Санкт-Петербургский Государственный
Университет
Математико-механический факультет

Кафедра системного программирования

Оптимальное планирование инструкций для процессоров семейства IA-64 с использованием алгоритма A*

Дипломная работа студента 545 группы
Галанова Сергея Евгеньевича

Научный руководитель	.... /подпись/	к.ф.-м.н., доц. Д.Ю. Булычев
Рецензент	.... /подпись/	к.ф.-м.н., доц. Н.Ф. Фоминых
“Допустить к защите” зав. кафедры	.... /подпись/	д.ф.-м.н., проф. А.Н. Терехов

Санкт-Петербург
2007

Содержание

1 Введение **2**

2 Обзор существующих подходов **3**
- 2.1 Некоторые понятия 3
- 2.2 Списковое планирование 6
- 2.3 Целочисленное линейное программирование 6
- 2.4 Программирование ограничений 9
- 2.5 Метод ветвей и границ 10

3 Реализация **12**
- 3.1 Архитектура IA-64 12
- 3.2 Общая структура планировщика 15
- 3.3 Построение дэга зависимостей 15
  - 3.3.1 Анализ предикатов 17
  - 3.3.2 Анализ алиасов 20
- 3.4 Поиск оптимального плана 21
- 3.5 Алгоритм A* 23
- 3.6 Эвристики 24
  - 3.6.1 Эвристика критического пути 24
  - 3.6.2 Эвристика доступных ресурсов 25
  - 3.6.3 Гибридная эвристика 25
  - 3.6.4 Дополнительная эвристика 26
- 3.7 Оптимизации 26
  - 3.7.1 Генерация пустых операций 26
  - 3.7.2 Эквивалентные планы 27
  - 3.7.3 Выделение изоморфных поддэгов 28
  - 3.7.4 Модификации эвристики 29
- 3.8 Некоторые детали реализации 29

4 Результаты **31**

5 Заключение **33**

1 Введение

Одной из важнейших и оказывающих большое влияние на производительность целевого кода стадий компилятора является стадия планирования инструкций. При выполнении планирования (в конце работы компилятора, после выбора машинных инструкций), происходит переупорядочение инструкций, а также выбор для каждой инструкции вычислительного устройства, на котором она будет выполняться (если архитектура процессора предусматривает такую возможность), с целью максимально эффективного задействования имеющихся ресурсов (вычислительных устройств) процессора — исключения как можно большего числа циклов их простоя. В результате работы планировщика получается план инструкций.

Задача планирования встает для любых процессоров, имеющих конвейер инструкций. Для современных *многопусковых (multiple issue)* процессоров, позволяющих запускать несколько инструкций за один такт (за счет наличия нескольких вычислительных устройств одного типа и нескольких конвейеров), эта задача стоит особенно остро. Эффективное планирование — необходимое условие для задействования имеющегося в таких процессорах параллелизма (параллелизма уровня инструкций, Instruction Level Parallelism, ILP).

Известно, что задача оптимального планирования, даже в весьма простых (и непрактичных) формулировках, NP-трудна. Оптимальность планирования выражается в минимизации некоторой целевой функции — длины плана, обычно определяемой как время его выполнения в тактах процессора (хотя возможны и другие варианты — например, размер машинного кода или количество потребляемой при выполнении энергии). Основанный на эвристиках подход, повсеместно применяемый в промышленных компиляторах, — списковое планирование (list scheduling) — дает субоптимальные результаты. Хотя эти результаты в большинстве случаев весьма близки к оптимальным, существуют приложения, в которых можно пожертвовать увеличением времени компиляции ради ускорения кода (например, встроенные системы). Поэтому большой интерес представляют способы оптимального планирования инструкций, завершающегося за разумное время.

В данной работе рассматривается задача оптимального локального планирования инструкций для процессора Intel Itanium 2. Локальность выражается в том условии, что оптимально планируется каждый отдельный базовый блок графа потока управления. Данный процессор относится к классу EPIC (Explicitly Parallel Instruction Computing), который является усовершенствованием класса VLIW (Very Long Instruction Word).

Однако на код, предназначенный для таких процессоров, накладываются дополнительные ограничения по кодированию инструкций (шаблоны), что существенно усложняет работу планировщика (подробнее об EPIC будет сказано в разделе 3.1).

Предлагаемый в данной работе способ планирования основан на использовании эвристического алгоритма A*, позволяющего при наличии достаточно точной оценки длины оптимального плана найти сам этот план за допустимое время.

2 Обзор существующих подходов

Начнем с определения некоторых важных понятий и рассмотрения существующих подходов к решению задачи планирования.

2.1 Некоторые понятия

Направленным графом называется тройка $(V, E, \delta), \delta \in V \times V$, где V — множество *вершин* графа, E — множество *дуг* графа, а δ — отображение, сопоставляющее каждой дуге ее *начало* и *конец* (обозначаемые $src(e)$ и $dst(e)$ соответственно). Через $G.V$ и $G.E$ будем обозначать соответственно множества вершин и дуг графа G.

Для каждой вершины v определены множества *входящих* и *исходящих* дуг $in(v)$ и $out(v)$: $in(v) = \{e \in E \mid src(e) = v\}, out(v) = \{e \in E \mid dst(e) = v\}$.

Множеством *непосредственных последователей* вершины v называется множество $succ(v)$ концов ее исходящих дуг.

Множеством *непосредственных предшественников* вершины v называется множество $pred(v)$ начал ее входящих дуг.

Путем называется последовательность дуг графа $p = e_1, ..., e_n$ такая, что $dst(e_i) = src(e_{i+1}), 1 \leqslant i \leqslant n-1$. При этом вершина $beg(p) = src(e_1)$ называется *началом* пути, а вершина $end(p) = dst(e_n)$ — его *концом*. Вершина принадлежит пути, если она совпадает с началом или концом какой-либо его дуги.

Циклом называется путь p такой, что $beg(p) = end(p)$.

Путь называется *простым*, если он не содержит циклов.

Направленный граф, не содержащий циклов, называется *дэгом (dag, directed acyclic graph)*.

Множество вершин дэга, не имеющих входящих дуг, называется множеством его *корней*. Будем обозначать множество корней дэга G через $roots(G)$.

Множество вершин дэга, не имеющих исходящих дуг, называется множеством его *листьев*. Будем обозначать множество листьев дэга G через $leaves(G)$.

Пусть на графе определена длина дуг, то есть отображение $len\colon E \to \mathbb{R}_+$, сопоставляющее каждой дуге некоторое неотрицательное число, называемое *длиной* этой дуги. *Длиной пути* $p = e_1, ..., e_n$ называется сумма длин его дуг $\sum_{i=1}^{n} len(e_i)$.

Пусть даны два множества вершин, A и B. Пусть P — множество всех путей с началами из A и концами из B: $P = \{p = e_1, ..., e_n \mid beg(p) \in A \wedge end(p) \in B\}$. Путь $cp \in P$ называется *критическим*, если его длина максимальна среди длин всех путей из P: $\forall p \in P\ len(p) \leqslant len(cp)$. В общем случае критических путей из A в B может быть много. Тем не менее, можно говорить о *длине критического пути* (так как длины всех критических путей равны). Мы будем использовать понятия критических путей между вершинами v_1 и v_2 ($A = \{v_1\}, B = \{v_2\}$), критических путей из данной вершины v дэга G ($A = \{v\}, B = leaves(G)$) и критических путей всего дэга G ($A = roots(G), B = leaves(G)$).

Предположим, что граф имеет единственную начальную и конечную вершины (то есть вершины, не имеющие, соответственно, входящих и исходящих дуг). В этом случае можно определить бинарные отношения доминирования и постдоминирования на множестве вершин графа. Вершина $v_1 \in V$ *доминирует* вершину $v_2 \in V$, если любой путь из начальной вершины в v_2 проходит через v_1. Вершина $v_1 \in V$ *постдоминирует* вершину $v_2 \in V$, если любой путь из v_2 в конечную вершину проходит через v_1.

Графом потока управления [9] называется направленный граф, вершинами которого являются инструкции процедуры, соединенные дугами тогда и только тогда, когда существует возможность передачи управления от первой инструкции ко второй.

