Санкт-Петербургский Государственный
Университет
Математико-механический факультет
Кафедра системного программирования

Локальное планирование инструкций с использованием профиля микроархитектуры

Дипломная работа студента 59 группы
Якушкина Сергея Игоревича

Научный руководитель:	.... (Булычев Д.Ю.)
Рецензент, к.ф.-м.н.:	.... (Фоминых Н.Ф.)
“Допустить к защите”, заведующий кафедрой, профессор,д.ф.-м.н.	.... (Терехов А.Н.)

Санкт-Петербург
2004

Содержание

1 Введение

В настоящее время бурно развивается область кодизайна, или совместной программно-аппаратной разработки (HSC, Hardware/Software Codesign). Появление данной области относят к началу 90-х годов, когда термин «кодизайн» был введен в обиход для описания совокупности проблем проектирования интегральных схем [5]. В это время дело шло к появлению кристаллов, которые могли вместить и центральный процессор, и другие требуемые подсистемы. Одни из первых программных систем, поддерживающих кодизайн, были разработаны в Стэндфордском университете и техническом университете города Брауншвейг. Эти системы предназначались для поддержки проектирования микропроцессорных систем, включающих, помимо центрального процессора, дополнительную интегральную схему (например, плату MPEG-кодирования), ориентированную на приложения (ASIC, Application-Specific Integrated Circuit). Ближе к концу 90-х годов направление кодизайна приобрело зрелость. Был разработан ряд важных алгоритмов, появились программные системы, поддерживающие разработку сложных микропроцессорных архитектур с произвольной топологией внутренних соединений и произвольным комбинированием центральных процессоров и ASIC. В настоящее время кодизайн становится основным подходом к созданию различных «систем на кристалле» (SoC, System on Chip), в особенности тех, которые базируются на вентильных матрицах с эксплуатационным программированием (FPGA, Field-Programmable Gate Array). Использование методов кодизайна позволяет заметно сократить время выхода на рынок, а также уменьшить сложность создания и цену интегральных схем.

Однако, в настоящее время существует ряд открытых проблем, решение которых требуется для дальнейшего развития подхода кодизайна. Одной из них является создание настраиваемых планировщиков.

Во многих современных архитектурах (VLIW[1], SuperScalar RISC[2], EPIC[3]) поддерживается одновременное исполнение нескольких инструкций за один такт, что позволяет увеличить производительность системы, выполняя за определенный интервал времени большее количество инструкций. Тем не менее, для определенных последовательностей ин-

[1]Very Large Instruction Word — архитектура с командными словами сверхбольшой длины

[2]Super Scalar Reduced Instruction Set Computer — скалярная архитектура, расширенная динамическим возможностями

[3]Explicitly Parallel Instruction Computing — эволюция архитектуры VLIW, которая абсорбировала в себе многие концепции суперскалярной архитектуры.

струкций время работы может увеличиться из-за задержек, вызванных конфликтами по ресурсам, т.е. отсутствием достаточного количества функциональных блоков в определенные моменты. В связи с этим компиляторы должны выполнять планирование инструкций. Таким образом, при генерации кода необходимо в значительной степени учитывать характерные особенности платформы, выполняя планирование инструкций и ресурсов под конкретную целевую архитектуру для уменьшения числа задержек.

Планировщик, зная количество функциональных единиц каждого вида, на каких из них и за сколько тактов выполняется та или иная команда, переупорядочивает действия таким образом, чтобы обеспечить оптимальную загрузку аппаратных средств. При переупорядочении, естественно, учитываются зависимости по данным.

Исторически планировщики в компиляторах создаются под каждый конкретный процессор/архитектуру. Однако, с появлением процессоров цифровой обработки сигналов (DSP, Digital Signal Processing) и процессоров, специализированных под конкретное приложение (ASIP, Application-Specific Instruction Set Processor), стало невозможно многолетнее научное исследование особенностей каждой архитектуры и разработка для нее специализированных методов планирования.

Решением обсуждаемой проблемы становятся настраиваемые планировщики, которые получают на вход описание целевого устройства на языке описания архитектуры (ADL, Architecture Description Language) и программу на целевом ассемблере. Результатом их работы является программа на целевом ассемблере, оптимальная по времени исполнения на целевой машине.

Отметим, что эффект от планирования инструкций особенно велик для DSP-процессоров. Это связано с тем, что алгоритмы цифровой обработки являются вычислительно-интенсивными, поэтому в соответствующих программах присутствуют выполняющиеся миллионы раз линейные участки кода со сложными вычислениями. Отсюда относительно небольшое повышение скорости работы на таких линейных участках дает значительный в абсолютном выражении выигрыш по времени.

Из сказанного выше ясно, что создание настраиваемых планировщиков является важной и актуальной проблемой.

Цель данной работы — создание параметризованного макроархитектурой средства для выполнения локального, управляемого профилем микроархитектуры планирования инструкций. Для достижения поставленной цели необходимо разработать формализм для описания профиля микроархитектуры и параметризуемый им настраиваемый планировщик.

В качестве целевой архитектуры выбран многоконвейерный вычислитель, алгоритмы планирования для которого изучены недостаточно. Предлагаемый подход позволяет избежать чрезмерно большой вычислительной сложности алгоритмов планирования, обычно возникающей для многоконвейерных архитектур, за счет применения вероятностного планирования.

Сформулируем основные требования к планировщику:

- планирование по двоичному коду программы;
- двойная параметризация: макроархитектурой (параметризует дизассемблер) и профилем микроархитектуры (параметризует модуль планирования);
- обеспечение возможности прямолинейной автоматической генерации профиля микроархитектуры при VHDL-синтезе микроархитектуры;
- поддержка целевых устройств, имеющих несколько конвейеров;
- возможность автоматического порождения планировщика по макроархитектуре и профилю микроархитектуры целевого устройства.

Выполнение этих требований позволит обеспечить широкий круг используемых архитектур и избавит пользователей от необходимости переписывать архитектуру при изменении микроуровня реализации системы команд. В рамках используемой технологии это приведет к возможности получения оптимизированного исполняемого кода для порожденной (в виде VHDL-модели) микроархитектуры целевого устройства.

2 Обзор существующих подходов

Локальное планирование можно рассматривать как задачу поиска наилучшего допустимого плана[4]. В общем случае локальное планирование инструкций (local instruction scheduling) является NP-трудной задачей [18].

