оптимизации адресного кода зависят от конкретной последовательности доступов к памяти, получаемой после планировщика. Положение переменных в памяти и количество операций автоувеличения зависят от последовательности доступа, и возможно, что альтернативная последовательность команд будет более эффективна с точки зрения стоимости адресных вычислений. Поскольку оптимизации адресного кода приводят к новым инструкциям, они могут повлиять в дальнейшем на эффективность сжатия.

Результат приведённых здесь зависимостей таков, что практически любая последовательность фаз может приводить к неэффективному коду для определённых входных программ. Проблема порядка следования фаз ещё острее для процессоров с нерегулярной архитектурой, таких как большинство ЦПОС. Для преодоления этих проблем обычно прибегают к сопряжению фаз (phase coupling), что приводит к тесному обмену информацией между сопряжёнными фазами.

Простейший способ сопряжения фаз – итеративное выполнение нескольких фаз с возможностью изменения потенциально неправильных решений, основанных на пометках, сделанных последующими фазами. Другие подходы

пытаются оценить влияние каждой фазы на последующие фазы на ранней стадии. Наибольшая степень сопряжения фаз достигается, когда фазы фактически соединяются в одну, хотя такие алгоритмы тяжелы для разработки и, как правило, очень трудоёмки.

Существует целый ряд подходов к композиции фаз. Например, в [22] покрытие деревьев совмещается с распределением регистров и планированием для Texas Instruments DSP. Ещё один подход, основанный на алгоритме simulated annealing, по сути похожем на генетические алгоритмы, описан в [23].

Здесь мы остановимся на общем подходе, основанном на CLP (constraint logic programming) – системе решения определённого класса задач. К этому классу задач можно свести и задачи генерации кода для нерегулярных архитектур, в частности для ЦПОС [24], т.е. для случаев, когда имеется большое количество различных ограничений, которые нельзя выразить в общем виде. Рассмотрим пример такого описания. Пусть имеется инструкция вида:

```
X1:=X2+X3
```

где X1, X2 и X3 представляют различные наборы регистров, причём не все комбинации регистров, представляющих каждый из операндов, возможны, а имеются дополнительные ограничения. К примеру, пусть X1={a1,a2}, X2={b2, b2} и X3={c1,c2}. При этом обычно присутствуют ограничения вида: «если X2=b1 и X3=c1, то X1 может быть только a2», или же «X1=a1 только при X3=c2» и т.п. Эта ситуация может быть ещё более усложнена, если допустимые комбинации регистров зависят и от выполняемой параллельно инструкции. Искомые величины (X2) будем в дальнейшем называть переменными, а множество их возможных значений ({b1, b2}) – доменами. Решением задачи CSP называется набор значений для переменных X1..X3, удовлетворяющих всем ограничениям.

Например, в [25] система команд процессора описывается следующим списком:

```
(Op, R, [O1,…,On], ERI, Cons)
```

где Op обозначает имеющуюся в процессоре операцию, R – множество ресурсов для хранения результата (регистров или ячеек памяти), Oi – множество ресурсов для размещения входных данных, ERI – набор дополнительных ресурсов, используемых данной машинной операцией (допустим, шины или функциональные узлы), Cons – набор ограничений на значения Op, R, Oi и ERI.

Для решения задачи CLP в [28] использовался язык ECLiPSe [26], являющийся расширением языка PROLOG. При этом поиск решения задачи сводится к поиску некоторой маркировки, т.е. выбора для каждой исходной переменной одного из значений из связанного с ней домена. Например, переменная X1 из приведённого выше примера может быть промаркирована значением a1 из её домена {a1, a2}. ECLiPSe содержит библиотеку алгоритмов для поиска маркировки для случая конечных доменов значений. Более того, в ECLiPSe входит

набор обобщённых оптимизирующих процедур, получающих на вход стратегию маркировки l(V) на наборе переменных V вместе с целевой функцией cost(V) и вычисляющих оптимальную маркировку: minimize(l(V), const(V)).

Данный подход позволяет достигать впечатляющих результатов (разница по сравнению с ручным ассемблерным кодом всего порядка 20%). Из недостатков следует отметить заметные затраты ресурсов и времени (порядка минут) на компиляцию тестовых программ из набора DSPStone 27.

6. Заключение

В данной работе мы рассмотрели основные особенности процессоров ЦПОС и методы оптимизации, основанные на использовании этих особенностей. В частности, особенности адресной арифметики дают возможность использования методов оптимизации распределения смещений (GOA и SOA). Алгоритмы сжатия кода позволяют использовать свойство ограниченного параллелизма команд в условиях нерегулярного кодирования. Такие оптимизации, как частичное дублирование данных, позволяют учитывать специфику расширенной гарвардской архитектуры. Кроме того, общие свойства программ для встраиваемых систем (небольшие размеры, отсутствие динамических библиотек и мягкие требования ко времени компиляции) позволяют изучать и использовать ресурсоемкие алгоритмы оптимизации, в частности, сопряжение фаз генерации кода. При построении компиляторов для ЦПОС архитектур применение рассмотренных методов является важнейшим элементом достижения эффективности генерируемого кода.

