Научно-исследовательский институт системных исследований РАН

На правах рукописи

Шумаков Сергей Михайлович

Оптимизация объектного кода для процессорных архитектур с поддержкой параллелизма на уровне команд

Специальность 05.13.11 – математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени кандидата физико-математических наук

Научные руководители: член-корреспондент РАН, профессор Бетелин В.Б., доктор физико-математических наук Галатенко В.А.

Москва – 2002

Оглавление

1. Введение

Актуальность темы.

Поддержка параллелизма на уровне команд в (микро)процессорных архитектурах - одно из основных направлений повышения быстродействия компьютеров. Такие микропроцессоры используются как в гражданских областях, так и при создании перспективных образцов вооружения и военной техники, в том числе для систем обработки информации в радиолокаторах и гидроакустических комплексах. Примером отечественного микропроцессора с поддержкой параллелизма на уровне команд является 1В577 и модели на его основе.

С аппаратной точки зрения наиболее перспективными в данном классе архитектур признаются процессоры с длинным командным словом. Их быстрое развитие ставит сложные задачи перед разработчиками инструментальных средств программирования. Некачественный компилятор способен нивелировать преимущества, предоставляемые высокопроизводительной аппаратурой. Укажем, что компания Hewlett-Packard, параллельно с созданием архитектуры IA-64, выделяет миллионы долларов на развитие инструментальных средств.

Трудность генерации эффективного кода для микропроцессорных архитектур с длинным командным словом проистекает в первую очередь из того, что традиционные компиляторы генерируют последовательный код, уступающий по качеству оптимальному в 5-10 раз, в то время как для достижения максимальной эффективности кода необходимо объединять в длинные командные слова элементарные операции, относящиеся к разным операторам исходного языка и разным итерациям циклов.

Таким образом, для полного использования преимуществ микропроцессоров с длинным командным словом нужны новые исследования, новые методы оптимизации объектного кода.

Цели диссертационной работы.

Основными целями диссертационной работы являются:

- исследование методов оптимизации объектного кода для микропроцессорных архитектур с параллелизмом на уровне команд;
- разработка и реализация постпроцессора, выполняющего оптимизацию объектного кода для процессоров с длинным командным словом;
- разработка и реализация средств настройки оптимизирующего постпроцессора на различные целевые архитектуры;

Научная новизна.

Предложен алгоритм планирования потока команд для процессоров с длинным командным словом путем перебора планов выполнения по методу динамического программирования. Наиболее актуальная область применения разработанного алгоритма планирования – генерация объектного кода для цифровых процессоров обработки сигналов с длинным командным словом.

Предложены новые средства описания процессорных архитектур с длинным командным словом, делающие возможным выделение архитектурно-независимого ядра оптимизирующего постпроцессора.

Практическая ценность.

Реализован оптимизирующий компилятор для отечественного микропроцессора 1В577. Этот компилятор используется в НИИСИ РАН, КБ "Корунд".

Апробация.

Основные положения диссертационной работы докладывались на международной конференции «Параллельные вычисления и задачи управления», Москва, ИПУ РАН 2001 и на семинаре «Современные сетевые технологии», МГУ, 2001.

Публикации.

По теме диссертации опубликовано 5 печатных работ - [4], [5], [6], [7], [19].

Объем и структура работы.

Диссертация состоит из 4 глав, заключения и списка литературы.

Первая глава является введением.

Вторая глава представляет собой обзор методов оптимизации кода для процессоров с поддержкой параллелизма на уровне команд. Особое внимание уделяется методам усиления программного параллелизма на уровне команд, а также методам планирования потока команд.

В третьей главе подробно рассматривается реализация компилятора для процессора 1В577 с оптимизирующим постпроцессором. В ней описана структура компилятора, рассмотрены оптимизации, реализованные в постпроцессоре, в частности, алгоритм планирования потока команд перебором планов выполнения по методу динамического программирования. Здесь же описаны средства настройки постпроцессора на целевую архитектуру и применение этих средств для настройки на архитектуру процессора 1В577.

В главе 4 дается сравнение реализованного алгоритма планирования с другими известными алгоритмами, а также приводятся результаты измерений эффективности кода, полученного с помощью постпроцессора. Полученный объектный код для различных тестовых задач сравнивается по времени выполнения с кодом, написанным вручную опытным программистом, а также с кодом других имеющихся компиляторов.

В заключении излагаются основные результаты диссертационной работы.

Список литературы насчитывает 70 названий.

2. Обзор методов оптимизации кода для процессоров с поддержкой параллелизма на уровне команд

Процессоры, способные одновременно и независимо выполнять несколько элементарных команд, обладают исключительно высоким потенциалом производительности и находят все более широкое применение. О процессорах такого типа говорят, что они поддерживают параллелизм на уровне команд (Instruction Level Parallelism, ILP). Далее для краткости они будут называться ILP-процессорами. Класс ILP-процессоров включает суперскалярные процессоры и процессоры с очень длинным командным словом (Very Large Instruction Word, VLIW), к числу которых относятся, в частности, многие модели цифровых процессоров обработки сигналов (ЦПОС).

Важное преимущество ILP по сравнению с параллелизмом многопроцессорных архитектур заключается в том, что программный параллелизм на уровне команд извлекается (аппаратурой или компилятором) автоматически, без дополнительных усилий со стороны прикладных программистов, в то время как использование параллелизма многопроцессорных архитектур подразумевает переписывание приложений.

Для реального использования высокой производительности ILP-процессоров необходимы компиляторы с языков высокого уровня, способные генерировать эффективный код. Применение одних лишь традиционных методов оптимизации кода оказывается совершенно недостаточным. Например, согласно [3] или [50], типичный компилятор для ЦПОС (поддерживающий только традиционные оптимизации) генерирует код, который по времени выполнения может уступать

оптимальному в 5-10 и более раз.

В течение последних лет прилагаются значительные усилия по разработке специальных методов оптимизации программ для ILP-процессоров, направленных на выявление и расширение программного параллелизма на уровне команд. Данная глава содержит обзор таких методов.

2.1. ILP-платформы

Общие свойства ILP-процессоров - способность одновременно и независимо выполнять несколько операций и наличие нескольких функциональных устройств различных типов, таких как, например, устройство обмена с памятью, арифметическое устройство и др. В выполнении каждой команды участвует определенный набор функциональных устройств. Процессор может выполнять команды $c_1, ..., c_n$ одновременно, если:

- процессор имеет достаточно функциональных устройств для их совместного выполнения.
- ни одна из команд c_i не использует в качестве входных операндов результаты других команд $c_1, ..., c_n$;

ILP-процессоры могут различаться многими характеристиками, которые существенны с точки зрения применимости и эффективности рассматриваемых далее методов оптимизации. Данный раздел содержит краткий обзор типов ILP-процессоров и их свойств.

Одним из исторически первых видов процессорного параллелизма был конвейерный параллелизм, основанный на том, что выполнение команды разбивалось на этапы, на каждом из которых использовались определенные функциональные устройства (рис. 2.1). Средства

конвейеризации обеспечивали совмещенный режим выполнения команд, когда эти команды оказывались независимыми друг от друга. При этом разработчики стремились добиться того, чтобы среднее количество тактов на выполнение команд в конвейере равнялось 1, т.е. чтобы темп выдачи команд составлял одну команду на такт.

Пусть исполнение команды состоит из 3-х этапов
по 1 процессорному такту на каждый:
1) чтение команды из памяти (Ч);
2) декодирование (Д);
3) исполнение (И).

Последовательное исполнение команд									
Этапы	Ч1	Д1	И1	Ч2	Д2	И2	Ч3	Д3	И3
Такты →	1	2	3	4	5	6	7	8	9
Конвейерное исполнение команд									
Устройство чтения:	Ч1	Ч2	Ч3	Ч4	Ч5	Ч6	Ч7		
Устройство декодирования:		Д1	Д2	Д3	Д4	Д5	Д6	Д7	
Устройство исполнения:			И1	И2	И3	И4	И5	И6	И7
Такты →	1	2	3	4	5	6	7	8	9
Конвейерное суперскалярное исполнение команд									
Устройство чтения 1:	Ч1	Ч2	Ч3	Ч4	Ч5	Ч6	Ч7		
Устройство декодирования 1:		Д1	Д2	Д3	Д4	Д5	Д6	Д7	
Устройство исполнения 1:			И1	И2	И3	И4	И5	И6	И7
Устройство чтения 2:	Ч1	Ч2	Ч3	Ч4	Ч5	Ч6	Ч7		
Устройство декодирования 2:		Д1	Д2	Д3	Д4	Д5	Д6	Д7	
Устройство исполнения 2:			И1	И2	И3	И4	И5	И6	И7
Такты →	1	2	3	4	5	6	7	8	9

Рис. 2.1. Последовательное и параллельное исполнение команд

Естественным развитием средств конвейерной обработки явились процессоры с множественной выдачей команд на исполнение (multiple issue processors) - суперскалярные и VLIW-процессоры. Суперскалярный процессор исполняет обычный последовательный код, но может выбирать

в нем и выдавать на выполнение одновременно несколько команд - не более *n*, где *n* - темп выдачи команд данного процессора. Различаются суперскалярные процессоры с упорядоченной и неупорядоченной выдачей команд на исполнение. Процессор первого типа выдает команды на исполнение в точности в том порядке, в котором они закодированы в программе. На каждом такте на исполнение выдается от 1 до *n* очередных команд с учетом возможности их параллельного исполнения. Процессор второго типа анализирует команды в пределах некоторого "окна" - текущего фрагмента входной программы - выбирая в нем для выдачи на исполнение от 1 до *n* команд с учетом связей по данным и возможности параллельного исполнения.

При разработке суперскалярных процессоров обычно преследуют цель обеспечить бинарную совместимость с предшествующими поколениями (скалярными или суперскалярными) данного модельного ряда процессоров (см. [60]). Суперскалярный процессор выполняет (без перекомпиляции) программный код для предшествующей модели, обеспечивая более высокую производительность.

VLIW-процессоры отличаются от суперскалярных тем, что код для них организован в виде последовательности очень длинных командных слов, каждое из которых содержит несколько элементарных команд (операций). Забота о корректном заполнении командных слов возлагается на компилятор (или программиста, пишущего на ассемблере). VLIW-процессоры в целом производительнее суперскалярных, поскольку исключается динамический анализ зависимостей по данным и функциональным устройствам во время выполнения. Однако реальная эффективность выполнения программы целиком зависит от качества кода, сгенерированного компилятором.

Следует отметить, что и для суперскалярных процессоров применение ILP-оптимизаций при компиляции дает существенное повышение

производительности (см. [64], [67]). Повышению эффективности исполнения на суперскалярных ЭВМ может способствовать также встраивание избыточной информации о программе, доступной во время компиляции и позволяющей процессору динамически производить дополнительные оптимизации. Пример применения этого подхода можно найти в [59].

VLIW-процессор способен работать с большей эффективностью, чем суперскалярный, поскольку у него нет необходимости заниматься динамическим анализом кода. Суперскалярный процессор, тем не менее, превосходит его в качестве планирования команд, поскольку имеет больше информации. Так, при статическом анализе невозможно предсказать случаи непопадания в кэш при чтении из памяти, из-за чего при выполнении возможны простои, в то время как динамический планировщик в этом случае может запустить другие готовые к исполнению команды. Компилятор не имеет права поменять местами команду чтения из памяти с последующей командой записи в память, поскольку адрес записи, возможно, совпадает с адресом чтения. Динамическому планировщику эти адреса уже известны, следовательно, он обладает большей свободой переупорядочения команд. Еще одно преимущество суперскалярных процессоров заключается в поддержке механизма предсказания ветвлений (branch prediction) и выполнения по прогнозу ветвления (control speculation). Аппаратура выбирает направление ветвления исходя из частоты предыдущих ветвлений в этой точке и с упреждением исполняет команды из более вероятной ветви. Это дает ускорение, если прогноз был верен. При неверном прогнозе аппаратура аннулирует результаты упреждающих вычислений.

Концепция явного параллелизма на уровне команд (EPIC - Explicitly Parallel Instruction Computing) возникла из стремления объединить преимущества двух типов архитектур. Идеология EPIC заключается в том,

чтобы, с одной стороны, полностью возложить составление плана выполнения команд на компилятор, с другой стороны, предоставить необходимые аппаратные средства, позволяющие при статическом планировании на стадии компиляции использовать механизмы, подобные тем, которые применяются при динамическом планировании в суперскалярных архитектурах (см. [18]).

В разд. 2.6.4 и 2.6.6 рассматриваются некоторые характерные для EPIC-архитектур аппаратные средства

- поддержка упреждающего выполнения команд на основе прогноза направления ветвления (control speculation);
- поддержка выполнения по прогнозу данных (data speculation);
- поддержка условного выполнения (predicated execution)

и их использование при статическом планировании.

С точки зрения применимости различных методов оптимизации существенно различие между VLIW-процессорами с регулярной и нерегулярной организацией командного слова. Последние отличаются тем, что на параллельно исполняемые инструкции налагаются дополнительные ограничения. В частности, для параллельно исполняемой инструкции могут допускаться в качестве операндов не все сочетания регистров или не все способы адресации, которые возможны в такой же инструкции, если параллельно с ней не закодированы другие инструкции. Такие особенности характерны, в частности, для цифровых процессоров обработки сигналов, где, в целях ускорения выборки команд, а также для сокращения общего размера кода и энергопотребления, проектировщики стремятся минимизировать длину командного слова и используют для этого нерегулярные способы кодирования. Нерегулярная организация командного слова (см. например, [61]) и связанные с ней ограничения параллелизма исполнения, называемые ограничениями кодирования,

существенно усложняют задачу генерации эффективного кода при компиляции ([50],[68]).

Еще одна разновидность ILP-архитектур - кластерные архитектуры, где функциональные устройства поделены на группы (кластеры), и с каждым кластером связан набор локальных регистров, недоступных для функциональных устройств других кластеров (см. [63]). На рис. 2.2 изображен пример кластерной архитектуры.

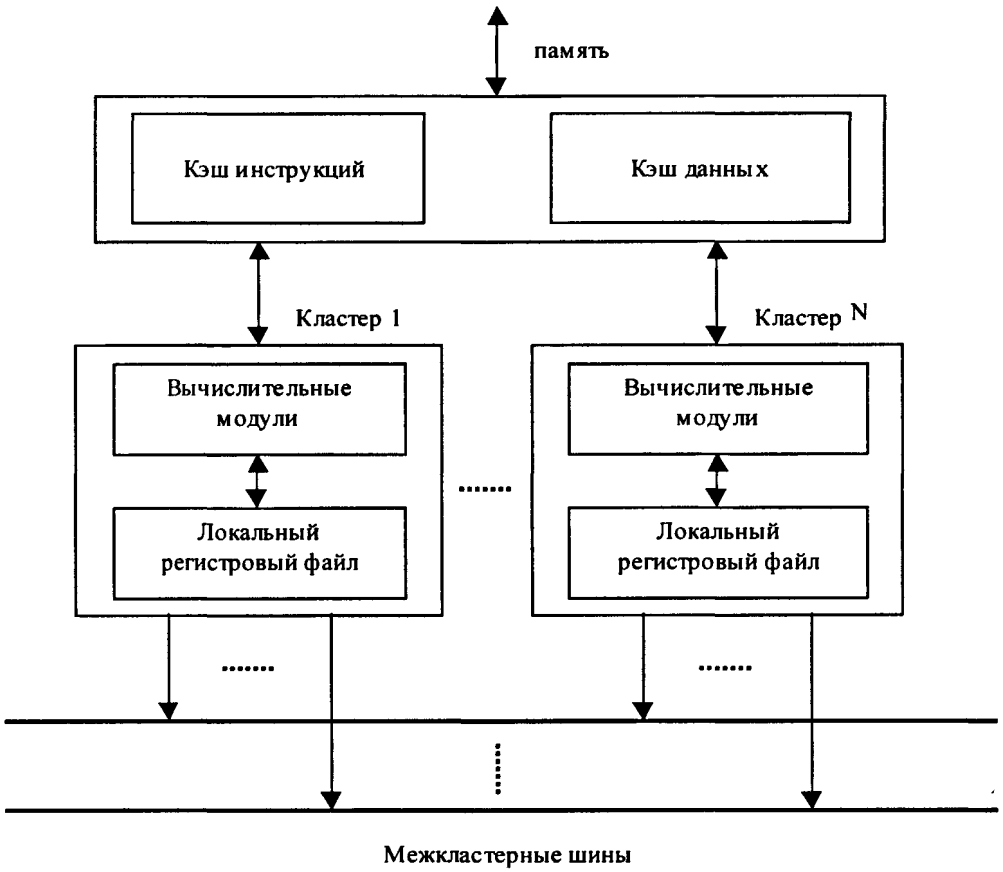


Рис. 2.2. Кластерная архитектура

Специфическая проблема, возникающая при генерации кода для кластерных архитектур - минимизация обменов данных между регистровыми файлами разных кластеров.

2.2. Критерии оптимизации кода

Подходы, используемые при оптимизации кода, могут существенно зависеть от критериев оптимизации. Обычно рассматривают три критерия или их комбинации с некоторыми приоритетами:

- минимизация времени выполнения программы;
- минимизация размера кода;
- минимизация энергопотребления.

Последний критерий существен при компиляции приложений для встроенных автономных систем. Размер кода, как правило, имеет второстепенное значение. Далее в основном будет рассматриваться критерий минимизации времени выполнения с учетом возможных ограничений на размер кода.

Локальные методы оптимизации, применяемые в пределах линейных участков, обычно направлены на сокращение одновременно и времени выполнения, и размера кода. Методы реорганизации кода (такие как развертка циклов, встраивание функций и др. - см. разд., 2.5.1, 2.5.3), направлены на ускорение работы компилируемой программы ценой увеличения размера выходного кода.

Возможны и другие, более специальные критерии и ограничения. Например, в работах [48] и [49] рассматривается метод планирования инструкций в условиях, когда для некоторых из них заданы начальные и/или конечные времена $Tmin_i$, $Tmax_i$, так что инструкция i должна сработать не позднее момента $Tmax_i$ и не ранее момента $Tmin_i$. Подобные ограничения характерны для систем реального времени, где определенные действия должны совершаться в пределах заданных временных интервалов.

Фактор скорости компиляции, по мнению многих авторов ([50], [54], [67] и др.), для ILP-процессоров следует считать второстепенным. В особенности это справедливо в контексте компиляции для ЦПОС. С одной стороны, генерация оптимального кода для них существенно затрудняется из-за ограничений параллельного исполнения, с другой стороны, эффективность результирующего кода для них имеет гораздо более важное значение, чем скорость компиляции.

2.3. Круг проблем, связанных с оптимизацией кода для ILP-процессоров

Прежде чем перейти к рассмотрению основных задач, относящихся к ILP-оптимизации, рассмотрим в общих чертах схему работы компилятора, которая представлена на рис. 2.3 (см., например, [9],[10]). Компилятор для ILP-процессора объединяет в себе стандартные механизмы компиляции, имеющие смысл для всех целевых платформ, и специализированные методы анализа и оптимизации, направленные на выявление, усиление и использование параллелизма на уровне команд.

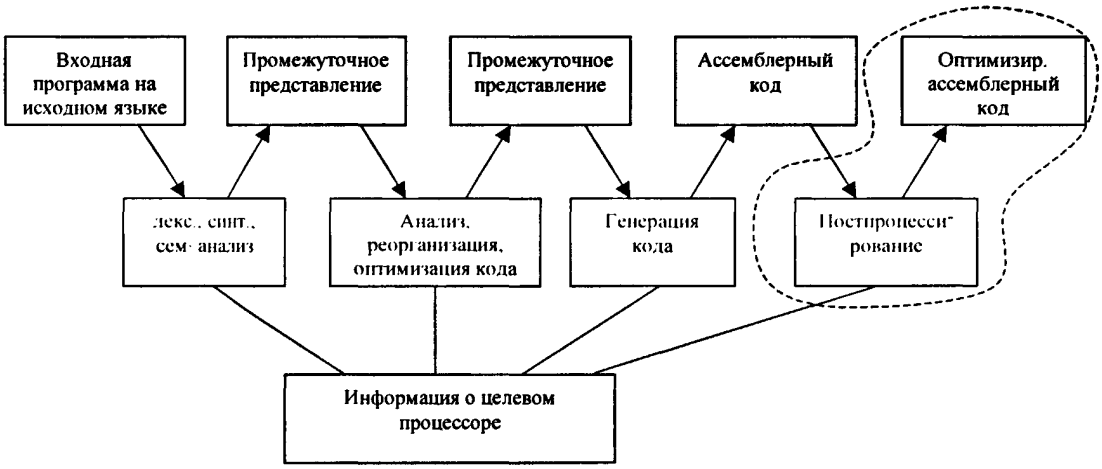


Рис. 2.3. Примерная схема компиляции; постпроцессирование – необязательный этап

На первом этапе проводится лексический, синтаксический и семантический анализ программы на входном языке и строится ее промежуточное представление.

В качестве промежуточного представления может использоваться, например, список, элементы которого соответствуют элементарным инструкциям реальной или гипотетической машины. Элементы промежуточного представления содержат информацию об операндах инструкции, о ее связях с другими инструкциями и т.п. В качестве элементов могут фигурировать также вспомогательные сущности, например, отметки о начале и конце циклов, метки и т.п.

Затем проводятся оптимизации в терминах промежуточного представления. Примеры стандартных оптимизаций, поддерживаемых большинством современных компиляторов, - удаление избыточного кода, свертка константных вычислений, выделение общих подвыражений, вынесение инвариантных вычислений из циклов, понижение мощности операций и др. [70]. В ILP-компиляции особое внимание уделяется методам усиления программного параллелизма в телах циклов, которые подробно рассматриваются в разд. 2.5.

В контексте ILP наибольший интерес представляет оптимизирующее преобразование, называемое планированием. В ходе планирования последовательность элементарных команд, сформированная традиционными методами компиляции, переупорядочивается, и команды группируются таким образом, чтобы обеспечить максимально быстрое параллельное исполнение. При этом учитываются связи между командами по данным и по управлению, а также аппаратные возможности параллельного исполнения команд. В применении к компиляции для VLIW-процессоров данное преобразование кода называют также распараллеливанием (code parallelization) или сжатием (code compaction).

Оптимизированное промежуточное представление преобразуется в ассемблерный код.

Применяются также (см. [56]) оптимизации на уровне ассемблерного кода (постпроцессирование). В ходе постпроцессирования кода, сгенерированного при помощи универсального компилятора, выполняются машинно-зависимые оптимизации. Такой подход позволяет ускорить создание оптимизирующего компилятора для нестандартной целевой платформы.

Существенной характеристикой большинства реализаций, как промышленных, так и экспериментальных, является настраиваемость компонентов компилятора на свойства и систему команд целевого процессора.

Перечислим коротко основные методы анализа, реорганизации и оптимизации кода, применяемые в ILP-компиляторах. Более подробно они рассматриваются в последующих разделах.

1. Выделение областей планирования. Область планирования - это фрагмент или множество фрагментов программы, в пределах которых применяется алгоритм планирования. В простейшем случае в качестве таких областей используются линейные участки в смысле [1] или [7] -

последовательности команд, содержащие не более одной метки (в начале) и не более одной команды перехода (в конце). Однако в пределах линейного участка не всегда можно найти достаточно команд, способных исполняться параллельно. Поэтому разработчики компиляторов стремятся выделить более крупные области планирования, объединяющие несколько линейных участков. Различные типы областей планирования рассматриваются в разделе 2.4.

2. Реорганизации кода, направленные на удлинение линейных участков и расширение областей планирования - преобразования циклов, встраивание функций и др., см. разделы 2.5.1, 2.5.2.

3. Усиление параллелизма в пределах выделенных областей. Поскольку параллельное исполнение инструкций возможно только при условии их независимости по данным, то в пределах областей проводятся реорганизации кода, направленные на частичное снятие зависимостей по данным между инструкциями - переименование регистров, исключение индуктивных переменных в циклах и др. Наиболее эффективны эти реорганизации в применении к телам развернутых циклов. Эти вопросы рассматриваются в разделе 2.5.3.

4. Планирование команд в пределах выделенных областей. Различают методы локального планирования (в пределах линейных участков) и глобальное планирование (в пределах расширенных областей), где применяется перемещение команд между линейными участками с использованием аппаратных и программных средств для сохранения корректности программы. Планированию команд посвящен раздел 2.6.

2.4. Области планирования

В традиционных компиляторах планирование, как правило, осуществляется в пределах линейных участков [2]. Однако для ILP-

процессоров такой подход может приводить к потерям производительности. Характерная частота переходов в программах нечисленных приложений, например, составляет примерно 20%, т.е. средняя длина линейного участка - 5 команд. С учетом связей по данным, которые вероятнее всего присутствуют между этими командами, степень естественного программного параллелизма оказывается невысокой. Для того чтобы привести степень программного параллелизма в соответствие с уровнем имеющегося аппаратного параллелизма, в компиляторах для ILP-процессоров реализуют планирование в рамках более широких областей кода, объединяющих несколько линейных участков, так что инструкции могут в результате перемещаться из одного участка в другие. При этом обычно стремятся максимально ускорить выполнение вдоль наиболее часто исполняемых ветвей программы. Надо заметить, что подавляющая часть из доступных экспериментальных результатов, подтверждающих преимущества глобального планирования по сравнению с локальным, относятся к приложениям нечисленного характера. Эффективность глобального планирования в компиляции численных приложений требует дополнительных исследований.

Для того чтобы перемещения инструкций между линейными участками были корректны, применяются определенные приемы, ограничения и аппаратные средства, которые рассматриваются в разд. 2.6.3, 2.6.4. В этом разделе будут рассмотрены типы областей, для которых выработаны эффективные методы планирования, а также способы построения областей.

Введем два понятия, которые используются в определениях областей: *точка слияния* - команда, на которую управление может прийти более чем из одного места; *точка ветвления* - команда условной передачи управления.

Область планирования состоит из одного или более линейных

участков, которые в исходной программе могут быть расположены последовательно или произвольно. Области различаются по структуре своего потока управления и по способу формирования. Наиболее известные типы областей - суперблоки, трассы, гиперблоки, древовидные области и регионы - имеют два общих признака: ациклический граф управления и один головной участок, из которого достижимы все остальные.

Ниже перечислены типы областей и их основные характеристики:

Суперблок [38], [44]

- может содержать только одну точку слияния - точку входа в начале головного линейного участка;
- имеет прямолинейный граф управления. Команды ветвления могут передавать управление в другие суперблоки, но не на команды того же суперблока.

Трасса [35], [36], [38] отличается от суперблока тем, что может содержать более одной точки слияния.

Гиперблок [58] - суперблок, который может включать условно исполняемые участки. Метод гиперблоков эффективен для процессоров, поддерживающих условное выполнение.

Древовидная область (treegion) [25], [40], [41], [43], имеет древовидный граф управления и включает не более одной точки слияния (в начале головного участка). Древовидные области могут формироваться путем реорганизации входной программы; при этом также могут использоваться данные профилирования.

Регион [27], [29] - область с произвольным ациклическим графом управления. Отличительная черта метода регионов - поддержка вложенных регионов (например, внутренних циклов). Метод регионов применяется, в частности, в компиляторе для IA-64 [29], где его

реализация существенно опирается на аппаратные средства поддержки параллелизма.

Одна из идей, на которой основываются методы глобального планирования, заключается в том, что код можно реорганизовать таким образом, чтобы сократить время выполнения вдоль одних путей за счет замедления вдоль других. Если решения принимаются в пользу ускорения наиболее частых путей, то за счет этого можно достичь сокращения времени выполнения программы в целом. Такой подход может быть неприемлем в приложениях реального времени, где возможны ограничения на время выполнения вдоль любого, даже самого редкого пути исполнения [67].

При формировании областей используются данные профилирования по частоте выполнения переходов, что делает актуальной задачу эффективного получения данных профилирования. В работе [34] предлагается экономный метод профилирования передач управления для ILP-процессоров. Метод не требует аппаратной поддержки и основан на добавлении минимального необходимого числа дополнительных линейных участков, содержащих зондирующий код для регистрации передач управления. Зондирующий код организуется таким образом, чтобы при выполнении обеспечивалось его максимальное распараллеливание.

Рассмотрим более подробно способы формирования двух типов областей - суперблоков и древовидных областей.

Суперблоки

Понятие суперблока соответствует определению расширенного линейного участка. Расширенный линейный участок есть последовательность линейных участков $B_1 \ldots B_k$, такая что для $1 \leq i < k$ B_i - единственный предшественник B_{i+1}. Отличительная черта суперблоков заключается в способах их формирования. С учетом данных

профилирования, точки слияния в исходной программе удаляются путем создания копий соответствующих участков. При этом стремятся выделить суперблоки, расположенные вдоль трасс - наиболее часто исполняемых путей на графе управления. Пример формирования суперблока из [44] приведен на рис. 2.4.

