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

С. Е. Галанов
sgsoftware@mail.ru

В настоящей статье рассматривается задача оптимального планирования инструкций для процессоров семейства IA-64 на примере процессора Intel Itanium 2. Для планирования используется известный эвристический алгоритм A^*, ранее не применявшийся к подобной задаче. В статье предлагаются эвристики, позволяющие оценить длину оптимального плана инструкций и, следовательно, сократить время работы алгоритма. Производится сравнение результатов планирования с результатами, получаемыми некоторыми промышленными компиляторами.

Введение

Одной из важнейших и оказывающих большое влияние на производительность целевого кода стадий компилятора является стадия планирования инструкций. При выполнении планирования 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 процессора и исключения как можно большего числа циклов их простоя. В результате работы планировщика получается план инструкций.

Данное исследование выполнено при поддержке лаборатории системного программирования и информационных технологий СПбГУ (СПРИНТ), созданной при содействии компании Intel.
© С. Е. Галанов, 2008

Задача планирования актуальна 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 [12]. 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 области, 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 связи с этим интересен алгоритм A^* [4], который является обобщением классического алгоритма Дейкстры для поиска оптимального пути в графе.

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, что существенно усложняет работу планировщика.

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.

1. Некоторые понятия

Сначала определим некоторые важные понятия, используемые в дальнейшем.

Направленным графом называется тройка $(V, E, \delta), \delta\colon E \to V \times V$, 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 [11] 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) [11] называется простой путь в графе потока управления. Этот путь может иметь произвольное число *точек входа* и *точек выхода* (то есть инструкций, имеющих непосредственного предшественника и непосредственного последователя соответственно, не принадлежащего пути).

Базовым блоком [11] называется след, имеющий единственную точку входа и единственную точку выхода.

Суперблоком [3] 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* [9] называется суперблок, содержащий инструкции с предикатами.

Планом инструкций (расписанием) [11] 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*.

Под *слотом* будем понимать место в группе инструкций для указания каждой отдельной инструкции.

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

Ограничения по ресурсам возникают из-за конечности 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. Обзор существующих подходов

В данном разделе будут рассмотрены основные подходы к эффективному планированию инструкций.

2.1. Списковое планирование

Списковое планирование (list scheduling) [11] 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), которая 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).

2.2. Целочисленное линейное программирование

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

Задача поиска оптимального плана сводится к задаче целочисленного линейного программирования, то есть к поиску целочисленного решения некоторой системы линейных неравенств, доставляющего минимум некоторой линейной функции.

Ограничения по данным и ресурсам (а также прочие виды ограничений) моделируются соответствующими неравенствами.

В работе [15], благодаря большому числу изощренных методов сокращения пространства перебора, авторам удалось с успехом применить рассматриваемый подход к задаче оптимального планирования для однопусковой машины с максимальной латентностью 3. Пакет из 7402 блоков (среди которых есть блоки с длиной около тысячи инструкций) обрабатывается за 98 секунд.

В работе [8] 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 (*макропланирование*) 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 см. раздел).

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 скромны. Блоки размером до 50 инструкций обрабатываются в среднем за время, не превышающее 10 секунд.

2.3. 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 пе-

ременной 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 [10] программирование ограничений применяется к планированию инструкций для многопускового процессора. Каждой инструкции сопоставляется переменная со значением, равным номеру такта, в котором будет запущена эта инструкция. Ограничения соответствуют зависимостям по данным и ограничениям по ресурсам. Кроме того, добавляется ряд дополнительных ограничений, позволяющих сократить время решения задачи.

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 закончив лишь несколько из них.

2.4. Метод ветвей и границ

Метод ветвей и границ (branch and bound) [13] 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 [2]. Однако время работы этого алгоритма оказывается слишком велико. Поэтому авторы также рассматривают эвристики, позволяющие существенно сократить пространство перебора, но не сохраняющие оптимальность.

В работе [14] метод ветвей и границ с успехом применяется к оптимальному локальному (на базовом блоке) и глобальному (на суперблоке и следе) планированию инструкций для многопускового процессора.

