Планирование команд и конвейеризация циклов на современных архитектурах

Арутюн Аветисян <arut@ispras.ru>

Аннотация. В статье предлагается метод планирования команд и конвейеризации циклов, основанный на расширяемой двухкомпонентной архитектуре планировщика – выявления и использования параллелизма на уровне команд. Компонент выявления параллелизма основан на подходе селективного планирования и состоит из ядра, поддерживающего перенос команд с созданием компенсационных копий, и модулей, реализующих дополнительные преобразования команд, включая спекулятивное и условное выполнение. Компонент использования параллелизма заключается в наборе эвристик, организующих выбор наилучшей команды для планирования на данной итерации планировщика. При описании разработанных компонент делается упор на улучшения базового подхода, необходимые для реализации предлагаемого метода в промышленном компиляторе. Приводятся экспериментальные результаты на платформах Intel Itanium и ARM.

Ключевые слова: планирование команд, конвейеризация циклов, спекулятивное выполнение, условное выполнение.

1. Введение

Для достижения высокой продуктивности вычислений в рамках одного ядра современного многоядерного процессора необходимо применение машинно-зависимых оптимизаций, позволяющих эффективно использовать ресурсы процессора. Такие оптимизации с необходимостью должны оперировать некоторой моделью ресурсов процессора и выполняться в дополнение к хорошо известным машинно-независимым оптимизациям (удаление избыточных выражений, оптимизация индуктивных переменных, вынос инвариантов и т.п.). Существует три основных машинно-независимых оптимизаций – автоматическая векторизация кода, планирование команд и конвейеризация циклов. Центральное место среди них занимает планирование команд, в комплексе с которым может выполняться и конвейеризация циклов. Векторизация кода среди этих оптимизаций стоит особняком, т.к. требует в основном хорошего анализа зависимостей по данным между массивами, анализа алиасов, и модели векторных устройств и регистров целевого процессора, в остальном представляя собой техническую задачу. Планирование команд, напротив, является NP-полной задачей, и основной сложностью при разработке алгоритма планирования является, во-первых,

создание инфраструктуры, поддерживающей все необходимые для выявления параллелизма преобразования команд, а во-вторых, создание эвристик, демонстрирующих хорошую производительность на репрезентативном наборе тестов. Последнему вопросу часто уделяется мало внимания при разработке алгоритмов планирования, а ведь именно правильное использование выявленного параллелизма дает в результате улучшение производительности программы.

В данной статье описывается разработанный нами метод планирования команд, основанный на известном подходе селективного планирования. К сожалению, базовая схема селективного планирования в том виде, как она изложена в литературе [9], не позволяет применить ее в промышленном компиляторе типа GCC [10], а также применить ее к архитектурам с явным параллелизмом команд и встраиваемым архитектурам. Основными недостатками, повлекшими за собой необходимость разработки нового метода, являются:

- Предполагаемая применимость базовых преобразований команд планировщика ко всем командам внутреннего представления компилятора. На практике для реальных программ это предположение оказывается неверным;
- Отсутствие механизма выбора лучшей на данном шаге команды для планирования из множества доступных, т.е. не разработана упомянутая выше задача использования выявленного параллелизма;
- Большая сложность, выражающаяся в значительном росте времени компиляции на реальных программах. Так, первоначальная версия алгоритма, реализованная нами для компилятора GCC, замедляла его в два раза. Для достижения приемлемой производительности оказалось необходимым переработать ядро алгоритма – изменить анализ зависимости по данным, анализ жизни переменных, базовые механизмы преобразования команд;
- Отсутствие преобразований, необходимых для выявления параллелизма на современных архитектурах с явным параллелизмом команд типа Itanium (спекулятивное выполнение) и на встраиваемых архитектурах типа ARM (условное выполнение).