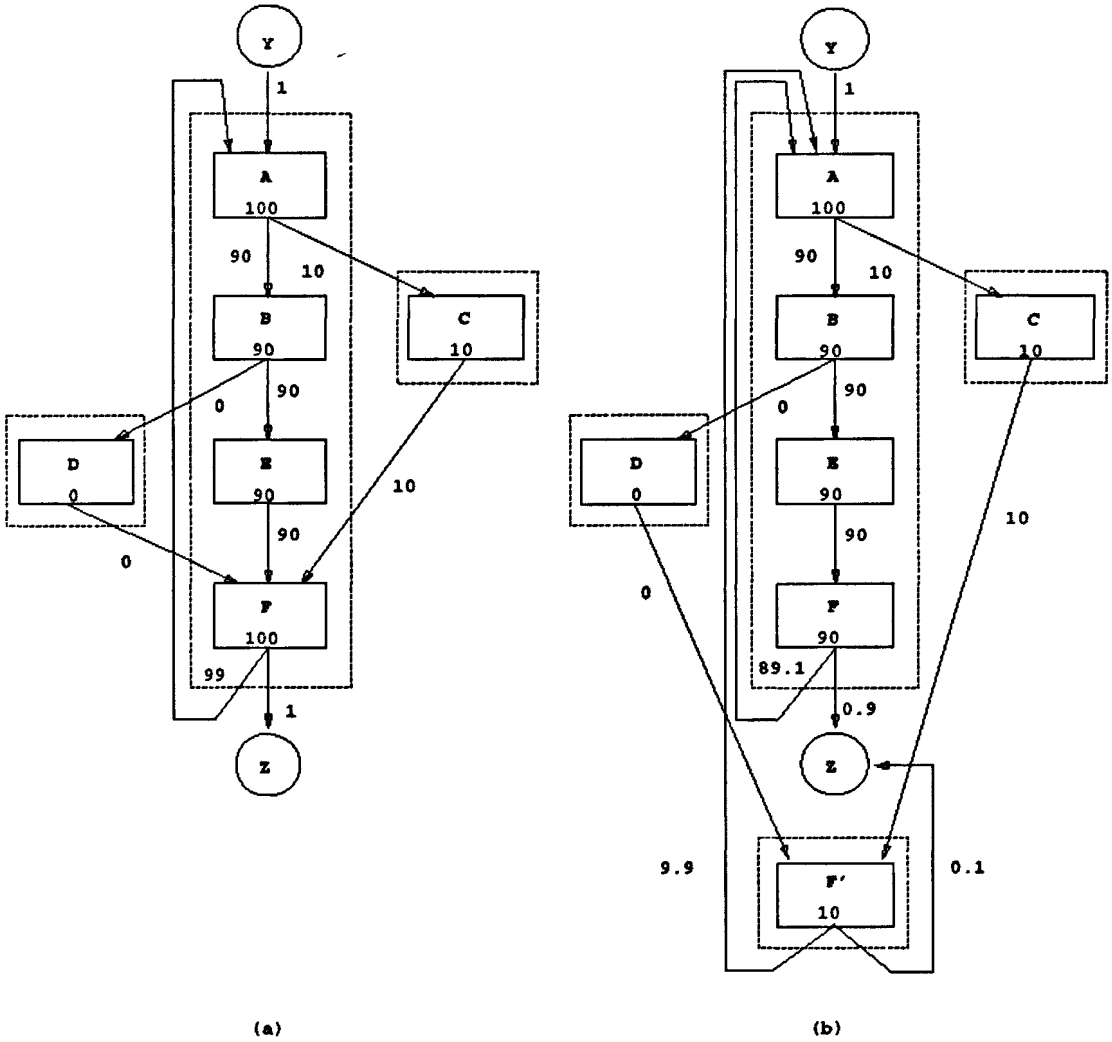


Рис. 2.4. Формирование суперблоков на основе данных профилирования

На рис. 2.4а показан граф управления для программного фрагмента, составляющего тело цикла, с указанием частот выполнения участков и переходов между ними. Из этой схемы видно, что наиболее часто выполнение следует вдоль пути A→B→E→F. Поэтому принимается

решение сформировать три суперблока: {A,B,E,F}, {C}, {D}. Для этого необходимо исключить точку слияния в F. На рис. 2.4б показано, как это достигается путем добавления копии F (F'). Этот прием называют "дублированием хвостов" (tail duplication). В конечном счете, из исходного программного фрагмента создается 4 суперблока: {A,B,E,F}, {C}, {D}, {F'}.

Древовидные области

Формирование древовидных областей проводится в два этапа. Сначала на основе статического анализа в графе управления выделяются имеющиеся древовидные участки. Далее, если доступны данные профилирования, выделенные участки искусственно наращивают методом "дублирования хвостов". При этом стремятся объединить участки вдоль наиболее часто исполняемых путей.

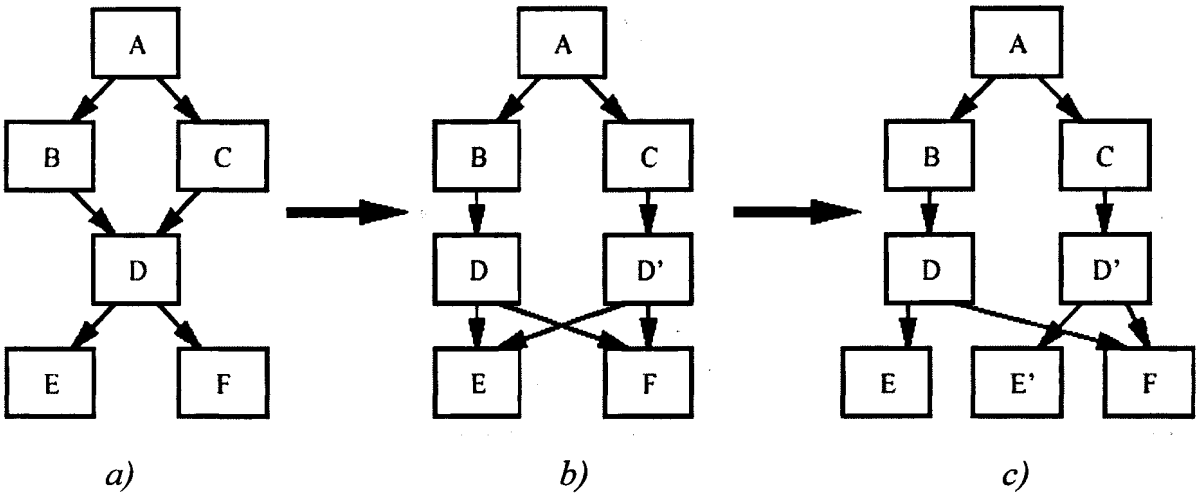


Рис. 2.5. Древовидная область

На рис. 2.5 приведен пример из [41], где показано наращивание первоначально выделенной области. Исходный программный фрагмент состоит из двух древовидных областей (а). Если исполнение преимущественно следует вдоль A→B→D→E, то желательно реорганизовать код, чтобы путь A→B→D→E попал в общую область, и

планировщик мог максимально использовать параллелизм на этом отрезке. На рис. 2.5b и рис. 2.5c показаны два этапа такого преобразования. Сначала создается копия D' участка D и формируется область, включающая путь A→B→D. Затем создается копия E' участка E и формируется область, включающая пути A→B→D→E и A→C→D'→E', а также область, состоящая из одного участка F.

Данные профилирования могут использоваться также на этапе планирования в древовидных областях, для того чтобы обеспечить максимально быстрое выполнение (и исключить задержки) преимущественно вдоль часто исполняемых путей.

Для того чтобы ограничить объем результирующей программы, при принятии решений о "дублировании хвостов", помимо данных профилирования, применяются и другие эвристики (см. [40]):

- допустимый общий коэффициент расширения не должен превышать некоторой заранее заданной величины;
- число путей исполнения в каждой древовидной области не должно превышать заданной величины;

если число предшественников участка в графе управления больше заданной величины, то дублирование участка не производится.

Аналогичные эвристики используются и при формировании областей других типов.

В [11] можно найти описание метода проникающего планирования (percolation scheduling), предполагающего глобальное переупорядочение кода для выявления параллелизма на уровне тела функции.

2.5. Усиление параллелизма в пределах областей планирования

Большинство из рассматриваемых в этом разделе методов применимы

в той или иной степени ко всем типам ILP-процессоров и видам областей планирования.

2.5.1. Преобразования циклов

Преобразования циклов, применяемые в ILP-компиляции, подробно рассмотрены в [44] и [67]. К ним относятся: развертка циклов, слияние и разбивка циклов, подгонка циклов, конвейеризация циклов. Все они имеют смысл независимо от наличия параллелизма в целевом процессоре, поскольку позволяют уменьшить общее число проверок завершения цикла и операций перехода. В компиляции для ILP-процессоров они приобретают дополнительную значимость, поскольку позволяют усилить программный параллелизм в теле цикла.

В примерах, иллюстрирующих смысл преобразований, использован язык Си, реально же они применяются на уровне промежуточного представления.

Развертка цикла (loop unrolling). Суть этого преобразования заключается в том, что тело цикла дублируется *n* раз, а число повторений соответственно сокращается во столько же раз (рис. 2.6). Число *n* называется коэффициентом развертки цикла.

```
for (i=0;i<100;i++)          for (i=0;i<100;i=i+4) {
  {a[i]=a[i]+c;}               a[i]=a[i]+c;
                     ==>       a[i+1]=a[i+1]+c;
                               a[i+2]=a[i+2]+c;
                               a[i+3]=a[i+3]+c;}
```

Рис. 2.6. Развертка циклов

В контексте ILP-компиляции он приобретает большее значение, поскольку позволяет использовать параллелизм команд, относящихся к разным итерациям цикла. Наиболее эффективно его применение в сочетании с другими преобразованиями, направленными на усиление

параллелизма (см. рис. 2.12).

Слияние циклов (loop fusion). Два расположенных последовательно цикла можно слить, если они имеют одинаковое число итераций и отсутствуют зависимости по данным, препятствующие объединению. Если тела сливаемых циклов не зависят друг от друга (рис. 2.7), появляется возможность спланировать параллельное выполнение команд, относящихся к разным циклам.

```
for (i=0;i<100;i++)              for (i=0;i<100;i++) {
  b[i]=b[i]+c;        ==>          b[i]=b[i]+c;
for (j=0;j<100;j++)                a[i]=a[i]*2;
  a[j]=a[j]*2;                   }
```

Рис. 2.7. Слияние циклов

Если граничные значения переменных двух циклов различаются, но ненамного, то применяют слияние с предварительной подгонкой одного из циклов.

Подгонка цикла (loop peeling). Подгонка цикла заключается в изменении граничных значений переменной цикла. Обычно подгонка применяется для того чтобы можно было выполнить слияние (рис. 2.8) или развертку цикла (если число итераций не кратно коэффициенту развертки).

```
for (i=0;i<100;i++)              for (i=0;i<100;i++) {
  b[i]=b[i+2]+c;      ==>          b[i]=b[i+2]+c;
for (j=0;j<102;j++)                a[i]=a[i]*2;}
  a[j]=a[j]*2;                   a[100]=a[100]*2;
                                 a[101]=a[101]*2;
```

Рис. 2.8. Слияние циклов с подгонкой одного из них

Программная конвейеризация цикла (software pipelining). Идея конвейеризации цикла заключается в том, что выполнение команд, относящихся к последующим итерациям, начинается раньше, чем завершается выполнение предшествующих итераций. Конвейеризация

применима в тех случаях, когда тело цикла можно разбить на группы команд, не зависящих друг от друга на разных итерациях.

На рис. 2.9 показан пример конвейеризации цикла. Команды, относящиеся к одной итерации исходного цикла, не могут выполняться параллельно в силу зависимостей по данным. Тело результирующего цикла составлено из команд, относящихся к трем смежным итерациям (i, i+1, i+2) и не зависящих друг от друга, так что их выполнение может быть спланировано параллельно. Число итераций, участвующих в конвейерном выполнении цикла, называется глубиной конвейеризацией (по аналогии с аппаратной конвейеризацией). Число итераций конвейеризованного цикла сокращается на n-1, где n - глубина конвейеризации, а в пролог и эпилог выносятся команды, относящиеся к начальным и завершающим итерациям исходного цикла.

```
                                a[0]=b[0]+2;
                                a[1]=b[1]+2;
                                d[0]=a[0]/n;
for (i=0;i<100;i++){            for (i=0;i<98;i++){
  a[i]=b[i]+2;                    f[i]=d[i]+a[i];
  d[i]=a[i]/n;          ==>       d[i+1]=a[i+1]/n;
  f[i]=d[i]+a[i];}                a[a+2]=b[i+2]+2;}
                                d[99]=a[99]/n;
                                f[98]=d[98]+a[98];
                                f[99]=d[99]+a[99]];
```

Рис. 2.9. Конвейеризация цикла

Конвейеризация, как и развертывание цикла, создает возможности для параллельного выполнения команд из разных итераций, но обладает тем преимуществом, что не увеличивает размер тела цикла.

Обзор методов конвейеризации циклов можно найти в работах [11], [17].

Разбивка циклов (loop distribution). В некоторых случаях может иметь смысл преобразование, обратное слиянию и называемое разбивкой циклов.

Это целесообразно, например, если тело цикла слишком длинное, и имеющееся число регистров недостаточно для размещения всех используемых в теле цикла переменных. В этом случае часть промежуточных значений приходится временно выгружать в память, а перед использованием в вычислениях загружать на регистры (в англоязычной литературе этот процесс обозначают термином register spilling). Благодаря разбивке цикла можно избежать дефицита регистров и выталкивания значений в память.

В примере, показанном на рис. 2.10, вторая команда не может быть выполнена параллельно с первой в силу зависимости по данным. В результате разбивки создаются циклы с более короткими телами и меньшим числом зависимостей по данным.

```
for (i=0;i<100;i++){                    for (i=0;i<100;i++)
  b[i]=b[i-1]+c;            ==>           b[i]=b[i-1]+c;
  a[i]=b[i]+2;}                         for (i=0;i<100;i++)
                                          a[i]=b[i]+2;
```

Рис. 2.10. Разбивка циклов

2.5.2. Встраивание функций

Встраивание функций широко применяется как метод оптимизации и в традиционных технологиях компиляции, поскольку при этом экономится время, затрачиваемое на передачу параметров, выполнение пролога и эпилога. В контексте ILP-компиляции этот подход имеет дополнительное преимущество, поскольку он позволяет спрямлять пути исполнения, создавая более длинные линейные участки и дополнительные возможности для распараллеливания кода.

2.5.3. Снятие зависимостей по данным

Алгоритмы планирования команд, используемые практически во всех компиляторах (см. разделы 2.6.1 и 2.7), работают с тремя видами

зависимостей (или связей) по данным между командами (см., например, [67] или [54]). Здесь будут рассмотрены только зависимости по обращениям к регистрам; о зависимостях по обращениям к памяти см. разд. 2.6.6.

Пусть имеется некоторая последовательность команд c_1, ..., c_n, подлежащих планированию. Планировщик может изменять порядок выполнения команд, не нарушая их частичной упорядоченности, которая определяется перечисленными далее зависимостями по данным. (При глобальном планировании планировщик также должен учитывать связи по управлению, препятствующие перемещению команд через точки ветвления; подробнее об этом см. разделы 2.6.3, 2.6.4).

1. Связи типа "чтение после записи". Команда c_j зависит от c_i, если c_i записывает значение в некоторый регистр r, а c_j читает это значение. Будем обозначать это отношение как $c_i \xrightarrow{\text{RAW}} c_j$.

Зависимости этого типа называются истинными, поскольку они отражают объективные связи по данным между операциями, реализующими компилируемую программу: выполнение c_j должно планироваться позже, чем выполнение c_i. Как правило, избавиться от них нельзя, однако существуют приемы, позволяющие сделать это в некоторых специальных случаях: дублирование переменной суммирования и индуктивных переменных, рассматриваемые далее в этом разделе.

2. Связи типа "запись после записи". Команда c_j зависит от c_i, если

- обе команды записывают значения в некоторый регистр r
- $j > i$
- имеется хотя бы одна команда c_k, которая читает значение r, записанное командой c_i.

Будем обозначать это отношение как $c_i \xrightarrow{\text{WAW}} c_j$. Выполнение команды c_j должно быть запланировано позже чем c_i, если имеет место

$c_i \xrightarrow{\text{WAW}} c_j$.

3. Связи типа "запись после чтения". Команда c_j зависит от c_i, если существует команда c_k, такая что имеет место $c_k \xrightarrow{\text{RAW}} c_i$ и $c_k \xrightarrow{\text{WAW}} c_j$. Будем обозначать это отношение $c_i \xrightarrow{\text{WAR}} c_j$. Смысл этой зависимости в том, что c_i читает некоторый регистр r, записанный ранее командой c_k, а c_j ($j > i$) записывает в r другое значение. Если имеет место $c_i \xrightarrow{\text{WAR}} c_j$, то c_j может быть выполнена не раньше c_i (т.е. одновременно или позднее c_i).

Последние два типа связей называют антизависимостями или ложными зависимостями, поскольку они возникают в результате того, что компилятор использует одни и те же регистры для хранения разных значений (или программист использует одни и те же рабочие переменные для хранения различных промежуточных значений).

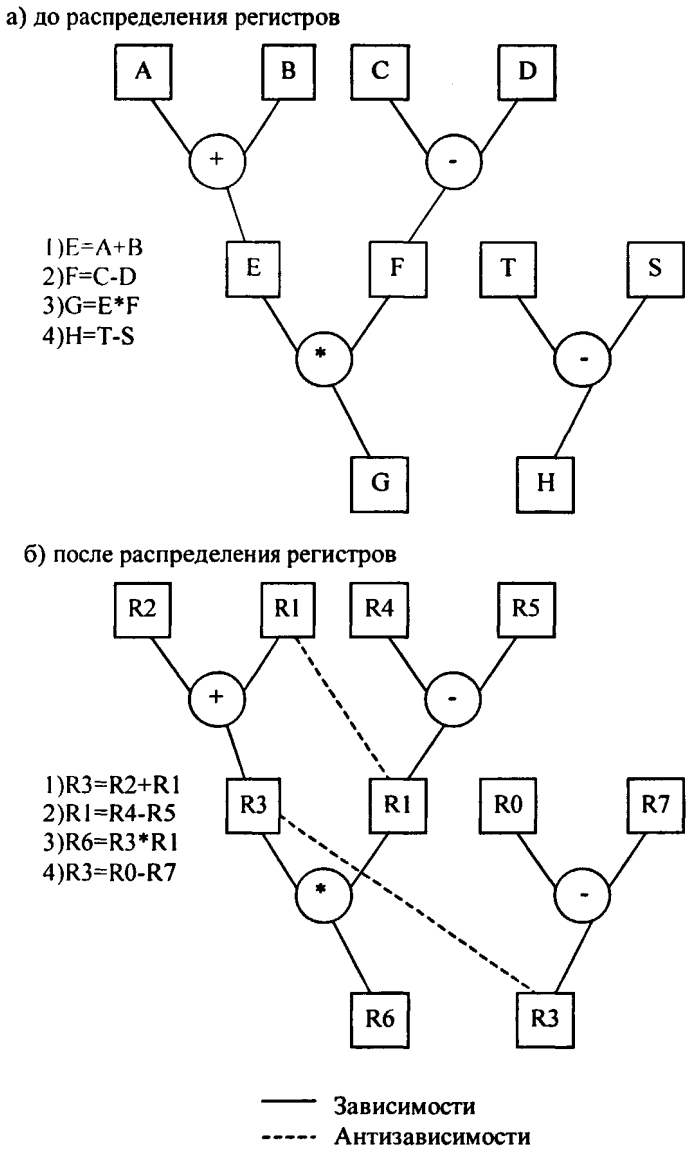


Рис. 2.11. Зависимости по данным (a) до распределения регистров, (b) после распределения регистров. После распределения регистров появляются антизависимости WAR (R3=R1+R2,R1=R4-R5) по R1 и WAW(R3=R1+R2,R3=R0-R7) по R3

На рис. 2.11 представлены примеры зависимостей всех типов и показано, как образуются антизависимости.

Зависимости по данным препятствуют параллельному исполнению команд и их переупорядочению при планировании. В случае, показанном

на рис. 2.11а), возможно параллельное выполнение команд (1,2,4) или (3,4). После распределения регистров (рис. 2.11б), антизависимости жестко определят порядок выполнения команд – 1), 2), 3), 4). Поэтому важно по возможности избавляться от них. Большинство из перечисленных далее преобразований направлено на снятие антизависимостей по данным внутри областей планирования.

Миграция команд. Если результат команды не используется в данной области, то ее можно переместить в области, которые выполняются реже. Для суперблоков этот метод применяется следующим образом [44]. Если результат операции не используется в суперблоке *S*, то она может быть удалена из *S*, при этом ее копии создаются во всех суперблоках, на которые управление может быть передано из *S*. При этом исключаются суперблоки, в которых результат заведомо не используется. Решение принимается с учетом оценок частот выполнения суперблоков. В результате в *S* исключаются все зависимости по данным, связанные с этой операцией.

Переименование регистров. Суть этого приема заключается в том, чтобы размещать разные значения в разных регистрах. Разумеется, его практическое применение ограничено числом доступных регистров.

Дублирование индуктивной переменной. Индуктивные переменные это переменные, представляющие собой выражения, линейно зависящие от переменной цикла, например, адресные выражения для доступа к элементам массива. В развернутом цикле при вычислении индуктивных переменных возникают зависимости по данным. В результате оказывается невозможным распараллеливание вычисления индуктивных переменных и доступа к памяти по ним. Положение можно исправить, если завести несколько экземпляров индуктивной переменной (в соответствии с коэффициентом развертки цикла).

Дублирование переменной суммирования. Если в цикле производится

суммирование или перемножение выражений, то при развертке цикла можно создать несколько экземпляров переменной суммирования для накопления частичных сумм или произведений [44]. В эпилоге цикла частичные суммы или произведения, соответственно, складываются или перемножаются. Этот прием применим к любой операции, обладающей свойствами коммутативности и ассоциативности.

На рис. 2.12 показано применение развертки цикла в сочетании с оптимизациями снятия зависимостей - переименованием регистров, дублированием переменной суммирования и индуктивной переменной. Код, полученный непосредственно после развертки, слабо поддается распараллеливанию из-за большого числа зависимостей по данным. В результате снятия зависимостей получается тело цикла, выполнение которого на идеальном процессоре (с неограниченными возможностями параллельного исполнения, без задержек) занимает 2 такта.

```
Исходный цикл:
  s=0;
  for (i=0;i<100;i++)
    {s=s+a[i];}

Ассемблерный код
      r1 = A
      r3 = 0
  L1: r2 = MEM(r1)
      r3 = r3 + r2
      r1 = r1 + 4
      blt (r1 A+4N) L1

Результат развертки с коэффициентом 3:
      r1 = A
      r3 = 0
  L1: r2 = MEM(r1)
      r3 = r2 + r3
      r1 = r1 + 4
      r2 = MEM(r1)
      r3 = r2 + r3
      r1 = r1 + 4
      r2 = MEM(r1)
      r3 = r2 + r3
      r1 = r1 + 4
      blt (r1 A+4N) L1

Результат снятия зависимостей по данным:
      r11 = A            ;; Размножение индуктивной
      r21 = A + 4        ;; переменной
      r31 = A + 8
      r3 = 0             ;; Размножение переменной
      r23 = 0            ;; суммирования
      r33 = 0
  L1: r2 = MEM(r11)
      r3 = r2 + r3
      r11 = r11 + 12
      r22 = MEM(r21)     ;; Переименование r2 -> r22
      r23 = r22 + r23
      r21 = r21 + 12
      r32 = MEM(r31)     ;; Переименование r2 -> r22
      r33 = r32 + r33
      r31 = r31 + 12
      blt (r1 A+4N) L1
      r3 = r3 + r23      ;; Сложение частичных сумм
      r3 = r3 + r33
```

Рис. 2.12. Развертка цикла и снятие зависимостей по данным

2.5.4. Соотношение программного и аппаратного параллелизма

Рассмотренные выше методы оптимизации направлены на усиление программного параллелизма на уровне команд, с тем чтобы максимально использовать имеющиеся в процессоре средства параллельного исполнения. Важный момент, который необходимо учитывать при их

практической реализации, - соотношение между фактическим уровнем аппаратного параллелизма целевого процессора и уровнем программного параллелизма. Набор оптимизаций и их параметры (такие как коэффициент развертки) следует соразмерять с аппаратными характеристиками, в первую очередь, с реальными возможностями параллельного исполнения команд и количеством доступных регистров. Например, дублирование переменной суммирования может не иметь смысла, если процессор не способен выполнять одновременно несколько сложений. Развертка цикла может привести к деградации производительности, если регистров недостаточно для размещения всех переменных увеличившегося тела цикла. В этом случае компилятор вынужден размещать переменные в памяти, и использовать дополнительные команды для загрузки их на регистры перед выполнением операций и выгрузки в память при изменении значений, что может свести на нет эффект усиления параллелизма (см. [26], [47]).

2.6. Планирование команд

2.6.1. Алгоритмы планирования

После того как области планирования выделены и проведены оптимизации снятия зависимостей по данным, выполняется планирование команд. Считается, что на этом этапе для каждой области известны ее входные (вычисленные ранее) и выходные (используемые вне данной области) объекты.

В ILP-компиляции в основном применяются различные варианты приоритетного (эвристического) списочного планирования (list scheduling). Процесс планирования состоит из трех шагов:

1. Вычисление зависимостей по данным и по управлению между

командами в пределах области планирования, которые обычно представляют в виде направленного ациклического графа. Зависимости по управлению препятствуют перемещению команд вокруг точек ветвления. Эти зависимости могут быть в дальнейшем частично сняты, т.е. при планировании команды могут перемещаться выше или ниже точек ветвления (см. далее разд.2.6.3, 2.6.4).

2. Вычисление приоритетов команд. Приоритеты определяют, в каком порядке будут планироваться готовые к исполнению команды на шаге 3. Способ вычисления приоритетов отражает эвристики планирования, от которых может существенно зависеть результат. Как правило, применяется эвристика планирования по критическому пути[1], когда приоритет команды определяется высотой[2] соответствующего узла в графе зависимостей. Команды с совпадающими приоритетами упорядочивают либо произвольным образом, либо по некоторому вторичному признаку. В [40] приводятся результаты исследований нескольких эвристик глобального планирования в древовидных областях:

1) по глубине команды в графе зависимостей;
2) по числу выходов из области по всем путям в дереве управления, исходящим из данной команды (exit count). Согласно этой эвристике, наибольший приоритет получают команды, находящиеся в корне дерева.
3) по частоте выполнения на основе данных профилирования (global weight). Приоритет отдается наиболее часто исполняемым командам, а

---

[1] критический путь - самый длинный путь на графе зависимостей по данным; критический путь не обязательно единственный.

[2] высотой узла называется самый длинный путь на графе зависимостей от этого узла до какого-либо концевого узла. Высота узла характеризует время, необходимое для выполнения всех зависящих от него вычислений на идеальном процессоре с неограниченным параллелизмом выполнения команд.

при равенстве частот команды упорядочиваются по глубине.

4) по взвешенной частоте выходов (weighted count). В качестве первичного критерия используется частота выполнения, а при равенстве частот - счетчик выходов.

Согласно результатам этой работы, наилучшие результаты в среднем дают эвристики 1) и 3); авторы рекомендуют использовать эвристику частоты исполнения, а если данные профилирования отсутствуют, то сортировать команды по глубине в графе зависимостей.

3. На последней фазе рассматриваются (в порядке своих приоритетов) готовые к исполнению команды и выдаются в выходной поток. Если целевой процессор является VLIW-процессором, то выходной поток состоит из длинных командных слов, каждое из которых может содержать несколько команд входного потока. Различаются следующие способы формирования выходного потока [41]:

- Сверху вниз / снизу вверх. В первом случае команда рассматривается после того, как все ее предшественники в графе зависимостей выведены в выходной поток. При планировании снизу вверх команда рассматривается после того как выведены все ее потомки. При глобальном планировании более естественным является метод сверху вниз.
- Перебор команд / заполнение командных слов. В первом случае планировщик на каждом шаге выбирает готовую к исполнению команду с максимальным приоритетом и подыскивает для нее подходящее командное слово в выходном потоке, так чтобы удовлетворялись связи по данным и ограничения по ресурсам (функциональным устройствам). Во втором случае планировщик последовательно заполняет командные слова. Среди готовых к исполнению команд подбираются команды, которые можно выполнить параллельно, и помещаются в очередное слово. Затем происходит заполнение следующего слова и т.д. Второй

подход, по-видимому, предпочтителен для процессоров, где команда может занимать ресурс в течение нескольких тактов, а также для VLIW-процессоров с нерегулярной структурой командного слова.

Иногда применяют дополнительную эвристику, выбирая из множества готовых к исполнению те команды, которые максимально расширят это множество на следующем шаге (см. [13]).

Интересный альтернативный алгоритм планирования, который предложен в работе [16], используется в системе построения компиляторов [9]. Это переборный алгоритм локального планирования, позволяющий находить наилучший по времени план выполнения. Идея его заключается в следующем. Пусть $G=(A,V,E)$ - смешанный граф, где A - множество вершин (операций линейного участка), V - множество дуг, соответствующих зависимостям по данным, E - множество ненаправленных ребер, таких что $[a,b] \in E$, если параллельное исполнение операций a, b невозможно в силу аппаратных ограничений. Из смешанного графа G можно при помощи перебора получить множество ориентированных графов, заменяя каждое из ненаправленных ребер из E на дугу, направленную в ту или другую сторону (если операции a, b невозможно выполнить параллельно, то либо a должна выполниться раньше b, либо наоборот). Каждый ациклический граф, полученный таким образом из G, однозначно определяет план выполнения линейного участка. Среди всех планов выбирается наилучший по времени выполнения.

Время работы алгоритма экспоненциально зависит от длины линейного участка, поэтому автор рекомендует использовать его для оптимизации циклов. Ограничением этого подхода является также предположение о том, что несовместимость команд (невозможность их параллельного исполнения) можно описать в виде бинарного отношения, что не верно, например, если процессор имеет несколько однотипных

функциональных устройств. Возможны ситуации, когда, скажем, команды *a, b, c* несовместимы, хотя любые две из них совместимы.

2.6.2. Координация планирования и распределения регистров

Процедуры планирования и распределения регистров при генерации кода для ILP-процессоров имеют в каком-то смысле конфликтующие цели. Для того чтобы полнее загрузить работой функциональные устройства, планировщик стремится максимально использовать программный параллелизм, а для этого необходимо, чтобы как можно большее число значений находилось на регистрах. При распределении регистров, напротив, необходимо сократить использование регистров, чтобы исключить выталкивание значений в память (а также их сохранение/восстановление при вызовах подпрограмм). Поэтому в компиляторе для ILP-процессора важно обеспечить правильное взаимодействие этих двух механизмов.

Если распределение регистров выполняется до планирования, то это может снизить программный параллелизм из-за введения антизависимостей по данным при переиспользовании регистров. С другой стороны, если планирование выполняется до распределения регистров, то зачастую генерируется код, требующий больше регистров, чем имеется в наличии. В таком случае распределитель регистров вынужден выталкивать часть значений в память и добавлять в программу команды загрузки и выгрузки этих значений, что может привести к деградации выходного кода.

Для решения этой проблемы применяются подходы, включающие повторное планирование, интеграцию обеих процедур или передачу информации между ними.

Повторное планирование. Перед распределением регистров выполняется предварительное планирование. На этом этапе для хранения

каждого значения используется уникальный виртуальный регистр, так что отрицательный эффект антизависимостей по данным исключен. Затем проводится распределение регистров и окончательное планирование, во время которого планируется исполнение дополнительных команд, которые могли быть сгенерированы в ходе распределения регистров ([31], [42]).

Распределение регистров с использованием информации, полученной от планировщика. В [66] описывается модификация классического решения задачи распределения регистров методом раскраски графа (см., например, [12]). Задача раскраски формулируется для графа, в котором представлены не только данные об областях жизни значений, но и информация о возможности параллельного исполнения команд (parallel interference graph). Переиспользование регистров по возможности исключается в тех случаях, когда оно влечет ограничение параллелизма. Информация о возможности параллельного исполнения вычисляется планировщиком. Аналогичный подход представлен в [65].

В [28] представлен алгоритм совместного планирования и распределения регистров, где и регистры, и функциональные устройства рассматриваются как ресурсы.

В [33] и [47] предлагаются специальные алгоритмы планирования для развернутых циклов, обеспечивающие контроль числа требуемых регистров, с тем чтобы исключить или минимизировать обмены с памятью.