3. Архитектура IA-64

В качестве целевой машины в данной работе используется процессор Intel Itanium 2 — представитель архитектуры IA-64 [6, 7].

Рассмотрим особенности этой архитектуры.

IA-64 является пока единственной реализацией подхода EPIC (Explicitly Parallel Instruction Computing). EPIC [5] 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. Другие важные проблемы, которые решает EPIC, — это 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. Кроме того, 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 четыре порта типа M, два I-порта, два F-порта, три B-порта и два 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.

4. Постановка задачи

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:

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

$$instr_i \neq nop \Rightarrow instr_i \neq instr_j, 1 \leqslant i, j \leqslant m, i \neq j. \quad (2)$$

Иными словами, в группу входят инструкции исходного блока и пустые операции, причем инструкции, отличные от пустых операций, не могут повторяться.

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:

$$\forall e \in DAG.E\ src(e), dst(e) \in Scheduled(P) \Rightarrow \\ cycle(dst(e)) \geqslant cycle(src(e)) + latency(e); \quad (3)$$

$$\forall instr \in Scheduled(P)\ port(instr) \in InstrPorts(instr); \quad (4)$$

$$\forall b \in bundles(P)\ template(b) \in Templates; \quad (5)$$

$$\forall G \in P |\{bundleNo(instr) \mid belongs(instr, G)\}| \leqslant 2. \quad (6)$$

Условие (3) выражает зависимости по данным, условие (4) — ограничения по ресурсам, условие (5) — ограничения по кодированию: каждый пакет должен иметь правильный шаблон. Условие (6) 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.

5. Использование алгоритма A^* для планирования

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^*. Кратко опишем его.

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 $finish$. При достижении целевой вершины работа алгоритма завершается.

Основной результат [4] заключается в том, что алгоритм всегда находит оптимальный путь, если на эвристику 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. Если же эвристика достаточно близка к настоящей длине кратчайшего пути, алгоритм A^* завершается очень быстро.

Для представления приоритетной очереди в данной работе выбрана реализация на основе биномиальной пирамиды. Для хранения промежуточных планов используется достаточно компактное представление, подробно описанное в [1].

5.1. 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.

Прежде чем рассматривать эвристики, докажем один простой вспомогательный факт. В дальнейшем будем исходить из частичного плана P, с порожденным дэгом $PDAG$. Пусть OP — соответствующий ему полный план (то есть полный план, достижимый из P в графе поиска).

Лемма 1. *Пусть имеется набор множеств неспланированных инструкций I_k. Обозначим через $N(I_k)$ номер такта в OP, в котором запущена последняя инструкция из I_k: $N(I_k) = \max\{cycle(i, OP)\}_{i \in I_k}$. Пусть для каждого $N(I_k)$ имеется оценка снизу $M(I_k)$ по всем возможным полным планам OP: $\forall OP : M(I_k) \leqslant N(I_k)$. Тогда верно, что величина $H = \max_k\{M(I_k)\} - len(P)$ удовлетворяет условию непереоценки, то есть является корректной эвристикой для алгоритма.*

Доказательство. Пусть p — путь в графе поиска из P в OP. Требуется доказать, что $H \leqslant len(p) = len(OP) - len(P)$. Но из определения $N(I_k)$ и $M(I_k)$ видно, что $\forall k : len(OP) \geqslant N(I_k) \geqslant M(I_k)$. Следовательно, $len(OP) \geqslant \max_k\{M(I_k)\}$.

Таким образом, $len(OP) - len(P) \geqslant \max_k\{M(I_k)\} - len(P) = H$. □

5.1.1. Эвристика критического пути

Рассмотрим какой-нибудь критический путь $cp = e_1, ..., e_n$ в дэге $PDAG$. Пусть $v_1, ..., v_{n+1}$ — последовательность вершин, лежащих на этом пути.