Несмотря на описанные недостатки, алгоритм селективного планирования имеет и достоинства: во-первых, следование общей схеме двухпроходного планирования сверху вниз и четкое разделение шагов по выявлению и использованию параллелизма программы позволяет на основе селективного планирования построить метод с расширяемой архитектурой, в котором легко можно добавлять новые преобразования команд, увеличивающие набор доступных для планирования команд и, следовательно, выявляющие больше параллелизма, а также добавлять новые эвристики выбора команды для планирования, улучшая использование параллелизма. Во-вторых, идея селективного планировщика о нескольких точках планирования (т.н. *фронте*

планирования) и возможности одновременного планирования команд в этих точках делает равноправными команды из разных базовых блоков и путей выполнения – одна точка планирования неизбежно отдает предпочтение командам из тех блоков, в которые эта точка попадает раньше. Следовательно, для регионов планирования без четко выраженных «горячих" путей можно строить в среднем лучшие расписания. Наконец, поддержка фронта планирования позволяет организовать конвейеризацию циклов с использованием той же инфраструктуры планировщика, причем с поддержкой сложного потока управления внутри цикла. Конечно, как будет показано далее, эвристики планирования при этом также нужно доработать.

Поэтому алгоритм селективного планирования был положен нами в основу предлагаемого в разделе 2 метода планирования команд и конвейеризации циклов, позволяющего добавлять новые преобразования команд и эвристики выбора лучшей команды. Раздел 3 посвящен добавлению к разработанному планировщику спекулятивного и условного выполнения команд. Раздел 4 представляет экспериментальные результаты, а раздел 5 заключает статью.

2. Метод планирования команд и конвейеризации циклов

2.1. Общая схема планирования

Итак, предлагаемая нами схема планирования по организации планировщика следует двухпроходному планированию сверху вниз, сохраняя поддержку нескольких точек планирования и разделяя этапы выявления и использования параллелизма. Самый верхний уровень алгоритма планирования выглядит следующим образом:

1. **Построение ациклических регионов планирования** по данной процедуре программы.
2. Планирование каждого региона отдельно:
    a. **Обход фронтом планирования региона** в топологическом порядке. При этом в точках разветвления потока управления фронт переходит сразу на все блоки-последователи, а в точках слияния – уже запланированные участки кода не получают новых точек планирования.
    b. **Для каждой точки планирования**, при наличии свободных ресурсов:
        i. **Выявление параллелизма** (сбор доступных для планирования команд в данной точке).
        ii. **Использование параллелизма** (выбор лучшей команды с помощью набора эвристик или переборных алгоритмов).
        iii. **Планирование выбранной команды**. При этом обновляются структуры данных планировщика, необходимые для первых двух шагов по сбору команд и выбору лучшей.

Наиболее интересная часть схемы заключается в цикле планирования пункта 2b, работающего для каждой точки планирования до заполнения ресурсов процессора на текущем моделируемом такте, и именно эту часть мы рассмотрим подробнее (см. рисунок 1). Пока стоит отметить, что расширение возможностей планировщика и учет реальных программ может изменить и остальные части. Например, построение регионов в пункте 1 и их обход в пункте 2 может быть расширен так, что планировщик будет поддерживать конвейеризацию вложенных циклов – для этого для каждого отдельного цикла строится свой регион, а обход регионов осуществляется по гнездам циклов, в порядке от внутренних к внешним. При этом необходимо учесть, что после планирования внутреннего цикла состав региона, соответствующего внешнему циклу, может измениться. Учет реальных программ может выразиться в том, что, например, отдельные виды циклов нельзя конвейеризовать (при наличии несводимого потока управления, возникающего из-за обработки исключений или применения операторов перехода), либо планируется настолько большая процедура, что максимальный размер региона (как по количеству базовых блоков, так и команд в них) необходимо ограничить.