Следом (trace) [9] называется простой путь в графе потока управления. Этот путь может иметь произвольное число *точек входа* и *точек выхода* (т.е. инструкций, имеющих непосредственного предшественника и непосредственного последователя соответственно, не принадлежащего пути).

Базовым блоком [9] называется след, имеющий единственную точку входа и единственную точку выхода.

Суперблоком [4] называется след, имеющий единственную точку входа и произвольное число точек выхода.

Некоторые архитектуры (в том числе IA-64) поддерживают *предикативное исполнение*. В этом случае инструкция может быть снабжена специальным маркером, задающим *предикатный регистр*. При наличии

такого маркера инструкция будет выполнена, только если соответствующий регистр имеет истинное значение. *Гиперблоком* [6] называется суперблок, содержащий инструкции с предикатами.

Планом инструкций (расписанием) [9] называется информация о порядке исполнения инструкций и назначенных им вычислительных устройствах. Более точное определение зависит от целевой машины. План может также включать *пустые операции*, отсутствовавшие среди исходных инструкций. *Планированием* называется процесс построения плана инструкций. Планирование называется *локальным*, если оно осуществляется внутри базового блока, после чего полученные планы объединяются в результирующий план. В противном случае планирование называется *глобальным*.

Для сохранения семантики программы при планировании должен быть выполнен ряд *ограничений*. К ограничениям относятся, прежде всего, ограничения, связанные с *зависимостями по данным* и *ограничения по ресурсам*.

Ограничения по ресурсам возникают из-за ограниченного набора вычислительных устройств. Например, процессор Itanium 2 может выполнять не более двух операций выгрузки в память за такт, а общее число параллельно исполняемых операций не превышает шести.

Зависимости по данным устанавливаются между инструкциями, адресующими одни и те же регистры или ячейки памяти. Различают три вида зависимостей по данным:

- *Прямая зависимость (flow dependence, read-after-write dependence)* — зависимость между двумя инструкциями, вторая из которых использует результат первой;

- *Антизависимость (anti-dependence, write-after-read dependence)* — зависимость между двумя инструкциями, вторая из которых сохраняет свой результат в ячейку, читаемую первой;

- *Выходная зависимость (output dependence, write-after-write dependence)* — зависимость между двумя инструкциями, сохраняющими результат в одну и ту же ячейку.

В каждом из этих трех случаев накладывается ограничение на порядок исполнения инструкций: зависимая инструкция должна исполняться не раньше той, от которой она зависит. Величина необходимой задержки в тактах называется *латентностью*. Инструкции, имеющие антизависимость, обычно можно исполнять в одном такте, то есть латентность равна нулю.

Для представления зависимостей по данным обычно используется *дэг зависимостей по данным (Data Dependence Dag)*, далее просто называемый *дэгом.* В узлах этого дэга находятся инструкции. Две инструкции соединены дугой, если между ними имеется зависимость. Дуге присваивается длина, равная соответствующей латентности.

2.2 Списковое планирование

Списковое планирование (list scheduling) [9] является общепринятым в промышленных компиляторах алгоритмом планирования, на практике дающим очень хорошие результаты, но не гарантирующим оптимальности.

В этом алгоритме имеется список инструкций с готовыми данными (*Data Ready Set*, DRS), то есть инструкций, для которых все зависимости по данным удовлетворены (но не обязательно удовлетворены зависимости по ресурсам). Изначально этот список состоит из корней дэга зависимостей.

Алгоритм последовательно заполняет слоты каждого такта процессора инструкциями. На каждом шаге выбирается какая-либо инструкция из списка готовых инструкций (какая именно — определяется на основе эвристики) и запускается в текущем слоте. Если инструкция может быть выполнена на различных устройствах, конкретное устройство также выбирается на основе эвристики.

Если ни одна инструкция из списка не может быть запущена в текущем слоте (из-за недостатка свободных устройств или ограничения по кодированию в случае EPIC), вставляется пустая операция.

После каждого шага (запуска очередной инструкции либо начала нового такта) к списку готовых инструкций добавляются инструкции, запуск которых стал возможным после удовлетворения соответствующих зависимостей по данным.

Работа алгоритма завершается, когда все исходные инструкции были спланированы.

В качестве эвристики обычно применяется длина критического пути инструкции (то есть сначала запускаются инструкции с более длинными критическими путями).

2.3 Целочисленное линейное программирование

Одним из подходов к оптимальному планированию является планирование с помощью целочисленного линейного программирования.

Задача *линейного программирования* — нахождение кортежа вещественных чисел, удовлетворяющих совокупности линейных неравенств и доставляющих минимум некоторой целевой функции, линейно зависящей от этих чисел:

$$\begin{aligned} P_F = \{x \mid Ax \geqslant b, x \in \mathbb{R}^n_+\}, c \in \mathbb{R}^n, b \in \mathbb{R}^m, A \in \mathbb{R}^{m+n}, \\ x \in P_F, \qquad (1) \\ \min z_{IP} = c^T x. \qquad (2) \end{aligned}$$

В задаче *целочисленного линейного программирования* к приведенным выше условиям добавляется условие целочисленности переменных x_i:

$$x \in P_F \cap \mathbb{Z}^n. \qquad (3)$$

Условие (2) может отсутствовать. В этом случае ищется просто решение системы неравенств.

В общем случае эта задача NP-трудна, однако для большого числа случаев решение может быть найдено за полиномиальное время.

Посмотрим, как можно свести нашу задачу (оптимальное планирование инструкций) к линейному программированию.

Пусть m — верхняя граница длины оптимального плана. Ее можно найти, применив сначала "легкий" алгоритм планирования (например, списковое планирование).

Будем отождествлять число i и инструкцию с номером i.

Переменными будут являться $x_{ij}, i \in V, j \in [1:m]$, где x_{ij} принимает значение 1, если инструкция i запущена в такте j, и 0 — иначе.

Далее задаются ограничения. Например, каждая инструкция должна быть запущена ровно один раз. Это выражается следующим условием:

$$\sum_{i \in V, j \in [1:m]} x_{ij} = |V|.$$

Тот факт, что процессор может выполнять не более r инструкций за такт выражается следующей группой условий:

$$\sum_{i \in V} x_{ik} \leqslant r, \forall k \in [1:m].$$

Номер такта, в котором запущена инструкция i задается как $\sum_{j \in [1:m]} j x_{ij}$. Тогда условие зависимости по данным между инструкциями i_1 и i_2 с латентностью $l_{i_1 i_2}$ записывается следующим образом:

$$\sum_{j\in[1:m]} jx_{i_2j} + l_{i_1i_2} \leqslant \sum_{j\in[1:m]} jx_{i_1j}.$$

Аналогичным образом записываются остальные ограничения.

Рассмотренный подход был успешно применен в работе [14]. В ней рассматривается локальное планирование для однопусковой машины, с максимальной латентностью 3.

Планировщик определяет нижнюю и верхнюю границы длины оптимального плана и перебирает длины (от меньшей к большей), пытаясь решить задачу ЦЛП с фиксированной длиной плана. Таким образом, условие минимизации целевой функции в такой формулировке отсутствует.

Поскольку время решения задачи зависит от ее размера нелинейно, ее уместно разбить на несколько подзадач меньшего размера (если это возможно), а затем объединить их решения. Одним способом такого разбиения является разделение дэга по узлу, который постдоминирует корень дэга и доминирует лист (дэг приведен в *стандартную форму*, то есть имеет единственный корень и лист — фиктивные вершины, которые потом можно будет удалить). Эта инструкция определяет *барьер*: все инструкции, находящиеся "выше" нее в дэге должны быть запущены раньше, а находящиеся "ниже" — позже. Таким образом, поддэги, разделенные такими барьерными вершинами, можно планировать независимо.

Для каждой инструкции определяется нижняя и верхняя границы номеров тактов, в которых она может быть запущена. Тогда вместо глобального значения m, фигурирующего в исходной формулировке можно использовать отдельное значение (значительно меньшее m) для каждой инструкции. Это позволяет существенно сократить число переменных в задаче.