В компиляторе для архитектуры TriMedia [41] реализован комбинированный подход, где распределение глобальных регистров осуществляется обычным методом раскраски графа, а распределение локальных регистров (для областей жизни, не выходящих за границы линейных участков) осуществляется непосредственно планировщиком, который постоянно отслеживает уровень дефицита регистров (register pressure) и динамически перевычисляет приоритеты команд. Как только уровень дефицита регистров превышает некоторый порог, приоритеты

команд пересчитываются таким образом, чтобы преимущество отдавалось командам, выполнение которых приведет к снижению дефицита. При низком уровне дефицита регистров увеличивается приоритет команд, которые открывают новые области жизни, что повышает возможности распараллеливания кода (чем больше значений находится на регистрах, тем больше команд, готовых к исполнению).

При генерации кода для кластерных архитектур возникает проблема минимизации обмена данных между регистровыми файлами разных кластеров. Решение этой проблемы предлагается в [63], где планирование совмещается с назначением кластеров. Аналогичный подход используется в [52].

2.6.3. Глобальное планирование

Преимущество глобального планирования заключается в том, что оно применяется к более крупным фрагментам программы, состоящим из нескольких линейным участкам. В результате появляется возможность более эффективного использования аппаратных средств параллельного исполнения. Для того чтобы это преимущество реально работало, необходимо уметь перемещать команды вокруг точек ветвления. Рассмотрим более подробно соответствующие проблемы и решения.

Перенос команды ниже точки ветвления является частным случаем миграции команд, которая описана выше в разд. 2.5.2. Более интересен случай переноса команды выше предшествующей точки ветвления. Этот вид переноса может быть полезен по двум причинам:

- в предшествующем линейном участке, возможно, имеется свободная позиция в одном из командных слов, куда можно поместить рассматриваемую команду;
- если команда имеет задержку (т.е. ее результат можно использовать только по истечению некоторого числа тактов), то выгодно

запланировать ее выполнение как можно раньше, в особенности, если она находится на критическом пути.

Перенос команд выше точки ветвления можно назвать упреждающим выполнением (в англоязычной литературе используется термин speculative execution или control speculation), поскольку команда выполняются раньше, чем становится известно, будет ли передано управление в участок, где она изначально находилась.

Рассмотрим ограничения, которые должны соблюдаться при переносе команд выше точки ветвления, на примере (рис. 2.13).

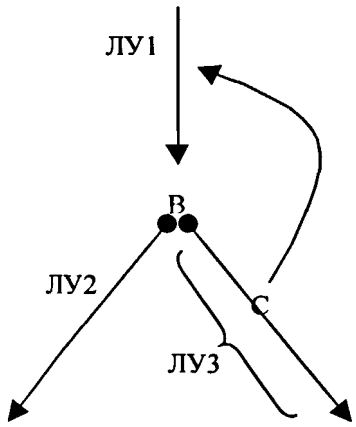


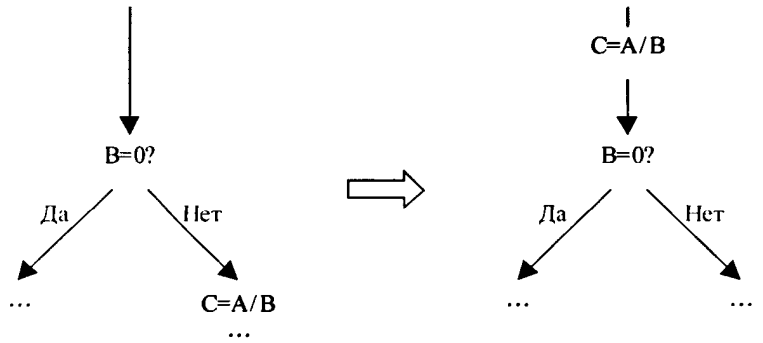
Рис. 2.13. Перенос команды С выше точки ветвления В

Перенос команды С выше точки ветвления В (в участок ЛУ1) возможен при соблюдении двух ограничений (см. [38], [44], [57]):

Ограничение 1. Прежнее значение регистра, в который С записывает свой результат, не требуется больше ни одной команде в ЛУ1 или в других участках, куда может быть передано управление из ЛУ1.

Ограничение 2. Команда С не может вызвать прерывание, которое приведет к завершению программы.

Некорректность переноса команды при несоблюдении второго ограничения можно проиллюстрировать на простом примере рис. 2.14.



а) Исходная программа

б) Команда C=A/B перенесена выше ветвления по условию B=0

Рис. 2.14. Перенос команды "C=A/B" выше команды ветвления некорректен, так как это приведет к аварийному завершению программы при B=0

Планирование с соблюдением обоих ограничений называют "ограниченной фильтрацией" (restricted percolation). При соблюдении обоих ограничений число команд, перенос которых реально возможен, оказывается невелико, что значительно снижает эффективность глобального планирования. Поэтому важное значение имеют методы, позволяющие снять эти ограничения.

Ограничение 1 можно снять путем переименования регистров - для значения, которое вычисляется в команде C, назначается другой регистр.

Для снятия ограничения 2 необходимы средства аппаратной поддержки, описанные в следующем разделе.

2.6.4. Аппаратная поддержка глобального планирования

Упреждающее выполнение применяется как в динамическом (аппаратном) планировании в суперскалярных процессорах с неупорядоченной выдачей команд, так и в статическом планировании во время компиляции. Здесь будут рассмотрены характерные для EPIC-архитектур аппаратные расширения, которые позволяют планировать упреждающее выполнение команд во время компиляции. Существуют две

модели поддержки упреждающего выполнения - модель неограниченной фильтрации (general percolation, см. [44]) и модель защищенного планирования (sentinel scheduling, см. [57]).

Первая модель предполагает наличие в аппаратуре непрерываемых версий всех команд. Если планировщик переносит команду выше точки ветвления, то он использует ее непрерываемую версию. Упреждающее выполнение записей в память не допускается. Исключительные ситуации, если они возникают, игнорируются. Получаемые при этом неправильные результаты могут сказаться на дальнейшем выполнении, например, привести позднее к прерываниям или просто к неверным результатам.

Вторая модель обеспечивает корректность статического планирования с применением упреждающего выполнения и включает следующие аппаратные расширения:

1. В кодировку команд добавляется бит упреждающего выполнения. Если он установлен, то говорят, что команда выполняется в режиме упреждающего выполнения, в котором прерывания не выдаются. Если возникает исключительная ситуация, то в регистре результата устанавливается бит прерывания. То же происходит, если бит прерывания установлен хотя бы в одном из регистров-операндов. При выполнении команд в обычном режиме должны выдаваться прерывания, если хотя бы один из входных регистров имеет установленный бит прерывания (или если исключительная ситуация возникла при выполнении самой команды).

2. К каждому регистру добавляется бит прерывания, который устанавливается при выполнении команд в режиме упреждающего выполнения, если возникает исключительная ситуация, и может быть опрошен при помощи специальной команды check_exception(R). С каждым регистром связывается также поле диагностики, в которое заносится программный адрес возникновения исключительной ситуации. Бит прерывания вместе с полем данных должны сохраняться и

восстанавливаться при временном сохранении регистра в памяти.

3. К системе команд добавляется команда для опроса бита прерывания заданного регистра check_exception(R). Если бит установлен, то возникает прерывание.

При наличии перечисленных средств компилятор получает возможность закодировать упреждающее выполнение команды C следующим образом:

- в новой позиции команды C (выше точки ветвления) помещается ее вариант с битом упреждающего выполнения;
- в прежней позиции команды C помещается команда check_exception, для того чтобы обеспечить прерывание, если оно должно произойти. Если имеется команда, выполняемая в обычном режиме и использующая результат команды C, то команда check_exception не генерируется.

Пример из [57], показанный на рис. 2.15, демонстрирует применение указанных средств при планировании.

```
A: if (r2==0) goto L1
B: r1 = mem(r2)
C: r3 = mem(r4)
D: r4 = r1+1
E: r5 = r3*9
F: mem(r2+4) = r4
```

a)

```
*B: r1 = mem(r2)
*C: r3 = mem(r4)
*D: r4 = r1+1
*E: r5 = r3*9
 A: if (r2==0) goto L1
 F: mem(r2+4) = r4
 G: check_exception(r5)
```

b)

*Рис. 2.15. Планирование команд с упреждающим выполнением: a) исходный код; b) код, полученный в результате планирования. Буквы A-G служат для идентификации команд. Символом * отмечены команды с признаком упреждающего выполнения*

На рис. 2.15 показан код до и после планирования (в предположении,

что исключительные ситуации возможны только при обращениях к памяти). Команды F и G обеспечивают проверку исключительных ситуаций, которые могли возникнуть при упреждающем выполнении команд B, C.

Для того чтобы разрешить упреждающее выполнение команд записи в память, предусматривается аппаратное расширение, основанное на введении дополнительных полей в элементы буфера записи в память.

Важным преимуществом модели защищенного планирования является поддержка адекватной диагностики исключительных ситуаций с указанием адреса фактического возникновения, даже если команда-источник выполнялась в упреждающем режиме.

По результатам тестов, приведенным в [57], применение защищенного планирования по сравнению с ограниченной фильтрацией на задачах нечисловой обработки дает существенное ускорение результирующих программ. Например, для процессора с темпом выдачи 8 команд на такт коэффициент ускорения[3] при использовании защищенного планирования был от 18 до 135% (в среднем на 57%) выше, чем при планировании с ограниченной фильтрацией.

Для численных приложений, не содержащих большого числа условных переходов во внутренних циклах (таких как операции над матрицами), разница коэффициентов ускорения оказалась незначительной. Коэффициенты ускорения для этих приложений оказались достаточно

---

[3] Тестирование проводилось для базового процессора с темпом выдачи 1 команда на такт. Затем оно повторялось для модельных процессоров с темпом выдачи 2, 4, 8 команд на такт без ограничений на параллельное выполнение команд - с применением ограниченной фильтрации и с применением защищенного планирования. При этом вычислялись коэффициенты ускорения по сравнению с базовой машиной - Kr(n) и Ks(n), где n = 2,4,8 - темп выдачи команд, символ r относится к ограниченной фильтрации, s - к защищенному планированию. Указанные процентные значения показывают, на сколько Ks(8) выше чем Kr(8).

высокими уже при ограниченной фильтрации. Для двух численных приложений с значительным количеством условных переходов во внутренних циклах улучшение коэффициента ускорения составило 36-38%.

Другое аппаратное расширение, предназначенное для поддержки глобального планирования - средства условного выполнения команд ([22], [67]). Например, процессор TM1000 [41] позволяет задавать в команде предикатный операнд, в качестве которого может выступать любой из 128 регистров. В зависимости от значения младшего бита этого операнда результат операции будет записан или проигнорирован. Аналогичные возможности поддерживаются в процессоре IA-64 ([45], [69]) и др.

Компилятор может слить условно выполняемый линейный участок с объемлющим участком, заменив составляющие его инструкции условно выполняемыми (рис. 2.16). В результате зависимости по управлению фактически заменяются зависимостями по данным, которые существенно удобнее с точки зрения планировщика.

```
c:= вычислить(условие)          c:= вычислить(условие)
        [c] goto then_label             [c]  B_THEN
            B_ELSE                      [!c] B_ELSE
            goto join_label
then_label: B_THEN
join_label: ...

            a)                              b)
```

Рис. 2.16. Реализация конструкции "if (условие) then B_THEN else B_ELSE" при помощи условных переходов (a) и условным выполнением (b). Обозначение [c] указывает, что команда будет выполнена только если предикат c имеет значение "истина"

В работе [53] рассматривается метод, позволяющий для заданной иерархии вложенных конструкций if-then-else выбрать оптимальные (по

средней скорости выполнения результирующей программы) способы реализации.

Поддержка условного выполнения делает особенно эффективным планирование в древовидных областях [41], [40]. Для каждого линейного участка древовидной области вычисляется предикат, который приписывается всем командам этого участка. В результате исключается большая часть условных переходов и появляется возможность планировать параллельное исполнение различных ветвей дерева.

2.6.5. Метод доминантного параллелизма при планировании в древовидных областях

Недостаток метода "дублирования хвостов" заключаются в генерации избыточного кода. При использовании упреждающего исполнения это приводит к нерациональному расходованию вычислительных ресурсов и может замедлить исполнение. В ряде случаев планировщик способен исключить отрицательные последствия при помощи метода доминантного параллелизма (dominator parallelism) [40]. Если команды из некоторого линейного участка BB_i и его копии $BB_{i'}$ можно поднять в доминирующий участок BB_k (являющийся общим предком BB_i и $BB_{i'}$), то можно оставить только по одному экземпляру команд. Пример ситуации, когда этот прием применим, показан на рис. 2.17.

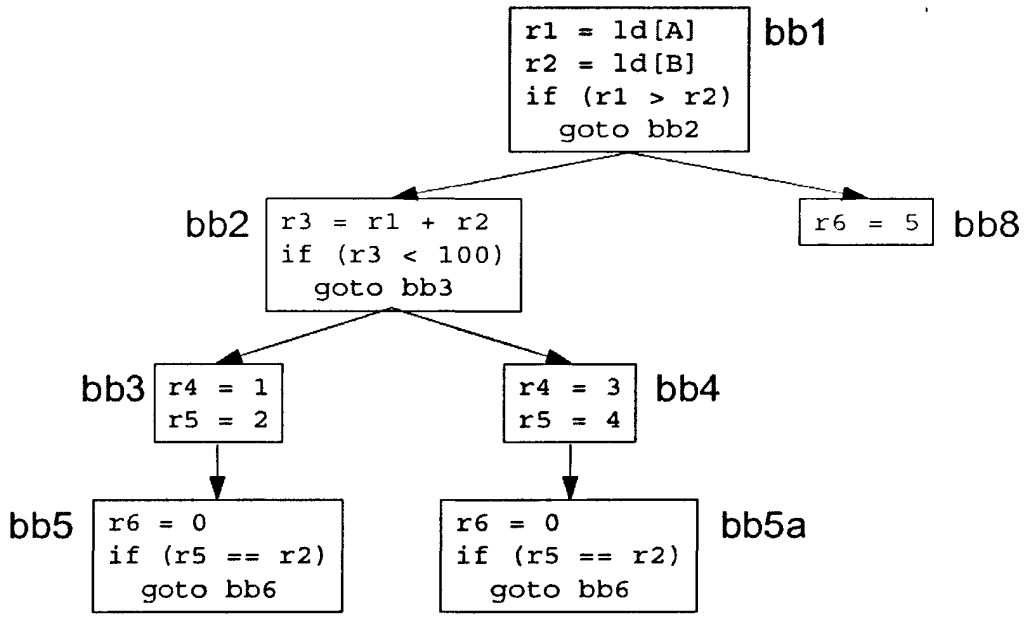


Рис. 2.17. Доминантный параллелизм. Команду r6=0 из линейного участка BB5 и его копии BB5' можно поднять в BB2 (или BB1) и исключить их дублирование

2.6.6. Планирование по прогнозу значений данных

Упреждающее выполнение команд по прогнозу данных (data speculation) широко применяется в динамическом аппаратном планировании в суперскалярных процессорах. Смысл этого приема заключается в том, что значение объекта используется до того, как оно записано, в предположении, что значение останется прежним. Разумеется, если значение все-таки изменяется, процессор обеспечивает перевычисление команд, выполненных с упреждением. EPIC-архитектуры предоставляют аппаратную поддержку для использования аналогичного механизма при статическом планировании в компиляторе по отношении к объектам, хранящимся в памяти, см. [45] или [69].

```
...
MEM(R0) = R1
...
R2=MEM(R3)
R4=R2+R2
...
```

a)

```
...
R2=MEM(R3)' ; упреждающее
            ; чтение
...
MEM(R0)=R1
check(address(R3)),L1
R4=R2+R2
```

b)

Рис. 2.18. Перестановка операций чтения и записи с использованием аппаратной поддержки упреждающего чтения

При наличии таких аппаратных средств компилятор получает возможность переупорядочивать команды чтения и записи памяти. Это может быть полезно по нескольким причинам:

- как можно раньше выполнить команды считывания из памяти, лежащие на критическом пути;
- использовать свободную позицию в командном слове,
- исключить задержки, связанные с чтением данных из памяти.

На рис. 2.18а показан пример последовательности вычислений, содержащий команды записи в память (MEM(R0) = R1) и чтения из памяти (R2=MEM(R3)). Компилятор, вообще говоря, не имеет права запланировать чтение раньше записи, если он не имеет точной информации о том, что команда записи не перепишет считываемое значение.

Аппаратная поддержка упреждающего считывания из памяти может состоять из следующих средств (см. [18], [45]):

- имеется команда упреждающего считывания, которая считывает значение по указанному адресу и запоминает этот адрес во внутренней таблице процессора;
- при записи в память и при обычном считывании содержимое таблицы

проверяется, и если в ней присутствует указанный адрес, то соответствующий элемент таблицы помечается как недействительный (или просто исключается)

- имеется команда, позволяющая узнать, не было ли записи по заданному адресу с момента упреждающего считывания по этому адресу; если запись была, то выполнить переход по указанному адресу. Подразумевается, что по этому адресу должен находиться компенсирующий код.

На рис. 2.18b показан пример перестановки чтения и записи памяти с использованием этих средств. Команда чтения (в упреждающем режиме) помещается до команды записи. На месте прежнего положения команды считывания (до использования результата) помещена команда проверки адреса. Если оказалось, что по этому адресу была произведена запись, то управление передается по метке L1, где размещается сгенерированный компилятором компенсирующий код.

2.7. Особенности генерации кода для ЦПОС

Особенности генерации эффективного кода для ЦПОС связаны как с нерегулярностью схем кодирования, так и с другими характерными чертами этих процессоров, такими как наличие специализированных команд, нескольких пространств памяти и др. В этом разделе представлены специфические подходы, применяемые в компиляторах для ЦПОС.

Наиболее важной в контексте компиляции для ЦПОС представляется задача планирования (сжатия) кода. Специфика планирования для VLIW-процессоров с нерегулярными схемами кодирования связана с многочисленными ограничениями на параллельное исполнение команд, из-за которых существенно снижаются возможности использования

внутреннего параллелизма программы при генерации кода. В ЦПОС также в меньшей степени представлены аппаратные расширения для поддержки параллелизма - условное выполнение, поддержка упреждающего выполнения. При этом именно для данного класса процессоров, в силу специфики областей применения, генерация оптимального кода имеет особенно важное значение.

В силу перечисленных причин в компиляторах для ЦПОС вместо эвристического глобального планирования более разумным представляется применение алгоритмов для поиска точного оптимального решения в пределах линейных участков.

В [54] приводятся следующие аргументы в пользу локального планирования:

- Методы глобального планирования так или иначе опираются на локальные методы, которые сами по себе в контексте компиляции для ЦПОС еще недостаточно изучены и разработаны. Поэтому разумно начать с их детального исследования и оценки.
- Популярные глобальные методы разрабатывались для регулярных VLIW-процессоров с высоким уровнем аппаратного параллелизма и проявили свою высокую эффективность для этого класса архитектур, в то время как в ЦПОС уровень параллелизма как правило значительно ниже из-за ограничений кодирования.
- В глобальном планировании для сохранения корректности программы в нее приходится добавлять компенсирующий код. Это может привести к существенному увеличению размера программы, что неприемлемо, если объем памяти ограничен.

К этому можно добавить, что:

- Преимущества глобального планирования экспериментально подтверждены для программ нечисленного характера; для программ

численных расчетов выигрыш от глобального планирования, вероятно, не столь значителен.

- Применение глобального планирования наиболее эффективно при наличии аппаратных расширений для его поддержки.

В [54] рассматривается подход к задаче локального планирования для определенного класса ЦПОС, основанный на методах целочисленного линейного программирования (см. [15]) и позволяющий находить кратчайший выходной код для процессора с заданными ограничениями на параллельное исполнение команд. Хотя время нахождения решения экспоненциально зависит от длины линейного участка, по мнению авторов, применение подобных подходов в компиляции для ЦПОС оправдано. Важная положительная черта предлагаемого метода заключается в поддержке вариантов команд - если процессор предоставляет несколько различных типов команд для выполнения одной и той же операции, то при поиске оптимального решения обеспечивается перебор вариантов.

В [30] и [37] также представлены реализации локальной оптимизации кода для ЦПОС на основе методов линейного программирования. В этих реализациях совмещается решение задач распределения регистров, распараллеливания и выбора команд (code selection) с учетом наличия в процессоре составных операций типа A=A+B*C.

Другая проблема, обусловленная нерегулярным характером кодирования, - способ представления ограничений на параллельное исполнение команд в планировщике. Если в регулярных ILP-архитектурах эти ограничения достаточно легко описать и представить в виде общего числа функциональных устройств каждого вида и наборов устройств, занимаемых каждой командой, то для процессоров с нерегулярным кодированием их рациональное представление составляет более сложную задачу. В [68] описывается подход, позволяющий свести нерегулярный

набор ограничений к регулярному представлению путем определения набора искусственных аппаратных ресурсов и приписывания соответствующего мультимножества ресурсов каждому виду команд процессора. Недостатком предлагаемой реализации является то, что ее применимость ограничена слишком жесткими предположениями о структуре системы команд.

Характерной особенностью многих ЦПОС является малое число регистров, их специализация или кластерная организация. Это также требует особых подходов при распараллеливании и выборе кода ([23], [52]).

Как показано в [30], применение некоторых оптимизаций, считающихся машинно-независимыми (таких как свертка констант, исключение общих подвыражений), в контексте компиляций для ЦПОС требует особых подходов с учетом специфики организации командного слова и числа доступных регистров в конкретном целевом процессоре.

Поскольку при программировании для ЦПОС обычно налагаются ограничения на размер кода, то при реорганизациях циклов и встраивании функций необходимо учитывать этот фактор. С этой точки зрения интересна работа [51], где рассматривается методика встраивания функций с контролируемым ростом объема кода. Аналогичные методики могут быть полезны и для преобразований циклов.

Проблема генерация оптимального кода для ЦПОС не сводится только к задаче сжатия кода с учетом возможностей параллельного исполнения. Хороший компилятор для ЦПОС должен уметь эффективно использовать их архитектурные особенности - решать задачу оптимального размещения программных данных в пространствах памяти [50], поддерживать языковые расширения для указания пространства памяти в декларациях переменных [42], решать задачу выбора кода с учетом имеющегося набора команд в процессорах с системами команд для

специальных приложений - Application Specific Instruction Set Processors (ASIP), (см. [23],[24]), выделять циклические буферы, оптимизировать способы адресации при обращениях к памяти (см. [37], [50], [55]) и др.

2.8. О роли языковых расширений

В различных реализациях компиляторов с языка Си для ILP-архитектур делаются попытка отразить на уровне входного языка специфику этих процессоров и особенности программирования для них ([20], [42]). Операции над комплексными, векторными, матричными данными, явно выраженные в терминах исходного языка, могут быть непосредственно отражены в эффективные связки команд ILP-процессоров.

Комплексный тип данных зафиксирован в последнем стандарте языка Си [46]:

```
#include <complex.h>
complex float x, y = 1.0 + 3.0 * I;
```

Для комплексного типа определен набор обычных операций и библиотечных функций.

Векторные и матричные операции, привлекательные с точки зрения возможности напрямую использовать параллелизм на уровне команд, к сожалению, плохо «встраиваются» в синтаксис языка Си, поскольку имена массивов трактуются как описатели. Поэтому, например, в стандарте ANSI Numerical C, разработанном группой NCEG (Numerical C Extension Group) для численных приложений, введены понятия итератора и оператора суммирования, отражающие семантику матричных операций.

Пример описания и использования итератора:

```
iter I = N;
A[I] = sin (2 * PI * I / N)
```

Этот фрагмент программы эквивалентен следующему тексту на стандартном Си:

```
int i;
for (i = 0); i < N; i++)
{
  A[i] = sin (2 * PI * i / N);
}
```

Пример вычисления произведения матриц с использованием итераторов и операторов суммирования:

```
iter I=N, J=N, K=N;
A[I][J] = sum (B[I][K] * C[K][J]);
```

Суммирование здесь производится по итератору K. В общем случае суммирование проводится по всем свободным итераторам, т.е. итераторам, которые не встречаются в том же итераторе вне оператора суммирования.

В работе [42] также предлагаются некоторые другие расширения, отражающие специфику ЦПОС – пространства памяти, циклические буферы.

Положительные стороны такого рода расширений — краткость и естественность исходных текстов, возможность эффективно отобразить высокоуровневые операции в оптимальный для заданной целевой платформы код.

2.9. Сводка методов оптимизации для процессоров с поддержкой параллелизма на уровне команд

Анализ работ, посвященных оптимизации кода для процессоров с параллелизмом на уровне команд показывает, что для достижения

наилучших результатов необходимо применение комплекса оптимизаций, среди которых можно выделить следующие классы.

- Преобразования циклов, направленные на усиление программного параллелизма – развертка циклов, слияние и разбивка циклов, конвейеризация циклов.
- Преобразования, направленные на ослабление зависимостей по данным – перемещение кода, переименование регистров, дублирование переменных суммирования в циклах и др.
- Применение методов глобального планирования потока команд на основе алгоритма списочного планирования.
- Использования аппаратных средств поддержки упреждающего выполнения и упреждающего чтения данных.
- Совмещение планирования команд и распределения регистров.
- Применение локального планирования на основе целочисленного линейного программирования;
- Реализация языковых расширений.

При выборе конкретных методов и их параметров необходимо учитывать уровень аппаратного параллелизма и другие особенности целевого процессора, а также характер типичных приложений.

3. Компилятор с оптимизирующим постпроцессором - детальное описание

В этой главе рассматривается общая структура компилятора с оптимизирующим постпроцессором, а также собственно процесс компиляции. Основное внимание уделяется постпроцессированию кода, сгенерированного базовым Си-компилятором, а также эвристикам,

позволяющим ускорить работу постпроцессора. В разделе 3.1 даются общие сведения о компиляторе; раздел 3.4 описывает работу постпроцессора; настройка постпроцессора на архитектуру процессора 1В577 рассмотрена в разделе 3.5.

3.1. Характеристика процессора 1В577

Процессор 1В577 является цифровым сигнальным процессором с поддержкой параллелизма на уровне команд. Ниже перечислены его свойства, существенные с точки зрения выбора методов оптимизации и генерации кода:

- имеется несколько функциональных устройств, способных работать параллельно;
- поддерживается очень длинное командное слово (Very Large Instruction Word, VLIW), в котором могут содержаться операции для разных функциональных устройств.
- операции, закодированные в одном командном слове, выполняются параллельно, то есть команда *i* не может использовать результат команды *j*, если *i* и *j* находятся в одном командном слове.
- значение регистра не меняется в результате его чтения (это может быть неверно для машин, где некоторые регистры образуют стек).
- значение регистра не может меняться самопроизвольно или портиться по истечении некоторого "срока хранения".

Возможность параллельного исполнения нескольких операций ограничивается набором функциональных устройств процессора и тем, какие из них участвуют в выполнении каждой операции. Сложность оптимизации усугубляется наличием ограничений кодирования. Имеются ограничения на использование регистров или режимов адресации, которые

допускаются в отдельно стоящей команде, но недопустимы в такой же команде, если она скомбинирована с другими в общем командном слове.

3.2. Общие сведения о компиляторе для 1В577

Компилятор для процессора 1В577 реализован на основе базового Си-компилятора, в который был интегрирован оптимизирующий постпроцессор ассемблерного кода, выполняющий оптимизации, специфические для платформ с длинным командным словом.

В качестве базового Си-компилятора использован известный свободно распространяемый продукт gcc, имеющий развитые общеоптимизирующие средства. Этот компилятор используется для генерации последовательного ассемблерного кода, который затем постпроцессируется с целью упаковки команд в длинные командные слова.

Общая схема компиляции с языка высокого уровня показана на рис. 3.1.

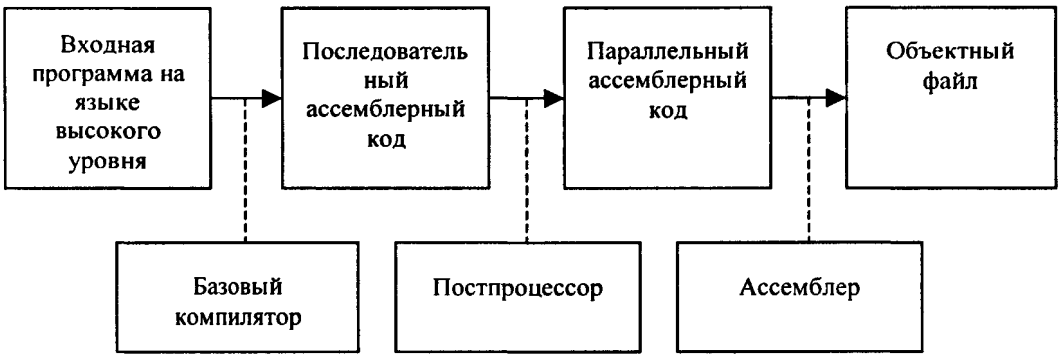


Рис. 3.1. Схема компиляции с языка высокого уровня

Ниже перечислены оптимизации, реализованные в постпроцессоре:

1. Планирование потока команд: объединение элементарных команд, из которых состоит сгенерированный базовым компилятором код, в длинные командные слова.

Планирование команд совмещается в постпроцессоре со следующими дополнительными оптимизациями:

2. Подбор вариантов команд. Постпроцессор может заменять элементарные команды на другие, эквивалентные им, если это позволяет в результате получить более эффективный код.
3. Модификация команд. Примером модификации может служить замена способа адресации при обращении к памяти.
4. Удаление лишних команд (результат которых не используется).

Примеры оптимизаций, выполняемых постпроцессором, можно найти в разд. 3.4.1 и в главе 4.

Отметим, что в современных версиях компилятора gcc поддерживается планирование потока команд и генерация параллельного кода для архитектур с параллелизмом на уровне команд — но только при условии отсутствия ограничений кодирования. Поэтому для рассматриваемого целевого процессора средства планирования, имеющиеся в базовом компиляторе, неприменимы. Существенно также, что gcc не поддерживает использование механизмов варьирования и модификации команд при планировании. Наконец, как отмечено в разд. 2.7, метод списочного планирования, используемый в gcc, не обеспечивает максимальной оптимизации кода для процессоров данного класса.

Реализация дополнительных оптимизаций на уровне ассемблерного кода, а не на уровне промежуточного представления gcc, обусловлена тем, что одной инструкции промежуточного представления может соответствовать несколько ассемблерных команд, что может приводить к снижению эффективности планирования.

3.3. Роль базового компилятора

Несмотря на то, что базовому компилятору при данном подходе отводится вспомогательная роль, конечный результат в существенной степени зависит от свойств сгенерированного им последовательного ассемблерного кода.