Ясно, что $N(\{v_1\}) \geqslant len(P) = M(\{v_1\})$. Далее, из определения латентности следует, что $N(\{v_{i+1}\}) \geqslant N(\{v_i\}) + latency(e_i)$. Отсюда по индукции получаем, что $N(\{v_n\}) \geqslant len(P) + len(cp) = M(\{v_n\})$. Таким образом, по лемме, величина $H_{cp} = len(cp)$ удовлетворяет условию непереоценки и является корректной эвристикой. Будем называть ее *эвристикой критического пути*.

В качестве примера рассмотрим блок инструкций, изображенный на рис. 1 (рядом приведен дэг зависимостей для него).

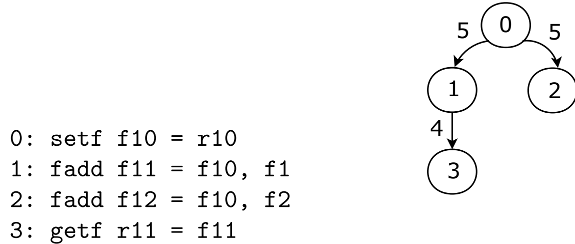


Рис. 1. Блок с «длинным» дэгом.

Длина критического пути в этом дэге равна 9. Следовательно, длину оптимального плана можно оценить числом 10 (поскольку длина пустого плана равна 1). Это значение совпадает с истинной длиной оптимального плана для данного блока (он приведен на рис. 2).

5.1.2. 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

```
{ mmi.
  setf f10 = r10 // => M2
  nop.m
  nop.i
  ;;
} // cycle 1
{ mfi
  nop.m
  fadd f11 = f10, f1 // => F0
  nop.i
}
{ mfi.
  nop.m
  fadd f12 = f10, f1 // => F1
  nop.i
  ;;
} // cycle 6
{ mmi.
  getf r11 = f11 // => M0
  nop.m
  nop.i
  ;;
} // cycle 10
```

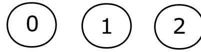
Рис. 2. Оптимальный план.

классу, которые могут быть запущены параллельно в одном такте. Обозначим это количество для класса k через M_k. Множество всех классов обозначим $InstrClasses$.

Рассмотрим произвольное множество I неспланированных инструкций. Пусть I_k — множество инструкций из I, принадлежащих классу k. Минимальное число тактов, необходимое для размещения I_k, равно $\lceil |I_k|/M_k \rceil$. Обозначим величину $\max_k \lceil |I_k|/M_k \rceil - 1$ через $H_r(I)$.

Возьмем теперь I равным всему множеству неспланированных инструкций. Ясно, что $N(I) \geqslant len(P) + H_r(I)$. Таким образом, из леммы получается, что значение $H_r = H_r(I)$ удовлетворяет условию непереоценки. Назовем его *эвристикой доступных ресурсов.*

Рассмотрим блок инструкций, изображенный на рис. 3.




```
0: fadd f11 = f10, f1
1: fadd f12 = f10, f2
1: fadd f13 = f10, f3
```

Рис. 3. Блок с «широким» дэгом.

Длина критического пути в этом дэге равна 0, что дает оценку длины оптимального плана, равную 1. Однако использование эвристики доступных ресурсов позволяет сделать более точную оценку. В этом блоке имеются инструкции одного класса (в который входит большинство инструкций с плавающей точкой). Значение M_k для этого класса равно 2. Таким образом, $H_r = \lceil 3/2 \rceil - 1 = 1$, что дает правильную оценку длины оптимального плана, равную 2 (рис. 4).

```
{ mfi
  nop.m
  fadd f11 = f10, f1 // => F0
  nop.i
}
{ mfi.
  nop.m
  fadd f12 = f10, f2 // => F1
  nop.i
  ;;
} // cycle 1
{ mfi.
  nop.m
  fadd f13 = f10, f3 // => F0
  nop.i
  ;;
} // cycle 2
```

Рис. 4. Оптимальный план.