Далее из дэга удаляются *излишние дуги*, то есть такие дуги, из начала которых можно прийти в конец, по пути, который длиннее этой дуги. Ясно, что эти дуги не накладывают новых ограничений на значения переменных (по сравнению с ограничениями, накладываемыми дугами соответствующего пути), однако эти ограничения все же присутствуют в задаче, увеличивая ее размер (и следовательно, время решения).

В ряде случаев число ограничений можно еще дальше сократить, преобразовав некоторые поддэги. Рассмотрим пару вершин дэга, между которыми существует более одного пути, и все пути содержат хотя бы одну вершину кроме этих двух. Такая пара вершин задает *регион* — множество вершин, являющихся последователями первой вершины и предше-

ственниками последней. При выполнении определенных условий можно заменить регион на линейный участок, полученный в результате оптимального планирования этого региона. Очевидно, что это преобразование может серьезно сократить число излишних дуг в дэге.

Кроме того, в некоторых случаях, если длина критического пути региона меньше реального числа тактов, требуемого для его выполнения, можно вставить дугу с соответствующей длиной из начальной вершины региона в конечную.

Эти и некоторые другие специфические преобразования позволили авторам добиться очень хороших результатов: пакет из 7402 блоков (среди которых есть блоки с длиной около тысячи инструкций) обрабатывается за 98 секунд.

В работе [5] ЦЛП применяется в задаче планирования для процессора Itanium.

Ввиду сложности точной формулировки задачи ЦЛП для процессоров EPIC планирование осуществляется в два этапа. На первом этапе (*макро-планирование*) ищется предварительный план минимальной длины, в котором учтены зависимости по данным и ограничения ресурсов, но который пока не является корректным планом из-за ограничений по кодированию. На втором этапе (*упаковка, bundling*) происходит формирование корректного плана, состоящего из пакетов, которые содержат точную информацию о распределении инструкций между тактами (подробнее об особенностях IA-64 написано в разделе 3.1).

Строго говоря, такое разбиение задачи на два этапа, должно приводить к потере оптимальности, хотя авторы утверждают, что на использованных ими тестах решения были оптимальными.

Полученные результаты более скромные, чем полученные в рассмотренной ранее работе. Блоки размером до 50 инструкций обрабатываются в среднем за время, не превышающее 10 секунд.

2.4 Программирование ограничений

Программирование ограничений (constraint programming) — это задача поиска значений переменных, удовлетворяющих определенным ограничениям.

Имеется n переменных $\{x_1, ..., x_n\}$, конечное множество возможных значений (*домен*) $dom(x_i)$ для каждой переменной x_i и набор *ограничений* $\{C_1, ..., C_r\}$, задающих ограничения на возможные комбинации переменных. Требуется найти значение для каждой переменной из ее домена так, чтобы все ограничения были выполнены.

Обычно задача решается алгоритмом с возвратом. При этом при установке переменной значения происходит *продвижение констант*: в ограничения подставляются уже известные значения переменных. После этого домены оставшихся переменных сужаются так, чтобы полученные ограничения были совместными.

В работе [7] программирование ограничений применяется к планированию инструкций для многопускового процессора. Каждой инструкции сопоставляется переменная со значением, равным номеру такта, в котором будет запущена эта инструкция.

Рассмотрим ограничения. Ограничения задаются, прежде всего, зависимостями по данным и ограничениями по ресурсам. К этим базовым типам ограничений (которых достаточно для корректного решения задачи) добавляются дополнительные, позволяющие сократить время решения задачи. К этим ограничениям относятся следующие:

- *Ограничения по расстоянию (distance constraints)*, так же, как и в [14], добавляются для пары инструкций, если можно определить, что для выполнения региона, задаваемого этой парой, требуется больше тактов, чем получается из длины критического пути.

- *Ограничения для безопасного удаления (safe pruning constraints)* — специальный вид ограничения, позволяющий уменьшить домен переменной.

- *Ограничения по доминированию (dominance constraints).* Дэг разбивается на изоморфные поддэги. При выполнении некоторых условий можно сразу задать порядок исполнения соответствующих инструкций этих поддэгов (без нарушения оптимальности). Это ограничение и задает порядок для каждой пары соответствующих инструкций.

Авторами были получены весьма хорошие результаты: на тестах из пакета SPEC (которые включали блоки длиной до 2600 инструкций) их планировщик работал около двух часов, не завершившись лишь на нескольких из них.

В работе [8] авторы применяют свой подход для глобального планирования на суперблоке.

2.5 Метод ветвей и границ

Метод ветвей и границ (branch and bound) [12] перебирает возможные варианты решения в явном виде. Среди этих вариантов выбирается

оптимальный. При этом используются различные методы сокращения пространства перебора, позволяющие отбросить варианты, заведомо не являющиеся оптимальными.

К процессорам EPIC этот метод был применен в работе [3]. Однако время работы этого алгоритма оказывается слишком велико. Поэтому авторы также рассматривают эвристики, позволяющие существенно сократить пространство перебора, но не сохраняющие оптимальность.

В работе [13] метод ветвей и границ с успехом применяется к оптимальному локальному (на базовом блоке) и глобальному (на суперблоке и следе) планированию инструкций для многопускового процессора.

Алгоритм работает следующим образом. Сначала определяются нижняя и верхняя границы длин оптимального плана. Затем перебираются значения в найденном интервале и методом ветвей и границ ищется план соответствущей длины. Первый найденный план будет, очевидно, оптимальным.

Для каждой инструкции определяется ее *время выхода (release time)*, то есть наименьший номер такта, в котором инструкция может быть запущена, и *крайний срок (deadline)*, то есть наибольший номер такта, в котором инструкция может быть запущена так, чтобы было соблюдено текущее ограничение на длину искомого плана.

Для сокращения пространства перебора используются следующие методы:

- *Сокращение интервала (range tightening)*. После запуска инструкции времена выхода других инструкций изменяются (например, если незапущенная инструкция имела время выхода C, и мы начали такт $C+1$, время выхода можно увеличить до $C+1$).

- *Релаксированное планирование (relaxed scheduling)*. Рассматривается задача с более простыми ограничениями. Если она не имеет решения, текущую ветвь рассматривать не надо.

- *Доминирование, основанное на истории (history-based domination)*. При выполнении ряда условий можно сразу отбросить ветвь дерева перебора, если корень этой ветви в некотором смысле "хуже", некоторого уже исследованного узла, для которого выяснилось, что он не приводит к решению.

- *Превосходство инструкции (instruction superiority)*. Предположим, что в некотором узле дерева была исследована ветвь, порожденная запуском некоторой инструкции i, и эта ветвь не дала результата. Может оказаться, что эта инструкция i "превосходит" некоторую

другую (также готовую к запуску) инструкцию j. Тогда ветвь, порожденную запуском j, можно не исследовать.

Для глобального планирования применяются также другие методы, которые мы здесь не рассматриваем.

3 Реализация

В этом разделе рассматривается реализация планировщика. Сначала посмотрим на архитектуру целевого процессора.

3.1 Архитектура IA-64

В качестве целевой машины в данной работе используется процессор Intel Itanium 2 — представитель архитектуры IA-64 [2, 1]. Рассмотрим особенности этой архитектуры.

IA-64 является пока единственной реализацией подхода EPIC (Explicitly Parallel Instruction Computing). EPIC [11] представляет собой новый подход к дизайну архитектуры процессора, являющийся развитием подхода VLIW (Very Long Instruction Word) и призванный устранить его недостатки (в первую очередь, относящиеся к области вычислений общего назначения).

Обычные суперскалярные процессоры требуют весьма сложной аппаратуры для динамического построения плана исполнения инструкций (то есть переупорядочения исходного списка инструкций и назначения им вычислительных устройств) с целью максимально эффективного задействования имеющегося в исполняемой программе параллелизма. Главная идея, на которой основан EPIC, состоит в возложении роли по построению плана (и некоторых других задач) на компилятор. Другими важными проблемами, которые решает EPIC, являются обеспечение совместимости кода между разными представителями одного семейства процессоров и минимизация размера кода.

EPIC предлагает следующие механизмы для эффективного выполнения этих задач:

- Явное указание групп параллельно исполняемых операций и устройств.