Наиболее распространенным решением задачи эффективного планирования инструкций для конвейерных архитектур является алгоритм спискового планирования (list scheduling), предложенный еще в 1986 г. в работе [14]. Рассматриваются два варианта алгоритма: планирование по тактам (instruction-driven), планирование по инструкциям (operation-driven). Последний вариант более прост в реализации и чаще всего дает результат несколько лучше. Сложность метода $O(n^2)$, где n — число инструкций. Экспериментальный анализ области применимости спискового планирования и некоторых его модификаций приведен в работе [19].

Основой эффективного алгоритма спискового планирования являются эвристики и правильный выбор между ними, что представляет собой машинно-зависимую часть алгоритма. В работе [27] предлагается использовать контролируемое обучение планировщика с целью определения весов эвристик для целевой архитектуры. Утверждается, что данный подход дает превосходные результаты на практике. В [26, 28] экспериментально изучены различные методы обучения.

Перечисленные выше методы основаны на графовых моделях (graph-based) и используют эвристики. Альтернативным подходом является применение целочисленного линейного программирования (ILP, Integer Linear Programming). Возможно как точное решение проблем планирования и распределения регистров, так и аппроксимация [25, 17]. Наиболее известны две формулировки задач кодогенерации с помощью целочисленного линейного программирования: планирование и распределение ресурсов с использованием целочисленного линейного программирования (SILP, Scheduling and Allocation with Integer Linear Programming) [31] и оптимальный синтез архитектуры с ограничениями на интерфейс (OASIC, Optimal Architectural Synthesis with Interface Constraints) [20, 21]. В работе [22] приведено экспериментальное сравнение этих подходов.

В недавних работах [24, 15] изучается интеграция планирования инструкций и других стадий кодогенерации: выбора инструкций (instruction selection) и распределения ресурсов (resource allocation). Но,

[4]Допустимый план — последовательность инструкций, не нарушающая частичного порядка, заданного графом зависимостей инструкций по данным.

несмотря на ведущееся изучение интегрированных подходов, отдельные подзадачи кодогенерации изучены недостаточно, что связано с постоянным появлением новых архитектур с нестандартными возможностями, например, с внеочередным выполнением инструкций (out-of-order execution). Развитие архитектур приводит к устареванию базовых моделей и формализаций и необходимости их расширения.

Следует также отметить, что рост производительности компьютеров позволяет использовать недоступные ранее алгоритмы. В работе [30] предлагается вероятностный метод поиска планов (RBF, Randomized Backward and Forward scheduling), который оказалось возможно расширить на случай многоконвейерных архитектур. В следующей главе данный метод будет описан более подробно.

Отметим, что настраиваемый планировщик работает по описанию архитектуры целевого процессора и описанию его системы команд, поэтому необходим специальный язык описания структуры процессора (набора функциональных устройств, стадий конвейеров, ресурсов) и системы команд (времени работы инструкций и устройств, которые они занимают). На настоящий момент стандартов в этой области нет — почти во всех программных средствах используются авторские языки описания архитектуры (ADL, Architecture Description Language). Сравнительный анализ языков дается в [29], где их подразделяют на три группы: структурные, поведенческие и смешанные. Одним из первых широко известных языков описания архитектуры является MARIL [8], а из последних разработок следует упомянуть ISDL [11], Expression [10, 12], TDL [23], PADLA [2, 1], AVIV [13].

3 Локальное планирование

В данном разделе описываются алгоритмы, а также модель настраиваемого планировщика, удовлетворяющая сформулированным требованиям.

Планирование команд — преобразование ассемблерной программы с целью более эффективного использования внутреннего параллелизма процессора. Заметим, что такой параллелизм появляется только при генерации микроархитектуры, поэтому хорошо спланированная программа не может появиться без принятия во внимание ее свойств. Построение хорошего планировщика по информации, извлекаемой в момент генерации микроархитектуры, является открытой задачей. Далее будет описан предлагаемый подход к ее решению.

3.1 Технология прототипирования PADLA

Под *прототипом* встроенной системы будем понимать совокупность исполняемого машинного кода и модели вычислительного устройства при условии, что данная модель исполняет этот код корректно, но, возможно, неэффективно. Кроме того, будем требовать, чтобы переход от модели устройства к его аппаратной реализации осуществлялся автоматически. Таким образом, прототип превращается в готовую систему тогда, когда его эффективность удовлетворяет заданным характеристикам задачи.

В рамках технологии предлагается способ автоматизированного получения прототипа, основанный на извлечении макроархитектуры аппаратной платформы из реализации на языке высокого уровня. Такой подход позволяет исключить стадию настройки средств разработки на целевую платформу, поскольку система команд и код реализации для нее порождаются одновременно. Кроме того, помимо собственно прототипирования рассматриваются задачи оптимизации, решение которых может быть автоматизировано в рамках данного подхода.

Под *макроархитектурой* будем понимать архитектуру процессора с точки зрения компилятора с языка высокого уровня. Соответственно, *микроархитектура* есть реализация макроархитектуры на языке описания оборудования. Здесь уместна аналогия между данными уровнями спецификации устройства и реализацией языков программирования: в то время как макроархитектура аналогична языку, микроархитектура соответствует его интерпретатору. В такой ситуации язык описания оборудования можно определить как язык реализации этого интерпретатора.

Граница между макро и микроархитектурой, таким образом, может перемещаться между двумя крайними случаями, одним из которых является ситуация, когда целевая программа написана на языке описания оборудования (и, следовательно, макроархитектура совпадает с микроархитектурой), а другим, — когда процессор является интерпретатором исходного языка (например, аппаратная Java-машина). Следовательно в каждом конкретном случае необходимо заранее договориться, где именно проходит эта граница. Как было сказано выше, будем считать, что исходная программа реализована на императивном языке высокого уровня (например, C). В такой ситуации на уровень макроархитектуры устройства попадают следующие элементы:

- набор программно доступных ресурсов (регистров, памятей) и их свойства (разрядность, количество экземпляров, принцип агрегирования и т.д.);
- режимы адресации;
- набор команд, их семантика, правила порождения двоичного кода и ассемблерного представления.

Это обусловлено тем, что только вышеперечисленные свойства существенны для генерации кода с языка высокого уровня. Более тонкие детали реализации процессора (например, конвейеризация и параллелизм на уровне инструкций) не только не оказывают существенного влияния на порождение кода[5], но и вообще могут быть еще не известны на момент кодогенерации в силу того, что финальные характеристики устройства пока не ясны.

Схема получения прототипа системы приведена на рис. 1. Рассмотрим основные этапы подхода более подробно.

Прежде всего, по тексту реализации системы на языке высокого уровня порождается описание макроархитектуры процессора и ассемблерный код (см. рис. 1, A). Этот процесс является критическим для описываемого подхода, поскольку порождение “неудачной” архитектуры или неэффективного кода является недопустимым. Вместе с тем существуют весомые аргументы в пользу того, что данная задача может быть решена более успешно, чем задача автоматической настройки компилятора на целевую платформу.