5.1.3. Гибридная эвристика

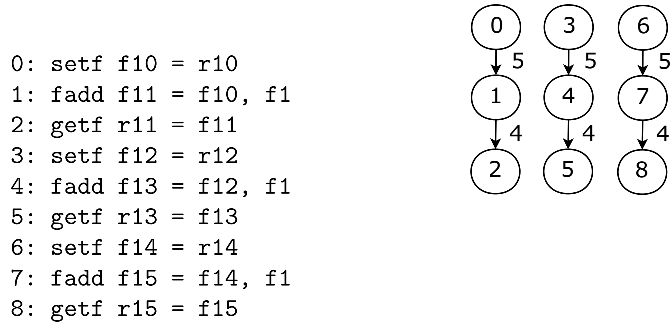
Желательно найти эвристику, которая объединяет достоинства эвристик критического пути и доступных ресурсов, так как существуют дэги, для части которых эффективна первая эвристика, а для другой части — вторая. Приведем здесь такую эвристику.

Очевидно, что для каждой неспланированной инструкции i $N\{(i\}) \leqslant mc_i = len(P) + cpl(PDAG, i)$, где $cpl(PDAG, i)$ — длина самого длинного пути в порожденном дэге для частичного плана P от корней до инструкции i.

Рассмотрим произвольное число $mc \in [1 : cpl(PDAG) + 1]$. Обозначим через R_{mc} множество неспланированных инструкций i, для которых $mc_i \geqslant mc$. Из рассуждений, аналогичных проведенным в предыдущем разделе, следует, что $N(R_{mc}) \geqslant mc + H_r(R_{mc}) = M(R_{mc})$. Отсюда с помощью леммы получаем, что величина $H_g = \max_{mc}(mc + H_r(R_{mc})) - len(P)$ является корректной эвристикой. Назовем эту величину *гибридной эвристикой*.

Эта эвристика не хуже эвристик критического пути и доступных ресурсов. Действительно, при $mc = cpl(PDAG)$ множество R_{mc} пусто, и значение $M(R_{mc}) - len(P)$ совпадает с эвристикой критического пути. Если же взять $mc = 1$, получим, что R_{mc} совпадает со всем множеством неспланированных инструкций, и $M(R_{mc}) - len(P)$ становится равным значению эвристики доступных ресурсов.

Приведем пример блока (рис. 5), для которого гибридная эвристика оказывается лучше рассмотренных ранее. Для этого примера



```
0: setf f10 = r10
1: fadd f11 = f10, f1
2: getf r11 = f11
3: setf f12 = r12
4: fadd f13 = f12, f1
5: getf r13 = f13
6: setf f14 = r14
7: fadd f15 = f14, f1
8: getf r15 = f15
```

Рис. 5. Блок с «длинным» и «широким» дэгом.

i	mc_i
0	1
1	6
2	10
3	1
4	6
5	10
6	1
7	6
8	10

mc	R_{mc}	$H_r(R_{mc})$	$M(R_{mc})$	H_g
1	все	2	3	2
6	$\{1, 2, 4, 5, 7, 8\}$	2	8	7
10	$\{2, 5, 8\}$	2	12	11

Рис. 6. Расчет эвристики.

$H_{cpl} = 9$, а $H_r = 2$ (инструкция getf может исполняться только на M2, то есть только одна за такт). Построим таблицу значений M для различных значений mc (достаточно рассмотреть только значения 1, 6 и 10, потому что путей другой длины в дэге нет).

Таким образом, гибридная эвристика равна 11, что дает оценку длины оптимального плана, равную числу 12. Это значение совпадает с истинной длиной (рис. 7).

```
{ mmi.
0: setf f10 = r10 // => M2
3: setf f12 = r12 // => M3
   nop.i
   ;;
} // cycle 1
{ mmi.
6: setf f14 = r14 // => M2
   nop.m
   nop.i
   ;;
} // cycle 2
{ mfi
   nop.m
1: fadd f11 = f10, f1 // => F0
   nop.m
} // cycle 6
{ mfi.
   nop.m
4: fadd f13 = f12, f1 // => F1
```

```
   nop.m
   ;;
} // cycle 6
{ mfi.
   nop.m
7: fadd f15 = f14, f1 // => F0
   nop.m
   ;;
} // cycle 7
{ mmi.
2: getf r11 = f11 // => M2
   nop.m
   nop.i
   ;;
} // cycle 10
{ mmi.
5: getf r13 = f13 // => M2
   nop.m
   nop.i
   ;;
} // cycle 11
{ mmi.
8: getf r15 = f15 // => M2
   nop.m
   nop.i
   ;;
} // cycle 12
```

Рис. 7. Оптимальный план.