- Наличие у компилятора точной информации о латентностях операций и имеющихся регистрах.

- Предикативное и спекулятивное исполнение инструкций.
- Помощь компилятора процессору в предсказании направления ветвления и выборе стратегии размещения данных в кэш-памяти.

Процессор Itanium 2 является вторым представителем архитектуры IA-64 (Intel Itanium). Он имеет 128 64-битных регистров общего назначения, 128 регистров для вычислений с плавающей точкой, 64 однобитных предикатных регистра (для поддержки предикативного исполнения) и восемь 64-битных регистров ветвления (используемых для указания целевого адреса при ветвлении). Itanium 2 содержит шесть арифметико-логических устройств общего назначения, два целочисленных устройства, одно устройство для выполнения операций сдвига, четыре порта памяти (позволяющих запускать две операции загрузки и две операции выгрузки за такт), четыре устройства для выполнения операций с плавающей точкой и набор устройств для выполнения мультимедийных операций.

Большинство арифметических инструкций имеют латентность 1, большинство инструкций с плавающей точкой — 4, загрузки из памяти и выгрузки в память, в случае работы с кэшем первого уровня, имеют латентность 1.

Под *группой инструкций (instruction group)* будем понимать совокупность инструкций, предназначенных для параллельного исполнения. Инструкции упаковываются в *пакеты (bundles)*. Каждый пакет имеет фиксированный размер (128 бит) и содержит три инструкции. Для пакета указывается его *шаблон (template)*, задающий типы вычислительных устройств, назначаемых инструкциям. Для каждой из трех инструкций, составляющих пакет, в шаблоне записывается один из следующих типов: M, I, F, B, L, X. Также после каждой инструкции в шаблоне может быть указан *стоп*, задающий границы между группами инструкций. Не все возможные комбинации типов и положений стопов допустимы. Itanium 2 поддерживает следующие шаблоны: MII, MII_, MI_I, MI_I_, MLX, MLX_, MMI, MMI_, M_MI, M_MI_, MFI, MFI_, MMF, MMF_, MIB, MIB_, MBB, MBB_, BBB, BBB_, MMB, MMB_, MFB, MFB_ (знак подчеркивания здесь соответствует положению стопа). Под типом инструкции будем понимать соответствующий тип, записанный в шаблоне. Следует заметить, что шаблоны MLX и MLX_ особенные в том смысле, что тип X виртуален (то есть не существует инструкции такого типа). Эти шаблоны используются для записи инструкции `movl`, занимающей два слота (и имеющей тип L).

Правила выделения групп инструкций следующие:

- каждый стоп начинает новую группу, которая состоит из некоторого набора инструкций, последовательно идущих за этим стопом;

- длина группы инструкций не может превышать максимально допустимого значения для данного процессора (для Itanium 2 это шесть инструкций);

- группа инструкций не может пересекаться более чем с двумя пакетами;

- если для исполнения запускаемой в текущем такте инструкции не хватает ресурсов или ее входное значение еще не вычислено, вставляется неявный стоп.

При запуске инструкция назначается определенному порту. Типы портов совпадают с типами инструкций. Порты назначаются последовательно внутри группы инструкций (например, первая M-инструкция в группе назначается порту M0, вторая F-инструкция — порту F1 и т. д.) В ряде случаев порты могут меняться (то есть сначала назначаются порты M2 и M3, а потом — M0 и M1). Всего имеется 4 порта типа M, 2 I-порта, 2 F-порта, 3 B-порта и 2 L-порта.

Для каждой инструкции задаются допустимые порты, на которых она может быть запущена. Например, большинство арифметических инструкций могут быть запущены на всех M- и I-портах. Инструкции с плавающей точкой могут быть запущены на F-портах. Загрузки из памяти могут быть запущены на портах M0 и M1, а выгрузки в память — на M2 и M3.

В качестве примера рассмотрим следующий фрагмент кода.

```
{ mmi
    add r1 = r2, r3             // => M0
    add r4 = r5, r6             // => M1
    add r7 = r8, r9             // => I0
}
{ m.mi
    add r9 = r2, r5 ;;          // => M2
    ld4 r2 = [r1]               // => M0
    add r8 = r9, r10            // => I0
}
{ mfi.
    sub r6 = r7, r4             // => M1
    fadd f3 = f4, f5            // => F0
```

```
    add r12 = r1, r4 ;;          // => I1
}
```

Пакеты обрамлены фигурными скобками. В начале каждого пакета указан его шаблон. Стоп обозначается точкой с запятой. В комментарии к каждой инструкции указан порт, на котором она будет запущена.

3.2 Общая структура планировщика

Представляемый в данной работе планировщик обрабатывает готовый ассемблерный код процессора Itanium 2, выполняя планирование каждого базового блока независимо от остальных. В базовых блоках допускаются инструкции с предикатами (поэтому эти блоки являются гиперблоками) и посторонние выходы. Однако порядок инструкций относительно выходов не меняется (если только выход не является независимым от этой инструкции, то есть инструкция и операция выхода не имеют несовместные предикаты).

Работа планировщика состоит из следующих четырех этапов:

1. разбора исходного ассемблерного файла и выделения в нем блоков;
2. построения дэга зависимостей для каждого блока;
3. поиска оптимального плана для каждого блока;
4. вывода результирующего ассемблерного файла.

Первый и последний этапы тривиальны, и мы не будем на них останавливаться. Перейдем к рассмотрению основных частей программы.

3.3 Построение дэга зависимостей

Дэг зависимостей (далее просто дэг) позволяет абстрагироваться от конкретного представления и порядка инструкций исходной программы. Для построения дэга используется стандартный алгоритм, аналогичный описанному в [9], сканирующий блок инструкций и вставляющий в вершину, соответствующую очередной инструкции дуги из инструкций, от которых зависит данная инструкция. Главная задача состоит, таким образом, в поиске этих инструкций.

В ходе работы алгоритма поддерживается текущее состояние, состоящее из следующих компонент:

- $lastWritesToReg$ — отображение, сопоставляющее каждому регистру те инструкции, которые писали в этот регистр последними (таких инструкций может быть много из-за наличия предикатов — подробнее об этом будет сказано в разделе 3.3.1).

- $allReadsFromReg$ — отображение, сопоставляющее каждому регистру все инструкции, которые из него читали.

- $allLoads$ и $allStores$ — все инструкции, которые читали из памяти и писали в память соответственно.

- $otherState$ — состояния анализатора предикатов и анализатора алиасов, о которых будет сказано ниже.

Будем понимать под *ячейками* как ячейки памяти (адрес такой ячейки всегда задается значением некоторого регистра), так и регистры процессора. Каждая инструкция имеет свои *входные ячейки* (то есть ячейки, из которых она читает значение) и *выходные ячейки* (ячейки, в которые она пишет значение).

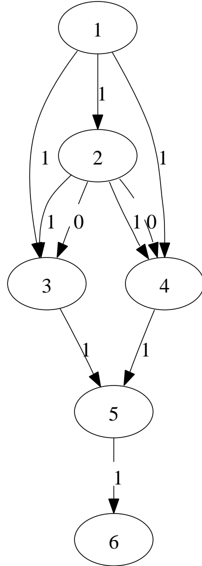
Прямые зависимости вычисляются для каждой входной ячейки путем объединения последних инструкций, писавших в эту ячейку. Выходные и антизависимости вычисляются для каждой выходной ячейки объединением соответственно последних писавших в ячейку и всех читавших из ячейки инструкций. Далее из получившихся множеств удаляются инструкции, имеющие предикат, несовместный с предикатом текущей инструкции.

После построения дэга из него удаляются излишние дэги (аналогично [14]).

Нетривиальным представляется вопрос о поиске инструкций, использовавших заданную ячейку памяти. Поскольку невозможно установить во время компиляции, содержат ли два регистра (задающие адреса двух ячеек) одинаковое значение, приходится использовать пессимистичную оценку и считать, что они всегда одинаковы (кроме случаев, когда можно доказать обратное). Поэтому в качестве инструкций, писавших в ячейку памяти последними, используются вообще все инструкции, писавшие в память, кроме тех, для которых анализатор алиасов установил, что ячейки не совпадают.