[5] Точнее, существует простой конструктивный критерий качества выбора инструкций и распределения регистров с точки зрения планирования: порожденный код не должен содержать антизависимостей [16].

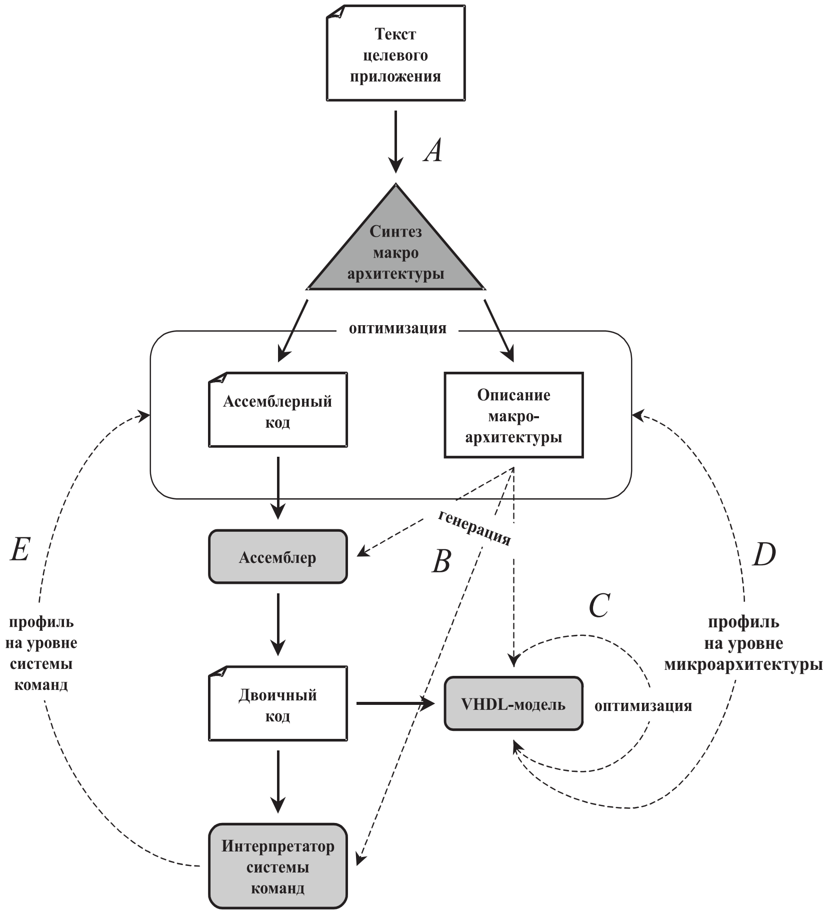

Рис. 1: Схема технологии

Действительно, основной сложностью при настройке компилятора на платформу для встроенного приложения является использование расширенных возможностей этой платформы — нестандартных ресурсов, режимов адресации и команд. Эти возможности вводятся разработчиками оборудования с целью достижения большей производительностью. Но поскольку окончательный набор программ, работающих на данной платформе, ее разработчику недоступен, правильное их использование в каждом конкретном случае может превращаться в нетривиальную задачу. С другой стороны, даже небольшой анализ текста реализации позволяет определить, какие конкретно расширенные возможности целевой платформы необходимы для синтеза эффективного кода. Подобного рода анализ, проводимый с целью выбора целевого процессора среди множества уже готовых, является обычным делом. Таким образом, сложность задачи настройки компилятора происходит не из-за сложности приложения, а из-за несогласованности расширенных возможностей универсальной платформы с его потребностями.

3.2 Модель целевого устройства и профиль микроархитектуры

В данном разделе рассматривается общая модель целевой архитектуры, в рамках которой разрабатывается микроархитектурная параметризация алгоритма планирования.

Код для суперскалярных процессоров содержит последовательность команд, которая не определяет конечный временной порядок, в котором будут выполняться команды. Конкретный порядок исполнения задается процессором динамически.

В отличие от программ для суперскалярных процессоров, код VLIW представляет собой точный план того, как процессор будет выполнять программу — план исполнения, который компилятор создает статически во время компиляции. Код точно указывает, когда будет выполнена каждая операция, какие функциональные устройства будут работать и какие регистры будут содержать операнды. Компилятор VLIW создает такой план выполнения (POE, plan of execution), имея полное представление о процессоре, причем создает этот план так, чтобы добиться требуемой записи выполнения (ROE, record of execution) —- последовательности событий, которые действительно происходят во время работы программы.

В нашей модели целевое устройство имеет многоконвейерный тип архитектуры. Целевое устройство состоит из N разных конвейеров, работающих параллельно. Каждый такт происходит выборка из памяти N новых инструкций. Конвейер устроен следующим образом:

- каждый уровень конвейера — это один такт выполнения команды;
- количество уровней конвейера равно длине самой "длинной" команды;
- на каждом уровне конвейера инструкция занимает определенные микроресурсы целевого устройства;
- команда не может стоять на одном уровне конвейера больше одного такта;
- остановка конвейера на один такт производится командой NOP, состоящей из одного пустого такта; можно остановить конвейер на несколько тактов, подав NOP несколько раз;

- команды на N конвейеров распределяются следующим образом: i-я команда — на i-ый конвейер;

- каждая команда i-ого конвейера инкрементирует счетчик команд так, чтобы он указывал на следующую за ней команду $i+1$-ого конвейера.

Планировщик использует информацию о микроархитектуре, а именно профиль микроархитектуры. Под *профилем макроархитектуры или микропрофилем* будем понимать характеристики данной архитектуры процессора с точки зрения параллелизма на уровне инструкций. Он включает:

- описание доступного набора микроресурсов;

- описание процесса исполнения инструкций на каждом конвейере по стадиям.