В дальнейшем мы планируем использовать подход сопряжения фаз с помощью решения задачи CLP при разработке компилятора на базе платформы GCC для разрабатываемого заказчиком закрытого ЦПОС. Из возможных трудностей стоит отметить, что, несмотря на универсальность метода CLP [28], существующий алгоритм ограничивается одним базовым блоком и в некоторых аспектах для принятия решений используются эвристики. Это говорит о незавершённости разработки данного подхода. Как признают авторы CLP [28], эвристики должны быть в дальнейшем сведены к общей процедуре поиска оптимальной маркировки. Кроме того, наш целевой процессор имеет существенно большее количество регистров общего назначения, чем тот, который использовали авторы данного подхода, что может привести к затруднению задачи выбора оптимальной маркировки при распределении регистров. По всей видимости, потребуется внесение дополнительных ограничений для устранения этой проблемы.

Литература

1. *E. Lee*. Programmable DSP Architectures: Part I. IEEE ASSP Magazine, pp. 4-19, October 1988.
2. *E. Lee.* Programmable DSP Architectures: Part II. IEEE ASSP Magazine, pp. 4-14, January 1989.

3. *S. Davidson, D. Landskov, B.D. Shriver, P.W.Mallett*. Some Experiments in Local Microcode Compaction for Horizontal Machines. IEEE Trans. on Computers, vol. 30, no. 7, 1981, pp. 460-477.
4. *Khalid Ismail*. DSP Architecture Implications on Compilation.
5. *Richard M. Stallman*. Using and Porting the GNU Compiler Collection.
6. *Ashok Sudarsanam*. Code Optimization Libraries For Retargetable Compilation For Embedded Digital Signal Processors.
7. *C. Liem*. Retargetable Compilers for Embedded Core Processors. Kluwer Academic Publishers, 1997.
8. *Алла Солонина, Дмитрий Улахович, Лев Яковлев*. Алгоритмы и процессоры цифровой обработки сигналов. БХВ-Петербург, 2001.
9. *Vasanth Bala, Norman Rubin*. Efficient Instruction Scheduling Using Finite State Automata. 1995.
10. *P. Briggs*. Register Allocation via Graph Coloring. PhD Thesis, RICE University, 1992.
11. *M. Saghir, P. Chow, C. Lee*. Exploiting Dual Data-Memory Banks in Digital Signal Processors. In Proceedings of the ACM SIGARCH Conference on Architectural Support for Programming Languages and Operating Systems, 1996.
12. *Rainer Leupers*. Retargetable Code Generation for Digital Signal Processors. Kluwer Academic Publishers, 1997.
13. *S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang*. Storage Assignment to Decrease Code Size. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 186-195, June 1995.
14. *R. Leupers, P. Marwedel*. Algorithms for Address Assignment in DSP Code Generation. In Proceedings of the International Conference on Computer-Aided Design, 1996.
15. *Шумаков С.М*. Обзор методов оптимизации кода для процессоров с поддержкой параллелизма на уровне команд. сентябрь 2003, http://www.citforum.ru/programming/digest/20030430/
16. V. Allan, R. Jones, R. Lee, S. Allan. Software Pipelining.
17. *B. Ramakrishna Rau*. Iterative Modulo Scheduling: An Algorighm For Software Pipelining Loops.
18. *A. Aiken, A. Nicolau, S. Novack*. Resource-Constrained Software Pipelining.
19. *T. Proebsting, C. Fraser*. Detecting Pipeline Structural Hazards Quickly. 1994.
20. *Rainer Leupers*. Code Optimization Techniques for Embedded Processors. Kluwer Academic Publishers, 2000.
21. *R. Ramakrishna Rau*. Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops.
22. *G. Araujo, S. Malik*. Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architectures. 1995.
23. *R. Leupers*. Register Allocation for Common Subexpressions in DSP Data Paths. 2000.
24. *R. Leupers, P. Marwedell*. Retargetable Compiler Technology for Embedded Systems.
25. *R. Leupers, S. Bashford*. Graph based Code Selection Techniques for Embedded Processors.
26. *M. Wallace, S. Novello, J. Schimpf*. ECLiPSe: A Platform for Constraint Logic Programming.
27. *V. Zivonovich, J.M. Velarde, C. Schläger, H. Meyr*. DSPStone – A DSP-oriented Benchmarking Methodology. 1994.
28. *S. Bashford, R.Leupers*. Phase-Coupled Mapping of Data Flow Graphs to Irregular Data Paths. 1999.