Ниже приведен пример блока и дэг зависимостей для него. Рядом с дугами указаны их латентности. Дуги, соответствующие антизависимостям и выходным зависимостям, нарисованы пунктирными линиями. Заметим, что дуги $1 \to 3$, $1 \to 4$ и две дуги с антизависимостями из вершины 2 являются излишними и будут удалены из итогового дэга.

```
1: add r1 = r2, r3
2: cmp p1,p2 = r1, r7
3: add r1 = r1, r2 if p1
4: add r1 = r1, r3 if p2
5: sub r4 = r1, r5
6: add r4 = r7, r2
```



3.3.1 Анализ предикатов

Основная задача анализатора предикатов состоит в разрешении вопроса о том, содержат ли два предикатных регистра в разных точках исполнения программы взаимно исключающие значения (то есть являются несовместными). Для ответа на этот вопрос к состоянию в алгоритме построения дэга добавляется состояние предикатов[1].

Выделяются два множеста: *базовых предикатов* и *зависимых предикатов.* Базовые предикаты — это некоторые неизвестные значения. Эти значения могут либо приходить в блок извне через предикатные регистры (при текущей архитектуре планировщика вся работа происходит внутри блока и информация о других блоках отсутствует, поэтому о значениях входных регистров ничего не известно), либо являться результатом операций сравнения двух регистров внутри блока. Зависимые предикаты — это настоящие предикатные регистры, значения которых выражаются некоторой логической функцией через значения базовых предикатов.

[1] основная идея алгоритма проверки совместимости предикатов предложена в [6]

Algorithm 1 Алгоритм построения дэга

```
sub buildDAG(Block)
  DAG ← emptyGraph;
  for ∀instr ∈ Block do
    addVertex(DAG, instr);
    for ∀(src, lat) ∈ getConflicts(DAG, instr) do
      addEdge(DAG, src, instr, lat);
    end for
    updateStateAfterIssue(DAG, instr);
  end for
end sub
sub getConflicts(DAG, instr)
  conflicts ← [];
  for ∀opnd ∈ inputs(instr) do
    if opnd = Register(r) then
      addRAW(conflicts, lastWritesToReg(r));
    else if opnd = Memory(r) then
      addRAW(conflicts, allStores);
    end if
  end for
  for ∀opnd ∈ outputs(instr) do
    if opnd = Register(r) then
      addWAR(conflicts, allReadsFromReg(r));
      addWAW(conflicts, lastWritesToReg(r));
    else if opnd = Memory(r) then
      addWAR(conflicts, allLoads);
      addWAW(conflicts, allStores);
    end if
  end for
  removeWithIncompatPred(conflicts, instr);
  returnconflicts;
end sub
```

В качестве примера рассмотрим следующий блок:

```
1: add r1 = r2, r3
2: add r4 = r5, r6
3: cmp p1,p2 = r1, r4
4: and p4 = p2, p3
5: cmp p3,p2 = cmp r3, r6
```

В этом примере первая операция `cmp` имеет некоторое значение $u1$ (оно является базовым предикатом). Предикат $p1$ (зависимый) имеет то же самое значение. Предикат $p2$ является его отрицанием. Предикату $p3$ к моменту исполнения четвертой строки не присваивается никакого значения, поэтому он считается входным, и его значение может быть любым (значение базового предиката $u0$). Предикат $p4$ является конъюнкцией предикатов $p2$ и $p3$. Ниже приведено состояние предикатов после выполнения четвертой строки блока.

$$\text{Базовые предикаты:} u0, u1$$
$$\text{Зависимые предикаты:}$$
$$p1 = u1$$
$$p2 = \neg u1$$
$$p3 = u0$$
$$p4 = \neg u1 \land u0$$

Во втором операторе сравнения предикаты $p3$ и $p2$ получают новые значения. Заметим, что предикат $p4$ по-прежнему ссылается на “старые” значения этих предикатов. Вот состояние предикатов после выполнения пятой строки блока.

$$\text{Базовые предикаты:} u0, u1, u2$$
$$\text{Зависимые предикаты:}$$
$$p1 = u1$$
$$p2 = \neg u3$$
$$p3 = u3$$
$$p4 = \neg u1 \land u0$$

Таким образом, состояние предикатов состоит из списка базовых предикатов и леса деревьев, выражающих значения зависимых предикатов через базовые. При обработке очередной инструкции, затрагивающей значения предикатов, к текущему состоянию добавляются базовые предикаты (если инструкция является инструкцией сравнения или ссылается на предикат, которому не было присвоено значение) и новые деревья (в соответствии с семантикой инструкции).

Теперь посмотрим, как можно решить нашу задачу. Два предиката являются несовместными, если они не могут одновременно быть истинными, то есть их конъюнкция имеет значение $false$, независимо от значений базовых предикатов. Сначала строится эта конъюнкция. Затем алгоритм пытается упростить получившееся дерево, используя набор тождеств (таких как $a \wedge true = a$ или $a \wedge \neg a = false$). Если дерево упростилось до константы, ответ получен. Иначе происходит попытка доказать, что значение конъюнкции ложно, перебором всех возможных значений базовых предикатов, участвующих в дереве (если их не очень много — в текущей реализации их количество ограничено шестью, то есть всего 64 комбинации значений). Если и эта попытка оказалась безуспешной, предикаты считаются совместными.

Рассмотрим теперь вопрос о хранении последних инструкций, писавших в некоторый регистр. Поскольку при наличии предикатов блок содержит в себе несколько потоков управления, может быть несколько таких инструкций (в зависимости от того, по какому пути была достигнута текущая точка исполнения). Когда запускается инструкция без предиката, эта инструкция становится единственной писавшей в свои выходные регистры. Если же запускается инструкция с предикатом, она добавляется к спискам инструкций, писавших в ее выходные регистры.

3.3.2 Анализ алиасов

Задача анализатора алиасов состоит в проверке равенства двух адресов в памяти в разных точках исполнения программы. Поскольку в Itanium 2 используется только простая адресация (без смещений и масштабирований), то есть каждый адрес задается значением некоторого регистра процессора, задача сводится к определению для двух регистров в двух точках программы, равны ли они.

Ясно, что в общем случае эта задача неразрешима. В рассматриваемом планировщике используется следующая простая проверка: если два регистра являются результатом суммирования некоторого общего (базового) регистра и различных констант, эти два регистра различны. Этой проверки достаточно для большинства случаев (она обычно срабатывает,

когда идет обращение к памяти на стеке — адреса различных элементов на стеке являются суммами указателя стека и некоторых константных смещений).

Для работы анализатора поддерживается состояние алиасов, которое включает в себя все соотношения описанного выше вида. При тестировании двух регистров на одинаковость производится поиск в списке соотношений записей для этих регистров. Если они найдены и имеют одинаковый базовый регистр и отличающиеся смещения, регистры и соответствующие ячейки памяти считаются различными. При запуске очередной инструкции сложения или копирования значения (которое эквивалентно сложению с числом 0) к этому списку добавляются соответствующие соотношения.

3.4 Поиск оптимального плана

Формализуем задачу поиска оптимального плана. Пусть дан дэг зависимостей DAG для блока $B = I_1, ..., I_n$.

Зафиксируем множество типов портов

$$Types = \{M, I, F, B, L, X\}$$

и множество портов

$$Ports = \{M0, M1, M2, M3, I0, I1, F0, F1, B0, B1, B2, L0, L1\}.$$

Через $InstrPorts(instr)$ будем обозначать множество тех портов, на которых может быть запущена инструкция $instr$.

Группой инструкций будем называть последовательность инструкций и соответствующих им типов $\{(instr_i, type_i)\}_{i=1}^{m}$, для которой выполнены следующие условия:

$$instr_i \in B \cup \{nop\}, 1 \leqslant i \leqslant m \quad (4)$$

$$instr_i \neq nop \Rightarrow instr_i \neq instr_j, 1 \leqslant i, j \leqslant m, i \neq j \quad (5)$$

Для каждой инструкции $instr$ группы может быть определен соответствующий ей порт $port(instr)$.