Структура профиля микроархитектуры задается на языке Objective Caml [7] следующим образом:

```
// описание микроресурса
type
    micro_resource = {
        // название
        name : string;
        // тип микроресурса
        mr_type : micro_type;
    }

// описание набора доступных микроресурсов
// целевого устройства и количество каждого микроресурса
type
    micro_resources = (int * micro_resource) array

// описание набора занимаемых
// инструкцией микроресурсов
type
    micro_resource_descriptor =
          // либо список микроресурсов с указанием их количества
        | Indexed of (micro_resource * int) list
          // либо один экземпляр определенного микроресурса
        | Any_of  of  micro_resource
```

```
        // либо некоторое количество определенного микроресурса
        | Some_of of  micro_resource * int

// описание набора ресурсов макроархитектуры,
// читаемых или записываемых инструкцией
type
    resource_descriptor =
        // список ресурсов с указанием экземпляров;
        // Il.Insn.Expr.t - стандартный тип внутреннего
        //                  представления макроархитектуры,
        //                  описывающий обращение к ресурсу
        Il.Insn.Expr.t list

// описание микроинструкции
type
    micro_command = {
        // описание инструкции
        // Il.Insn.t - стандартный тип внутреннего
        //             представления макроархитектуры,
        //             описывающий инструкцию
        insn : Il.Insn.t;
        // читаемые ресурсы
        read : resource_descriptor;
        // записываемые ресурсы
        write : resource_descriptor;
        // занимаемые микроресурсы
        inner : micro_resource_descriptor;
    }

// описание стадии конвейера
// (набор исполняемых микроинструкций)
type
    stage = micro_command list

// описание конвейера (набор стадий)
type
    pipeline = stage array

// описание профиля (набор конвейеров)
type
    micro_profile = pipeline array
```

Из приведенного примера видно, что микропрофиль состоит из описания микроресурсов и описания конвейеров. Описание микроресурсов — массив записей с указанием типа микроресурса и количества микроресурсов данного типа. Конвейеры задаются по стадиям, число которых может быть различно. Стадия задается набором исполняющихся на ней микроинструкций. Основным элементом микропрофиля являются описания микроинструкций. Они состоят из

- ссылки на описание инструкции в макроархитектуре;
- списка читаемых в данном такте ресурсов;
- списка записываемых в данном такте ресурсов;
- множества занимаемых в данном такте микроресурсов.

Записываемые/читаемые ресурсы используются для построения графа зависимостей. Они представлены в виде выражений, зависящих от параметров инструкции. Например, следующему действию

$$r(i) \leftarrow r(i + k * 2) + r(j)$$

соответствуют чтения $r(i + k * 2)$, $r(j)$ и запись $r(i)$.

Информация по чтению/записи может быть получена из анализа семантики инструкции в макроархитектуре. Это делается на стадии синтеза микроархитектуры.

3.3 Алгоритм построения графа зависимостей

При планировании выполняются только допустимые перестановки инструкций, т.е. перестановки, не нарушающие семантику программы. Для их нахождения строится ориентированный ациклический граф зависимостей по данным (DDD, Data Dependence Directed Acyclic Graph). В вершинах располагаются инструкции, а наличие дуги между двумя вершинами означает зависимость по данным между ними. Следовательно, переставить можно только те инструкции, которые не связаны путем в этом графе.

В отличие от распространенных подходов, используемая технология не предусматривает разбиения инструкций на микрокоманды, поэтому построение графа зависимостей ведется по сложным многотактовым инструкциям, семантика которых описана на уровне регистровых передач (RTL, Register Transfer Level). Информацию о чтении и записи ресурсов необходимо извлекать отдельно, поскольку в описании семантики инструкции может идти доступ к любым ресурсам макроархитектуры. Конкретный экземпляр ресурса возможно определить только по целевой программе, так как он зависит от параметров инструкции. Приведем поясняющий пример инструкции (написан на PADLA):

```
instruction add_fib:
    addr:imm ->
    action {mem(addr), pc <- mem(addr-1)+mem(addr-2), pc+size}
    repr {"add_fib", "["addr"]"}
```

Инструкция вычисляет следующее число Фибоначчи и записывает значение в память по указанному адресу. Ее использование на целевом ассемблере будет выглядеть так:

```
    add_fib [100]
```

В примере `addr=100`, поэтому читаются ресурсы `mem(98)`, `mem(99)` и записывается `mem(100)`. По приведенному примеру видно, что без анализа семантики и конкретизации параметров инструкции невозможно определить конкретные экземпляры ресурсов, а значит и построить граф зависимостей. Иногда конкретный экземпляр ресурса возможно определить только в момент исполнения программы, например `mem(r1+r2)`, тогда будем считать, что изменяется каждое поле ресурса — `mem(?)`.

В нашем случае анализ семантики выполняется на этапе синтеза микроархитектуры и необходимая информация включается в профиль микроархитектуры (микроинструкция читает `micro_command.read` и пишет `micro_command.write`). Однако, для построения графа зависимостей необходимо описание именно инструкции, поэтому выполняется преобразование микропрофиля в форму, где каждой инструкции соответствует описание вариантов ее исполнения на разных конвейерах, собранное из микроинструкций.

Многотактовость инструкций приводит еще к одной особенности построения графа зависимостей. Дугам графа сопоставляются числа, показывающие минимальную задержку от начала первой инструкции до начала второй. Эта задержка может быть отрицательной. Такое допущение позволяет выполнять VLIW-планирование, помещая зависимые инструкции в одном такте.

Рассмотрим пример. Пусть инструкция `B` выполняется после `A`:

```
Инструкция A       Инструкция B
write   r1               -
    -               read    r1
```

Видно, что, несмотря на зависимость между `A` и `B` (`A` пишет `r1`, а `B` читает `r1`), минимальная задержка равна 0 и инструкции можно исполнить одновременно.

Приведем алгоритм построения графа зависимостей:

```
Reads = empty;
Writes = empty;

// последовательная обработка всех инструкций
// данного линейного участка
for (inst = 0; inst < instruction_count; inst++)
{
  // цикл по всем стадиям выбранной инструкции
  for (stage = 0; stage < stageCount(inst); stage++)
  {
    // выборка всех чтений ресурсов для
    // инструкцией inst на стадии stage
    Ireads = getReads(inst, stage);
    // обработка каждого ресурса из множества чтений
    for each rc in Ireads do
    {
      // добавить прямые зависимости (запись-чтение)

      // цикл по инструкциям из списка последних записей в ресурс rc
      for each j in getLast(Writes,rc) {
         // добавить в граф зависимость между инструкциями j и inst
         addDependency(dag, j, inst);
      }

      // Добавить в список Reads чтение
      // ресурса rc инструкцией inst
      Reads(rc) += inst;
    }
    // выборка всех записей ресурсов для
    // инструкцией inst на стадии stage
    Iwrites = getWrites(inst, stage);
    // обработка каждого ресурса из множества записей
    for each rc in Iwrites do
    {
      // Добавить анти-зависимости (чтение-запись)
```

```
      // цикл по инструкциям из списка последних чтений ресурса rc
      for each j in (getLast(Reads,rc)) do
         // добавить в граф зависимость между инструкциями j и inst
         addDependency(dag, j, inst);

      // Добавить выходные зависимости (чтение-чтение)

      // цикл по инструкциям из списка последних записей в ресурс rc
      for each j in (getLast(Writes,rc)) do
         // добавить в граф зависимость между инструкциями j и inst
         addDependency(dag, j, inst);

      Writes(rc) = inst;
      Reads(rc) = empty;
    }
  }
}
```

Для построения графа последовательно обходятся все инструкции по стадиям (в естественном порядке их исполнения). Отображения `Reads` и `Writes` задают, какие инструкции последними читали/писали каждый ресурс. На каждом шаге алгоритма найденные зависимости заносятся в граф, а множества обновляются.