Во-первых, базовый компилятор должен обладать развитыми общеоптимизирующими средствами, порождая пусть и чисто последовательный, но, тем не менее, достаточно экономный код. Компилятор gcc удовлетворяет этому требованию — в нем реализован общепринятый набор методов оптимизации: алгоритмы оптимизации циклов и переходов, развертка циклов (см. разд. 2.5.1), понижение мощности операций, выделение общих подвыражений, свертка констант и др.

В дополнение к этим оптимизациям были также реализованы специфические для целевой платформы 1В577 оптимизации — поддержка аппаратных циклов, встроенное деление.

Во-вторых, можно усилить возможности постпроцессора (или ускорить его работу), если встроить в промежуточный код дополнительную информацию, которую компилятор накапливает во время обработки программы. В модуль генерации кода компилятора gcc были добавлены средства для вывода следующих видов аннотирующей информации в выходной ассемблерный код.

1. Информация о связях по данным между линейными участками. Зная, что некоторое значение не используется за пределами линейного участка, постпроцессор может более свободно планировать команды на этом участке.

2. Отметки начала и конца циклов. Постпроцессор может применить более дорогостоящие средства анализа и оптимизации к телам внутренних циклов.

3. Информация о неявных входных и выходных регистрах ассемблерных команд. Компилятор генерирует списки дополнительных (не фигурирующих явно в ассемблерном тексте) входных и выходных значениях. В частности, это существенно для команд перехода с возвратом, при помощи которых осуществляется вызов подпрограмм. Для них компилятор сообщает имена регистров, на которых передаются и возвращаются результаты. В списки входных и выходных регистров включается также указатель стека, память, системные регистры, используемые командой перехода с возвратом.

В-третьих, очень важно, чтобы генерируемые компилятором команды допускали эффективное комбинирование в длинные командные слова. Для этого необходимо, например, учитывать ограничения на форматы параллельных команд и генерировать преимущественно те варианты последовательных команд, которые могут быть скомбинированы с другими командами. Полезно также располагать потенциально комбинируемые команды по возможности ближе друг к другу. Эти требования обеспечиваются с помощью таких средств базового компилятора, как *планирование* и *распределение регистров*.

Планирование потока команд в базовом компиляторе. В ответ на развитие архитектур, обладающих свойствами параллелизма, в gcc было реализовано локальное списочное планирование (см. разд. 2.6.1). Средства описания целевых платформ позволяют задавать информацию о типах и количестве функциональных устройств и об их участии в выполнении каждой команды. Поскольку не поддерживается в необходимом объеме задание информации об ограничениях кодирования, то решение проблемы

распараллеливания кода средствами gcc для данного целевого процессора невозможно. Тем не менее, средства планирования gcc были использованы для переупорядочения кода таким образом, чтобы команды, использующие разные функциональные устройства, были расположены близко друг к другу, с тем, чтобы облегчить задачу планирования на стадии постпроцессирования. Это было особенно существенно для первых версий постпроцессора, когда планирование, осуществлялось в пределах относительно небольшого “окна” кода (см. разд. 2.1).

Распределение регистров. Из-за ограниченной длины командного слова возможны ограничения на форматы адресации и использование регистров в параллельных командах. Так, из 800 комбинаций регистров возможных в команде умножения процессора 1В577:

```
FMPY.S Di,Dj,Dk ;; Dk=Di*Dj, i,k=0, ..., 9; j=0, ..., 7
```

допустимы только 16, если умножение выполняется параллельно со сложением.

Соответственно, в описании целевой платформы распределение регистров задано таким образом, чтобы в командах умножения использовались преимущественно эти 16 комбинаций. Таким образом, постпроцессор получает относительно широкие возможности для комбинирования умножения со сложением.

Некоторые виды ограничений на использование регистров в параллельных командах учесть довольно сложно. Например, две пересылки данных в память или из памяти в процессоре 1В577 могут выполняться параллельно только при условии, что в них используются адресные регистры из двух разных банков: r0-r3 и r4-r7. В ходе компиляции на этапе распределения регистров сложно предсказать, какие из пересылок в дальнейшем могут быть скомбинированы, чтобы назначить подходящие адресные регистры. Единственное, что удается довольно легко обеспечить при настройке компилятора, - это равномерное

использование адресных регистров из обоих банков; для этого достаточно указать соответствующий порядок выделения регистров, например: r0, r4, r1, r5, r2, r6, r3, r7.

Отметим, наконец, поддержку в gcc ряда полезных языковых расширений, таких как комплексный тип данных, ассемблерные вставки и др.

3.4. Постпроцессирование

Основная цель постпроцессирования заключается в распараллеливании последовательного ассемблерного кода, сгенерированного базовым компилятором. Распараллеливание кода выполняется путем планирования потока команд в сочетании с оптимизациями подбора вариантов команд, модификации команд и, при необходимости, удаления лишних команд.

В разд. 3.4.1 на примерах поясняется смысл оптимизаций, выполняемых постпроцессором.

В разд. 3.4.2 вводятся основные понятия, используемые при описании постпроцессора.

В 3.4.3 описана общая последовательность обработки входного ассемблерного файла постпроцессором.

В разд. 3.4.4 дается способ представления аппаратных ограничений, т.е. условий, при которых команды могут быть помещены в одно командное слово.

Разд. 3.4.5 — 3.4.8 посвящены описанию алгоритма планирования, реализованного в постпроцессоре. В разд. 3.4.5 формулируется модель участка планирования, вводится понятие плана выполнения, доказываются достаточные условия эквивалентности двух планов выполнения. В разд.

3.4.6 приведено описание алгоритма планирования перебором планов выполнения по методу динамического программирования. В 3.4.7 обсуждается способ учета аппаратных задержек при планировании. Поскольку число возможных планов выполнения быстро растет с ростом размера участка, то полный перебор планов выполнения потребовал бы слишком много времени и памяти. Поэтому большое значение при данном подходе играют методы сокращения перебора, которые обсуждаются в разд. 3.4.8.

Разд. 3.4.9 посвящен оптимизации варьирования (подбора вариантов) команд.

В разд. 3.4.10 описана реализация модификации команд в постпроцессоре.

3.4.1. Примеры оптимизаций, выполняемых постпроцессором

В примере 4.1 показан ассемблерный код до и после постпроцессирования.

Пример 3.1. Последовательный код для 1В577 и эквивалентный код, обеспечивающий одновременное выполнение нескольких элементарных действий.

Последовательный код:

Номера команд	Команды	Комментарии
1	Move x:(r0)+,d4.s	; пересылка данных из памяти в регистр d4.
		; Запись "x:(r0)+" означает обращение по шине x
		; к слову, адрес которого находится в регистре
		; r0. Символ "+" означает, что после вычисления
		; адреса значение r0 увеличится на 1.
2	Fadd.s d0,d1	; d1:=d1+d2
3	Fmpy.s d4,d7,d0	; d0:=d4*d7
4	Fsub.s d3,d2	; d2:=d2-d3
5	Fmpy.s d5,d6,d3	; d3:=d5*d6
6	Move d3.s,x:(r4)+	; Запись в память значения d3.

Параллельный код:

Номера команд	Команды				
1,4,5	Fmpy	d5,d6,d3	fsub.s	d3,d2	x:(r0)+,d4.s
2,3,6	Fmpy	d4,d7,d0	fadd.s	d0,d1	d3.s,x:(r4)+

(В записи параллельной команды имя команды move опускается.)

Оптимизация *подбора вариантов команд* заключается в том, что одни элементарные команды могут заменяться на другие, эквивалентные им, если это позволяет в результате получить более эффективный код. Например, в процессоре 1В477 пересылка из одного регистра данных в другой может выполняться командами tfr или move, которые обладают разными свойствами комбинируем ости с прочими командами. Постпроцессор выбирает тот или иной способ пересылки с учетом возможности скомбинировать ее с окружающими командами.

Примером *модификации команд* может служить замена способа адресации при обращении к памяти. Так, если входной ассемблерный код содержит последовательность команд вида

```
move x:(r4),d1.s     ;занести в d1 значение из памяти по адресу,
                     ;хранящемуся в регистре r4 (по шине x).
...
move (r4)+n4         ;увеличить r4 на значение регистра n4
```

и между ними не используется регистр r4 и не устанавливается регистр n4, то можно исключить вторую команду, модифицировав при этом способ адресации в первой следующим образом:

```
move                 ; Здесь "x:(r4)+n4" означает обращение к памяти
x:(r4)+n4,d1.s       ;по шине x, по адресу, хранящемуся в регистре r4,
                     ;и увеличение r4 на значение регистра n4
```

Это дает выигрыш как в быстродействии, так и в размере выходного кода.

Удаление лишних команд выполняется редко, поскольку эту оптимизацию выполняет базовый компилятор. Тем не менее, в отдельных случаях gcc не все же генерирует лишние команды на фазе распределения регистров и перегрузки значений. Поэтому данная оптимизация (практически не требующая дополнительных вычислительных затрат) включена в состав постпроцессора.

3.4.2. Основные понятия

В этом разделе определяются понятия, используемые при описании алгоритма планирования.

Элементарная команда - это команда, рассматриваемая постпроцессором как неделимое действие.

VLIW-строкой назовем неупорядоченный набор элементарных команд, способных исполняться параллельно и независимо. В записи выходного ассемблерного кода элементарные команды VLIW-строк упорядочиваются согласно синтаксическим правилам ассемблера. В результате ассемблирования из них формируются длинные командные

слова процессора. В частном случае VLIW-строка может содержать одну элементарную команду.

VLIW-последовательность - это последовательность VLIW-строк.

Комбинирование - это объединение элементарных команд в VLIW-строки с учетом аппаратных ограничений и зависимостей по данным.

Линейный участок - это VLIW-последовательность, не содержащая меток, и содержащая не более одной команды передачи управления, которая располагается в конце последовательности.

3.4.3. Последовательность обработки входного ассемблерного файла

Постпроцессирование ассемблерного кода состоит из нескольких этапов. На первом этапе входная ассемблерная программа подвергается синтаксическому и лексическому анализу и переводится во внутреннее представление - список, содержащий элементы исходной программы: метки, команды, директивы ассемблера, вспомогательные директивы, сгенерированные компилятором для постпроцессора. Если входная программа содержит комбинации параллельно исполняемых команд, они разбиваются на элементарные команды.

Далее в этом внутреннем представлении выделяются линейные участки. Границами линейных участков служат метки, а также вспомогательные директивы, отмечающие начало и конец циклов.

Затем каждая команда и каждый линейный участок в целом снабжаются аннотирующей информацией, состав которой определяется потребностями алгоритмов и методов оптимизации. В частности, для каждой команды приводится набор ее входных и выходных регистров, наборы регистров, по которым она создает аппаратные задержки или чувствительна к задержкам, созданным предыдущими командами, множество функциональных единиц процессора, которые нужны для выполнения команды и т.п. Для линейного участка указывается, например,

является ли он телом цикла. При аннотировании используется информация, содержащаяся в описании целевого процессора (разд. 3.5).

Каждый линейный участок передается оптимизатору линейных участков, который преобразует его в последовательность VLIW-строк (во внутреннем представлении). Если последняя VLIW-строка оптимизированного участка создает аппаратные задержки или первая команда чувствительна к аппаратным задержкам, то эта информация передается соседним участкам и учитывается при их оптимизации.

Обработка линейных участков проводится в порядке убывания уровня вложенности циклов, чтобы минимизировать ограничения, связанные с задержками, при оптимизации внутренних циклов.

Наконец, оптимизированная таким образом программа выводится в выходной файл в виде ассемблерного кода.

3.4.4. Аппаратная совместимость

При планировании необходимо соблюдать ограничения аппаратной совместимости, которые определяют, может ли данное множество элементарных команд выполняться параллельно. Описание этих ограничений должно обеспечивать простой способ проверки корректности результирующих VLIW-строк. Для некоторых архитектур (например, 1B577) составление такого описания - весьма непростая задача.

Одно из ограничений алгоритма планирования, соответствующее аппаратному ограничению большинства ILP-процессоров, заключается в том, что параллельно исполняемые команды не должны записывать значения в один и тот же регистр. Многократное чтение считается допустимым. Более того, регистр может быть входным для одной элементарной команды и выходным для другой команды из той же VLIW-строки: читающая команда использует прежнее значение регистра до того, как оно будет изменено пишущей командой.

Следующее ограничение определяется набором функциональных устройств и других ресурсов процессора. Постпроцессор использует описание набора аппаратных устройств и ресурсов, а также информацию о ресурсах, требуемых для выполнения каждой элементарной команды. При комбинировании элементарных команд проверяется, что их совокупные требования по ресурсам не превышают возможностей процессора.

Этого простого подхода, к сожалению, оказывается недостаточно, чтобы учесть многочисленные ограничения кодирования процессора 1В577 (например, запрет на параллельное исполнение операций разной точности). Для решения этой проблемы вводятся описания запрещенных комбинаций ресурсов (см. разд. 3.5.4). Использование запрещенных комбинаций оказалось чрезвычайно удобным как для компактного описания корректных VLIW-строк, так и для рекурсивного их порождения из заданного множества элементарных команд.

В дальнейшем изложении при упоминании VLIW-строк или VLIW-последовательностей везде подразумевается, что речь идет об аппаратно корректных VLIW-строках или их последовательностях.

3.4.5. Модель линейного участка и постановка задачи планирования

Задачу, которую решает алгоритм планирования, неформально можно сформулировать следующим образом: преобразовать входную VLIW-последовательность в эквивалентную ей выходную VLIW-последовательность, наилучшую из возможных с точки зрения заданной функции стоимости.

Необходимо отметить, что для длинных участков поиск точного оптимума связан с перебором слишком большого числа вариантов, поэтому для длинных участков ставится задача приемлемой оптимизации за приемлемое время (см. раздел 3.4.8).

Будем считать, что для целевого процессора определены:

- Множество регистров **R** [4].
- Система команд процессора.
- Функция compatible, позволяющая для заданного подмножества команд процессора определить, являются ли они аппаратно совместимыми (см. разд. 3.4.4).
- Функция, определяющая стоимость VLIW-строки, а также операции сравнения и сложения стоимостей VLIW-строк.

Участок программы B, подлежащий планированию, определяется как четверка

$B = (C_B, P_B, I_B, O_B)$

где

$C_B = \{a, b, ...\}$ - конечное множество элементарных команд.

Хотя участок может содержать одинаковые команды, с точки зрения процедуры планирования все команды различимы. Для каждой команды a определено множество ее входных регистров In_a и выходных регистров Out_a [5].

P_B: $C_B \rightarrow N$ - отображение, задающее для каждой команды номер командного слова, в котором она закодирована в исходном программном фрагменте.

$I_B \subset \boldsymbol{R}$ - множество входных регистров, значения которых определены при входе в B.

---

[4]Под регистрами понимаются все объекты, в которые может производиться запись - в том числе, например, память, регистр состояния, порты ввода-вы-вода и т.п.

[5] Команда может записывать значения в несколько выходных регистров. Например, команда сложения записывает результат в регистр данных и признак результата в регистр состояния. Кроме того, не все записываемые значения фактически используются.

$O_B \subset \boldsymbol{R}$ - множество выходных регистров, значения которых, вычисленные в B, используются в других программных участках, выполняемых после B.

Отображение множества команд на множество натуральных чисел P: $C_B \rightarrow N$ однозначно определяет последовательность VLIW-строк, составленных из элементов C_B:

$$W = (W_1, ..., W_k),$$

$$\text{где } W_j = \{c \in C_B | \, P(c) = j \}$$

И, наоборот, по заданной последовательности VLIW-строк (W_1, ..., W_k), где $W_j \subset C_B$ и $W_i \cap W_j = \varnothing$ при $i \neq j$, $i,j \leq k$, однозначно определяется отображение $P{:}C_B \rightarrow N$. Таким образом, существует взаимно однозначное соответствие между отображениями из C_B в N и последовательностями VLIW-строк, составленных из элементов C_B.

Определение. *Планом выполнения* участка B=(C_B, P_B, I_B, O_B) называется отображение $P{:}C_B \rightarrow N$ или соответствующая ему последовательность VLIW-строк.

План P_B будем называть *исходным* планом выполнения участка B.

Будем предполагать, что линейный участок содержит фиктивную команду *start*, которая записывает значения во все входные регистры: $Out_{start} = I_B$, $In_{start} = \varnothing$, и фиктивную команду *stop*, читающую все выходные регистры: $In_{stop} = O_B$, $Out_{stop} = \varnothing$. Считается, что исходный план выполнения содержит VLIW-строки $W_0 = \{start\}$ и $W_{k+1} = \{stop\}$.

Определение. План P обладает свойством *определенности входных данных*, если для всякой команды a, которая читает некоторый регистр r, существует команда b, которая пишет в регистр r и исполняется раньше a:

$$\forall\, a \in C_B, \forall\, r \in In_a \;\; \exists\, b \in C_B : \; P(b) < P(a) \wedge r \in Out_b$$

Будем считать, что исходный план выполнения P_B с учетом предположения о наличии команд *start* и *stop* обладает указанным свойством.

Определение. План P назовем *корректным*, если:

1. в соответствующей VLIW-последовательности (W_1, ..., W_k) все VLIW-строки составлены из аппаратно совместимых команд:

$$\forall\, i, \quad \text{compatible}(W_i) = \text{истина}$$

2. никакие две команды, принадлежащие одной и той же VLIW-строке, не записывают значения в одни и те же регистры:

$$P(a) = P(b) \Rightarrow Out_a \cap Out_b = \varnothing$$

Будем считать, что исходный план всегда корректен.

План P назовем *допустимым*, если он корректен и обладает свойством определенности входных данных.

Будем предполагать пока, что исходная последовательность VLIW-строк представляет собой линейный участок (базовый блок) в смысле [1], [21], [14], т.е. не содержит команд передачи управления, и управление извне может быть передано только на первую VLIW-строку.

Определим на C_B отношение зависимости по данным: b зависит от a по регистру r относительно заданного плана выполнения P, $a \xrightarrow{\text{RAW(P,r)}} b$, если

- $r \in Out_a \;\&\; r \in In_b$
- $P(b) > P(a)$
- $\exists\, c \in C_B$: $r \in Out_c \;\&\; P(a) < P(c) < P(b)$

Будем обозначать через $a \xrightarrow{\text{RAW(r)}} b$ зависимости по регистру r относительно исходного плана P_B. Будем также писать $a \xrightarrow{\text{RAW}} b$, если

$\exists\, r \in \boldsymbol{R}$: $a \xrightarrow{\text{RAW(r)}} b$.

Далее мы рассмотрим свойства отношения RAW для допустимых планов выполнения, введем понятие эквивалентности планов выполнения и покажем, что эквивалентные планы выполнения линейного участка вычисляют одинаковые значения выходных регистров на этом участке.

Утверждение 1. Если план P корректен, то всякая команда по каждому своему входному регистру зависит относительно P не более чем от одной команды.

Пусть имеются команды a, b, $c \in C_B$ и регистр $r \in \boldsymbol{R}$, такой что $a \xrightarrow{\mathrm{RAW(P,r)}} c$ и $b \xrightarrow{\mathrm{RAW(P,r)}} c$. По определению отношения RAW(P,r) это возможно, только если $P(a) = P(c)$. Но поскольку $Out_a \cap Out_c$ содержит r, следовательно, не пусто, то это противоречит предположению о корректности плана P.

Утверждение 2. Если план P обладает свойством определенности входных данных, то любая команда по каждому своему входному регистру зависит относительно P по крайней мере от одной команды.

Утверждение вытекает из определений отношения RAW(P,r) и свойства определенности входных данных.

План выполнения $R{:}C_B \rightarrow N$ не противоречит плану P, если он сохраняет отношения зависимостей по всем регистрам, т.е. если из $a \xrightarrow{\mathrm{RAW(P,r)}} b$ следует $a \xrightarrow{\mathrm{RAW(R,r)}} b$.

Отношение непротиворечивости, очевидно, рефлексивно.

Утверждение 3. Отношение непротиворечивости симметрично на множестве допустимых планов выполнения участка B.

Пусть P, R - допустимые планы выполнения участка B, и R не противоречит P. Докажем, что P не противоречит R.

Рассмотрим произвольные команды a, $b \in C_B$ и регистр r, такой что $a \xrightarrow{\mathrm{RAW(R,r)}} b$. Покажем, что $a \xrightarrow{\mathrm{RAW(P,r)}} b$.

Из утверждений 1 и 2 следует, что для заданной команды b и ее входного регистра r существует одна и только одна команда c, такая что $c \xrightarrow{\text{RAW(P,r)}} b$. Поскольку R не противоречит P, то $c \xrightarrow{\text{RAW(R,r)}} b$. Из свойства корректности R и утверждения 1 следует, что $c=a$, следовательно, $a \xrightarrow{\text{RAW(P,r)}} b$.

Утверждение 4. Отношение непротиворечивости транзитивно на множестве планов выполнения участка B.

Доказательство следует из определения непротиворечивости планов.

Из утверждений 3 и 4 следует, что отношение непротиворечивости на множестве допустимых планов выполнения заданного линейного участка является отношением эквивалентности. Соответственно, будем называть взаимно не противоречащие планы *эквивалентными*.

Обозначим через $\mathbf{P}_B$ множество корректных планов, эквивалентных P_B и не содержащих пустых VLIW-строк. Обозначим через cost(P) стоимость последовательности VLIW-строк, соответствующей плану P.

Цель рассматриваемого алгоритма планирования можно сформулировать следующим образом. Для заданного участка $B=(C_B, P_B, I_B, O_B)$ найти допустимый план выполнения P', имеющий минимальную стоимость:

$$\text{cost}(P') = \min_{P \in \mathbf{P}_B}(\text{cost}(P))$$

Существование решения гарантируется конечностью и непустотой множества $\mathbf{P}_B$, которое содержит, по крайней мере, исходный план P_B.

Можно показать, что при выполнении участков $B=(C_B, P_B, I_B, O_B)$ и $B'=(C_B, P, I_B, O_B)$ будут вычислены одинаковые значения всех регистров из O_B, если план P эквивалентен P_B.

Опишем модель выполнения линейного участка подобно тому, как это сделано в [2], и покажем, что при любом плане выполнения $P' \in \mathbf{P}_B$ будут

вычислены те же значения всех регистров из O_B, что и при исходном плане, если B не содержит лишних команд.

Определение. Команда a участка B является *лишней* относительно плана выполнения P, если она не является командой *stop* и множество команд, зависящих от нее относительно P, либо пусто, либо содержит только лишние команды.

Если в участке B нет лишних команд относительно исходного плана, то их не будет и относительно любого плана из $\mathbf{P}_B$, что следует из определений лишних команд и эквивалентности планов.

Из определения лишних команд легко следует также, что любая команда a, следующая за командой *stop* ($P(a) \geq P(stop)$), является лишней. Поэтому, если линейный участок B не содержит лишних команд, то в любом $P \in \mathbf{P}_B$ команда *stop* будет последней.

Пусть $\mathbf{A}_\mathrm{R}$ - множество символов, обозначающих имена регистров. Свяжем с командой *start* набор операторов, присваивающих каждому регистру из I_B символ из $\mathbf{A}_\mathrm{R}$, обозначающий имя этого регистра:

$$q_1 \leftarrow q_1$$

$$\ldots$$

$$q_p \leftarrow q_p$$

где $\{q_1, \ldots, q_p\} = I_B$.

Свяжем с каждой командой a, кроме *start*, набор операторов присваивания

$$q_1 \leftarrow a_1\, r_1 \ldots r_n$$

$$\ldots$$

$$q_k \leftarrow a_k\, r_1 \ldots r_n$$

где k - число выходных операндов команды a,

$\{q_1,..., q_k\} = Out_a$;

$r_1,...,r_n$ - упорядоченная последовательность регистров из In_a; $a_1,..., a_k$ - символы n-местных операций из некоторого конечного алфавита $\mathbf{A}_C$, не пересекающегося с $\mathbf{A}_R$.

Определим значение $v_{t,P}(q)$ регистра q непосредственно после момента времени t относительно допустимого плана выполнения P следующим образом.

- При $t < 0$ $v_{t,P}(q)$ не определено.
- При $t \geq 0$

  1) если $\exists\ a$: $P(a) = t \wedge q \in Out_a$, то $v_{t,P}(q) = a_i\ v_{t\text{-}1,P}(r_1)\ ...\ v_{t\text{-}1,P}(r_n)$, где r_1, ..., r_n - упорядоченная последовательность входных регистров команды a;

  2) иначе если $v_{t\text{-}1,P}(q)$ определено, то $v_{t,P}(q) = v_{t\text{-}1,P}(q)$

  3) иначе $v_{t,P}(q)$ не определено.

Поскольку план P обладает свойством определенности входных данных, все $v_{t\text{-}1,P}(r_i)$ определены.

В случае (1) будем говорить, что значение регистра q вычислено как результат команды a.

Назовем значением $v_P(q)$ регистра q относительно плана выполнения P значение $v_{tmax,P}(q)$, где $tmax = \min\limits_{a \in C_B} P(a)$.

Таким образом, значения регистров определяются как выражения в префиксной записи. Значения считаются равными, если они совпадают как последовательности символов.

Введем понятие *глубины* команды, которое понадобится для доказательства последующих утверждений.

Определение. Команда имеет глубину 0, если она не зависит ни от каких команд. Команда имеет глубину d, если максимальная глубина команд, от которых она зависит, равна d-1.

Глубина команды не зависит от того, какой план выполнения из $\mathbf{P}_B$ используется.

Утверждение 5. Пусть имеется линейный участок $B=(C_B, P, I_B, O_B)$, не содержащий лишних команд, и $P' \in \mathbf{P}_B$. Тогда значения регистров, вычисленные как результат некоторой команды относительно P и P', равны.

Пусть $a \in C_B$ и $q \in Out_a$. Нужно доказать, что имеет место равенство $v_{t,P}(q) = v_{t',P'}(q)$, где $t = P(a)$, $t' = P'(a)$.

Докажем это утверждение индукцией по глубине команды a.

Пусть a имеет глубину 0. Это означает, что a не имеет входных регистров, и значения выходных регистров, вычисляемые как ее результат, определяются следующими присваиваниями:

если $a = start$, то

$q_1 \leftarrow q_1$

...

$q_p \leftarrow q_p$

где $\{q_1, ..., q_p\} = I_B$.

иначе

$q_1 \leftarrow a_1$

...

$q_k \leftarrow a_k$

где k - число выходных операндов команды a,

$\{q_1,..., q_k\} = Out_a$;

$a_1,..., a_k$ - символы 0-местных операций, соответствующих команде a. Таким образом, значения, вычисляемые как результат команды a, являются символами, не зависящими от предшествующих вычислений и от номера VLIW-строки, в которой она выполняется.

Предположим, что утверждение верно для любой команды a, глубина которой не больше чем d-1. Докажем, что оно справедливо и для команд глубины d.

Пусть a - команда глубины d и пусть $P(a) = t$, $P'(a) = t'$. Значения регистра $q \in Out_a$, вычисляемые как результат a относительно P и P' по определению равны, соответственно

$v_{t,P}(q) = a_i\, v_{t\text{-}1,P}(r_1) \ldots v_{t\text{-}1,P}(r_n)$

$v_{t',P'(q)} = a_i\, v_{t'\text{-}1,P'}(r_1) \ldots v_{t'\text{-}1,P'}(r_n)$

где $r_1, ..., r_n$ - упорядоченная последовательность входных регистров команды a;

Для того чтобы значения совпадали, достаточно, чтобы выполнялись равенства

$v_{t\text{-}1,P}(r_1) = v_{t'\text{-}1,P'}(r_1)$

...

$v_{t\text{-}1,P}(r_n) = v_{t'\text{-}1,P'}(r_n)$

Значения $v_{t\text{-}1,P}(r_i)$ и $v_{t'\text{-}1,P'}(r_i)$ вычисляются как результаты некоторой команды b, от которой зависит a, причем b - одна и та же для P и P', поскольку планы эквивалентны. По определению глубина b не превышает d-1, следовательно, по предположению индукции, $v_{t\text{-}1,P}(r_i) = v_{t'\text{-}1,P'}(r_i)$ для всех $i = 1,...,n$. Тем самым утверждение 5 доказано.

Утверждение 6. Пусть имеется линейный участок $B=(C_B, P, I_B, O_B)$, не содержащий лишних команд, и $P' \in \mathbf{P}_B$.

Пусть $q \in O_B$, и в плане выполнения P команда a является последней командой, такой что $q \in Out_a$. Тогда и в плане P' a является последней командой, такой что $q \in Out_a$.

Команда *stop* является последней в P и в P'. По определению $a \xrightarrow{\mathrm{RAW}(P,q)} stop$. Поскольку P' эквивалентен P, то $a \xrightarrow{\mathrm{RAW}(P',q)} stop$, следовательно, и в P' команда a является последней командой, которая пишет значение в регистр q.

Утверждение 7. Пусть имеется линейный участок $B=(C_B, P, I_B, O_B)$, не содержащий лишних команд, и P' - план выполнения, эквивалентный P. Тогда $\forall\, q \in O_B\ v_P(q) = v_{P'}(q)$.

Пусть регистр $q \in O_B$. Значения $v_P(q)$ и $v_{P'}(q)$ вычисляются как результаты последней команды a, такой что $q \in Out_a$. Согласно утверждению 6 такая команда a - одна и та же для P и P'. Согласно утверждению 5, значения, вычисленные как результаты одной и той же команды в P и P', совпадают. Следовательно, $v_P(q) = v_{P'}(q)$.

Таким образом, было доказано, что значения выходных регистров линейного участка, вычисленные относительно любого плана выполнения из $\mathbf{P}_B$, одинаковы.

3.4.6. Алгоритм планирования

Определение. *Частичным планом выполнения* участка B называется отображение в $P':C'_B \rightarrow N$ (где $C'_B \subset C_B$), обладающее свойствами допустимости и сохраняющее отношение зависимости по данным на C'_B.[6]

---

[6] Рассматриваются только частичные планы, не содержащие пустые командные слова.

Алгоритм реализует перебор частичных планов выполнения по методу динамического программирования. Чтобы объяснить работу алгоритма, воспользуемся аналогией с одним из вариантов известного алгоритма списочного планирования [32], где на каждом шаге из множества готовых к исполнению элементарных команд формируется и выводится в выходной поток одна VLIW-строка. При переборе частичных планов всякий раз строится не одна, а все возможные VLIW-строки, которые можно построить из готовых к исполнению элементарных команд. Поскольку требуется получить линейный участок, эквивалентный исходному, то при генерации очередной VLIW-строки из элементарных команд c_1, ..., c_n необходимо соблюдать по крайней мере следующие условия.