Длину группы инструкций G, то есть число элементов в ней будем обозначать $len(G)$.

Определим отношение принадлежности инструкции группе следующим образом:

$$belongs(instr, G) \Leftrightarrow \exists i \in [1 : len(G)], p \in Ports : (instr, p) = G_i$$

Значение p в случае истинности правой части этого выражения называется портом, на котором запущена инструкция $instr$, и обозначается $port(instr, G)$.

Частичным планом P назовем последовательность групп инструкций такую, что каждая инструкция из блока принадлежит не более чем одной группе. Длина этой последовательности называется *длиной частичного плана* ($len(P)$).

Будем говорить, что инструкция $instr \in B$ принадлежит частичному плану $P = \{G_i\}_{i=1}^{k}$ (или *instr запущена* в P, $scheduled(instr, P)$), если $\exists i \in [1:k] : belongs(instr, G_i)$. Номер i в этом случае называется номером такта для $instr$ ($cycle(instr, P)$). Множество всех запущенных в P инструкций обозначается $Scheduled(P)$.

Пакетом называется пара $(template, instrs)$, где

$$template\colon [1:m] \to Types \cup \{_\}, instrs\colon [1:3] \to B \cup \{nop\}$$

, состоящая из последовательности типов инструкций и стопов (*шаблона*) и последовательности из трех инструкций.

Для частичного плана P может быть найдена последовательность пакетов $bundles(P)$, состоящих из тех же инструкций (в том же порядке) и имеющих шаблоны, соответствующие типам инструкций (со стопами между группами инструкций). Номер пакета, которому принадлежит инструкция $instr$ обозначается $bundleNo(instr)$.

Частичный план $P = \{G_i\}_{i=1}^{k}$ называется *корректным*, если выполнены следующие условия:

$$\forall e \in DAG.E\ src(e), dst(e) \in Scheduled(P) \Rightarrow cycle(dst(e)) \geqslant cycle(src(e)) + latency(e) \quad (6)$$

$$\forall instr \in Scheduled(P)\ port(instr) \in InstrPorts(instr) \quad (7)$$

$$\forall b \in bundles(P)\ template(b) \in Templates \quad (8)$$

$$\forall G \in P |\{bundleNo(instr) \mid belongs(instr, G)\}| \leqslant 2 \quad (9)$$

Условие (6) выражает зависимости по данным. Условие (7) выражает ограничения по ресурсам. Условие (8) выражает ограничения по кодированию: каждый пакет должен иметь правильный шаблон. Условие (9) выражает тот факт, что группа не может пересекаться более чем с двумя пакетами (откуда, в частности, следует, что длина группы не превышает шести инструкций).

Корректные частичные планы далее будем называть просто планами.

План P, для которого $Scheduled(P) = B$, называется *полным планом.*

Определим на множестве планов следующие две *элементарные операции*:

- операция запуска инструкции $issueOp(instr, type, P)$, которая добавляет к последней группе P пару $(instr, type)$;

- операция завершения группы $endGroup(P)$, которая добавляет новую пустую группу к P.

Эти две операции определены, только если результатом их применения является корректный частичный план.

Дэгом зависимостей, порожденным данным планом, называется исходный дэг зависимостей с модифицированными дугами. Для каждой спланированной инструкции латентности исходящих из нее дуг уменьшены на число тактов, истекшее с момента ее запуска. Если это число больше или равно начальной латентности, дуга удаляется из дэга.

Определим *граф поиска* следующим образом. Вершинами этого графа являются все возможные планы, а также специальная выделенная вершина $finish$. Две вершины, отличные от $finish$ соединены дугой тогда и только тогда, когда вторая из них получена из первой применением одной элементарной операции. Кроме того, имеются дуги из каждого полного плана в $finish$.

Теперь можно сформулировать задачу поиска оптимального плана. Необходимо найти кратчайший путь в графе поиска от начального плана (пустого) до конечного ($finish$). Для решения этой задачи предлагается воспользоваться алгоритмом A*.

3.5 Алгоритм A*

Алгоритм A* [10] является обобщением классического алгоритма Дейкстры для поиска оптимального пути в графе.

Даны две вершины графа, $start$ и $finish$. Требуется найти путь из $start$ в $finish$, имеющий минимальную длину.

Алгоритм поддерживает приоритетную очередь путей, начинающихся в стартовой вершине. В качестве приоритета пути $p = e_1, ..., e_n$ выступает оцениваемая длина кратчайшего полного пути из $start$ в $finish$, первые n дуг которого совпадают с данным путем. Эта оценка складывается из двух величин: $f(x) = g(x) + h(x)$, где $g(x) = len(p)$, а $h(x)$ — *эвристика*, оценивающая кратчайшее расстояние от вершины $x = end(p)$ до $finish$.

Для того, чтобы алгоритм нашел оптимальный путь, на эвристику должно быть наложено ограничение: она не должна превосходить длину кратчайшего пути: $\forall p : beg(p) = x, end(p) = finish\ h(x) \leqslant len(p)$ (мы будем называть это ограничение *условием непереоценки*).

Если эвристика равна нулю, получается обычный алгоритм Дейкстры (осуществляющий полный перебор путей). Если же эвристика достаточно близка к настоящей длине кратчайшего пути, алгоритм A* завершается очень быстро.

3.6 Эвристики

Как видно из описания алгоритма, для эффективного поиска пути требуется найти эвристику, достаточно точно оценивающую (снизу) кратчайшее расстояние от произвольного плана до конечного. Эту задачу можно переформулировать следующим образом: для данного плана необходимо оценить минимальное количество тактов (или групп инструкций), требуемое для запуска оставшихся неспланированными инструкций.

Рассмотрим возможные эвристики.

3.6.1 Эвристика критического пути

Рассмотрим какой-нибудь критический путь $cp = e_1, ..., e_n$ в дэге $PDAG$, порожденном текущим планом. Пусть $v_1, ..., v_{n+1}$ — последовательность вершин, лежащих на этом пути. Обозначим через mc_i нижнюю границу номеров тактов, в которых может быть запущена инструкция v_i.

В качестве mc_1 разумно взять текущую длину плана $len(P)$. Из определения латентности видно также, что если mc_i — нижняя граница для v_i, то $mc_{i+1} = mc_i + latency(e_i)$ будет нижней границей для v_{i+1}. По индукции легко доказывается, что этой системе уравнений удовлетворяет следующий набор значений mc_i: $mc_i = len(P) + \sum_{j=1}^{i-1} latency(e_j)$. В частности, $mc_{n+1} = len(P) + \sum_{j=1}^{n} latency(e_j)$. Отсюда получаем, что минимальное расстояние до такта, в котором можно запустить последнюю инструкцию выбранного пути равно $\sum_{j=1}^{n} latency(e_j)$, то есть длине этого пути. Таким образом, оценка $H_{cp} = len(cp)$, удовлетворяет условию непереоценки и может быть выбрана в качестве эвристики для алгоритма. Эта эвристика называется *эвристикой критического пути.*

3.6.2 Эвристика доступных ресурсов

Факторизуем множество инструкций процессора по отношению равенства множеств портов, на которых могут быть запущены инструкции. Например, арифметико-логические инструкции, которые могут быть запущены на всех портах типа M и I, составляют один класс, а инструкции загрузки и некоторые другие инструкции, которые могут быть запущены на портах M0 и M1 — другой. Всего для Itanium 2 может быть выделено 13 классов.

Количество портов, соответствующих каждому классу, является верхней границей количества инструкций, принадлежащих этому классу, которые могут быть запущены параллельно в одном такте. Обозначим это количество для класса k через M_k.

Пусть ni_k — число неспланированных инструкций, принадлежащих классу k. Тогда минимальное число тактов, требуемое для запуска этих инструкций, равно ni_k/M_k. Каждое из этих чисел является, таким образом, нижней границей на расстояние от текущей точки до конечной. Максимум этих чисел также является нижней границей и удовлетворяет условию непереоценки. Он называется *эвристикой доступных ресурсов* (и обозначается H_r).