3.4 Алгоритм вероятностного планирования

Алгоритм является основной планировщика — от его выбора зависит не только производительность, но и тип поддерживаемых архитектур. Поскольку планировщик должен быть интегрирован в существующую среду разработки встроенных систем с поддержкой сложных инструкций, к алгоритму естественным образом предъявляются следующие требования:

- параметризуемость на уровне профиля микроархитектуры;
- поддержка широкого круга архитектур (от VLIW до SuperScalar RISC);
- поддержка многотактовых команд;
- приемлемое время работы планировщика;

- обеспечение возможности поиска оптимального плана (вне зависимости от занимаемого времени).

В обзоре было показано, что поставленным требованиям в лучшей степени отвечает алгоритм вероятностного планирования (stochastic list scheduling), предложенный в статье [30]. Базовый алгоритм поддерживает достаточно ограниченную архитектурную модель. Далее предлагается модификация указанного метода, позволяющая выполнять планирование в случае наличия многотактовых инструкций и целевого устройства с несколькими конвейерами.

При планировании необходимо определить возникновение конфликтов по микроресурсам для нахождения места новой инструкции в плане исполнения. Для выполнения данной задачи строится отображение

$$F(r,t,p): \ R \times T \times P \ \rightarrow \ \mathbf{Z}$$

где R — микроресурсы, T — такты, P — конвейеры.

Отсутствие конфликтов формально определяется так:

$$\forall \, r \in R, t \in T, p \in P: \ F(r,t,p) \geq 0$$

Для определения отсутствия конфликтов по плану исполнения строится информация о занятых на каждом такте микроресурсах. При добавлении инструкции на определенное место (такт, конвейер) в план выполняется проверка наличия свободных микроресурсов, и, если она успешно прошла, информация о занятых микроресурсы обновляется.

Приведем алгоритм планирования:

```
/* Описание используемых функций:
 add_nop_to_each_pipe - добавляет на каждый конвейер по инструкции NOP.

 try_to_schedule - пытается запланировать инструкцию без конфликтов.
    Действует потактово, начиная с op.start_cycle. В каждом такте ищет
    NOP'ы на всех конвейерах, и последовательно пробует заменить их на op.
    Если место найдено, возвращает его Place = (pipe_num, cycle_num)

 schedule.set - заменить NOP, стоящий в плане на месте Place,
    инструкцию на op.

 new_ready_in_dag - помечает op как выполненную, и возвращает появившиеся
    готовые инструкции (с помеченными предшественниками)
```

```
 find_start_cycles - вычисляет начальный такт - максимум суммы
    (время предшественника в плане + задержка до инструкции op)

 stachastic_choose - Строит приоритетный список инструкций из DRS
    у которых (start_cycle <= cur_cycle). Если построенный список пуст,
    то возвращает empty. Иначе произвольно выбирает одну и возвращает ее.
*/

/*
  Вход: 1) DAG инструкций с относительными задержками между ними.
           Задержки могут быть < 0, они определяются зависимостями
           по данным между микроинструкциями.
        2) Описание ФИУ (функциональных исполнительных устройств),
           которые занимают инструкции в каждом такте.
        Также передается структура конвейера.
  Выход:
    1) План по тактам для каждого конвейера
*/

find_schedule_candidate() {
  DRS = {набор инструкций без предшествующих};
  Cur_cycle = 0;

  while (DRS != 0)
  {
    // добавить на каждый конвейер NOP
    add_nop_to_each_pipe();
    while ((op = stochastic_choose(DRS, Cur_cycle)) != empty)
    {
      Place = try_to_schedule(op);

      if (Place == eof) // если место в плане не найдено
      {
        /* обновить в DRS начальный такт операции op */
        op.start_cycle = Cur_cycle+1;
      } else {
        // занести op в план
        schedule.set(op, Place);
        // убрать из DRS
        DRS -= op;
        // Найти новые "готовые инструкции"
```

```
        Delta_DRS = new_ready_in_dag(op);
        // Вычислить начальные такты для них
        for each New_op in Delta_DRS
        {
          New_op.start_cycle = find_start_cycles(New_op);
        }
        // Обновить DRS
        DRS += Delta_DRS;
      }
    }
    Cur_cycle ++;
  }
}

/* Ищет лучший план в течении определенного времени */
find_schedule
{
  best_time = +infinity;
  while (not time_limit_exceed)
  {
    // найти нового кандидата
    candidate = find_schedule_candidate();

    if (candidate.time < best_time) {
      best_candidate = candidate;
      best_time = candidate.time;
    }
  }
  return best_candidate;
}
```

Планирование ведется в течении заданного промежутка времени. Многократно повторяется поиск плана методом спискового планирования с элементами случайности (в выборе инструкции из набора готовых и в выборе места в плане). По истечении времени выдается лучший из найденных кандидатов плана исполнения.

4 Реализация

Созданный настраиваемый локальный планировщик реализует предложенные алгоритмы и обладает некоторыми особенностями, которые будут описаны ниже.

В качестве окружения в данном проекте используется технология разработки программно-аппаратных систем на основе языка описания макроархитектуры (Processor Architecture Description Language) и соответствующий набор утилит. Указанная технология позволяет порождать набор средств разработки по формальному описанию системы команд процессора: ассемблер, дизассемблер [3], симулятор, а также VHDL-модель устройства, реализующего данную систему команд [4]. Планировщик, как и вся система, реализован на современном функциональном языке программирования Objective Caml, разработанном в INRIA (Национальный Институт Информатики и Автоматизации, Франция).