1. Условие готовности. Все команды, от которых c_1, ..., c_n зависят по данным, выведены ранее.
2. Условие неразрушения данных. Ни одна из команд c_1, ..., c_n не должна разрушать значение объекта, пока не выведены все команды, которые используют этот объект.

Номер	Команда	Комментарий	Выведена?	Может быть выведена в данном состоянии?
0	START	(фиктивная)	+	
1	move x:(r0),d1.s	; d1:=x:(r0)	+	
2	move y:(r1),d2.s	; d2:=y:(r1)	-	только вместе с 7.
3	fadd.s d1,d2	; d2:=d2+d1	-	нет
4	fmpy.s d6,d7,d0	; d0:=d6*d7	-	да
5	move d2.s,x:(r4)	; x:(r4):=d2	-	нет
6	fmpy.s d4,d4,d2	; d2:=d4*d4	+	
7	fadd.s d2,d3	; d3:=d2+d3	-	да
8	move (r5)+	; r5:=r5+1	-	да

Рис. 3.2. Пример возможного состояния планирования линейного участка. Выведенные команды отмечены символом "+". Справа

показано, какие команды могут быть выведены в этом состоянии

Рис. 3.2 иллюстрирует применение условий готовности и неразрушения данных. Если команды 0, 1 и 6 уже выведены, то на очередном шаге

- безусловно можно вывести команды 4, 7, 8;
- команду 2 можно вывести только совместно с командой 7 (команда 2 разрушает значение регистра d2, установленное командой 6 и используемое командой 7, которая еще не выведена);
- команду 3 нельзя вывести, потому что не соблюдается условие готовности данных - не вычислено входное значение d2 (команду 3 можно вывести только после 2);
- команду 5 нельзя вывести также из-за нарушения условия готовности по d2 (ее можно вывести только после 3).

Составим все возможные VLIW-строки из элементарных команд 2, 4, 7, 8, соблюдая ограничения аппаратной совместимости:

Номера команд	VLIW-строка		
2	move y:(r1),d2.s		
4	fmpy.s d6,d7,d0		
7	fadd.s d2,d3		
8	move (r5)+		
2,4	fmpy d6,d7,d0		y:(r1),d2.s
2,7	fadd.s d2,d3		y:(r1),d2.s
4,7	fmpy d6,d7,d0	fadd.s d2,d3	
4,8	fmpy.s d6,d7,d0		(r5)+
2,4,7	fmpy d6,d7,d0	fadd.s d2,d3	y:(r1),d2.s
4,7,8	fmpy d6,d7,d0	fadd.s d2,d3	(r5)+

Отбросим теперь комбинации, в которых присутствует команда 2, но отсутствует 7. Получается, что, с учетом требований аппаратной

совместимости, готовности и неразрушения данных, в данном состоянии можно сгенерировать любую из перечисленных выше VLIW- строк, кроме (2) и (2,4).

Таким образом, при формировании возможных VLIW-последовательностей имеется большая свобода выбора. Чтобы получить оптимальную выходную последовательность, необходимо организовать перебор возможных VLIW-последовательностей и выбор наилучшей из них в смысле определения, сформулированного в разд. 3.4.5.

Для организации перебора строится граф состояний, где вершинам S соответствуют подмножества команд из C_B, выведенных в выходной поток[7]. Будем называть вершины этого графа также *наборами* или *состояниями перебора.* Начальное состояние соответствует пустому набору команд, конечное – набору C_B. Будем далее отождествлять состояния (вершины графа) и соответствующие им подмножества команд, которые считаются выведенными в данном состоянии. Дуга из S в S' соответствует построению VLIW-строки из множества команд (S' – S). Оптимальный план выполнения вычисляется как самый дешевый путь из начального состояния в конечное. Стоимость пути S_0, ..., S_k определяется суммой стоимостей VLIW-строк [8]:

$$\sum_{i=0}^{k-1} \mathrm{cost}(S_{i+1} - S_i)$$

---

[7] Команда готова к исполнению, если все команды, от которых она зависит по данным, выведены. Множество готовых к исполнению команд определяется только тем, какие команды из C_B уже выведены (и не зависит от плана их выполнения). Следовательно, можно отождествить состояния перебора, которым соответствуют одинаковые множества выведенных команд.

[8] Стоимость необязательно должна представляться скалярным числовым значением. Существенным требованием к определению стоимостей и операций на над ними является требование монотонности сложения: если α, β, γ - стоимости последовательностей командных слов, и $\alpha<\beta$, то должно выполняться $\alpha+\gamma<\beta+\gamma$.

Рис. 3.3 показывает набор S_0, соответствующий состоянию обработки линейного участка, которое изображено на рис. 3.2. На рис. 3.3. представлены также наборы, непосредственно достижимые из S_0.

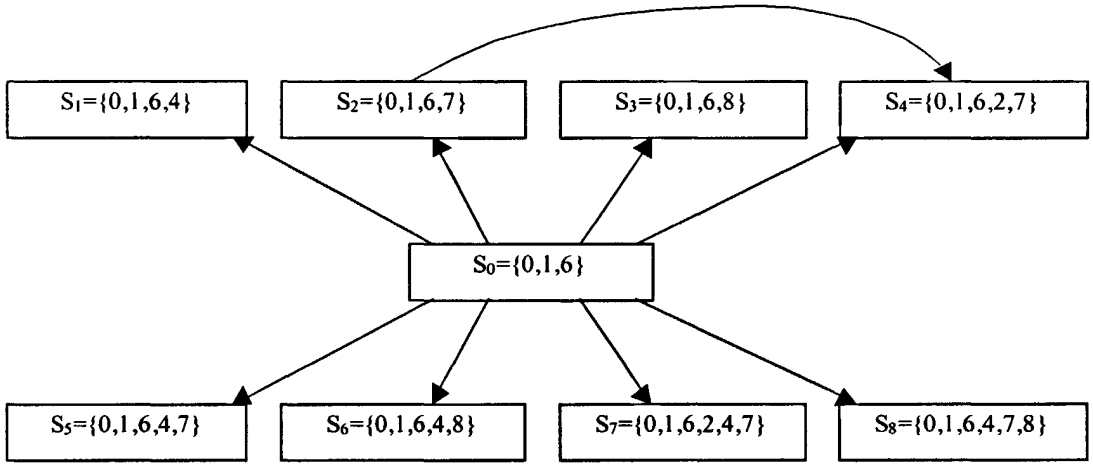


Рис. 3.3. Фрагмент графа наборов. Показан набор S_0, соответствующий состоянию, которое изображено на рис. 3.2, а также все наборы, непосредственно достижимые из него

Так как длина дуги определяется как время выполнения VLIW-строки, составленной из команд $c_1,...,c_n$, то длина дуги из S_0 в S_4 на рис. 3.3 равна времени выполнения VLIW-строки, составленной из элементарных команд 2, 7, т.е. строки

```
fadd.s d2,d3 y:(r1),d2.s
```

(см. рис. 3.2).

Из одного набора в другой может вести несколько путей. Например, из S_0 в S_4 (рис. 3.3) можно попасть непосредственно или через вершину S_2.

Назовем начальной вершиной (S_{begin}) пустой набор, а конечной (S_{end}) - набор, содержащий все команды линейного участка. Каждому пути, ведущему из S_{begin} в S_{end}, соответствует определенная VLIW-последовательность, эквивалентная исходной. Задача состоит в том, чтобы

найти самый короткий путь, соответствующий самой быстрой VLIW-последовательности.

Важно отметить, что по крайней мере один путь из начальной вершины в конечную существует. Он определяется VLIW-последовательностью, которая соответствует исходному плану выполнения P_B. Будем называть этот путь тождественным.

В рамках поставленной задачи имеет смысл рассматривать только часть графа, состоящую из наборов, которые достижимы из начальной вершины. Поскольку множество путей, исходящих из вершины S_i, зависит только от содержимого ее набора и не зависит от путей, ведущих в S_i, то рассматриваемый алгоритм поиска пути можно отнести к методам динамического программирования.

Рассмотрим алгоритм планирования более детально. Идея состоит в том, чтобы строить все наборы, непосредственно достижимые из ранее построенных, двигаясь от начальной вершины и перебирая существующие наборы в порядке неубывания их мощности.

Каждому создаваемому набору S_i приписывается следующая информация:

1. стоимость набора $\mathrm{Cost}(S_i)$ - длина кратчайшего (из построенных до сих пор) пути из S_{begin} в S_i;
2. набор S_k=$\mathrm{Pred}(S_i)$, такой что дуга (S_k, S_i) принадлежит кратчайшему пути из S_{begin} в S_i.

Построив таким образом все пути из S_{begin} в S_{end}, можно затем восстановить (в обратном направлении) оптимальный путь: S_{end}, $\mathrm{Pred}(S_{\mathrm{end}})$, $\mathrm{Pred}(\mathrm{Pred}(S_{\mathrm{end}}))$, ... Соответствующую ему последовательность VLIW-строк также можно восстановить из наборов, через которые проходит кратчайший путь.

На рис. 3.4 показан фрагмент графа наборов и отмечен кратчайший путь из начальной вершины в конечную. Из последовательности наборов S_{end}={1,2,3,4,5,6,7,8}, Pred(S_{end})={1,2,3,5,6}, Pred(Pred(S_{end}))={1,2,3}, Pred(Pred(Pred(S_{end})))={}, можно восстановить оптимальную последовательность VLIW-строк.

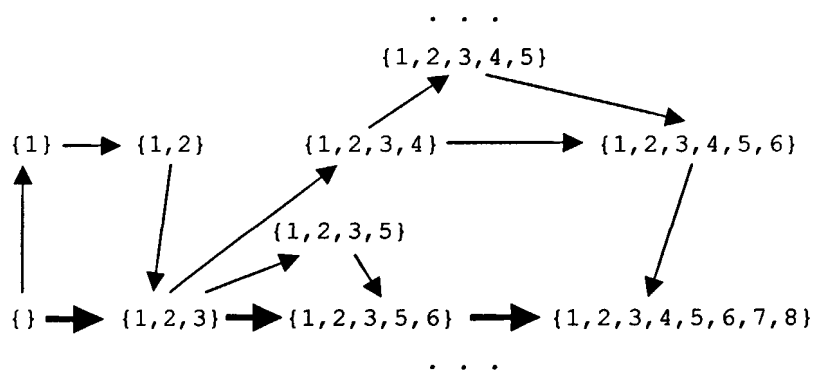


Рис. 3.4. Фрагмент графа наборов. Отмечен кратчайший путь от начальной вершины к конечной

Ниже приведен алгоритм построения графа наборов с полным перебором всех путей из начальной вершины в конечную. Далее в п. 3.4.7 описано уточнение алгоритма, позволяющее учесть аппаратные задержки, а в п. 3.4.8 рассматриваются реализованные способы сокращения перебора.

{

Создать начальную вершину S_{begin} - пустой набор.

Перебирать существующие наборы в порядке неубывания их мощности; для каждого набора S вычислить множество всех исходящих из него дуг и создать все непосредственно достижимые наборы следующим образом.

{

Построить для набора S множество команд, удовлетворяющих условию готовности данных.

Из этого множества сформировать все аппаратно корректные VLIW-строки, исключая строки, противоречащие условию неразрушения данных. Пусть $W_1,..., W_n$ - множество полученных в результате VLIW-строк.

Для всех W_i, i=1,...,n выполнять следующее.

{

Вычислить набор $S_i = S \cup \{c_1, ..., c_m\}$, где $c_1, ..., c_m$

- элементарные команды, из которых состоит W_i.

Если S_i еще не существует в множестве построенных наборов, то:

{

добавить S_i к имеющимся наборам;

вычислить $\mathrm{Cost}(S_i) = \mathrm{Cost}(S) + \mathrm{cost}(W_i)$;

положить $\mathrm{Pred}(S_i)=S$.

}

Если S_i уже имеется в множестве построенных наборов, то это означает, что найден другой путь из S_{begin} в S_i. В этом случае:

{

вычислить стоимость найденного пути из S_{begin} в S_i:

$\mathrm{Cost}'(S_i) = \mathrm{Cost}(S) + \mathrm{Cost}(W_i)$.

если $\mathrm{Cost}'(S_i) < \mathrm{Cost}(S_i)$, то положить

$\mathrm{Cost}(S_i)=\mathrm{Cost}'(S_i)$, $\mathrm{Pred}(S_i)=S$.

}

}

}

Построить VLIW-последовательность, соответствующую кратчайшему пути из S_{begin} в S_{end}, по наборам S_{end}, Pred(S_{end}), Pred(Pred(S_{end})), ...

}

Перебор в порядке неубывания мощности наборов является важным условием. Он гарантирует, что к моменту, когда мы начинаем строить дуги из набора S, уже пройдены все возможные пути из S_{begin} в S (по определению все пути в графе направлены от наборов меньшей мощности к наборам большей мощности); следовательно, стоимость S уже определена как длина кратчайшего пути из S_{begin} в S.

3.4.7. Учет аппаратных задержек

Корректность VLIW-строки не всегда определяется ей самой и набором S. Результат операции может быть доступен не сразу, а только через несколько тактов. Например, в программе для процессора 1B577, если непосредственно после VLIW-строки W_1, где есть присваивание адресному регистру r4

```
move d1.1,r4
```

следует строка W_2, в которой r4 используется для формирования адреса, то ассемблер вставит между W_1 и W_2 пустую команду. Таким образом, в этом контексте стоимость строки W_2 возрастает.

Следовательно, для правильной оценки стоимостей нужно знать, какие элементарные команды вошли в предыдущую VLIW-строку. Поэтому в определение вершины включается еще один элемент - множество регистров, использование которых в следующей строке ограничено. В результате вместо одной вершины, соответствующей набору S, можно получить несколько вершин:

$\{S, regs_1\},..., \{S, regs_k\}$

где $regs_i$ - множества регистров, использование которых в следующей VLIW-строке ограничено. Множества $regs_i$ определяются составом VLIW-строки, соответствующей последней дуге пути, по которому мы пришли в набор S.

Вершина [S,{r4}] отличается от вершины [S,{}] тем, что время выполнения VLIW-строк, содержащих адресацию по регистру r4, увеличивается соответственно аппаратной задержке (см. рис. 3.5).

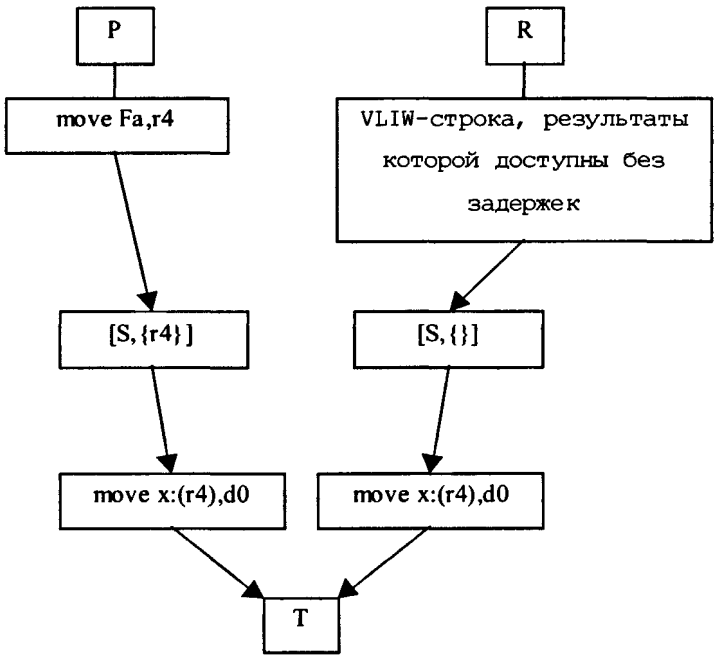


Рис. 3.5. Длина дуги [S,{r4}]→T больше, чем длина дуги [S,{}]→T, хотя им соответствует одна и та же VLIW-строка "move x:(r4),d0". Это позволяет учесть аппаратную задержку, связанную с установкой значения r4 после выполнения "move Fa,r4"

Так как команд, результат которых доступен не сразу, немного, то общее число вершин возрастает незначительно. Притом, это "последействие" исчезает на следующей строке. В дальнейшем изложении вершины будут по-прежнему отождествляться с наборами обработанных

команд, пренебрегая для простоты этим дополнительным фактором, влияющим на стоимость VLIW-строк.

3.4.8. Сокращение перебора

Число вершин в графе состояний, который приходится строить в процессе перебора частичных планов, очень быстро растет в зависимости от числа команд в C_B. При этом построение каждой вершины и каждой дуги требует довольно больших вычислительных затрат. Рассматриваемые далее методы оптимизации перебора направлены, с одной стороны, на сокращение времени обработки каждой вершины (где под обработкой вершины понимается построение всех исходящих из нее дуг и вершин-преемников), с другой - на сокращение числа вершин в графе.

Для сокращения размера рассматриваемой части графа и ускорения поиска применяются следующие эвристики:

- просеивание наборов;
- ограничение числа наборов одинаковой мощности.
- исключение бесполезных состояний.

Первые два подхода носят эвристический характер и предполагают сужение пространства поиска, т.е. исключение из рассмотрения некоторых допустимых планов выполнения из $\mathbf{P}_B$. Эти подходы были реализованы в первой версии постпроцессора.

Третий подход предполагает сокращение перебора путем исключения из графа перебора только таких состояний, из которых конечная вершина не достижима. Его реализация во второй версии постпроцессора позволила добиться премлемого времени постпроцессирования и практически отказаться от применения первых двух подходов, которые “включаются” только при обработке очень длинных линейных участков (порядка нескольких сотен команд).

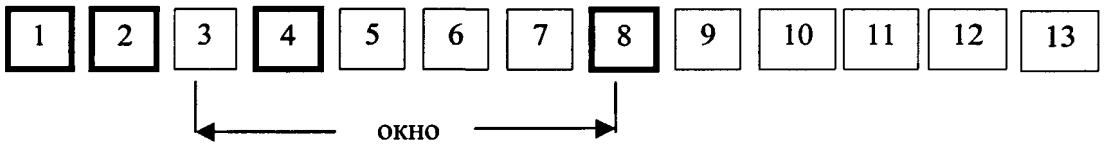
Просеивание наборов. Можно попытаться выделить какое-либо свойство наборов, и искать пути, проходящие только через наборы с этим свойством, называемые далее выделенными наборами. Естественно, свойство должно быть определено так, чтобы всегда нашелся хотя бы один путь, и чтобы стоимость лучшего пути, проходящего через выделенные наборы, не слишком отличалась от стоимости оптимального пути. Важно также, чтобы свойство проверялось достаточно легко, и желательно, чтобы выделенные наборы можно было кодировать короче, чем произвольные.

При анализе работы постпроцессора было замечено, что в оптимальных выходных линейных участках команды редко смещаются (по отношению к своему положению в исходном линейном участке) более чем на 15-20 позиций.

Объясняется это следующим образом. Если участок маленький, то сдвигаться некуда. Если участок большой, то задействованы все регистры, и зависимости по данным не дают команде передвинуться. Даже если связи по данным позволяют сместить команду далеко от ее исходного положения, это, как правило, приводит к получению неоптимального плана выполнения. Вслед за ней придется переместить другие команды, которые от нее зависят, что увеличит продолжительность жизни значений (расстояние между командой, пишущей значение, и последней командой, читающей это значение), уменьшит число допустимых строк в дальнейшем и, вероятнее всего, в результате общий итог ухудшится - будут выбираться "лучшие из худших".

Исходя из этих наблюдений, выделим наборы со следующим свойством. Определим размер окна набора как разность между максимальным номером команды, входящей в набор, и минимальным номером команды, не входящей в него (рис. 3.6). Номер команды определяется ее положением в исходном линейном участке. Если разность не определена или отрицательна, то размер окна полагается нулевым.

Будем считать выделенными наборы, у которых размер окна не превышает заданного положительного параметра W_{MAX}. По крайней мере один путь, проходящий только через выделенные наборы, существует для любого W_{MAX}, поскольку всегда имеется тождественный путь, проходящий через наборы с нулевым размером окна:



Жирными рамками отмечены команды, принадлежащие набору. Размер окна в данном случае равен 5.

Рис. 3.6. Окно набора и его размер

Мощность заданного таким образом множества выделенных наборов ограничена сверху функцией, линейно зависящей от числа команд в исходном участке L_B. Следовательно, время поиска на множестве выделенных наборов зависит от L_B почти линейно (т. к. независимо от способа кодирования, размер представления набора зависит от L_B.

Ограничив рассматриваемую часть графа множеством выделенных наборов, получим также выигрыш во времени, которое затрачивается на построение списка команд, удовлетворяющих свойству готовности данных, поскольку область их поиска ограничивается размером окна.

Ограничение числа наборов одинаковой мощности. Это дополнительное ограничение введено по той причине, что просеивание не всегда оказывается достаточным. Если в исходном линейном участке есть длинная последовательность команд, не зависящих друг от друга, то можно получить $2^{W_{MAX}}$ наборов одинаковой мощности. Именно эта

константа - $2^{W_{MAX}}$ - фигурирует в определении линейной функции, ограничивающей число выделенных наборов.

Чтобы избежать рассмотрения слишком большого числа наборов, постпроцессор ограничивает количество наборов одной мощности, оставляя не более N_1 наиболее удачных (с наименьшими стоимостями) и не более N_2 "самых выделенных" (с наименьшими размерами окон). Первые нужны, чтобы сохранить оптимальные пути, последние - для того, чтобы гарантировать нахождение хотя бы одного пути (напомним, что к числу "самых выделенных" относится тождественный путь, который существует для любого линейного участка). Если оставлять только наиболее удачные пути, то существует риск, что все они заведут в тупики.[9]

Номер	Команда	Комментарий	Обработана?	Может быть обработана в данном состоянии?
1	move x:(r0),d0.s	; d1:=x:(r0)	-	только вместе с 4
2	fadd.s d0,d1	; d1:=d1+d0	-	нет
3	move y:(r1),d0.s	; d0:=y:(r1)	+	
4	fmpy.s d0,d1,d2	; d2:=d1*d0	-	нет

Рис. 3.7. Пример тупикового набора

На рис. 3.7 показан пример линейного участка и тупикового набора (S={3}). Если обработана только команда 3, то из оставшихся элементарных команд нельзя составить ни одной VLIW-строки, удовлетворяющей условиям готовности и неразрушения данных:

- для команды 2 не готово значение d0, поскольку не обработана команда 1;

[9] Тупик - это набор, отличный от конечной вершины, из которого нельзя построить ни одной дуги в другие наборы. Тупиковый набор характеризуется тем, что множество строк, удовлетворяющих свойствам готовности и неразрушения данных, для него пусто.

- для команды 4 не готово значение d1, поскольку не обработана команда 2.

Данные для команды 1 готовы, но VLIW-строку, составленную из одной этой команды придется отбросить, поскольку она испортит значение d0, которое требуется для команды 4.

Следует отметить, что свойства удачности и выделенности не противоречат друг другу. Пути с наименьшим размером окна лежат в окрестности неоптимального тождественного пути. Но через те же наборы могут проходить и самые короткие пути. Рассмотрим, например, случай, когда оптимальное решение заключается в том, чтобы комбинировать по несколько подряд стоящих элементарных команд. Тогда соответствующий путь будет проходить через наборы, имеющие нулевой размер (рис. 3.8).

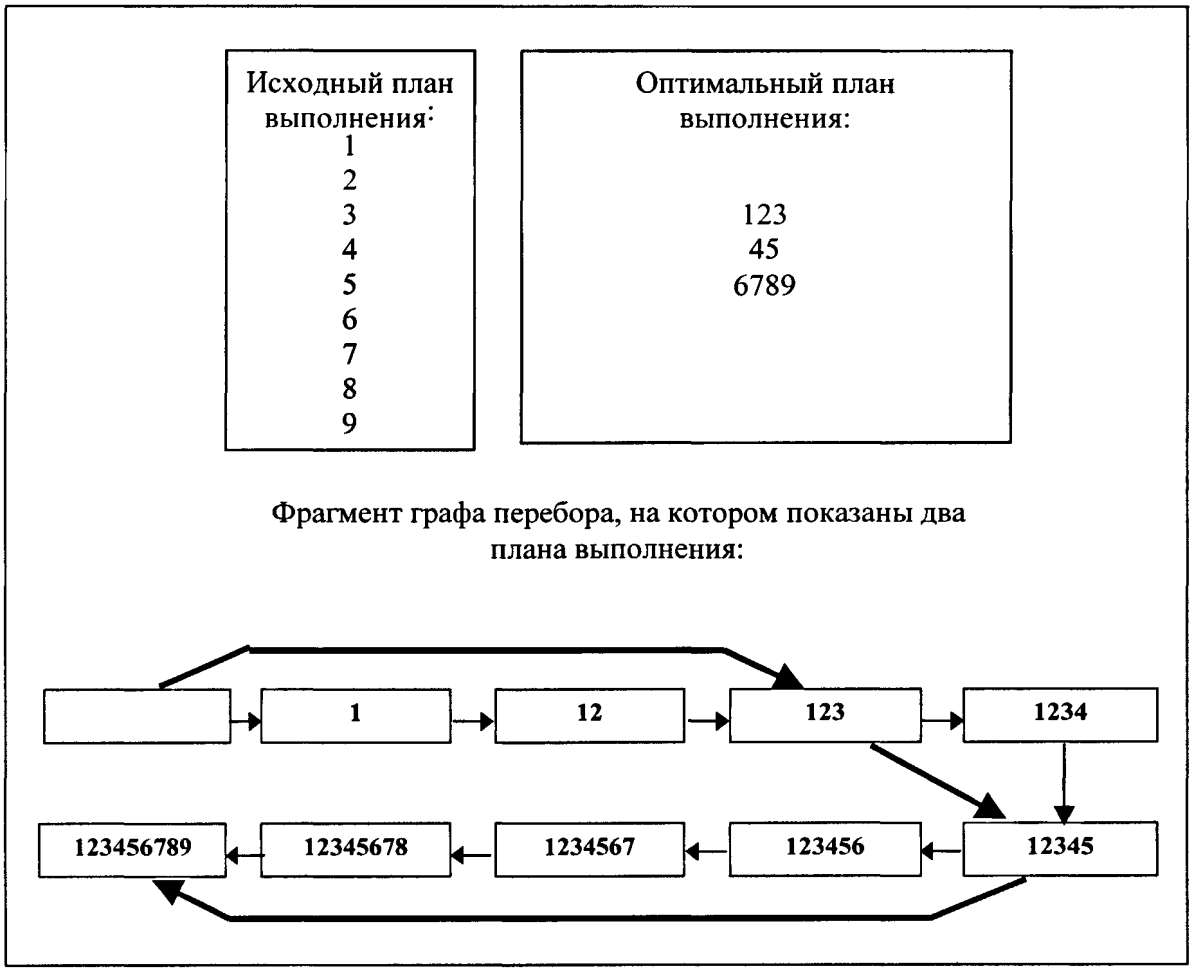


Рис. 3.8. Пример оптимального пути, проходящего через наборы с нулевым размером окна

Оправданность выбранного подхода экспериментально подтверждается тестированием постпроцессора на программах с длинными линейными участками. При разумном выборе параметров оптимизации результаты оказываются такими же, как при полном переборе, при том что время постпроцессирования может сокращаться в десятки и сотни раз.

Исключение бесполезных состояний. В отличие от эвристических способов ограничения перебора, которые были рассмотрены выше,

описанные далее оптимизации не сужают пространство решений $\mathbf{P}_B$, исключая лишь бесполезные состояния.[10]

Один из ключевых типов сущностей, с которыми приходится иметь дело при построении графа, - области жизни значений. Назовем читателями регистра r, записанного командой a, команды, которые зависят от a по регистру r:

$$\text{Readers}(a,r) = \{b \in C_B \mid a \xrightarrow{\text{RAW(r)}} b\}$$

Областью жизни регистра r назовем пару $L(r) = (a, \text{Readers}(a,r))$. Линейный участок может содержать несколько областей жизни одного регистра, если тому на протяжении участка присваиваются различные значения.

С точки зрения сохранения зависимостей по данным при построении частичных планов важное значение имеет понятие *открытой* области жизни. Область жизни $L(r)=(a, \text{Readers}\ (a,r))$ называется открытой в состоянии S, если a уже выведена в выходной поток, а хотя бы одна из команд-читателей - еще нет: $a \in S$ и $\text{Readers}(a, r) \not\subset S$.

Пусть в некотором состоянии S область жизни $L(r)=(a, \text{Readers}(a,r))$ открыта и пусть $\exists\ b \notin S$, готовая к исполнению и такая что $r \in Out_b$. При построении состояний-преемников можно строить только такие VLIW-строки, которые либо не содержат b, либо включают ее одновременно со всеми оставшимися командами из $\text{Readers}(a,r)$ - иначе b испортит значение регистра r, которое еще требуется командам из множества $\text{Readers}(a,r)$. Можно сформулировать это требование иначе: ни в каком состоянии перебора не может быть открыто более одной области жизни одного и того же регистра.

---

[10] Под бесполезным понимается состояние, из которого конечное состояние недостижимо - все пути из него ведут в тупики, то есть в неконечные состояния, из которых не исходит ни одной дуги.

Поскольку областей жизни в большом линейном участке много, на их отслеживание затрачивается значительное время. Можно сократить число отслеживаемых областей жизни, если в исходном участке некоторые регистры всегда используются совместно. Например, если группа регистров (r_i, r_{i+1}) используется для хранения "длинных" значений, то области жизни $L(r_{i+1})$ можно исключить из рассмотрения, а следить только за областями жизни $L(r_i)$.

Еще более значительной экономии вычислительных затрат можно достичь за счет использования отношений следования и несовместимости на множестве команд.

Определение. Команды a, b несовместимы, a $\nparallel$ b, если их параллельное выполнение невозможно: $P(a) \neq P(b)\ \forall\ P \in \mathbf{P}_B$.

Определение. Команда a строго предшествует b, $a \prec b$, если $P(a) < P(b)\ \forall\ P \in \boldsymbol{P}_{\mathrm{B}}$ (в любом допустимом плане исполнения a выполняется раньше, чем b). Будем также говорить в этом случае, что b строго следует за a.

Определение. Команда a нестрого предшествует b, $(a \preceq b)$, если $P(a) \leq P(b)\ \forall\ P \in \mathbf{P}_B$ (в любом допустимом плане исполнения a выполняется не позже, чем b). Будем также говорить в этом случае, что b нестрого следует за a.

Допустим, что мы умеем вычислять эти отношения, и посмотрим, как ими можно воспользоваться для оптимизации перебора.

Оптимизация 1. *Уменьшение числа областей жизни, отслеживаемых при переборе.* Пусть имеется область жизни $L(r)$=(a, Readers(a,r)) и все другие команды, пишущие в регистр r, либо строго предшествуют a, либо строго следуют за всеми командами из Readers(a,r). Тогда область $L(r)$ можно исключить из рассмотрения при переборе.