3.6.3 Гибридная эвристика

Желательно найти эвристику, которая объединяет достоинства эвристик критического пути и доступных ресурсов, так как существуют дэги, для части которых эффективна первая эвристика, а для другой части — вторая. Приведем здесь такую эвристику.

Сначала определим для каждой неспланированной инструкции i нижнюю границу mc_i номеров тактов, в которых инструкция может быть запущена. Эта граница равна сумме текущего номера такта и длины самого длинного пути в порожденном дэге от корней до данной инструкции.

Далее для каждого из полученных значений mc границ выделим множество всех неспланированных инструкций со значением mc_i, не превышающим данного значения. Все эти инструкции могут быть спланированы, в лучшем случае, к такту mc. Кроме того, остаются инструкции с нижней границей, большей mc. Для них мы можем рассчитать минимальное число mcr требуемых тактов с помощью процедуры, применяемой при расчете эвристики доступных ресурсов. Очевидно, что $mc+mcr$ является нижней границей длин полных планов, достижимых из текущего плана. Выберем максимум M этих значений (для всех возможных

значений mc). Значение $H_g = M - len(P)$ назывется *гибридной эвристикой*. Как было показано выше, оно удовлетворяет условию непереоценки и, следовательно, является корректной эвристикой для нашего алгоритма.

Очевидно, что эта эвристика не хуже эвристик критического пути и доступных ресурсов.

3.6.4 Дополнительная эвристика

Дополнительная эвристика используется для уточнения основной (гибридной) в случае, если для двух разных планов значения основной эвристики оказались одинаковыми. Опишем эту эвристику.

Пусть $PDAG$ — дэг, порожденный текущим планом. Рассмотрим всевозможные пути в этом дэге, которые кончаются в его листьях. Пусть cpl — длина критического пути дэга. Из определения критического пути следует, что cpl не меньше длины каждого из рассматриваемых путей. Пусть np_i — количество путей длины i. Дополнительная эвристика совпадает с вектором $(np_{cpl}, ..., np_0)$. Если при сравнении двух планов (с целью выбора того плана, в направлении которого алгоритм должен продолжать поиск) их основные эвристики оказались равны, сравниваются их дополнительные эвристики. Выбирается план с меньшим ее значением (они сравниваются в лексикографическом порядке).

Смысл этой эвристики состоит в выборе в первую очередь планов с меньшими путями или меньшим количеством наиболее длинных путей.

3.7 Оптимизации

В этом разделе мы рассмотрим различные оптимизации, проводимые для уменьшения графа поиска.

3.7.1 Генерация пустых операций

Для каждого плана (кроме такого, в котором последняя группа заполнена) среди инструкций, могущих быть запущенными, всегда присутствует пустая операция, причем она может быть запущена на порте любого типа (если это позволяют ограничения по кодированию). Это приводит к существенному росту числа вариантов и разрастанию графа поиска. Поэтому чрезвычайно большое значение имеет ограничение генерации пустых операций, не приводящее к потере оптимальности.

Предположим, что для текущего плана нет готовых по данным инструкций, а первая готовая инструкция появится через k тактов. Это

означает, что текущий такт и $k-1$ последующих тактов будут полностью состоять из пустых операций. Основная идея оптимизации генерации пустых операций заключается в том, что типы пустых операций, заполняющих такт, не имеют значения.

Разобьем множество тактов, в которых необходимо сгенерировать пустые операции на три класса. К первому классу отнесем текущий такт, к третьему — последний из последовательности пустых тактов, а ко второму — все промежуточные. Второй класс может оказаться пустым (если $k < 3$), а первый и третий — совпадающими (если $k = 1$).

Ясно, что такты, принадлежащие второму классу, могут быть заполнены пустыми операциями произвольным образом, например, по три операции в такте, так что границы групп инструкций совпадают с границами пакетов. Текущий такт можно заполнить пустыми операциями до конца пакета.

С последним тактом ситуация несколько сложнее, поскольку он граничит с тактом, в котором появятся готовые инструкции, а для него имеет значение шаблон пакета. Поэтому необходимо генерировать варианты, в которых такт, следующий за последним пустым, начинается в различных позициях пакета (на границе пакета, во второй позиции и в третьей позиции). К счастью, в Itanium 2, существует только по одному возможному шаблону, в которых предусмотрена одна граница между группами внутри пакета после первой и второй позиций (это M_MI и MM_I соответственно; есть еще парные шаблоны со стопом в конце пакета, но они имеют общий префикс до первого стопа с этими двумя, поэтому с нашей точки зрения ничем не отличаются). Таким образом, достаточно породить всего три варианта последовательностей из пустых тактов: с последним пустым тактом, занимающим весь пакет, занимающим первую позицию пакета (с типом M) и занимающим первые две позиции пакета (обе с типами M).

Вторая оптимизация, связанная с пустыми операциями, состоит в следующем. Предположим, что для некоторого плана имеются готовые по данным инструкции, но ни одна из них не сможет быть запущена в текущем такте из-за ограничений по ресурсам. В этом случае можно сразу заполнить текущий такт пустыми операциями аналогично рассмотренной только что оптимизации.

3.7.2 Эквивалентные планы

Граф поиска, который рассматривался до настоящего момента, имел структуру дерева: невозможно прийти в одну вершину двумя различными путями (кроме вершины $finish$). Для уменьшения графа поиска

можно ввести понятие эквивалентности планов: если два плана эквивалентны в некотором смысле, им соответствует одна вершина графа. Граф поиска в этом случае будет являться дэгом.

Часто бывает так, что для некоторого плана имеется набор готовых однотипных инструкций. Планировщик будет пытаться запустить эти инструкции во всевозможных комбинациях. Для сокращения числа комбинаций можно определить эквивалентность планов следующим образом.

Для того, чтобы быть эквивалентными, планы должны иметь одинаковое число групп, и соответствующие группы — эквивалентными. Эквивалентность двух групп определяется так: они должны состоять из одного и того же набора инструкций (порядок может отличаться) и иметь одинаковые последовательности типов портов.

Проверка на эквивалентность запускается очень часто и, потому, должна быть эффективной. Для ее реализации используется специальная оптимизированная структура плана, описанная в разделе 3.8.

3.7.3 Выделение изоморфных поддэгов

Предположим, что дэг зависимостей содержит два изоморфных поддэга, то есть задано соответствие между вершиными этих поддэгов, при котором соотвествующие вершины относятся к одному классу и дуги, соединяющие две вершины одного поддэга, одинаковы с дугами, соединяющими соответствующие им вершины другого поддэга.

При выполнении определенных условий инструкции из изоморфных поддэгов можно, не опасаясь потерять оптимальность, запускать в любом выбранном порядке, то есть установить порядок запуска внутри каждой пары соответствующих инструкций. Фактически это эквивалентно вставке в дэг дуг с нулевой латентностью между вершинами каждой такой пары. В частности, в одном очень часто встречающемся случае упомянутые условия всегда выполнены [13].

Речь идет о наборе изолированных однотипных вершин дэга (то есть вершин, не имеющих входящих и исходящих дуг, кроме, быть может, одинаковых дуг из некоторой общей вершины и одинаковых дуг в некоторую другую общую вершину). Такие наборы выделяются в процессе конструирования дэга и далее используются при генерации планов-последователей каждого плана.

3.7.4 Модификации эвристики

Для оптимизации используются также некоторые модификации основной и дополнительной эвристик, позволяющие более эффективно управлять направлением поиска. К этим модификациям относяться следующие:

- Искуственное повышение основной эвристики у планов, полученных слишком ранним завершением группы (это не нарушает корректности алгоритма, так как конечный план не может быть получен применением операции *endGroup*).
- Шаблону MMI отдается предпочтение перед MII.

3.8 Некоторые детали реализации

Данный планировщик был реализован на языке Objective Caml. В этом разделе приведено внутреннее представление некоторых наиболее важных структур. Начнем с рассмотрения представления плана.

```
type schedule =
  {
    groups : group list;
    operations : BitSet.t;
    opsCount : int;
    realOpsCount : int;
  }
```

Поле `groups` является собственно списком групп инструкций. Остальные поля являются производными от `groups` и хранятся с целью повышения эффективности. Поле `operations` является битовым вектором инструкций исходного блока, принадлежащих плану (число инструкций в блоке обычно не превышает двух сотен). Поля `opsCount` и `realOpsCount` содержат счетчики количества всех инструкций и непустых инструкций в плане соответственно.