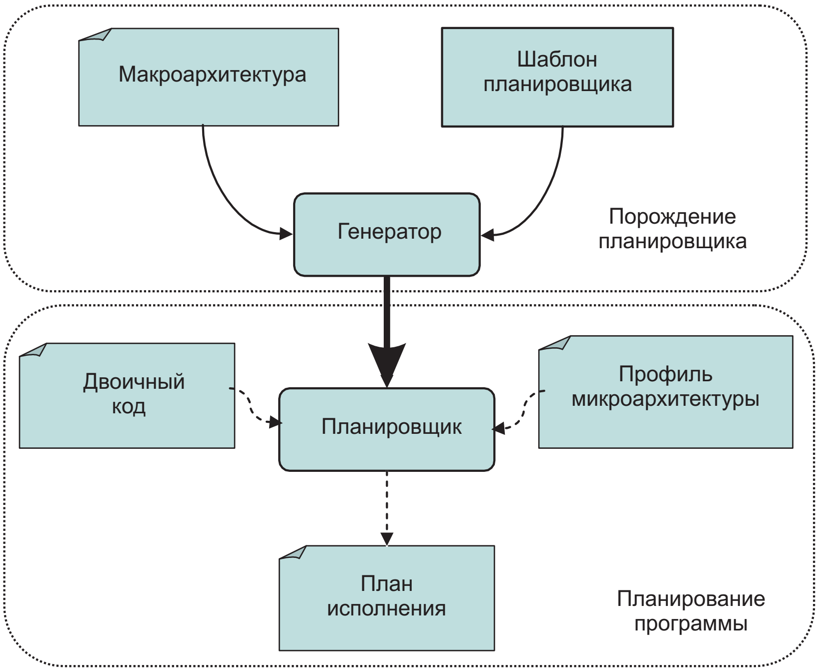

Рис. 2: Схема работы настраиваемого планировщика

Настраиваемый планировщик реализован в виде генератора планировщика по шаблону (см. рис. 2). Генератор анализирует макроархитектуру, выделяя из нее описания инструкций. Далее по шаблонам порождается текст планировщика на языке Objective Caml и планировщик компилируется.

Рис. 3: Схема планировщика

Планировщик получает на вход двоичный код и профиль микроархитектуры (см. рис. 3). Двоичный код дизассемблируется, а микропрофиль преобразуется в описание доступных ресурсов микроархитектуры и список микропрофилей инструкций. В микропрофиль входят:

- список чтений/записей по тактам (используется для построения графа зависимостей);
- список занимаемых микроресурсов по тактам (используется для определения конфликтов микроресурсов).

По дизассемблированному коду (запись исполнения) строится граф зависимостей с учетом списков чтений/записей. Далее выполняется вероятностное планирование. При добавлении новой инструкции в план исполнения производится проверка на отсутствие конфликтов по микроресурсам, и, если свободное место (такт, конвейер) в плане исполнения найдено, новая инструкция заносится в план. Когда план найден, он сравнивается с наилучшим кандидатом. Поиск новых кандидатов продолжается до истечения времени, после чего лучший из найденных кандидатов выводится.

Планировщик состоит из следующих основных модулей:

POE — отвечает за хранение и добавление новых инструкций в план исполнения;

DAGRC — строит граф зависимостей;

DRS — обрабатывает список готовых к планированию инструкций;

NRS — обрабатывает список не готовых к планированию инструкций;

MODEL — проверяет конфликты по микроресурсам, выполняет поиск свободного места в плане исполнения;

SCHDR — реализует алгоритм поиска плана исполнения;

PROCESS — выполняет преобразование профиля микроархитектуры.

Дополнительно планировщик рассчитывает вероятностные характеристики полученной последовательности планов: среднее число шагов в плане исполнения, максимальную длину плана, корень из дисперсии.

5 Результаты

Продемонстрируем работу планировщика на примерах. Зафиксируем программу и макроархитектуру, после чего покажем, как изменяется план исполнения программы в зависимости от выбранной микроархитектуры.

Рассмотрим следующую макроархитектуру. Система команд состоит из умножения (`mul`), сложения (`add`), загрузки из памяти (`ld`) и сохранения в память (st). Регистровый файл (`r`) размером 256 регистров по 4 байта, а память (`mem`) 16 Кб. Описание макроархитектуры на PADLA будет выглядеть так (правила кодирования опущены для краткости):

```
resource
  r(0..255)     : 32;
  mem(0..2^15) : 8 little;
  pc            : 16 ip (mem);
type
  imm:32;

instruction mul: r(i), r(j) ->
    action {r(i), pc <- (r(i)*r(j))[:32], pc+size}
    repr   {"mul", "r"i, "r"j}
instruction nop: ->
    action {pc <- pc+size}
    repr   {"nop"}
instruction add: r(i), r(j) ->
    action {r(i), pc <- r(i)+r(j), pc+size}
    repr   {"add", "r"i, "r"j}
instruction ld:  r(i), r(j) ->
    action {r(i), pc <- mem(r(j))[4], pc+size}
    repr   {"ld", "r"i, "[r"j"]"}
instruction st: r(i), r(j) ->
    action {mem(r(i))[4], pc <- r(j), pc+size}
    repr   {"st", "[r"i"]", "r"j}
```

В качестве целевой программы возьмем вычисление координат прямолинейно движущейся точки. Система уравнения записываются так:

$$\begin{cases} xPos = xPos + (xVel * t) \\ yPos = yPos + (yVel * t) \end{cases}$$

где

$(xPos, yPos)$ — координаты точки;

t — время движения;

$(xVel, yVel)$ — вектор скорости движения.

Программа для вычисления значений $xPos$, $yPos$ записывается на целевом ассемблере так:

```
Номер        Инструкция         Комментарий

  0          ld r0 [x_vel]      ; r0 = mem[x_vel]
  1          ld r1 [t]          ; r1 = mem[t]
  2          ld r2 [x_pos]      ; r2 = mem[x_pos]
  3          ld r3 [y_vel]      ; r3 = mem[y_vel]
  4          mul r0 r1          ; r0 *= r1
  5          add r0 r2          ; r0 += r2
  6          ld r2 [y_pos]      ; r2 = mem[y_pos]
  7          st [x_pos] r0      ; mem[x_pos] = r0
  8          mul r1 r3          ; r1 *= r3
  9          add r1 r2          ; r1 += r2
 10          st [y_pos] r1      ; mem[y_pos] = r1
```