5.1.4. 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.

5.2. Оптимизации

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

5.2.1. Генерация пустых операций

Для каждого плана (кроме такого, в котором последняя группа заполнена) среди инструкций, которые могут быть запущены, 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: а) с последним пустым тактом, занимающим весь пакет, б) занимающим первую позицию пакета (с типом M) и в) 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.

5.2.2. 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, а соответствующие группы быть 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.

5.2.3. Выделение изоморфных поддэгов

Предположим, что дэг зависимостей содержит два изоморфных поддэга, то есть задано соответствие между вершинами 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 [14].

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.

6. Результаты

Данный планировщик был опробован на наборе тестов SPEC[1].

Ассемблерный код был получен с помощью компиляторов open64[2] (версия 4.0) и gcc[3] (версия 4.2.2), с выставленными максимальными уровнями оптимизации. Сначала для каждого блока строился дэг зависимостей. Далее вычислялась нижняя граница

[1]http://www.spec.org.
[2]http://www.open64.net.
[3]http://gcc.gnu.org.

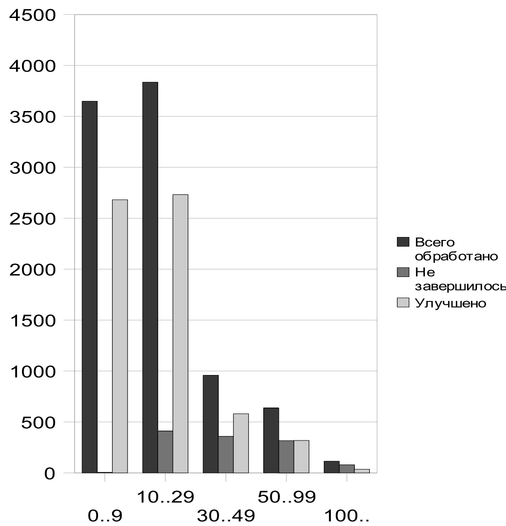


Рис. 8. Диаграмма результатов для open64.

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. Поскольку точный анализ зависимостей для низкоуровневого кода представляет значительные трудности (главным образом из-за недостаточности информации для анализа указателей), появлялись лишние зависимости между инструкциями, и поэтому для части блоков план был ухудшен. Для планировщика устанавливалось ограничение по времени в 5 секунд.

Результаты тестирования приведены в табл. 1.

В табл. 2 и 3 приведены более детальные результаты, распределенные по различным длинам блоков.

Заключение

В представляемой работе был разработан оптимальный планировщик инструкций для процессоров класса EPIC. Был предложен ряд эвристик и методов сокращения пространства перебора. Плани-

Таблица 1. **Результаты тестирования**

	open64	gcc
Количество блоков	92616	39909
Уже оптимально	83422 (90.1%)	33056 (82.8%)
Улучшено	6347 (6.9%)	5529 (13.8%)
Ухудшено	0	10 (0.03%)
Не завершилось	1169 (1.3%)	749 (1.9%)
Общее время работы	2 ч 17 мин	1 ч 38 мин

Таблица 2. **Результаты для open64**

Размер	0...9	10...29	30...49	50...99	100...
Количество обработанных блоков	3649	3836	958	636	115
Не завершилось	5 (0.14%)	410 (10.7%)	358 (37.4%)	317 (49.9%)	79 (68.7%)
Улучшено блоков	2681 (73.5%)	2732 (71.2%)	580 (60.6%)	318 (50%)	36 (31.3%)
Общее улучшение в тактах	2828	3643	1151	831	408

Таблица 3. **Результаты для gcc**

Размер	0...9	10...29	30...49	50...99	100...
Количество обработанных блоков	825	4169	1197	514	148
Не завершилось	3 (0.4%)	208 (5%)	253 (21.1%)	212 (41.2%)	73 (49.3%)
Улучшено блоков	705 (85.5%)	3526 (84.6%)	924 (77.2%)	300 (58.4%)	74 (50%)
Общее улучшение в тактах	753	4678	1977	1151	1073