Оптимизация 2. *Уменьшение размеров областей жизни.* Если сформулированное выше условие выполняется не для всех команд из Readers(a,r), то в Readers(a,r) можно выделить подмножество команд, для которых оно не выполняется, и отслеживать только их.

Сокращение числа и размеров отслеживаемых областей жизни проводится до начала перебора.

Оптимизация 3. *Вычисление множества команд, готовых к исполнению.* Во время обработки состояний можно сузить множество команд, готовых к исполнению, и ускорить его вычисление следующим образом. Пусть имеется состояние S. Множество готовых к исполнению команд E для него определяется как $\hat{S}-\hat{E}$, где $\hat{S}$- множество невыполненных, $\hat{E}$- множество неготовых. Множество неготовых, в свою очередь, определяется как множество команд, которым строго предшествует хотя бы одна из невыполненных: N

$$\hat{E} = \{a \in C_B \mid \exists\, b \in \hat{S} : b \prec a\}$$

Тем самым мы сужаем множество готовых к исполнению команд, по сравнению с подходом, основанном только на анализе готовности входных данных для команд. В примере 4.2 далее в этом разделе будет показано, как это помогает сократить перебор.

Оптимизация 4. *Дополнительный контроль VLIW-строк.* При построении множества всех допустимых VLIW-строк W_i в состоянии S будем проверять следующее соотношение:

$$W'_i \subset (S \cup W_i)$$

где W'_i - множество команд, нестрого предшествующих командам из W_i:

$$W'_i = \{a \in C_B \mid a \leq b \;\forall\; b \in W_i\}$$

Такая проверка обеспечивает, что команды a, b, такие что $a \preceq b$ и $b \preceq a$, всегда помещаются в одну VLIW-строку.

Применение этой проверки демонстрируется в примере 4.4 далее в этом разделе.

Будем называть перебор планов выполнения с использованием оптимизаций 1-4 на основе отношений несовместимости и предшествования команд *оптимизированным перебором*. Перебор без применения этих правил будем называть *неоптимизированным*.

Рассмотрим способы вычисления отношений несовместимости и предшествования. Фактически будут использоваться их приближения, являющиеся подмножествами определенных выше отношений. В разделе 3.4.9 обсуждаются возможные уточнения способов вычисления отношений с учетом вариантности команд.

Приближенные отношения несовместимости и предшествования определим при помощи следующих правил вывода

Инициализирующие правила:

I. Команды a и b несовместимы, если они либо несовместимы аппаратно, либо производят запись в один регистр:

$$(!\text{compatible}\ (\{a, b\})\ \text{или}\ Out_a \cap Out_b \neq \emptyset) \Rightarrow a \not| b.$$

II. Нестрогое предшествование рефлексивно:

$$\forall\, a \in C_B\ a \preceq a$$

III. Строгое предшествование содержит отношение зависимости по данным:

$$a \xrightarrow{\text{RAW}} b \Rightarrow a \prec b$$

Правила вывода:

IV. Нестрогое предшествование включает строгое предшествование:

$$a \prec b \Rightarrow a \preceq b$$

V. Нестрогое предшествование транзитивно:

$$\exists\, c\colon (a \preceq c \text{ и } c \preceq b) \Rightarrow \mathrm{a} \preceq \mathrm{b}$$

VI. Строгое предшествование содержит пересечение нестрогого предшествования и несовместимости:

$$a \preceq b \text{ и } a \nparallel b \Rightarrow \mathrm{a} \prec \mathrm{b}$$

VII. Строгое предшествование содержит пересечение строгого предшествования на нестрогое и произведение нестрогого предшествования на строгое:

$$\exists\, c\colon (a \prec c \preceq b \text{ или } a \preceq c \prec b) \Rightarrow a \prec b$$

Правило VII вместе с двумя предыдущими гарантирует транзитивность приближенного отношения строгого предшествования.

VIII. Отношение несовместимости содержит отношения строгого предшествования:

$$a \prec b \Rightarrow a \nparallel b$$

IX. Отношение несовместимости симметрично:

$$a \nparallel b \Rightarrow b \nparallel a$$

Следующие два правила позволяют вычислить отношения предшествования, превышающие рекурсивное замыкание отношения зависимостей по данным:

X. Если команды $a \preceq b$ обе пишут в регистр r, то все команды из Readers (a, r) нестрого предшествуют b:

$$(a \preceq b) \text{ и } (r \in Out_a \cap Out_b) \Rightarrow$$

$$\forall\, c \in \mathrm{Readers}(a,r)\; c \preceq b.$$

XI. Если команды a, b пишут в регистр r, и a строго предшествует хотя бы одной команде из Readers (b, r), то a строго предшествует b:

$$(r \in Out_a \cap Out_b) \text{ и } (\exists\, c \in \text{Readers}\,(b, r)\colon a \prec c) \Rightarrow a \prec b$$

Вычисление отношений начинается с применения инициализирующих правил I, II, III, после чего правила вывода IV - XI применяются итеративно до достижения неподвижной точки.

Рассмотрим примеры, демонстрирующие эффект применения отношений.

Пример 3.2.

c_1 | D := 5

c_2 | A := D*D

c_3 | B := A*D

c_4 | A := 3

c_5 | C := A*B

Если не использовать отношение предшествования, то в начальном состоянии список готовых к исполнению команд будет состоять из c_1 и c_4. Если допускается комбинирование присваиваний констант, то планировщик сгенерирует в начальном состоянии следующие VLIW-строки: $\{c_1\}$, $\{c_4\}$, $\{c_1, c_4\}$, соответствующие им состояния $S_1 = \{c_1\}$, $S_2=\{c_4\}$, $S_3= \{c_1, c_4\}$, из которых S_2 и S_4 - тупиковые.

Вычислим отношения предшествования для этого участка:

Из правила II (свойство рефлексивности) и из правила III (строгое предшествование содержит отношение зависимости по данным) получаем:

$$c_1 \preceq c_1,\ c_2 \preceq c_2,\ c_3 \preceq c_3,\ c_4 \preceq c_4,\ c_5 \preceq c_5$$

$$c_1 \prec c_2,\ c_1 \prec c_3,\ c_2 \prec c_3,\ c_3 \prec c_5,\ c_4 \prec c_5$$

Из правила IV получаем дополнительно:

$$c_1 \preceq c_2, c_1 \preceq c_3, c_2 \preceq c_3, c_3 \preceq c_5, c_4 \preceq c_5$$

Из правила V (транзитивность нестрогого предшествования) получаем:

$$c_1 \preceq c_5, c_2 \preceq c_5$$

Поскольку c_5 несовместимо ни с c_1, ни с c_2, то по правилу VI

$$c_1 \prec c_5, c_2 \prec c_5$$

Правило VII в данном случае не добавляет ничего. Получаем, что

$$c_1 \prec c_2 \prec c_3 \prec c_5 \text{ и } c_4 \prec c_5$$

По правилу X добавляем $c_2 \prec c_4$, по правилу IV — $c_2 \preceq c_4$.

По правилу XI добавляем $c_3 \preceq c_4$ и (по правилу VI) $c_3 \prec c_4$. Таким образом, получаем линейный порядок $c_1 \prec c_2 \prec c_3 \prec c_4 \prec c_5$. Используя правило для вычисления множества готовых к исполнению команд (см. выше «Оптимизация 3. *Вычисление множества команд, готовых к исполнению*»), получим граф перебора, содержащий единственный путь от S_0 к S_5, соответствующий исходному плану исполнения.

Очевидно, что при планировании более сложных линейных участков могут возникать бесполезные состояния, отстоящие от тупиков достаточно далеко, т.е. экономия, получаемая от исключения бесполезных состояний за счет применения описанного выше метода для вычислений множества преемников каждого состояния, может быть очень велика.

Заметим, что экономия времени достигается не только за счет уменьшения числа состояний в графе, но и за счет ускорения их обработки, поскольку в большинстве случаев при использовании отношений сокращается число готовых к исполнению команд и, следовательно, число VLIW-строк, которые нужно создать и проверить на совместимость.

В таблице 4.1 приведена статистика числа состояний в графе перебора и времени компиляции при неоптимизированном и оптимизированном

переборе. В столбце 1 приведен размер модуля (число команд в неоптимизированном ассемблерном коде) и максимальная длина линейного участка. В столбцах 2-3 показано число состояний и число тупиков, а также время компиляции модуля при неоптимизированном переборе. В столбцах 4-5 приведены те же сведения для оптимизированного перебора. В последней строке приведены сводные данные для набора из 70 тестовых модулей общим размером более 1800 строк на языке Си.

Таблица 3.1 Число состояний и время компиляции при неоптимизированном и оптимизированном переборе

Тесты	**Неоптимизированный перебор**		**Оптимизированный перебор**	
Число команд/ макс.участок	Число состояний/ тупиков	Время компиляции	Число состояний/ тупиков	Время компиляции
118/118	121050/3221	159.39 сек	3435/23	3.02 сек.
215/189	48263/3137	75.85 сек.	1356/11	4.01 сек.
277/63	22016/276	16.07 сек.	9034/26	5.64 сек.
108/33	2020/90	1.69 сек.	1258/52	1.13 сек.
67/28	390/12	0.55 сек.	321/8	0.54 сек.
54/25	316/14	0.44 сек.	304/11	0.43 сек.
Набор из 70 тестов 9681/661	469653/20298	9 мин.	140831/5135	3 мин.

Важно отметить, что принципиально невозможно определить на множестве команд бинарное отношение предшествования, не сужающее пространства решений $\mathbf{P}_B$ и позволяющее отсеивать все бесполезные состояния. Это можно продемонстрировать на следующем примере.

Пример 3.3

c_1 | A := 2

c_2 | D := A*A ;; с разрушением B

c_3 | B := 3

c_4 | E := B*B ;; с разрушением A

Множество выходных регистров этого линейного участка $O_B = \{D, E\}$. В командах c_2, c_4 определено по два выходных регистра - $Out_{c2} = \{D, B\}$, $Out_{c4} = \{E, A\}$.

Первую пару команд можно поменять местами со второй. Отношение строгого порядка, не запрещающее такую перестановку, содержит не более двух пар команд - (c_1, c_2) и (c_3, c_4). Оно не гарантирует от попадания в тупики, позволяя, например, на первом шаге выдать в выходной поток пару $\{c_1, c_3\}$.

Пример 3.4. Рассмотрим линейный участок *B* вида

```
{start}
{a="A:=B", b="B:=A"}
{stop}
```

где $I_B = O_B = \{A, B\}$.

Очевидно, в любом допустимом плане P команды *a, b* должны выполняться одновременно: $P(a) = P(b)$. Вычислив отношение нестрогого порядка для этого линейного участка (с применением правила X), получим, что $a \preceq b$ и $b \preceq a$. Применяя правило дополнительного контроля VLIW-строк (см. выше Оптимизация 4), удается исключить при переборе генерацию VLIW-строк, содержащих только одну из команд *a* или *b*, сокращая таким образом число бесполезных состояний.

3.4.9. Подбор вариантов команд

Возможности для комбинирования можно существенно расширить, вводя варианты элементарных команд.

Вариантом элементарной команды называется другая элементарная команда, которая выполняет то же действие, но обладает другими свойствами комбинируемости. Например, в процессоре 1В577 присваивание значения одного регистра другому можно выполнить при помощи арифметической команды tfr или при помощи команды пересылки move. Команда tfr комбинируется с пересылками данных; move, напротив, комбинируется с арифметическими командами. Во многих случаях эти команды взаимозаменяемы. Воспользовавшись этим свойством, можно создавать дополнительные комбинации из готовых к исполнению команд.

Заменим каждую команду линейного участка списком ее вариантов. Все варианты должны читать одинаковые регистры; множества выходных регистров у них могут различаться. Например, команда tfr, помимо присваивания значения одного регистра другому, устанавливает также код условия, а move - нет. Важно, что, если регистр записывается несколькими вариантами, то во всех случаях в него записывается одно и то же значение. Если некоторый вариант команды записывает какой-то объект и потом это значение используется, то для данного линейного участка постпроцессор не будет рассматривать варианты, не записывающие этот регистр.

В постпроцессоре для 1В577 варианты применяются довольно широко. Например, вместо команды очистки регистра "clr d0" может использоваться команда вычитания "sub d0,d0", которая может выполняться параллельно с умножением. Команды обмена с памятью имеют варианты для доступа к памяти по шинам x: и y:, которые комбинируются друг с другом. Команда умножения вещественных данных имеет два варианта, один из которых записывает код условия, а второй - нет; второй вариант аппаратно совместим со сложением.

С учетом поддержки вариантов, правила вывода X, XI, используемые для расчета отношений, формулируются несколько иначе:

X. Если команда a пишет в регистр r, $a \preceq b$, и все варианты команды b пишут в регистр r, то все команды из Readers (a, r) нестрого предшествуют b.

XI. Если все варианты команды a пишут в регистр r, команда b пишет в регистр r и a строго предшествует хотя бы одной команде из Readers (b, r), то $a \prec b$.

Расчет отношений можно сделать более точным, если вычислять также отношения между командами и вариантами команд и учитывать несовместимость по динамическим ресурсам (о динамических ресурсах см. 3.5.8). Можно рассчитывать предшествование не в форме бинарных отношений, а как функцию расстояния Dist: C_B x $C_B \rightarrow N$, которая для каждой пары команд (a, b) дает оценку снизу для разности номеров строк ($\min_{P \in \mathbf{P}_B} P(b) - P(a)$), в которые могут быть запланированы команды a, b.

Более точных расчетов решено было не делать по двум причинам. Во-первых, расчет отношений сам по себе требует значительных затрат времени и памяти, и его усложнение имеет смысл до какого-то предела. Во-вторых, как показано в разд. 3.4.8 (примере 4.3), невозможно определить бинарное отношение предшествования на множестве команд, полностью устраняющее бесполезные состояния при переборе планов выполнения.

3.4.10. Модификация команд

Механизм вариантов оказался удобен еще для одной цели: модификации команд. Поясним применение модификации на примере, приведенном в разд. 3.4.1 — замена последовательности команд

- `move x:(r4),d1.s` (записать в d1 значение из памяти, считанное по адресу, хранящемуся в регистре r4)
- `move (r4)+n4` (увеличить r4 на значение n4)

на одну команду `move x:(r4)+n4,d1.s`, которая выполняет оба действия.

Модификация в данном случае работает следующим образом. Вместо команды "`move (r4)+n4`" создается пара вариантов {"`move (r4)+n4`"; "`.replace '(r4)' '(r4)+n4'`"}, где второй вариант является псевдокомандой подстановки режима адресации ('(r4)' → '(r4)+n4'). Подстановке сопоставляются дополнительные ограничения, которые позволяет комбинировать ее только с командами пересылки в память/из памяти вида "move @:(r4),*" или "move *,@:(r4)", где "*" обозначает любой регистр, а "@" - шину x или y.

Если вариант ".replace '(r4)' '(r4)+n4'" был скомбинирован с командой пересылки в память или из памяти, то при выводе последней в выходной ассемблерный файл в ней производится подстановка '(r4)' → '(r4)+n4'.

3.5. Настройка постпроцессора на архитектуру 1В577.

В этом разделе рассматривается состав и внешнее представление информации о целевой архитектуре в постпроцессоре. Как уже отмечалось, состав информации определяется набором оптимизаций и способами их реализации. Например, для комбинирования команд необходимо знать, какие функциональные устройства нужны для выполнения команды каждого вида. В то же время, поскольку в постпроцессоре семантика входной программы не анализируется, то информация о семантике команд не нужна.

Описание целевой платформы для постпроцессора состоит из следующих элементов:

- описание регистров, классов регистров, соглашений о связях;
- описание ресурсов (функциональных устройств процессора и др.);
- описания свойств команд и модификаторов(о модификаторах см. разд. 3.5.7) а также набор предикатов, описывающих различные виды операндов;
- описание множеств взаимозаменяемых вариантов команд (см. разд. 3.5.6)
- описание динамических ресурсов (см. разд. 3.5.8)

3.5.1. Регистры

Множество регистров задается в виде файла, содержащего последовательность определений вида

```
define_reg (<идентификатор регистра>, "<имя регистра>")
```

где

<идентификатор регистра> - имя, используемое для обозначения регистра в таблице команд и других таблицах.

"*<имя регистра>*" - имя, используемое в отладочных целях.

В файле описания регистров, как и в файлах, описывающих другие характеристики целевой машины, допускаются также комментарии в стиле языка Си.

Пример 3.5. Фрагмент файла с описаниями регистров процессора 1В577 (регистры данных):

```
/* Data registers of 1B577 */
define_reg (REG_D0L, "d0l")
define_reg (REG_D0M, "d0m")
define_reg (REG_D0H, "d0h") define_reg (REG_D1L, "d1l")
```

```
define_reg (REG_D1M, "d1m") define_reg (REG_D1H, "d1h")     ...
```

Порядок описаний задает упорядоченность на множестве регистров, которая существенно используется в таблице классов регистров (см. далее п. 3.5.2)

Помимо реальных физических регистров процессора, в описание машины могут вводиться фиктивные регистры, служащие для специальных целей.

Так, оперативная память с точки зрения постпроцессора является регистром с зарезервированным идентификатором REG_MEMORY:

```
define_reg (REG_MEMORY, "memory")
```

При этом считается, что команда, производящая запись в память, не только пишет, но и читает этот регистр. Это сделано для того, чтобы команды записи в память не переупорядочивались при постпроцесстровании (поскольку не исключено, что они записывают в одно и то же место физической памяти). Таким же образом описываются и другие случаи, когда команда переписывает содержимое некоторого регистра лишь частично или в зависимости от некоторого условия.

Для правильной обработки команд с побочным эффектом (например, вызовов подпрограмм) вводится фиктивный регистр REG_OUT:

```
define_reg (REG_OUT, "out")
```

В определениях команд, имеющих побочные эффекты, указано, что они читают и пишут регистр REG_OUT. Тем самым обеспечивается, что при постпроцессировании они не будут переупорядочены.

В разделе 3.5.8 объясняется, как при помощи еще нескольких псевдорегистров реализуются некоторые специфические ограничения целевого процессора.

Доступ к различным системным регистрам может осуществляться на побитовом уровне. Например, в процессоре 1B577 32-битный системный регистр SR включает биты признаков результата. Арифметические

команды устанавливают определенные биты, а различные команды условного перехода считывают их.

Чтобы не вводить избыточные связи по данным между командами оптимизируемой программы, в описании машины целесообразно описать каждый бит системного регистра как отдельный регистр.

Биты системного регистра задаются также в виде макросов, значениями которых являются битовые маски, соответствующие порядковому номеру регистра в таблице регистров.

```
#define RMSK_EMPTY 0x00000000
#define RMSK_C     0x00000001
#define RMSK_V     0x00000002
#define RMSK_Z     0x00000004
#define RMSK_N     0x00000008
#define RMSK_I     0x00000010
#define RMSK_LR    0x00000020
#define RMSK_R     0x00000040
#define RMSK_A     0x00000080
...
```

3.5.2. Классы регистров

Как правило, в множестве физических регистров процессора можно выделить группы (классы) - адресные регистры, индексные регистры, регистры данных и т.п. Классы регистров различаются по своему назначению, типу хранимых значений и другим свойствам. Существенно, что регистры разных классов могут физически перекрываться. Так, в процессоре 1В577 каждый регистр для хранения плавающих данных d*i*.s (*i*=0,...,9) соответствует трем регистрам d*i*.l, d*i*.m, d*i*.h, в каждом из которых может находиться 32-битное целое.

Информация о классах регистров задается в виде таблицы, каждая строка которой описывает один класс и имеет следующий формат:

```
REGCLASS (<идентификатор класса>,
          <формат>,
          <массив имен>,
```

```
<число регистров>,
<начальный регистр>,
<ширина>,
<шаг>,
<ресурсы>,
<задержка>)
```

где

<идентификатор класса> - символическое имя класса.

<формат> - формат для печати имен регистров этого класса в смысле функции printf языка Си. Формат должен включать спецификацию для вывода целого (порядковый номер регистра внутри класса). Например, формат "d%d.l" задает печать регистров с номерами 0, 1, ... и т.д. как d0.l, d1.l и т.д.

<массив имен> - имя массива с именами регистров этого класса (для вывода в выходной ассемблерный файл). Массив указывается, если невозможно задать печатное представление регистра при помощи формата, который в таком случае должен иметь значение NULL. Массивы имен регистров, если они используются, должны быть определены и инициализированы в модуле программных компонентов описания машины.

<число регистров> - количество регистров в классе.

<начальный регистр> - идентификатор начального регистра (с номером 0) этого класса. Идентификатор должен быть определен в таблице регистров (см. п. 4.1.1).

<ширина> - сколько последовательных элементарных регистров (перечисленных в таблице регистров) занимает один регистр данного класса.

<шаг> - расстояние между последовательными регистрами класса в смысле таблицы регистров.

<ресурсы> - имена ресурсов (функциональных единиц), используемых командами, которые работают с регистрами данного класса (подробнее о ресурсах см. далее, раздел 3.5.4). Если ресурсов несколько, они перечисляются с разделителем "|".

<задержка> - задается ключевым словом YES (да) или NO (нет), определяющим, есть ли аппаратная задержка при записи в регистр этого класса.

Пример 3.6. Описания некоторых классов регистров процессора 1B577.

```
/*              format nametable N  first spa step resources    post */
RCLASS (DL_EXT, "d%d.l", NULL, 10, REG_D0L,1, 3,  RES_INT,         NO)
RCLASS (DM_EXT, "d%d.m", NULL, 10, REG_D0M,1, 3,  RES_INT,         NO)
RCLASS (DH_EXT, "d%d.h", NULL, 10, REG_D0H,1, 3,  RES_EMPTY,       NO)
RCLASS (DML_EXT,"d%d.ml",NULL, 10, REG_D0L,2, 3,  RES_INT,         NO)
RCLASS (DS_EXT, "d%d.s", NULL, 10, REG_D0L,3, 3,  RES_FLT,         NO)
RCLASS (DD_EXT, "d%d.d", NULL, 10, REG_D0L,3, 3,  RES_FLT,         NO)
RCLASS (R_EXT,  "r%d",   NULL,  8, REG_R0, 1, 3,  RES_EMPTY,      YES)
RCLASS (M_EXT,  "m%d",   NULL,  8, REG_M0, 1, 3,  RES_EMPTY,      YES)
RCLASS (N_EXT,  "n%d",   NULL,  8, REG_N0, 1, 3,  RES_EMPTY,      YES)
```

Здесь класс DL_EXT содержит 10 регистров, начиная с регистра REG_D0L с шагом 3 (REG_D0L, REG_D1L, ... REG_D9L, см. пример 3.5).

Значения, хранящиеся в регистрах этого класса, занимают по 1 регистру. Команды, работающие с регистрами этого класса, используют ресурс RES_INT (характеризующий команды целочисленной арифметики). Задержек при записи в регистры данного класса нет. На печать выводятся как d0.l, d1.l и т.д.

Класс DS_EXT содержит 10 регистров (начиная с REG_D0L, с шагом 3) для хранения плавающих значений. Значение занимает 3 последовательных регистра (REG_D*i*L, REG_D*i*M, REG_D*i*H, см. пример 3.5). Команды, работающие с регистрами этого класса, используют ресурс RES_FLT (характеризующий команды плавающей арифметики). Задержек

при записи в регистры данного класса нет. На печать выводятся как d0.s, d1.s и т.д.

3.5.3. Соглашения о связях

Для обработки вызовов подпрограмм необходимы сведения об использовании регистров - какие из них служат для передачи параметров, какие сохраняются/не сохраняются при обращении к подпрограммам. Эта информация задается двумя способами.

- в виде вспомогательных директив, выводимых компилятором в ассемблерный код, предназначенный для постпроцессирования;

Пример 3.7. Директива, содержащая информацию о неявных операндах команды перехода с возвратом.

```
jsr    Farr_in
POST extra_operands input d0.l input d1.l  \
   input r6 input sp input sr input memory output memory
```

Здесь "jsr Farr_in" - команда вызова подпрограммы. Директива "POST extra_operands" предназначена для постпроцессора; в ней перечислены дополнительные входные и выходные операнды предыдущей ассемблерной команды, не заданные явно.

- в виде макросов описания целевой машины в постпроцессоре.

Эта информация используется по умолчанию, если соответствующая директива отсутствует. Например, если вызов подпрограммы встретился в ассемблерной вставке (заданной оператором asm) исходной Си-программы.

Пример 3.8. Список входных регистров для команд вызова подпрограмм по умолчанию. В список входят регистры, на которых передаются входные параметры, память, системный

регистр, указатель программного стека, указатель аппаратного стека.

```
#define DEFAULT_CALL_PARAMETER_REGS \
REG_D0L, REG_D0M, REG_D0H, REG_D1L, REG_D1M, REG_D1H, REG_MEMORY,\
REG_SP, REG_SR, REG_R6, REG_M6
```

Аналогичным образом задается информация о выходных регистрах, а также о регистрах, не сохраняемых при вызовах подпрограмм.

На основе заданных таким образом соглашений о связях вычисляются стандартные множества регистров: call_used_reg_set, call_parameter_reg_set, call_result_reg_set. Эти множества могут использоваться в описаниях команд процессора.

Необходимо отметить, что первый способ (директивы, генерируемые компилятором) обеспечивает более точную информацию. Например, если из декларации функции известно, что она не имеет входных параметров, то компилятор не включит в список читаемых те регистры, на которых передаются параметры. Это значит, что при планировании не будут учитываться избыточные зависимости по этим регистрам.

3.5.4. Ресурсы

Под ресурсами понимаются функциональные устройства и другие аппаратные средства процессора, которые участвуют в выполнении команд. В оптимизирующем постпроцессоре механизм ресурсов используется для задания правил комбинируемости элементарных команд. Ресурсы также применяются для обозначения определенных свойств элементарных команд, участвующих в описании аппаратных ограничений на комбинирование. Задача формирования минимального набора абстрактных ресурсов, достаточного для описания ограничений на комбинируемость операций, рассматривается, например, в работе [68].

Поясним применение механизма ресурсов на примере процессора 1В577. Для большей наглядности будет использована упрощенная модель процессора.

Пример 3.9. Упрощенные правила комбинирования для процессора 1В577.

Одна VLIW-строка может содержать:

- одно плавающее умножение
- одно плавающее сложение или вычитание
- две пересылки данных

или

- одну любую арифметическую операцию
- две пересылки данных

или

- одну управляющую операцию

При этом различаются арифметические команды одинарной и двойной точности, и в одной VLIW-строке нельзя закодировать умножение со сложением, если они имеют разные режимы точности.

Правила комбинируемости для такого процессора можно задать при помощи следующего множества ресурсов.

Пример 3.10. Множество ресурсов для описания правил комбинирования из примера 4.9.

RES_FMPY - устройство умножения
RES_FADD - устройство сложения RES_MOVE - устройство пересылки данных
RES_SNGL - характеризует операцию одинарной точности RES_DBL - характеризует операцию двойной точности

Пример 3.11. Ресурсы, потребляемые командами (для примера 4.9).

Умножение одинарной точности:	RES_FMPY, RES_SNGL
Сложение одинарной точности:	RES_FADD, RES_SNGL
Вычитание одинарной точности:	RES_FADD, RES_SNGL
Умножение двойной точности:	RES_FMPY, RES_DBL
Сложение двойной точности:	RES_FADD, RES_DBL
Вычитание двойной точности:	RES_FADD, RES_DBL
Прочие арифметические команды:	RES_FADD, RES_FMPY
Пересылки данных:	RES_MOVE
Управляющие команды:	RES_FADD, RES_FMPY, RES_MOVE, RES_MOVE

Сформулируем теперь правила комбинируемости для примера 4.9 в терминах запрещенных мультимножеств ресурсов (мультимножество отличается от множества тем, что элементы могут входить в него с некоторой кратностью).

Пример 3.12. Запрещенные мультимножества ресурсов для описания правил комбинируемости из примера 4.9.

Запрещенное мультимножество	Комментарий
{2*RES_FMPY}	Нельзя использовать более одной единицы ресурса RES_FMPY
{2*RES_FADD}	Нельзя использовать более одной единицы ресурса RES_FADD
{3*RES_MOVE}	Нельзя использовать более двух единиц ресурса RES_MOVE
{RES_SNGL, RES_DBL}	Нельзя использовать одновременно ресурсы RES_SNGL и RES_DBL

Постпроцессор может объединять в одну VLIW-строку элементарные команды только при условии, что мультимножество потребляемых ими в совокупности ресурсов не включает ни одно из запрещенных мультимножеств. Нетрудно убедиться, что заданные таким образом описания ресурсов (пример 3.10), команд (пример 3.11) и запрещенных

мультимножеств (пример 3.12) обеспечивают соблюдение правил из примера 4.9.

В поисках оптимального выходного кода постпроцессор составляет и проверяет на соблюдение ограничений по ресурсам большое число потенциальных комбинаций из исходных элементарных команд. Следовательно, быстродействие постпроцессора существенно зависит от эффективной реализации таких проверок. В текущей версии постпроцессора используется представление мультимножеств в виде структурного типа, где кратность вхождения каждого ресурса хранится в битовом поле (рис. 3.9), размер которого должен быть на единицу больше, чем необходимо для хранения максимальной возможной кратности ресурса. Запрещенные мультимножества представлены константами этого типа, формируемыми по определенным правилам. Такое представление позволяет реализовать исключительно эффективное вычисление мультимножества ресурсов для комбинации элементарных команд (при помощи операции сложения), а также проверку его допустимости (при помощи операции "побитовое И").

2 бита	2 бита	3 бита	3 бита	3 бита
FMPY	FADD	MOVE	SNGL	DBL

Рис. 3.9. Представление мультимножеств ресурсов в виде структуры из битовых полей

Структурный тип, задающий представление мультимножеств ресурсов для целевого процессора, должен быть описан на языке Си. В принципе, возможна автоматическая генерация требуемого структурного типа по текстовому описанию запрещенных комбинаций как в примере 4.12.

3.5.5. Свойства команд

Информация о свойствах команд организована в виде двух таблиц: таблицы команд и таблицы групп команд. Основная часть информации сосредоточена во второй таблице, где описаны свойства, присущие однотипным (с точки зрения постпроцессора) командам. Выделение групп команд предусмотрено для того чтобы не дублировать многократно описания одинаковых свойств, присущих многим командам.