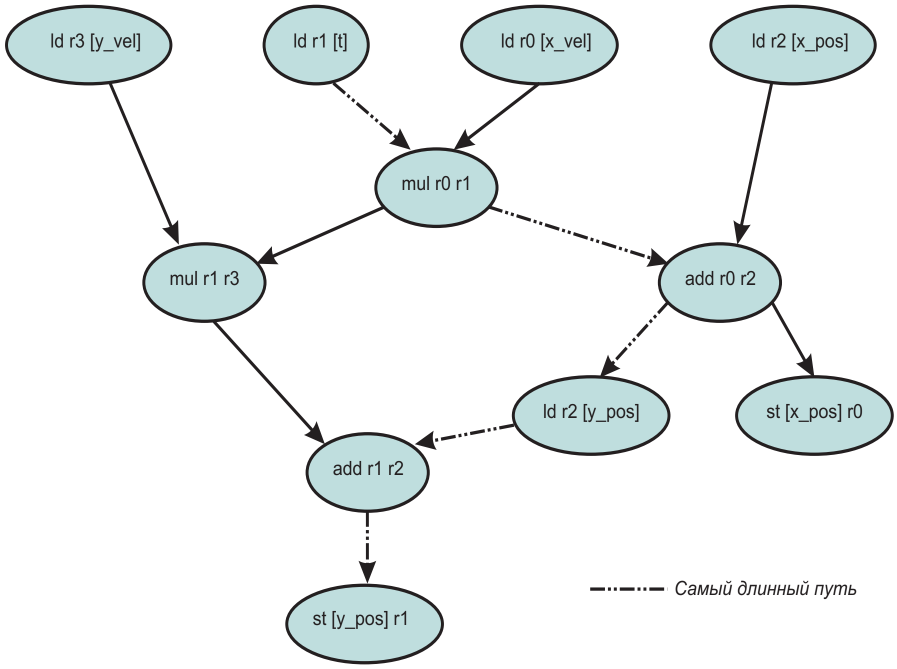

Рис. 4: Граф зависимостей программы

Если устройство не поддерживает параллелизма инструкций, то выполнение программы займет 11 тактов. Отметим, что длина плана исполнения не меньше глубины графа зависимостей (см. рис. 4), т.е. выполнение программы не может занять меньше 6 тактов.

5.1 Целевое устройство с одним конвейером

В случае конвейерной архитектуры увеличение производительности достигается разбиением инструкций на микрокоманды. Каждый такт на конвейер поступает одна новая инструкция, а текущие переходят на следующие стадии конвейера. Время исполнения инструкций доступа в память (`ld, st`) — 2 такта, остальные инструкции (`add, mul, mov`) работают 1 такт. Введем ограничение в нашу микроархитектуру: между инструкциями `ld` и `st` возникает задержка в 1 такт. Ограничение моделируется с помощью введения дополнительного микроресурса:

```
// описание дополнительного микроресурса,
// (далее мы на него ссылаемся как virtual_unit)
let micro_resources =
  [| ... ; (1, {name="VIRT_UNIT"; mr_type=1}); |] in
```

```
// добавление новых стадий инструкций
let micro_ld_stage1 =
  { ... ; inner=Any_of virtual_unit} in
let micro_st_stage0 =
  { ... ; inner=Any_of virtual_unit} in
```

После разрешения конфликтов микроресурсов получается следующий план:

```
Такты     Инструкции
  0:      ld r0 [x_vel]
  1:      ld r1 [t]
  2:      ld r2 [x_pos]
  3:      ld r3 [y_vel]
  4:      mul r0 r1
  5:      add r0 r2
  6:      ld r2 [r10]
  7:      nop
  8:      st [x_pos] r0
  9:      mul r1 r3
 10:      add r1 r2
 11:      st [r10] r1
 12:      nop
```

Задержка в 12 такте вызвана тем, что инструкция `st` выполняется 2 такта. Последнее действие в рассматриваемой программе — сохранение результатов вычислений в память, поэтому ее невозможно устранить. Однако возникла еще одна задержка в 7 такте. Попробуем устранить эту дополнительную задержку с помощью настраиваемого планировщика:

```
Такты     Инструкции
  0:      ld r3 [y_vel]
  1:      ld r0 [x_vel]
  2:      ld r1 [t]
  3:      ld r2 [x_pos]
  4:      mul r0 r1
  5:      add r0 r2
  6:      ld r2 [y_pos]
  7:      mul r1 r3
  8:      add r1 r2
  9:      st [x_pos] r0
```

```
10:      st [y_pos] r1
11:      nop
```

Найден оптимальный план исполнения (для данной архитектуры), его длина составляет 12 тактов. Длина исходного плана снижена на 7.7%. Данный план исполнения был найден на 11 итерации из 100 сделанных. Средняя длина планов составила 13.25, а корень из среднеквадратичного отклонения $\sigma = 0.68$.

5.2 Целевое устройство со скалярной архитектурой

Пусть целевое устройство состоит из двух одинаковых функциональных блоков, на каждом из которых может выполниться любая инструкция. Время исполнения инструкций доступа в память (`ld, st`) — 2 такта, остальные инструкции (`add, mul, mov`) работают 1 такт.

В результате работы настраиваемого планировщика был получен следующий план исполнения:

```
             Конвейер 0         Конвейер 1
  Такты
     0  |    ld r0 [x_vel]      ld r1 [t]
     1  |    ld r3 [y_vel]      ld r2 [x_pos]
     2  |    nop                mul r0 r1
     3  |    add r0 r2          ld r2 [y_pos]
     4  |    st [x_pos] r0      mul r1 r3
     5  |    add r1 r2          nop
     6  |    st [y_pos] r1      nop
     7  |    nop                nop
```

Найден действительно оптимальный план исполнения (для данной архитектуры), чем повторен результат приведенный в работах [30, 9]. Длина исходного плана снижена на 27.3%. Данный план исполнения был найден на 25 итерации из 100 сделанных. Средняя длина планов составила 9.53, а корень из среднеквадратичного отклонения $\sigma = 0.59$.

В заключение приведем фрагмент описания профиля микроархитектуры:

```
let micro_resources =
  [|
    2, {name="FU"; mr_type=1};
    2, {name="MU"; mr_type=1};
  |]
```

```
let build_micro_profile il =
  ...
  let micro_ld_stage0 = {
        insn=ld;
        read=[m;rj]; write=[];
        inner=Any_of fu
    } in
  let micro_ld_stage1 = {
        insn=ld;
        read=[]; write=[ri];
        inner=Any_of mu
    } in
  let micro_mul_stage0 = {
        insn=mul;
        read=[ri;rj];
        write=[ri];
        inner=Any_of fu
    }
  in
  let stage0 = [  ...
                  micro_ld_stage0;
                  micro_mul_stage0;
               ] in
  let stage1 = [  ...
                  micro_ld_stage1;
               ] in
  let pipeline = [|stage0; stage1|] in

  [|pipeline;pipeline|]
```

5.3 Целевое устройство с VLIW архитектурой

Теперь рассмотрим простую VLIW архитектуру. Все инструкции занимают 1 такт. Одновременно исполняется 4 инструкции. В процессоре 4 функциональных устройства: 2 целочисленных и 2 выборки из памяти.