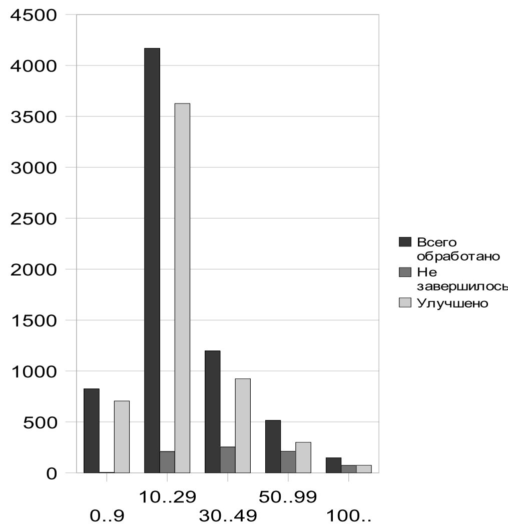


Рис. 9. Диаграмма результатов для gcc.

ровщик показал неплохие результаты на наборе тестов (хотя пока работает довольно плохо на очень больших блоках).

Список литературы

[1] *Галанов С. Е.* Оптимальное планирование инструкций для процессоров семейства IA-64 с использованием алгоритма A^*: Дипломная работа. СПбГУ, 2007. 34 с.

[2] *Haga S., Barua R.* EPIC Instruction Scheduling Based On Optimal Approaches // Workshop on EPIC Architectures and Compiler Technology. 2001.

[3] *Hank R. E., Mahlke S. A., Bringmann R. A.* Superblock formation using static program analysis // Proceedings of the 26th annual international symposium on Microarchitecture, Austin, Texas, United States, 1993. P. 247–255.

[4] *Hart P. E., Nilsson N. J., Raphael B.* A Formal Basis for the Heuristic Determination of Minimum Cost Paths // IEEE Transactions on Systems Science and Cybernetics SSC4 (2). 1968. P. 100–107.

[5] *Schlansker M. S., Ramakrishna B. R.* EPIC: An Architecture for Instruction-Level Parallel Processors // HPL Tech. Re-

port HPL-1999-111, Hewlett-Packard Laboratories, Jan. 2000. http://www.hpl.hp.com/techreports/1999/HPL-1999-111.pdf.

[6] Intel Itanium Architecture Software Developer's Manual. Volume 1, 2, 3, 2006.
http://www.intel.com/design/itanium/manuals/iiasdmanual.htm.

[7] Intel Itanium 2 Processor Reference Manual, 2004.
http://www.intel.com/design/itanium2/manuals/251110.htm.

[8] *Kastner D., Winkel S.* ILP-based Instruction Scheduling for IA-64 // Proceedings of the ACM SIGPLAN Workshop on Languages, Snowbird, Utah, USA, June 2001. P. 145–154.

[9] *Mahlke S. A., Lin D. C., Chen W. Y.* Effective compiler support for predicated execution using the hyperblock // Proceedings of the 25th annual international symposium on Microarchitecture, Portland, Oregon, United States, 1992. P. 45–54.

[10] *Malik A. M., McInnes J., Beek P.* Optimal Basic Block Instruction Scheduling for Multiple-Issue Processors using Constraint Programming // Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence, 2006. P. 279–287.

[11] *Muchnick S. S.* Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997. 856 p.

[12] *Garey M. R., Johnson D. S.* Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Co., San Francisco, CA, 1979. 338 p.

[13] *Sedgewick R.* Algorithms. Addison-Wesley Publishing Company, 1983. 551 p.

[14] *Shobaki G. O.* Optimal Global Instruction Scheduling Using Enumeration. PhD thesis, University of California Davis, 2006. 144 p.

[15] *Wilken K., Liu J., Heffernan M.* Optimal instruction scheduling using integer programming // PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, Vancouver, British Columbia, Canada, 2000. P. 121–133.