Таблица команд состоит из записей вида

```
define_ins (<идентификатор команды>, <идентификатор группы>, "<код операции>",
            .sys_iregs = <читаемые биты признака результата>,
            .sys_oregs = <записываемые биты признака результата>)
```

Пример 3.13. Описание инструкции abs.

```
define_ins (IORN_ABS, GRID_ABS, "abs",
            .sys_iregs = RMSK_EMPTY,
            .sys_oregs = RMSK_V | RMSK_Z | RMSK_N)
```

Здесь:

IORN_ABS - уникальный идентификатор команды;

GRID_ABS - идентификатор группы команд (должен присутствовать в таблице групп команд), такой же идентификатор группы может быть и у других команд;

RMSK_EMPTY - маска (пустая) читаемых битов признаков результата;

RMSK_V | RMSK_Z | RMSK_N - маска из трех записываемых битов признака результата.

Информация о читаемых/записываемых битах признака результата вынесена в таблицу команд для того чтобы сократить число описателей в таблице групп. Набор входных/выходных признаков результата

оказывается различным для многих команд, идентичных по остальным свойствам.

Таблица групп команд состоит из записей вида

```
GRID (<идентификатор группы>,
      .grname = "<текстовое имя группы>",
      .priority = <приоритет>,
      <описатель операндов>,
      <признаки>,
      <цена>,
      <описатель псевдокоманды>,
      <регистры, разрушаемые данной командой>,
      <дополнительные входные/выходные регистры>,
      <информация об аппаратных задержках>,
      <информация о потребляемых ресурсах>,
      <информация о динамических ресурсах>,
      <ограничения на следующую команду>,
      <варианты>)
```

Далее объясняется назначение и синтаксис полей в описателе группы.

<идентификатор группы> - уникальный идентификатор группы, по которому на нее можно сослаться из таблицы команд.

.grname = "<текстовое имя группы>" - задает текстовое имя группы, используемое в отладочных дампах.

.priority = <приоритет> - целое число, задает порядковый номер, используемый при выводе результирующих VLIW-строк. Например, если VLIW-строка содержит арифметическую команду и команду пересылки данных, то при выводе ассемблерного кода сначала следует арифметическая команда, а затем пересылка данных.

<описатель операндов> - выражение, задающее количество операндов и их свойства. В описателе используются предикаты, характеризующие операнды, а также ключевые слова IN (входной операнд), OUT (выходной операнд), INOUT (операнд, являющийся и входным, и выходным). Примеры:

```
operand (op_d07, INOUT),
```

один операнд, удовлетворяющий предикату op_d07 (регистр данных с номером от 0 до 7), являющийся одновременно входным и выходным.

```
two_operands (op_d07_float, IN, op_d07_l, OUT),
```

два операнда - входной на плавающем регистре данных d0-d7 и выходной на одном из регистров от d0.l до d7.l.

```
two_operands_with_check (op_d07_l, IN, op_d07_l, OUT, ops_d03_d47),
```

два операнда (входной и выходной) с дополнительным предикатом ops_d03_d47, задающим условие на всю совокупность операндов.

Все предикаты, используемые в описателях групп, должны быть реализованы на языке Си.

<признаки> - перечисление специальных признаков группы. Например, слово SIDE_EFFECTS_FLAG указывает, что команды этой группы имеют побочные эффекты. Слово no_flags указывает, что данная группа не имеет специальных признаков.

<цена> - выражение, задающее цену данной команды. Цена должна отражать критерий (или критерии) оптимизации. Она может соответствовать времени выполнения, размеру команды, или комбинации этих характеристик.

Цена задается выражением вида cost(c1,c2,c3). Стоимость комбинации из нескольких элементарных команд вычисляется как максимум первых компонентов (c1) плюс сумма вторых компонентов (c2). Третий компонент зарезервирован для настройки на другие архитектуры. Первый компонент рассматривается как основное время выполнения команды, второй - как штраф, связанный с особенностями формата команды. Например, основное время выполнения пересылки "память-регистр" равно 2 тактам; при использовании абсолютной адресации добавляется штраф, поскольку адрес в этом случае размещается в дополнительном командном слове, на считывание которого требуется добавочное время.

В принципе тип данных для представления цены и функцию вычисления стоимости для комбинации элементарных команд также следует считать параметрами целевой платформы. Функция для вычисления цены команды должна зависеть только от самой команды и не должна зависеть от того, какие команды выполняются до или после нее. Она также всегда должна давать неотрицательный результат.

<описатель псевдокоманды>. Псевдокоманды вводятся в описание целевой машины для реализации дополнительных оптимизаций, подробнее см. разд. 3.5.7.

Если описатель соответствует реальной команде (или группе команд) процессора, то в качестве значения должно быть задано ключевое слово normal_command. Если описатель соответствует псевдокоманде, то следует задать имя функции, реализующей эту псевдокоманду.

Функции, реализующие псевдокоманды, задаются на языке Си в отдельном программном модуле, который включается в состав постпроцессора.

<регистры, разрушаемые данной командой> - множество регистров, значения которых разрушаются в результате выполнения данной команды. Например, описатель .destroyed_regs = call_used_reg_set указывает, что команды данной группы разрушают множество регистров call_used_reg_set (регистры, не сохраняемые при обращениях к подпрограммам). Ключевое слово no_destroyed_regs указывает, что разрушаемых регистров нет.

<дополнительные входные/выходные регистры> - множества дополнительных входных и выходных операндов (регистров) команды, не задаваемых явно в ассемблерной команде. Например, некоторые команды могут читать/писать определенные системные регистры (помимо регистров признака результата).

<информация об аппаратных задержках> - пред- и постусловия, связанные с аппаратными задержками. Предусловия задаются перечислением регистров, по которым данная группа команд чувствительна к задержкам, если они были созданы предшествующими командами. Постусловия задаются перечислением регистров, по которым данная команда создает аппаратные задержки. Примеры применения этого механизма для реализации некоторых ограничений процессора 1В577 представлены в разд. 3.5.9.

<информация о потребляемых ресурсах> - перечисление ресурсов, потребляемых командами данной группы. Пример:

```
.resources = RES_FADD | RES_FLT | RES_DBL
```

Здесь RES_FADD соответствует устройству сложения, RES_FLT характеризует операции плавающей арифметики, а RES_DBL - операции двойной точности.

<информация о динамических ресурсах> - описатели динамических ресурсов, создаваемых и потребляемых командами данной группы. Подробнее о динамических ресурсах см. разд. 3.5.8.

<ограничения на следующую команду>. Механизм ограничений на следующую команду (в сочетании с псевдокомандами) используется для комбинирования двух операций записи в память в одном командном слове. Стандартным образом это сделать невозможно из-за принципиального свойства алгоритма комбинирования - запрет на объединение команд, которые пишут результат в один и тот же объект (в данном случае - память).

<варианты> - задаются идентификатором группы вариантов, на которые можно заменить данную команду, а также именем функции для преобразования операндов при переходе от одного варианта к другому. Подробнее о вариантах см. разд. 3.5.6.

Отметим, что во многих группах команд часть описателей являются вырожденными (пустыми). Для представления вырожденных описателей используются специальные ключевые слова, как показано в следующем примере.

Пример 3.14. Описатель группы команд целочисленной арифметики.

```
GRID (GRID_ABS,
        .grname = "abs (single INOUT-operand instructions)",
        .priority = 0,           /* при выводе VLIW-строки ставить
                                  * данную элементарную команду
                                  * на первое место
                                  */
        operand (op_d07, INOUT), /* один операнд, входной и выходной
*/
        no_flags,                /* нет специальных признаков */
        cost (2,0,0),            /* цена (2 такта) */
        normal_command           /* не псевдокоманда */,
        no_destroyed_regs,       /* разрушаемых регистров нет */
        no_extra_regs,           /* дополнительных (неявных)
                                  * операндов нет
                                  */
        no_pre_post,             /* пред- и постусловий по
                                  * аппаратным задержкам нет
                                  */
        .resources = RES_FADD | RES_FMPY | RES_INT,
                                 /* использует устройства сложения и
                                  * умножения, целочисленная операция
                                  */
        no_takes_gives,           /* динамические ресурсы не создаются
                                   * и не потребляются
                                   */
        no_nexts,                 /* ограничений на следующую
                                   * команду нет.
                                   */
        no_variants               /* вариантов нет. */
)
```

Важно отметить, что одна и та же команда процессора может иметь разное число операндов или, в зависимости от типов операндов, обладать разными свойствами с точки зрения комбинируемости с другими командами. В таких случаях используются множественные описатели, или описатели с продолжением. Описатель с продолжением - это последовательность описателей, в которой у второго и последующих описателей задан признак continue.

Ниже приведен пример описателя с продолжением для команд asl, asr (арифметический сдвиг влево и вправо) и lsl, lsr (логический сдвиг влево и вправо). Эти команды могут иметь следующие форматы:

```
<операция> <регистр>
      - сдвиг содержимого регистра на 1 бит;

  <операция> <регистр1>,<регистр2>
      - сдвиг содержимого регистра2 на число бит, заданное регистром1;

  <операция> <непосредственный операнд><регистр>
      - сдвиг содержимого регистра2 на число бит, заданное
        непосредственным операндом;
```

Здесь <операция> - это asl, asr, lsl или lsr. Первые два формата допускают комбинирование операции сдвига с параллельными пересылками данных. При использовании непосредственного операнда комбинирование с пересылками данных не допускается.

```
GRID (
GRID_ASL,
        .grname = "asl D",
        .priority = 0,
        operand (op_d07_1, INOUT),  /* Один операнд - регистр. */
        no_flags,
        cost (2,0,0),
        normal_command, /* not pseudo */
        no_destroyed_regs, no_extra_regs,
        no_pre_post,
        /* Набор ресурсов, допускающий комбинирование
```

```
            с пересылками данных: */
        .resources = RES_FADD | RES_FMPY | RES_INT,
        no_takes_gives,
        no_nexts,
        no_variants),

/* Продолжение - второй формат операндов */

  GRID (
GRID_ASL_H_D,
        .grname = "asl S,D",
        .priority = 0,
                        /* Два операнда - регистра. */
        two_operands (op_d07_h, IN, op_d07_l, INOUT),
        continue,       /* признак продолжения */
        cost (2,0,0),
        normal_command, /* not pseudo */
        no_destroyed_regs, no_extra_regs,
        no_pre_post,
        /* Набор ресурсов, допускающий комбинирование
           с пересылками данных: */
        .resources = RES_FADD | RES_FMPY | RES_INT,
        no_takes_gives,
        no_nexts,
        no_variants),

/* Продолжение - третий формат операндов */

  GRID (
GRID_ASL_I_D,
        .grname = "asl #shift,D",
        .priority = 0,
                        /* Первый операнд - константа,
                           второй операнд - регистр */
        two_operands (op_immediate, IN, op_d07_l, INOUT),
        continue,       /* признак продолжения */
        cost (2,0,0),   /* цена */
        normal_command, /* not pseudo */
        no_destroyed_regs, no_extra_regs,
        no_pre_post,
        /* Набор ресурсов, исключающий комбинирование: */
        .resources = RES_FADD | RES_FMPY | RES_INT | RES_RR | RES_RM,
```

```
no_takes_gives,        no_nexts,        no_variants),
```

3.5.6. Варианты

Некоторые операции могут быть реализованы в процессоре более чем одним способом. Примеры вариантов приводились в разд. 3.4.9.

Если предоставить постпроцессору информацию о взаимозаменяемости операций, то он получит дополнительные возможности построения комбинаций. На уровне описания целевой машины множества взаимозаменяемых команд задаются в виде файла, состоящего из записей вида:

```
define_vargroup (<идентификатор множества вариантов>, <число>,
                 <идентификатор описателя в таблице групп команд>,
                 ...)
```

где

<идентификатор множества вариантов> - уникальный идентификатор множества вариантов (взаимозаменяемых команд целевого процессора).

<число> - количество элементов в множестве.

<идентификатор описателя в таблице групп команд>, ... перечисление идентификаторов соответствующих описателей в таблице групп команд (см. разд. 3.5.5).

Пример 3.15. Описатель множества взаимозаменяемых команд.

```
define_vargroup (vargroup_move_tfr_1, 2, GRID_TFR, GRID_MOVE_D07L)
```

Здесь vargroup_move_tfr_1 - идентификатор множества вариантов, которое состоит из 2 эквивалентных команд. В таблице групп (разд. 3.5.5) описатели этих команд имеют идентификаторы GRID_TFR и GRID_MOVE_D07L.

Если при замене команды на ее вариант необходимо какое-либо преобразование операндов, то оно должно быть задано в виде Си-функции, имя которой указывается в описателе команды в таблице групп команд.

Отметим две важных особенности механизма вариантов.

1. Варианты поддерживаются на уровне отдельных команд, но не на уровне последовательностей или комбинаций команд.
2. Набор входных регистров для всех эквивалентных вариантов одинаков. Число записываемых регистров может различаться. Как указано в разд. 3.4.9, постпроцессор отбрасывает варианты, не записывающие регистры, значения которых в данном линейном участке используются.

3.5.7. Псевдокоманды (модификаторы)

Псевдокоманды вводятся в описание машины как варианты некоторых реальных машинных команд для реализации дополнительных оптимизаций. Смысл псевдокоманды заключается в том, чтобы выполнить определенное преобразование командной строки, в которую она попала в результате оптимизации линейного участка.

Поясним использование псевдокоманд на следующем примере.

Пример 3.16. В программе для 1В577 пару команд

```
move x:(r1),d5.s      ;;считать значение из памяти по адресу хранящемуся в r1, и
                      записать в регистр d5.s
move (r1)+            ;;увеличить r1 на 1.
```

можно заменить на одну команду

```
move x:(r1)+,d5.s
```

которая объединяет эффект двух первых команд, но выполняется по времени столько же, сколько первая из них.

Для того чтобы реализовать подобную оптимизацию, введем псевдокоманду с условным именем .replace

```
.replace <инкрементное адресное выражение>
```

и опишем ее как вариант команды move с операндом того же вида:

```
move <инкрементное адресное выражение>
```

Действие псевдокоманды .replace заключается в том, чтобы заменить простую косвенную адресацию в команде пересылки в той же командной строке на инкрементную - "(ri)" на "(ri)+", а использоваться она может только в комбинации с командой пересылки из памяти или в память (с косвенной адресацией по тому же адресному регистру).

Пример 3.17. Допустимая комбинация, включающая псевдокоманду .replace.

```
{move x:(r1),d5.s; .replace (r1)+}
```

При выводе ассемблерного файла такая комбинация будет преобразована в

```
move x:(r1)+,d5.s
```

Команда move (ri)+ и ее вариант - псевдокоманда .replace (ri)+ отличаются следующими свойствами.

1. Команда move может использоваться только самостоятельно, она не может комбинироваться с пересылкой данных. Время ее выполнения (т.е. цена) - два такта.
2. Псевдокоманда .replace, наоборот, может использоваться только в комбинации с пересылкой данных, использующей тот же самый адресный регистр. Это условие реализуется при помощи механизма динамических ресурсов, который описан в следующем разделе. Время выполнения псевдокоманды .replace (цена) нулевое.

Поскольку цена псевдокоманды .replace ниже, чем цена команды move, то постпроцессор будет стремится использовать ее, а не move, если только на линейном участке найдется пересылка данных с нужными свойствами, к которой можно присоединить .replace.

Как и обычные команды, псевдокоманды задаются при помощи описателей в таблице команд и в таблице групп команд (разд. 3.5.5)). Различие заключается в том, что описатель псевдокоманды в таблице групп содержит имя функции, которая реализует соответствующее преобразование командной строки. Функция задается в поле <*описатель псевдокоманды*>. Например, в описателе псевдокоманды .replace поле <*описатель псевдокоманды*> имеет вид:

```
.pseudo_fun = pseudo_replace,
```

где pseudo_replace - функция, реализующая данную псевдокоманду (замену способа адресации). Функции, реализующие все псевдокоманды, составляют часть описания целевой машины и должны быть заданы на языке Си.

В последующих разделах, если не оговорено противное, под словом "команды" подразумеваются как реальные машинные команды, так и псевдокоманды.

3.5.8. Динамические ресурсы

Механизм динамических ресурсов используется для описания тех ограничений на комбинирование операций, которые невозможно задать средствами обычных (статических) ресурсов, обсуждавшихся в разделе 3.5.4. На основе статических ресурсов можно определить лишь такие комбинации, которые обладают следующим свойством: любое подмножество команд из допустимой комбинации является допустимой комбинацией. На практике это не всегда так. В частности, псевдокоманда может входить в состав допустимой комбинации (см. пример 3.17), но сама по себе не составляет допустимой комбинации, поскольку псевдокоманда, по определению, является модификатором, применяемым к другой команде из своей строки.

В качестве динамического ресурса выступает значение (например, текстовая строка), однозначно соответствующее некоторому свойству или элементу машинной команды (например, адресному выражению в одном из операндов). Команда может предоставлять и/или потреблять динамические ресурсы различных видов. Если в описателе команды (см. разд. 3.5.5) указано, что она потребляет некоторый динамический ресурс, то она может употребляться только в комбинации с командой, предоставляющей в точности этот ресурс. У команды-потребителя всегда есть вариант, не требующий динамических ресурсов, который заведомо можно использовать, если в линейном участке нет команды, предоставляющей требуемый динамический ресурс. Команда, предоставляющая динамический ресурс, может употребляться независимо от того, присутствует ли в той же комбинации команда-потребитель этого ресурса.

Ниже приведен пример применения механизма динамических ресурсов.

Пример 3.18. Процессор 1B577 допускает параллельное выполнение пересылок из памяти в регистр по шинам X и Y:

```
move X:<адрес>,<регистр1> (1)
move Y:<адрес>,<регистр2> (2)
```

при условии что <адрес> в обеих командах один и тот же. Пара таких пересылок может быть записана в виде VLIW-строки

```
move X:<адрес>,<регистр1> Y:,<регистр2>
```

которая выполняется быстрее чем исходная пара команд.

Пример 3.19. Пара команд

```
move X:Fd,d1.s      ;чтение памяти по шине X, по адресу, заданному символом Fd
move Y:Fd,d4.s      ;чтение памяти по шине Y, по адресу, заданному символом Fd
```

эквивалентна VLIW-строке

```
move X:Fd,d1.s Y:,d4.s
```

Будем считать, что команда вида (1) в примере 4.18, которая читает из памяти по шине X, генерирует уникальный динамический ресурс, соответствующий заданному адресу. Для команды вида 2, которая читает из памяти по шине Y, определим в качестве варианта псевдокоманду .move, выполняющую то же действие, что и исходная команда, но потребляющую динамический ресурс, соответствующий заданному адресу.

В описателе для команд вида (1) "move X:<адрес>,<регистр>" укажем, что они предоставляют динамический ресурс с идентификатором "(<адрес>)". Элемент <информация о динамических ресурсах> в описателе команды (см. разд. 3.5.5) может выглядеть следующим образом:

```
dres (no_takes, give1(read_address, 0))
```

Ключевое слово dres обозначает начало описателя динамических ресурсов, который следует далее в скобках; слово no_takes указывает, что команда не потребляет динамических ресурсов. Ключевое слово give1 указывает, что команда предоставляет один вид динамических ресурсов, описатель которого задан далее в скобках в виде последовательности из двух элементов: первый элемент (read_address) - имя функции, вычисляющей значение ресурса по операнду команды, второй элемент - номер операнда, по которому вычисляется значение ресурса (операнды нумеруются с нуля). Функции для вычисления динамических ресурсов являются частью описания целевой машины; они должны быть реализованы на языке Си.

В описателе для команд вида (2) "move Y:<адрес>,<регистр>" укажем (при помощи набора статических ресурсов), что они не могут комбинироваться с командами move вида (1), но имеют вариант - псевдокоманду ".move Y:<адрес>,<регистр>", которая потребляет динамический ресурс "(<адрес>)" и может комбинироваться с командами move вида (1).

Таким образом, если на линейном участке есть команда move вида (1) с подходящим адресом, то постпроцессор имеет право использовать вариант .move. Определим цену псевдокоманды .move равной нулю, чтобы ей было отдано предпочтение, если имеются необходимые условия для ее использования.

Если же на данном линейном участке не окажется подходящей команды вида (1), предоставляющей нужный динамический ресурс, то вариант .move не будет употреблен (а будет использована исходная, более дорогостоящая, команда "move Y:Fd,d4.s").

3.5.9. Реализация аппаратных ограничений при помощи псевдорегистров

Использование определенных команд может быть запрещено после некоторых других команд. Например, в процессоре 1B577 непосредственно перед командой аппаратного цикла (DO или DOR) нельзя использовать команды загрузки адреса (LEA) и записи управляющих регистров (MOVEC) с определенными сочетаниями операндов.

Ограничения такого рода реализуются путем введения в описание машины фиктивных регистров и аппаратных задержек по ним. Рассмотрим, например, как реализуется указанное выше ограничение.

1. В список регистров вводится фиктивный регистр (с идентификатором REG_DO).
2. В описателе для группы команд (DO, DOR) указывается, что они чувствительны к задержкам по этому регистру: pre(REG_DO)
3. В описателях команд LEA, MOVEC с соответствующими сочетаниями операндов задается задержка по регистру REG_DO.

Тем самым гарантируется, что постпроцессор при оптимизации линейного участка не сгенерирует недопустимую последовательность команд.

4. Оценки эффективности

Эта глава посвящена сравнению алгоритма планирования с другими алгоритмами а также измерениям эффективности кода, сгенерированного компилятором с оптимизирующим постпроцессором для целевой платформы 1В577.

4.1. Сравнение с другими методами планирования

Сравним рассмотренный в главе 3 метод планирования с описанными в литературе алгоритмами, применяемыми в компиляторах для VLIW-платформ, в частности, с алгоритмом списочного планирования [32], методами, основанными на применении ЦЛП [30],[37],[54], и методом, основанным на применении дизъюнктивных графов [16].

4.1.1. Списочное планирование.

Описание алгоритма списочного планирования представлено в [32]; там же, а также в [41], можно найти описания нескольких его разновидностей. Все они обладают следующими двумя свойствами:

1) сохранение не только зависимостей (отношение RAW), но и антизависимостей (см. разд. 2.5.3) по данным.

2) "жадность" - алгоритм строит единственное решение из множества возможных, принимая решения на основе эвристически расставленных приоритетов.[11]

Использование списочного планирования в подавляющем большинстве компиляторов основано на общепринятом мнении о том, что жадные алгоритмы как правило находят решение, близкое к оптимальному. Однако исследования, проведенные авторами работы [32], показывают, что это не совсем так. Согласно результатам этих исследований, списочное планирование может давать план выполнения, существенно хуже оптимального, если длина линейного участка значительна, а степень аппаратного параллелизма невысока. На недостаточную оптимальность кода, генерируемого алгоритмом списочного планирования (в частности, для ЦПОС), указывают также авторы работ [30], [37], [54].

Рассмотрим пример, показывающий, почему списочное планирование может давать неоптимальный результат.

Пример 4.1. Неоптимальность, связанная с сохранением антизависимостей. Пусть имеется линейный участок B

c_1 | A:=MEM(R)

c_2 | B:=B+A

c_3 | A:=D+E

c_4 | F:=A+G

с множествами входных и выходных регистров $I_B = \{R, B, D, E, G\}$, $O_B = \{B, F\}$. Пусть в целевом процессоре поддерживается чтение из памяти

[11] Применяется также многократное выполнение списочного планирования (со случайным выбором для команд с равными приоритетами) и выбор наилучшего из вычисленных планов выполнения.

параллельно со сложением, а параллельное выполнение двух сложений невозможно. В этом случае алгоритм полного перебора планов даст решение

c_3 | A:=D+E

c_4,c_1 | F:=A+G; A:=MEM(R)

c_2 | B:=B+A

а алгоритм списочного планирования вернет в качестве результата исходную последовательность команд, поскольку он не имеет права переупорядочить команды c_1, c_3, связанные отношением WAW, а также команды c_2, c_3, связанные отношением WAR.[12]

Сохранение антизависимостей - принципиальное свойство списочного планирования. Без соблюдения этого требования возможны ситуации, когда множество готовых к исполнению команд окажется пустым, при том что множество невыведенных команд еще не пусто.

Следующий пример показывает, что теоретически списочное планирование (или любой алгоритм, который "не заглядывает вперед" более чем на некоторое ограниченное число командных слов) может давать вдвое худший план выполнения по сравнению с полным перебором планов.

Пример 4.2. Пусть имеется следующий линейный участок *B*,.

```
c1  | R0 := R2*R2
c2  | R0 := R0*R0
c3  | R0 := R0*R0
c4  | R0 := R0*R0
c5  | R0 := R0*R0
c6  | R0 := R0*R0
c7  | R0 := R0*R0
```

[12]Обычно антизависимости, подобные этой, устраняются методом переименования регистров. Однако при ограниченном числе регистров полное устранение антизависимостей не всегда возможно.

c_8 | R0 := R0*R0

c_9 | R0 := R0*R0

c_{10} | R1 := R0

c_{11} | R0 := R1*R1

c_{12} | R1 := R2

c_{13} | R1 := R1+R1

c_{14} | R1 := R1+R1

c_{15} | R1 := R1+R1

c_{16} | R1 := R1+R1

c_{17} | R1 := R1+R1

c_{18} | R1 := R1+R1

c_{19} | R1 := R1+R1

c_{20} | R1 := R1+R1

c_{21} | R2 := R1*R1

где I_B={R2}, O_B={R0, R2}, и допускается параллельное исполнение одной команды умножения и одной команды сложения или пересылки

Алгоритм перебора найдет для этого линейного участка следующее решение:

c_1, c_{12}	R0 := R2*R2	R1 := R2
c_2, c_{13}	R0 := R0*R0	R1 := R1+R1
c_3, c_{14}	R0 := R0*R0	R1 := R1+R1
c_4, c_{15}	R0 := R0*R0	R1 := R1+R1
c_5, c_{16}	R0 := R0*R0	R1 := R1+R1
c_6, c_{17}	R0 := R0*R0	R1 := R1+R1
c_7, c_{18}	R0 := R0*R0	R1 := R1+R1
c_8, c_{19}	R0 := R0*R0	R1 := R1+R1
c_9, c_{20}	R0 := R0*R0	R1 := R1+R1
c_{10}, c_{21}	R1 := R0	R2 := R1*R1
c_{11}	R0 := R1*R1	

При списочном планировании, независимо от аппаратного параллелизма, в качестве результата будет выдан результат, идентичный исходному линейному участку. Из-за требования сохранять отношения WAW, WAR, на каждом шаге готовой к исполнению оказывается только одна команда.

4.1.2. Методы планирования на основе ЦЛП.

Подход, основанный на применении ЦЛП, равно как и рассматриваемый здесь алгоритм перебора, позволяет находить наилучший план выполнения, обеспечивая поиск решения на всем пространстве $\mathbf{P}_B$. Реализация заключается в том, чтобы по входному линейному участку и описанию целевой архитектуры автоматически сгенерировать соответствующую систему уравнений и неравенств для задачи ЦЛП, применить ЦЛП-решатель, и полученное решение преобразовать в последовательность командных слов. Сложность реализации представляется довольно высокой, поскольку аккуратно отобразить пространство допустимых планов выполнения для нерегулярной VLIW-архитектуры на систему ЦЛП-ограничений весьма непросто.

В [30] и [37] представлены основанные на ЦЛП методы оптимизации, позволяющие совместно решать задачи планирования команд, подбора команд и распределения регистров. Возможность совместного решения этих трех задач оптимизации является важным преимуществом данного подхода.

Однако это довольно ресурсоемкий (по времени счета и памяти) метод, и, по имеющейся информации, в настоящее время существуют только его экспериментальные реализации. Нет сведений о его проверке на представительных наборах тестов с указанием затрат времени и памяти. Как показано в разд. 3.4.8, алгоритм перебора планов выполнения методом динамического программирования поддается эффективным оптимизациям, позволяющим довести его ресурсоемкость до уровня практической применимости в промышленных компиляторах.

Алгоритм перебора планов выполнения опробован в компиляторе для целевой платформы ЦПОС 1В577 и обладает приемлемыми временными характеристиками выполнения. Например, время компиляции набора из 70

тестов общим объемом более 1800 строк на языке Си для составляет 3 мин. на ЭВМ Pentium II 350Мгц, с объемом памяти 128Мб. При этом некоторые из тестов содержат линейные участки размером порядка 500 элементарных процессорных команд и более. Указанное время включает все проходы компиляции от препроцессирования до генерации ассемблерного кода. Эффективность генерируемого кода (см. разд. 4.2) составляет от 1.2 до 2.3 по отношению к коду, написанному вручную на ассемблере квалифицированным программистом.

4.1.3. Метод планирования с использованием дизъюнктивных графов.

Этот переборный алгоритм локального планирования предложен в работе [16] и используется в системе построения компиляторов [9]. (см. его краткое изложение в конце разд. 2.6.1). Преимуществом алгоритма перебора частичных планов по сравнению с алгоритмом [16] является более широкая область применимости. Алгоритм [16] существенно опирается на предположение о том, что информацию о несовместимости команд (невозможности их параллельного исполнения) можно представить в виде бинарного отношения. Он неприменим, если возможны ситуации, когда, например, команды *a, b, c* несовместимы, хотя любые две из них совместимы. В частности, он неприменим для процессоров, имеющих несколько однотипных функциональных устройств, и для некоторых процессоров с нерегулярными схемами кодирования.

4.2. Измерение эффективности кода для процессора 1В577

4.2.1. Цели и методика измерений

Измерения проводились с двумя основными целями:

- оценить, насколько код, генерируемый компилятором и постпроцессором, уступает по эффективности ассемблерным реализациям тех же алгоритмов, выполненным квалифицированным программистом;
- проанализировать свойства сгенерированного кода в сравнении с "идеальным" ассемблерным кодом, с тем чтобы выявить резервы оптимизации.

Для тестирования были использованы вычислительные задачи, типичные для ЦПОС-приложений, такие как сложение и умножение векторов; сложение, умножение, транспонирование матриц; преобразование Фурье, алгоритмы фильтрации, вычисление многочлена по схеме Горнера и др. В частности, использовались алгоритмы из опубликованных документов компании Motorola. Тестировались варианты программ, работающих с вещественными данными обычной точности и с комплексными данными.

Измерения производились средствами симулятора для процессора 1В577. Для каждой из тестовых задач были подготовлены:

- набор тестовых исходных данных;
- реализация тестируемого алгоритма в виде функции на языке Си;
- оптимальная или близкая к оптимальной ассемблерная реализация, написанная вручную. (Для большинства тестов оптимальность использованных ассемблерных реализаций может быть строго доказана.)

Помимо этого были подготовлены вспомогательные программные средства, обеспечивающие автоматизированный запуск серии тестов и вывод отчетов о временах выполнения.

Показателем эффективности скомпилированного кода для конкретного теста служит отношение

T_{comp} / T_{ass}

где T_{comp} - время выполнения (в тактах процессора) скомпилированной Си-функции, T_{ass} - время выполнения соответствующей подпрограммы, написанной вручную на ассемблере, в применении к тем же тестовым данным. Сходный подход к оценке эффективности кода применяется в работе [3].