В результате работы настраиваемого планировщика был получен следующий план исполнения:

```
          Слот 1     Слот 2     Слот 3          Слот 4
  Такты
```

```
0:   {nop;       nop;       ld r1 [t];     ld r0 [x_vel];}
1:   {nop;       mul r0 r1; ld r3 [y_vel]; ld r2 [x_pos];}
2:   {mul r1 r3; add r0 r2; nop;           nop;          }
3:   {nop;       nop;       nop;           ld r2 [y_pos];}
4:   {nop;       add r1 r2; st [x_pos] r0; nop;          }
5:   {nop;       nop;       st [y_pos] r1; nop;          }
```

Найден наилучший из возможных планов для данной программы (независимо от архитектуры), поскольку его длина совпадает с глубиной графа зависимостей по данным. Отсюда делаем вывод, что архитектура обеспечивает параллелизм инструкций, в наилучшей степени удовлетворяющий требованиям выбранного приложения.

Длина исходного плана снижена на 45.5%. Данный план исполнения был найден на 5 итерации из 100 сделанных. Средняя длина планов составила 6.86, а корень из среднеквадратичного отклонения $\sigma = 0.35$.

6 Заключение

В ходе данной работы был разработан алгоритм настраиваемого планирования, отвечающий поставленным требованиям, и формализм для описания профиля микроархитектуры.

На основе разработанного алгоритма реализован прототип системы настраиваемой генерации планировщиков, состоящий из

- генератора планировщиков на базе шаблонов;
- ядра, реализующего алгоритм планирования;
- модуля обнаружения конфликтов по микроресурсам;
- модуля построения графа зависимостей.

Настраиваемый планировщик поддерживает целевые устройства с несколькими конвейерами и имеет двойную параметризацию микроархитектурой и макроархитектурой.

Список литературы

[1] Булычев Д.Ю. Технология разработки программно-аппаратных систем на основе описания макроархитектуры. // Системное программирование. — СПб., 2004.

[2] Булычев Д.Ю. Язык описания макроархитектуры процессоров, версия 1.0. Технический отчет. — Институт Информационных Технологий СПбГУ. — 2002. — 22 с.

[3] Симановский А.А., Шапоренков Д.А. Синтез ассемблера и дизассемблера из описания макроархитектуры. // Системное программирование. — СПб., 2004.

[4] Смирнов М.Н. Порождение модели микроархитектуры из модели макроархитектуры. // Системное программирование. — СПб., 2004.

[5] Barabanov A., Bombana M., Fominykh N., Gorla G., Terekhov A. Reusable Objects for Optimized DSP Design // Embedded Microprocessor Systems. — IOS Press 1996.

[6] Вьюкова Н., Галатенко В., Самборский С., Шумаков С. Генерация эффективного кода для процессорных архитектур с явным параллелизмом. — М.: НИИСИ РАН. — 2001.

[7] Leroy H. The Objective Caml System. Documentation and User's Manual. // Institut National de Recherche en Informatique et en Automatique — 2003. — 481 p.

[8] Bradlee D., Henry R., Eggers S. The Marion System for Retargetable Instruction Scheduling. // Proc. of the 1991 ACM SIGPLAIN Conference on Programming Language Design and Implementation. — 1991. — P. 229-230.

[9] Brasier T., Sweany P., Beaty S., Carr S. A practical framework for combining instruction scheduling and register assignment. // Proc. of the 1995 PACT Conference on Parallel Architectures and Compilation Techniques — 1995.

[10] Grun P., Halambi A., Khare A., Ganesh V., Dutt N., Nicolau A. EXPRESSION: an ADL for System Level Design Exploration. — University of California, Irvine, Department of Information and Computer Science. — Technical Report №98-29. — 1998. — 26 p.

[11] Hadjiyiannis G., Hanono S., Devadas S. ISDL: an Instruction Set Description Language for Retargetability // Proc. of the 34th Annual Conference on Design Automation. — 1997. — P. 299-302.

[12] Halambi A., Grun P., Khare A., Ganesh V., Dutt N., Nicolau A. EXPRESSION: a Language for Architecture Exploration through Compiler/Simulator Retargetability // Proc. of the Conference on Design, Automation and Test in Europe. — 1999. — P. 485-490.

[13] Hanono S., Devadas S. Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator // Proc. of the 35th Annual Conference on Design Automation. — 1998. — P. 510-515.

[14] Gibbons P., Muchnick S. Efficient Instruction Scheduling for a Pipelined Architecture. // Proc. of the 1986 ACM SIGPLAIN Symposium on Compiler Construction. — 1991. — Vol. 21, No. 7 — P. 11-16.

[15] Hanono S. *Aviv: A retargetable Code Generator for Embedded Processors*. PhD thesis, 1999.

[16] Muchnik S. Advanced Compiler Design and Implementation. — Morgan Kaufmann Publishers. — 1997. — 855 p.

[17] Wilson T., Grewal G., Henshall S., Banerji D. An ILP-based Approach to Code Generation // Code Generation for Embedded Processors. — Kluwer Academic Publishers, 1995. — P. 103-117.

[18] Coffman E. Computer and Job-Shop Scheduling Theory. 1976.

[19] Cooper K., Schielke P., Subramanian D. An Experimental Evaluation of List Scheduling. 1998.

[20] Gebotys C., Elmasry M. Optimal VLSI Architectural Synthesis. 1992.

[21] Gebotys C., Elmasry M. Global Optimization Approach for Architectural Synthesis. 1993.

[22] Kastner D., Langenbach M. Integer Linear Programming vs. Graph-Based Methods in Code Generation. Technical report, 1998.

[23] Kastner D. A Hardware and Assembly Description Language. Number TDL 1.4. 2000.

[24] Kastner D. PROPAN: A Retargetable System for Postpass Optimizations and Analyses. 2000.

[25] Kastner D. ILP-based Approximations for Retargetable Code Optimization. 2001.

[26] McGovern A., Moss E., Barto A. Scheduling Straight-Line Code using Reinforcement Learning and Rollouts. 1999.

[27] Moss E., Cavazos J. Learning Policies for Local Instruction Scheduling. 1997.

[28] Moss E., Utgoff P., Cavazos J., Precup D., Stefanovic D., Brodley C., Scheeff D. Learning to Schedule Straight-Line Code. 1998.

[29] Qin W., Malik S. Architecture Description Languages for Retargetable Compilation. 2002.

[30] Schielke P. *Stochastic Instruction Scheduling.* PhD thesis, 2000.

[31] Zhang L. *SILP. Scheduling and Allocating with Integer Linear Programming.* PhD thesis, 1996.