Представление группы специально разработано для оптимизации требуемой памяти и эффективной реализации проверки эквивалентности групп.

```
type group =
  | ShortGroup of BFUnits.t * BFOps.t * BFOrder.t
  | LongGroup of BFUnits.t * BFOps.t * BFOps.t * BFOrder.t
```

Группа представляет из себя кортеж из трех или четырех 31-битных слов, имеющих следующий смысл.

Первое слово (*слово портов*) содержит информацию о количестве инструкций в группе (биты 0-2), количестве непустых инструкций в группе (биты 3-5) и последовательность портов, на которых будут запускаться инструкции этой группы (биты 6-29). Всего имеется 12 портов, поэтому для кодирования порта достаточно 4 бит. Группа содержит до шести инструкций, следовательно, для кодирования последовательности портов достаточно 24 бит.

Следующие одно или два слова (*слова операций*) кодируют инструкции, содержащиеся в группе (без учета их порядка). Если блок состоит из n инструкций, то для кодирования каждой инструкции требуется $\lceil log(n) \rceil$ бит. В текущей реализации размер блока ограничен 1024 инструкциями, что позволяет упаковать список инструкций в два слова. Если для представления списка достаточно одного слова, используется более компактное представление. Пустым операциями соответствует специальный код. Операции записаны в порядке возрастания их кодов.

Для проверки эквивалентности двух групп, таким образом, достаточно проверить на равенство слова портов и операций.

Последнее слово (*слово порядка*) содержит список индексов инструкций группы в отсортированный последовательности, позволяющий восстановить последовательность инструкций группы.

Рассмотрим теперь представление вершины графа поиска.

```
type point =
  {
    sched : schedule;
    drs : op list list;
    cands : (op * int * int) list;
  }
```

Поле `sched` содержит сам план. Следующие два поля используются для эффективной реализации поиска планов-последователей. Поле `drs` содержит список инструкций, готовых к запуску по данным. Эти инструкции разбиты на классы, каждый из которых содержит инструкции, которые можно запускать в произвольном порядке (см. раздел 3.7.3). Поле `cands` содержит инструкции, хотя бы один предшественник которых в дэге был запущен. Для каждой такой инструкции хранится количество еще не запущенных предшественников и максимальное число тактов, которые должны истечь, прежде чем данная инструкция сможет быть запущена (здесь не учитываются латентности предшественников, которые еще не были запущены).

Таблица 1: Результаты тестирования

	open64	gcc
число блоков	27747	11420
уже оптимально	25636 (92.4%)	8276 (72.5%)
улучшено	1494 (5.4%)	2086 (18.3%)
ухудшено	0	150 (1.3%)
не завершилось	184 (0.66%)	276 (2.4%)
общее время работы	23 мин	31 мин

Таблица 2: Результаты для open64

размер	0..9	10..29	30..49	50..99	100..
число обработанных блоков	844	956	177	108	20
не завершилось	0	27	81	59	16
улучшено блоков	679	696	75	39	3
общее улучшение в тактах	702	969	160	117	7

Ниже приведена схематичная реализация функции *succ*, возвращающей для данной вершины графа поиска ее последователей.

4 Результаты

Данный планировщик был опробован на пакете mesa, входящего в состав набора тестов SPEC FP. В нем активно используются вычисления с плавающей точкой, а значит, имеется большое число высоколатентных инструкций.

Ассемблерный код был получен с помощью компиляторов open64 и gcc. Для каждого блока была вычислена нижняя граница длины плана (с помощью алгоритма, рассчитывающего эвристику). Если эта граница совпадала с длиной исходного плана, блок не обрабатывался, так как построенный компилятором план уже был оптимальным. В противном случае для блока с помощью алгоритма A* искался оптимальный план. Часть блоков была ухудшена (из-за недостаточно точного расчета зависимостей). Для планировщика устанавливалось ограничение по времени в 5 секунд.

Результаты тестирования приведены в таблице 1.

В таблицах 2 и 3 приведены более детальные результаты, распределенные по различным длиным блоков.

Algorithm 2 Реализация операции *succ*

```
sub succ(pt)
  availableOps ← [];
  for ∀ops ∈ pt.drs do
    op ← fst(ops);
    availableOps ← availableOps ∪ canIssue(pt.sched, op);
  end for
  return map((issueOp pt), availableOps) ∪ endGroup(pt);
end sub
suf issueOp(pt, (op, port))
  newDrs ← removeFromDRS(pt.drs, op);
  newCands ← pt.cands;
  for ∀e ∈ DAG.outEdges(op) do
    if ∃c ∈ newCands : c.op = dst(e) then
      c.parentCount ← c.parentCount − 1;
      c.maxCycles ← max(c.maxCycles, latency(e));
    else
      newCands ← newCands ∪ {(op, |DAG.succ(op)|, latency(e)};
    end if
  end for
  return makePoint(Sched.issueOp(pt.sched, op, port), newDrs, newCands);
end sub
sub endGroup(pt)
  if not(canEndGroup(pt.sched)) then
    return [];
  end if
  newDrs ← pt.drs;
  newCands ← pt.cands;
  for ∀c ∈ pt.cands do
    if c.maxCycles > 0 then
      c.maxCycles ← c.maxCycles − 1;
    end if
    if c.maxCycles = 0 ∧ c.parentCount = 0 then
      newCands ← newCands\{c};
      putToDrs(newDrs, c.op);
    end if
  end for
  return makePoint(Sched.endGroup(pt.sched), newDrs, newCands);
end sub
```

Таблица 3: Результаты для gcc

размер	0..9	10..29	30..49	50..99	100..
число обработанных блоков	753	1769	332	178	63
не завершилось	0	31	86	98	60
улучшено блоков	635	1161	171	74	3
общее улучшение в тактах	644	1920	558	303	29

5 Заключение

В данной дипломной работе был разработан оптимальный планировщик инструкций для процессоров класса EPIC. Был предложен ряд эвристик и методов сокращения пространства перебора. Планировщик показал неплохие результаты на наборе тестов (хотя пока работает довольно плохо на очень больших блоках).

Список литературы

[1] *Intel Itanium 2 Processor Reference Manual*, 2004.

[2] *Intel Itanium Architecture Software Developer's Manual. Volume 1, 2, 3*, 2006.

[3] Steve Haga and Rajeev Barua. EPIC Instruction Scheduling Based On Optimal Approaches. *Workshop on EPIC Architectures and Compiler Technology 2001*, 2001.

[4] Richard E. Hank, Scott A. Mahlke, Roger A. Bringmann, John C. Gyllenhaal, and Wen mei W. Hwu. Superblock formation using static program analysis. In *MICRO 26: Proceedings of the 26th annual international symposium on Microarchitecture*, pages 247–255, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[5] D. Kastner and S. Winkel. ILP-based Instruction Scheduling for IA-64. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Snowbird, Utah, USA, June 2001.*, 2001.

[6] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *25th Annual International Symposium on Microarchitecture*, 1992.

[7] Abid M. Malik, Jim McInnes, and Peter van Beek. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. Technical report, School of Computer Science, University of Waterloo, 2005.

[8] Abid M. Malik, Jim McInnes, and Peter van Beek. Optimal super block instruction scheduling for multiple-issue processors using constraint programming. Technical report, School of Computer Science, University of Waterloo, 2006.

[9] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[10] Hart P.E., Nilsson N.J., and Raphael B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on System Science and Cybernetics*, pages 100–107, 1968.

[11] Michael S. Schlansker and B. Ramakrishna Rau. EPIC: An Architecture for Instruction-Level Parallel Processors. Technical report, 2000.

[12] Robert Sedgewick. *Algorithms*. Addison-Wesley Publishing Company, 1983.

[13] Ghassan Omar Shobaki. *Optimal Global Instruction Scheduling Using Enumeration*. PhD thesis, University of California Davis, 2006.

[14] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 121–133, New York, NY, USA, 2000. ACM Press.