4.2.2. Результаты измерений

Согласно результатам измерений на наборе тестов, отражающем реальные вычислительные задачи процессора 1В577, коэффициент эффективности скомпилированного кода находится в пределах от 1.2 до 2.2 (см. таблицу 5.1).

Таблица 4.1. Коэффициенты эффективности скомпилированных тестовых функций

	Коэффициент эффективности	
Тестовая задача	Вещественные данные	Комплексные данные
1. Сложение векторов	1.2	1.4
2. Умножение вектора на скаляр	1.7	1.5
3. Скалярное произведение векторов	1.7	1.5
4. Внешнее произведение векторов	1.7	1.5
5. Поэлементное произведение векторов	1.2	1.5
6. Умножение квадратных матриц	1.7	1.7
7. Транспонирование матрицы	1.4	1.5
8. Сравнение элементов массива с заданной константой	1.2	1.4
9. Вычисление многочлена по схеме Горнера	1.3	1.3
10. Вычисление 4 нецентральных моментов	1.6	-
11. Преобразование Фурье	-	2.1
12. Корреляция (FIR фильтр)	-	1.5
13. FIR фильтр со сдвигом	-	2.0
14. Степенной ряд N-го порядка	1.7	-
15. Сглаживание картинки фильтром 3x3	2.2	-

Остальные разделы этой главы посвящены анализу причин "отставания" скомпилированного кода от ассемблерной реализации и рассмотрению методов оптимизации, направленных на преодоление этих причин. Предварительные эксперименты показывают, что реализация дополнительных оптимизаций позволит во многих случаях значительно улучшить коэффициенты эффективности, указанные в табл. 5.1.

4.2.3. Конвейеризация и развертка циклов

Тщательная оптимизация циклов - ключ к получению эффективной программы. При написании ассемблерных программ вручную, опытный программист как правило использует метод конвейеризации циклов (см.

разд. 2.5.1), который в настоящее время не реализован ни в постпроцессоре, ни в базовом компиляторе.

Реализация эффективной конвейеризации циклов для процессора с ограничениями кодирования представляется довольно сложной задачей, требующей, как правило, перебора нескольких вариантов и выбора того из них, который дает максимальную степень параллельности. В то же время, в компиляторе gcc поддерживается развертка циклов, эффект которой, в принципе, сравним с эффектом конвейеризации.

Проблема заключается в том, что для эффективного использования результатов развертки необходима реализация еще двух дополнительных оптимизаций. Поясним это на примере характерного цикла обработки массивов, в котором вычисляется внешнее умножение вещественных векторов:

Пример 4.3. Программа внешнего умножения вещественных векторов.

```
for (j=0; j<M; j++) {
      sj = S[j];
      for (i=0; i<N; i++) {
        T[j][i] = P[i] * sj; /* Внутренний цикл */
      }
}
```

Компилятор, как и неискушенный программист, «запрограммирует» внутренний цикл следующим образом.

Пример 4.4. Внутренний цикл, сгенерированный компилятором без опции развертки циклов.

```
do N,_enddo
    move    x:(r1)+,d5.s       ;Чтение P[i] из памяти в регистр d5.
    fmpy.s  d4,d5,d1           ;Умножение sj на P[i], результат
                               ;заносится в регистр d1.
    move    d1.s,x:(r4)+       ;Запись результата (d1) в T[j][i].
enddo:
```

Зависимости по данным в этом цикле таковы, что параллельное исполнение команд в нем невозможно.

Если задана опция развертки циклов, то компилятор сгенерирует код, показанный далее.

Пример 4.5. Внутренний цикл, сгенерированный компилятором с опцией развертки циклов.

№	VLIW-строки		Комментарий
0	do	N/4,_enddo	
1	move	x:(r1),d5.s	; I-я итерация
2	fmpy.s	d5,d4,d1	
3	move	d1.s,x:(r4)	
4	move	x:(r1+1),d5.s	; (I+1)-я итерация
5	fmpy.s	d5,d4,d1	
6	move	d1.s,x:(r4+1)	
7	move	x:(r1+2),d5.s	; (I+2)-я итерация
8	fmpy.s	d5,d4,d1	
9	move	d1.s,x:(r4+2)	
10	move	x:(r1+3),d5.s	; (I+3)-я итерация
11	fmpy.s	d5,d4,d1 (r1)+n1	
12	move	d1.s,x:(r4+3)	
13	move	(r4)+n4	
14	_enddo		

Видно, что постпроцессору удалось сгенерировать только одну параллельную команду (номер 11): "fmpy.s d5,d4,d1 (r1)+n1", где умножение выполняется параллельно с продвижением адресного регистра r1 на 4. Вычисления же из соседних итераций скомбинировать не удалось.

Для сравнения приведем конвейеризованное тело из ассемблерной программы, написанной вручную опытным программистом.

Пример 4.6. Конвейеризованный внутренний цикл для программы внешнего умножения вещественных векторов.

```
;Разгон:
 move  x:(r1)+,d5.s   ;Чтение P[0] из памяти в регистр d5.
 fmpy.s d4,d5,d1      ;Умножение sj на P[0], результат в d1.
 move   x:(r1)+,d5.s  ;Чтение P[1] из памяти в регистр d5.
                      ;для следующего умножения
;Цикл:
    do N-2,_enddo
        fmpy.s d4,d5,d1 x:(r1)+,d5.s d1.s,y:(r4)+
        ; Эта VLIW-строка совмещает 3 элементарных команды:
        ;move    d1.s,x:(r4)+    ;Запись результата предыдущего
                                 ; умножения (d1) в T[j][i-1]
        ;fmpy.s  d4,d5,d1        ;Умножение sj (d4) на P[i] (d5)
        ;move    x:(r1)+,d5.s    ;Чтение P[i+1] из памяти в регистр d5
                                 ; для следующего цикла
    _enddo:

;Торможение:
 move d1.s,y:(r4)+     ;Запись результата в T[j][N-2].
 fmpy.s  d4,d5,d1      ;Умножение sj на P[N-1], результат в d1.
 move d1.s,y:(r4)+     ;Запись результата в  T[j][N-1].
```

В следующих двух разделах рассматриваются дополнительные оптимизации, реализация которых позволила бы в полной мере использовать эффект развертки для комбинирования команд в теле цикла:

- замена адресации со смещением на адресацию с постинкрементацией адресного регистра;
- перестановка обращений к памяти при наличии дополнительной информации о семантике программы.

4.2.4. Замена адресации со смещением на адресацию с постинкрементацией адресного регистра

Рассмотрим пару команд 3,4 из примера 5.5:

```
3        move    d1.s,x:(r4)
4        move    x:(r1+1),d5.s          ; (I+1)-я итерация
```

Объединению этих двух пересылок в одну параллельную команду препятствует то, что в команде 4 используется адресация со смещением: x:(r1+1). Процессор 1B577 не позволяет параллельно выполнять две пересылки данных, если хотя бы одна использует этот вид адресации. По этой же причине невозможно объединить пересылки 6,7 и 9,10.

В то же время, допускается параллельное выполнение пересылок, использующих адресацию с инкрементацией или декрементацией. Поэтому важной для данной платформы оптимизацией является замена адресации со смещением на инкрементальную адресацию. Существенно также то, что команда, содержащая адрес со смещением, занимает 2 слова и выполняется 6 тактов, в то время как команда с инкрементальной адресацией занимает 1 слово и выполняется 2 такта.

Следующий пример демонстрирует эффект замены способа адресации.

Пример 4.7. Развернутый внутренний цикл. Вместо адресации со смещением используется инкрементальная адресация.

До постпроцессирования:

```
0    do                              N/4,_enddo
1    move        x:(r1)+,d5.s        ;чтение P[i] из памяти в регистр d5
2    fmpy.s      d4,d5,d1            ;умножение sj (d4) на P[i] (d5)
3    move        d1.s,x:(r4)+        ;запись результата (d1) в T[j][i]

4    move        x:(r1)+,d5.s        ;чтение P[i+1] из памяти в регистр d5
5    fmpy.s      d4,d5,d1            ;умножение sj (d4) на P[i+1] (d5)
6    move        d1.s,x:(r4)+        ;запись результата (d1) в T[j][i+1]

7    move        x:(r1)+,d5.s        ;чтение P[i+2] из памяти в регистр d5
8    fmpy.s      d4,d5,d1            ;умножение sj (d4) на P[i+2] (d5)
9    move        d1.s,x:(r4)+        ;запись результата (d1) в T[j][i+2]
```

```
10   | move      x:(r1)+,d5.s      ;чтение P[i+3] из памяти в регистр d5
11   | fmpy.s    d4,d5,d1          ;умножение sj (d4) на P[i+3] (d5)
12   | move      d1.s,x:(r4)+      ;запись результата (d1) в T[j][i+3]
13   | _enddo:
```

Отметим, что здесь удалены за ненадобностью команды продвижения адресных регистров "(r1)+n1" и "move (r4)+n4". Постинкрементальная адресация "(r1)+" и "(r4)+" обеспечивает требуемое увеличение значений регистров.

После постпроцессирования:

```
0    | do        N/4,_enddo
1    | move      x:(r1)+,d5.s
2    | fmpy.s    d4,d5,d1
3,4  | move      d1.s,x:(r4)+      y:(r1)+,d5.s
5    | fmpy.s    d4,d5,d1
6,7  | move      d1.s,x:(r4)+      y:(r1)+,d5.s
8    | fmpy.s    d4,d5,d1
9,10 | move      d1.s,x:(r4)+      y:(r1)+,d5.s
11   | fmpy.s    d4,d5,d1
12   | move      d1.s,x:(r4)+
13   | _enddo:
```

Результат заметно лучше - удалось скомбинировать три пары пересылок данных.

4.2.5. Перестановки обращений к памяти

Более интенсивного комбинирования можно достичь, объединив тройки команд, например, команды с номерами 5, 3, 7:

```
5,3,7| fmpy.s    d4,d5,d1   d1.s,x:(r4)+    y:(r1)+,d5.s
```

Здесь, как и в теле конвейеризованного цикла (пример 4.6), одновременно выполняется запись в память результата умножения P[i-1]*sj → T[j][i-1], вычисляется произведение P[i]*sj и считывается из памяти P[i+1].

Однако такому комбинированию препятствует ограничение на перестановки команд доступа к памяти. Память рассматривается постпроцессором как один "регистр": считается, что любая команда, которая пишет значение в память изменяет этот "регистр", поэтому последующие команды считывания из памяти нельзя выполнить раньше этой команды.

Подобное ограничение гарантирует корректную работу с памятью, но существенно снижает свободу постпроцессора переставлять (и, следовательно, комбинировать) команды. Имея дополнительную информацию о семантике программы, во многих случаях можно было бы снять это ограничение и получить более эффективный параллельный код.

Поясним это подробнее на примере задачи о вычислении внешнего произведения векторов. Рассмотрим команды 3, 4 из примера 5.7 (до постпроцессирования):

```
3   |   move    d1.s,x:(r4)+      ;запись результата (d1) в T[j][i]
4   |   move    x:(r1)+,d5.s      ;чтение P[i+1] из памяти в регистр d5
```

Здесь адресные регистры r1 и r4 выполняют роль указателей на текущие элементы массивов P и T соответственно. Если ничего не известно о расположении массивов P и T, то нельзя выполнить команду 4 раньше, чем 3, поскольку не исключена возможность, что команда 3 пишет как раз в то слово, которое считывается в команде 4. Если же известно, что массивы P и T не могут перекрываться в памяти, то порядок выполнения команд 3, 4 может быть произвольным. Располагая этой информацией, постпроцессор мог бы сгенерировать более эффективный код.

Ниже показан развернутый внутренний цикл функции для вычисления внешнего произведения векторов. Показан результат постпроцессирования при условии, что используется инкрементальная

адресация и доступна информация о том, что массивы T и P не перекрываются.

```
          do N/4,_enddo

          ;P[i] считывается в регистр:
1         |  move    x:(r1)+,d5.s
          ;умножение и считывание данных для следующего умножения:
2,4       |  fmpy.s  d4,d5,d1        x:(r1)+,d5.s
          ;умножение, сохранение предыдущего произведения и считывание
          ;данных для следующего умножения:
5,3,7     |  fmpy.s  d4,d5,d1        d1.s,x:(r4)+    y:(r1)+,d5.s
          ;умножение, сохранение предыдущего произведения и считывание
          ;данных для следующего умножения:
8,6,10    |  fmpy.s  d4,d5,d1        d1.s,x:(r4)+    y:(r1)+,d5.s
          ;умножение и сохранение предыдущего произведения:
11,9      |  fmpy.s  d4,d5,d1        d1.s,x:(r4)+
          ;сохранение последнего произведения:
12        |  Move    d1.s,x:(r4)+
          _enddo:
```

Здесь тело цикла содержит всего 6 VLIW-строк (в расчете на 4 итерации неразвернутого цикла) против 13 в примере 5.5 и 9 в примере 5.7.

В стандарте языка C99 предусмотрен квалификатор restrict, который может относиться к указателю или массиву. Этот квалификатор обозначает эксклюзивный доступ: к области памяти, к которой обращаются посредством массива или указателя, декларированного с квалификатором restrict, не должно быть обращений по другим указателям или массивам. (Компилятор gcc поддерживает ограниченный синтаксис применения этого квалификатора.)

Пример 4.8. Использование квалификатора restrict в декларации функции для вычисления внешнего произведения векторов.

```
void vmult (int N, V1[restrtict N], V2[restrtict N],
                   V3[restrtict N][restrtict N]);
```

Научившись передавать постпроцессору информацию о том, что данный адресный регистр на данном участке кода соответствует restrict-указателю или массиву, можно достичь более эффективного планирования.

4.2.6. Оценка эффективности оптимизаций

Ниже приведена таблица, которая показывает на примере программы вычисления внешнего произведения векторов, какого эффекта можно ожидать от реализации рассмотренных выше оптимизаций.

Таблица 4.2. Эффект от реализации различных дополнительных оптимизаций

	количество тактов	коэффициент эффективности
Ручной ассемблерный код (пример 4.6)	3 такта на 1 итерацию	1.00
Результат компиляции без развертки цикла	5 тактов на 1 итерацию	1.67
Результат компиляции с разверткой и с применением инкрементальной адресации	17 тактов на 4 итерации	1.42
Результат компиляции с разверткой, применением инкрементальной адресации и перестановками обращений к памяти	14 тактов на 4 итерации	1.17

Исследование других тестов показывает, что дополнительные оптимизации в сочетании с разверткой цикла могут дать еще более значительное ускорение. Например, в задаче транспонирования матриц замена способа адресации в сочетании с разверткой цикла дает коэффициент эффективности 1.04, а в задаче сложения векторов при разрешении перестановок обращений к памяти коэффициент эффективности составляет 1.05.

Возможно, часть из рассмотренных выше оптимизаций (развертка, замена способа адресации) специфичны для рассматриваемого процессора и необязательно окажутся эффективными для других процессоров с явным параллелизмом, которые могут налагать другие ограничения на комбинирование команд; оптимизация, связанная с перестановками обращений к памяти, носит общезначимый характер. Тем не менее, накопление банка частных оптимизаций может позволить в будущем достаточно быстро создавать оптимизирующие компиляторы для таких процессоров, избирательно применяя подходящий набор оптимизаций.

4.2.7. Распределение регистров

В главе 3 говорилось об ограничениях на комбинирование элементарных команд, связанных с распределением регистров. При настройке компилятора распределение регистров задано таким образом, чтобы в командах умножения преимущественно использовались комбинации регистров, позволяющие объединять ее со сложением. Тем не менее, на линейном участке, содержащем большое количество операций умножения, достичь оптимального распределения регистров в компиляторе не удается. При ручном программировании на ассемблере оно достигается ценой проб и ошибок или за счет большого опыта программиста. Следующий пример демонстрирует эффект хорошего и плохого распределения регистров.

```
/* Вычисление 4 нецентральных моментов. */

void moment (register float *V, register long NV, float res[4]) { register float
m1=0.0, m2=0.0, m3=0.0, m4=0.0;
        while (NV--) {
                register float v = *(V++);
                register float v2 = v * v;
                m1 += v;
m2 += v2;
m3 += v*v2; m4 += v2*v2;
        }
        res[0]=m1; res[1]=m2; res[2]=m3; res[3]=m4;
}
```

Тело цикла в оптимальной программе, написанной вручную на ассемблере (5 тактов на итерацию):

```
do n4,_enddo1
        fmpy d4,d4,d0 fadd.s d4,d1
        fmpy d4,d0,d0 fadd.s d0,d2 fmpy d4,d0,d0 fadd.s d0,d3
        fadd.s d0,d7 x:(r4)+,d4.s
_enddo1:
```

Тело цикла в скомпилированной программе (8 тактов на итерацию):

```
do d1.1,L7
        move   x:(r1)+,d4.s
        fadd.s d4,d5
        fmpy.s d4,d4,d7
        fmpy   d7,d4,d1 fadd.s d7,d0
        fadd.s d1,d6
        fmpy.s d7,d7,d1
        fadd.s d1,d3
L7:
```

В программе, написанной вручную, все умножения совмещены со сложениями, в то время как в скомпилированной программе из-за неоптимального распределения регистров только одно умножение совмещено со сложением.

Можно рассматривать два пути решения этой проблемы, но оба они довольно сложны:

- выделение регистров в компиляторе с учетом перспективы дальнейшего комбинирования на стадии распределения;
- перераспределение регистров постпроцессоре (локальное или глобальное) с целью достижения максимальной плотности комбинирования внутренних циклов.

5. Заключение

В диссертационной работе получены следующие основные результаты.

1. Проведен анализ существующих методов оптимизации объектного кода для процессорных архитектур с параллелизмом на уровне команд. Выделены основные классы таких методов.
2. Разработан алгоритм планирования потока команд для процессоров с длинным командным словом путем перебора планов выполнения по методу динамического программирования. В алгоритме предусмотрены оптимизации, направленные на сокращение перебора, которые позволяют обеспечить приемлемое время обработки входных программ.
3. Реализован постпроцессор, выполняющий планирование и другие оптимизации объектного кода. В постпроцессоре выделено универсальное, архитектурно-независимое ядро.
4. Разработаны и реализованы средства настройки оптимизирующего постпроцессора на различные целевые архитектуры. Выполнена настройка на отечественный микропроцессор 1В577.
5. Разработан оптимизирующий кросс-компилятор с языка Си для отечественных целевых ЭВМ на основе микропроцессора 1В577. Компилятор позволяет сочетать преимущества программирования на языке высокого уровня с качеством объектного кода, близким к ручному, что подтверждено полученными экспериментальными оценками.

Таким образом, в диссертационной работе содержится решение задачи, имеющей существенное значение для оптимизации программ при

компиляции с языков высокого уровня - оптимизация объектного кода для процессорных архитектур с длинным командным словом.

6. Литература

1. Ахо А., Сети Р., Ульман Д., Компиляторы: принципы, технологии и инструменты. – М., Издательский дом "Вильямс", 2001.
2. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции, том 2. - М., Мир, 1978.
3. Балашов В.В., Капитонова А.П., Костенко В.А., Смелянский Р.Л., Ющенко Н.В. Метод и средства оценки времени выполнения оптимизированных программ. - Программирование #5, 2000, с. 52- 61.
4. Вьюкова Н.И., Галатенко В.А., Самборский С.В., Шумаков С.М. Алгоритм планирования потока команд для VLIW-процессоров. - М.: НИИСИ РАН, 2002.
5. Вьюкова Н.И., Галатенко В.А., Самборский С.В., Шумаков С.М. Генерация эффективного кода для процессорных архитектур с явным параллелизмом. - М.: НИИСИ РАН, 2001.
6. Вьюкова Н.И., Галатенко В.А., Самборский С.В., Шумаков С.М. О проблеме оптимизации кода для процессорных архитектур с явным параллелизмом. – Труды международной конференции "Параллельные вычисления и задачи управления". – М.: ИПУ РАН, 2001.
7. Вьюкова Н.И., Галатенко В.А., Самборский С.В., Шумаков С.М. Описание VLIW-архитектуры в оптимизирующем постпроцессоре. М.: НИИСИ РАН, 2001.
8. Грис Д. Конструирование компиляторов для цифровых вычислительных машин. - М., Мир, 1975.
9. Дорошенко А.Ю., Куйвашев Д.В. Архитектура модульного кросс-компилятора для микропроцессоров с очень длинным командным словом. - Проблемы программирования, 2000 г., N 3-4, с. 122-134 (на

укр. языке).

10. Дорошенко А.Ю., Куйвашев Д.В. Интеллектуализация векторизующих компиляторов для микропроцессоров с длинным командным словом. - Проблемы программирования, 2001 г., N 1-2, с. 138-151 (на укр. языке).

11. Евстигнеев В.А. Некоторые особенности программного обеспечения ЭВМ с длинным командным словом (обзор). - Программирование, 1991, N 2, стр. 69-80.

12. Ершов А.П. Введение в теоретическое программирование. - М., Наука, 1977.

13. Калашников Д.В., Машечкин И.В., Петровский М.И. Планирование потока инструкций для конвейерных архитектур. - Москва, МГУ, Вестник Московского университета, серия 15 (вычислительная математика и кибернетика), N4, 1999, с. 39-44.

14. Касьянов В.Н. Оптимизирующие преобразования программ. - М.: Наука, 1988 г.

15. Пападимитриу Х., Стайглиц К. Комбинаторная оптимизация. - М., Мир, 1985.

16. Скворцов С.В. Оптимизация кода для суперскалярных процессоров с использованием дизъюнктивных графов. - Программирование 1996, N 2, стр. 41-52.

17. Французов Ю.А. Обзор методов распараллеливания кода и программной конвейеризации. - Программировние, 1992, N 3, стр. 16-37.

18. Шланскер М.С., Рау Б.Р. Явный параллелизм на уровне команд. Открытые Системы, #11-12, 1999.

19. Шумаков. С.М. Обзор методов оптимизации кода для процессоров поддержкой параллелизма на уровне команд. М.: НИИСИ РАН, 2002.

20. ADSP-21000 Family. C Tools Manual. - Analog Devices, Inc. http://www.analog.com/publications/documentation/C-Tools_manual/books.htm

21. Aho A.V., Sethi R, Ullman J.D.: Compilers - Principles, Techniques, and Tools, Addison-Wesley, 1986

22. Allen J.R., Kennedy K., Porterfield C., Warren J.D. Conversion of Control Dependence to Data Dependence. Proceedings of the 10th ACM Symposium on Principles of Programming Languages. Jan. 1983, pp. 177-189.

23. Araujo G., Malik S. Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architectures. - 8th Int. Symp. on System Synthesis (ISSS), 1995, pp. 36-41.

24. Araujo G., Malik S., Lee M. Using Register-Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures. - In ACM IEEE Design Automation Conference (DAC), number 33, pp. 591-596, June 1996.

25. Banerjia S., Havanki W.A., Conte T.M. Treegion Scheduling for Highly Parallel Processors. - Proceedings of the 3rd International Euro-Par Conference (Euro-Par'97, Passau, Germany), pp. 1074-1078, Aug. 1997.

26. Benitez M. E., Davidson J. W. Target-specific Global Code Improvement: Principles and Applications. - Technical Report CS-94-42, Department of Computer Science, University of Virginia, VA 22903.

27. Bernstein D., Rodey M. Global Instruction Scheduling for Superscalar Machines. - Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation, pp.241-255, June 1991.

28. Berson D. A., Gupta R., Soffa M. L. Integrated Instruction Scheduling and Register Allocation Techniques. - In Proc. of the Eleventh International Workshop on Languages and Compilers for Parallel Computing, LNCS, Springer Verlag, Chapel Hill, NC, Aug. 1998.

29. Bharadwaj J., Menezes K. McKinsey C. Wavefront Scheduling: Path Based Data Representation and Scheduling of Subgraphs. The Journal of Instruction-Level Parallelism, May 2000.

30. Bruggen T., Ropers A. Optimized Code Generation for Digital Signal Processors. - Institute for Integrated Signal Processing, http://www.ert.rwth-aahen.de/coal .

31. Chang P. P., Mahlke S. A., Chen W. Y., Warter N. J., Hwu W. W. IMPACT: An architectural framework for multipleinstruction-issue processors," in Proceedings of the 18th International Symposium on Computer Architecture, pp. 266-275, May 1991.

32. Cooper K.D., Schielke P.J., Subramanian D. An Experimantal Evaluation of List Scheduling. - Department of Computer Science, Rice University, Houston, TX, USA: Technical Report, http://cs-tr.cs.rice.edu/Dienst/UI/2.0/Describe/ncstrl.rice-cs/TR98-326

33. Eichenberger A. E., Davidson E. S., Abraham S. G. Minimizing Register Requirements of a Modulo Schedule via Optimum Stage Scheduling. - International Journal of Parallel Programming, February, 1996.

34. Eichenberger A. E., Lobo S. M. Efficient Edge Profiling for ILP-Processors. - Proceedings of PACT 98, 12-18 October 1998 in Paris, France..

35. Fisher J. A. Global code generation for instruction-level parallelism: Trace Scheduling-2. - Tech. Rep. HPL-93-43, Hewlett-Packard Laboratories, June 1993

36. Fisher J.A. Trace scheduling: A Technique for Global Microcode Compaction. - IEEE Transaction on Computers, vol. 7, pp. 478-490, July 1981.

37. Gebotys C. H., Gebotys R. J. Complexities In DSP Software Compilation: Performance, Code Size, Power, Retargetability. 1060-3425/98, IEEE, 1998.

38. Grossman J.P. Compiler and Architectural Techniques for Improving the

Effectiveness of VLIW Compilation. – submitted to ICCD 2000.

39. Hartung J., Gay S.L., Haigh S.G. A Practical C Language Compiler/Optimizer for Real-Time Implementations on a Family of Floating-Point DSPs. - IEEE. Proceedings of of the International Conference on Acoustics, Speech and Signal Processing, New York, U.S., April 1988.

40. Havanki W. A., Banerjia S., Conte T. M. Treegion Scheduling for Wide Issue Processors. - Proc. 4th Intl. Symp. on HighPerformance Computer Architecture, Feb. 1998, pp. 266-276.

41. Hoogerbrugge J., Augusteijn L. Instruction Scheduling for TriMedia. - The Journal of Instruction-Level Parallelism, February 1999

42. Horst E., Kloosterhius W., Heyden J. A C Compiler for the Embedded R.E.A.L DSP Architecture. - Материал получен с телеконференции comp.dsp.

43. Hsu P.Y.T., Davidson E.S. Highly Concurrent Scalar Processing. - Proceedings of the 13th Annual International Symposium on Computer Architecture, pp. 386-395. June 1986.

44. Hwu W.W., Mahlke S.A., Chen W.Y., Chang P.P., Warter N.J., Bringmann R.A., Quelette R.G., Hank R.E., Kiyohara T., Haab G.E., Holm J.G., Lavery D.M. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. - The Journal of Supercomputing, vol. 7, pp. 229-249, May 1993.

45. IA-64 Application Developer's Architecture Guide. - Intel, May 1999.

46. ISO/IEC 9899:1999(E). Programming Languages - C. - ISO/IEC, 1999.

47. Kiyohara T., Gyllenhaal J. C. Code Scheduling for VLIW/ Superscalar Processors with Limited Register Files. Proceedings of the 25th International Symposium on Microarchitecture, Dec. 1992, pp. 197-201.

48. Leung A., Palem K.V. A fast algorithm for scheduling timeconstrained

instructions on processors with ILP. - In The 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT 98), Paris, October, 1998.

49. Leung A., Palem K.V. Scheduling Time-Constrained Instructions on Pipelined Processors. - Submitted for publication to ACM TOPLAS, 1999.

50. Leupers R. Code Generation for Embedded Processors. - ISSS 2000, Madrid/Spain, Sept. 2000.

51. Leupers R. Function Inlining under Code Size Constraints for Embedded Processors. – ICCAD'99, San Jose (USA), Nov 1999.

52. Leupers R. Instruction Scheduling for Clustered VLIW DSPs. -Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT'00).

53. Leupers R. Novel Code Optimization Techniques for DSPs.- 2nd European DSP Education and Research Conference, Paris/France, Sep 1998.

54. Leupers R., Marwedel P. Time-Constrained Code Compaction for DSPs. - 8th Int. System Synthesis Symposium(ISSS), 1995. Trans.on VLSI Systems, Vol. 5, no. 1, March 1997.

55. Liao S., Devadas S., Keutzer K., Tjiang S., Wang A. Storage Assignment to decrease code size. – ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 186-195, 1995.

56. Lin W.-Y., Lee C.G., Chow P. An Optimizing Compiler for the TMS320C25 DSP Chip. - Proceedings of the 5th International Conference on Signal Processing Applications and Technology, pp. I-689-I-694, October 1994.

57. Mahlke S. A., Chen W. Y., Hwu W. W., Rau B. R., Schlansker M. S. Sentinel Scheduling for VLIW and Superscalar Processors. - ASPLOS-V Conference Proceedings, October 1992.

58. Mahlke S.A., Lin D.C., Chen W.Y., Hank R.E., Bringmann R.A. Effective

Compiler Support for Predicated Execution Using the Hyperblock. - Proceedings of the 25th Annual International Workshop on Microprogramming (Portland, Oregon), pp. 45-54, Dec. 1992.

59. Martin M. M., Roth A., Fischer C. N. Exploiting Dead Value Information. - 30th International Symposium on Microarchitecture, pages 125--135, December 1997.

60. Moreno J.H. Dynamic Translation of tree-instructions into VLIW. IBM Research Report, 1996.

61. Motorola DSP96000 User's Manual. - Motorola, Inc., 1990.

62. Motorola DSP96KCC User's Manual. - Motorola, Inc., 1990.

63. Ozer E., Banerjia S. Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures. - Proceedings of the MICRO-31 - The 31th Annual International Symposiumon Microarchitecture, 1998.

64. Pai V. S., Adve S. Code Transformation to Improve Memory Parallelism. - The Journal of Instruction-Level Parallelism, May 2000.

65. Pendry A., Fenwick J. B., Norris J. C. Using SUIF as a Front-end Translator for Register Allocation and Instruction Scheduling Research. - In Second SUIF Compiler Workshop, Stanford, CA, August 1997.

66. Pinter S. Register Allocation with Instruction Scheduling: a New Approach. - Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, pages 248-257, 1993.

67. Pozzi L. Compilation Techniques for Exploiting Instruction Level Parallelism, a Survey. - Milano, Italy, 1998.

68. Rajagopalan S., Vachharajani M,. Malik S. Handling Irregular ILP Within Conventional VLIW Schedulers Using Artificial Resource Constraints. - CASES'00, November 17-19, 2000, San Jose, California.

69. Rao S. IA-64 Code Generation. – Electrical and Computer Engineering, June 2000.

70. Stallman R. Using and Porting GNU CC. - FSF, Boston, USA.