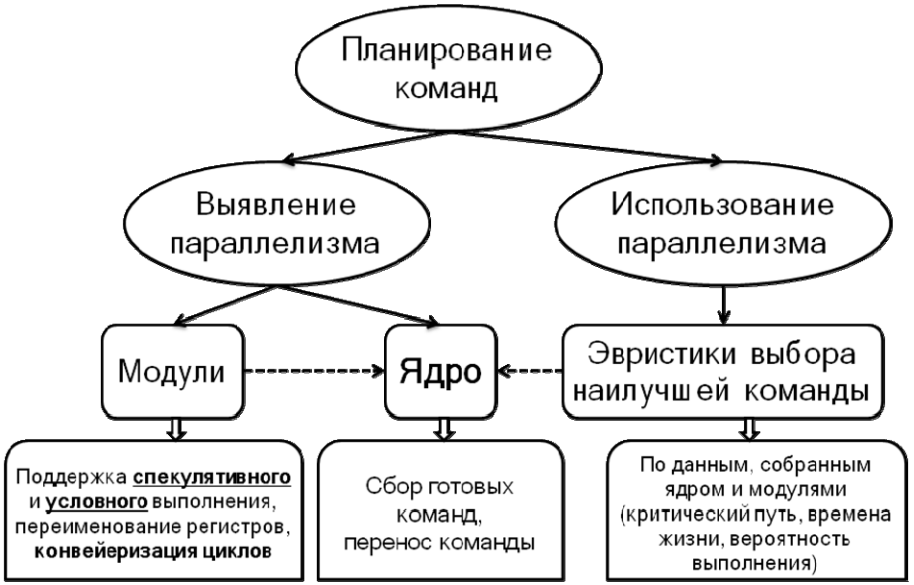

Рис. 1. Схема основного цикла планировщика.

При выявлении параллелизма (см. рисунок 1) основную часть нагрузки несет ядро планировщика, обеспечивающее сбор готовых для планирования команд

(возможно, с учетом одного или нескольких преобразований) и перенос команды в точку планирования с поддержанием корректности программы. В схеме алгоритма верхнего уровня это соответствует шагам 2b-i и 2b-iii. Ядро ответственно за два базовых преобразования команд – *клонирование* команд (т.е. возможность обеспечить такой перенос команды, который потребует создания *компенсационных копий*) и *унификацию* команд (т.е. при доступности одинаковых команд на разных путях выполнения планируется лишь одна копия команды), используя в основном данные анализа зависимостей для выполнения своих действий. Остальные преобразования команд обеспечиваются *модулями* расширения – это *переименование регистров* и связанный с ним *перенос через копии* (уже предложенные в [9] преобразования), *спекулятивное* выполнение, *условное* выполнение. Модули обращаются к ядру за необходимой информацией о свойствах команд, жизни регистров и т.п. для проведения преобразования, при этом модули могут добавлять свою информацию в структуры данных, поддерживаемые ядром. Конвейеризация циклов также может рассматриваться как модуль, но с точки зрения корректности получаемой программы в основном обеспечивается правильным построением регионов и выбором их порядка обхода.

Использование параллелизма (шаг 2b-ii) заключается в поддержке набора эвристических алгоритмов, приоритизирующих собранные команды, после чего для выдачи выбирается наиболее приоритетная команда. Возможен дополнительный перебор элементов множества готовых команд для упорядочивания выдаваемых команд в пределах одного цикла, как в [12]. Как и в случае модулей, применение эвристик, основанных на информации о критическом пути команды, относительных вероятностях выполнения отдельных команд, жизни регистров и т.п., требует обращения к ядру для поддержания всех необходимых данных в актуальном состоянии. Вообще, в описанной схеме расширение планировщика заключается в написании нового модуля или эвристики выбора, при этом у ядра запрашивается необходимая информация либо через ядро организуется ее сбор при обходе планируемого региона.

Таким образом, мы обобщили базовую схему селективного планирования, предложив организовать планировщик с явно выраженными частями выявления и использования параллелизма, который легко расширяется написанием новых модулей или эвристик выбора. В следующем подразделе мы опишем новые алгоритмы ядра планировщика, которые понадобились по сравнению с базовым подходом [9] для применения к реальным программам.

2.2. Алгоритмы ядра планировщика

К улучшениям алгоритмов ядра планировщика можно отнести выбор структур данных, учет различных типов команд, улучшения анализа зависимостей по данным, улучшение планирования выбранной команды и поддержка модулей переименования регистров и конвейеризации циклов. Опишем кратко

основные сделанные улучшения (более подробно каждое из сделанных изменений описано в [1-4, 13].

Структуры данных планировщика хранят информацию для регионов, базовых блоков, команд и *выражений* (правых частей команд) в массивах, которые индексируются по идентификатором соотвестствующих элементов. Наиболее интересным является понятие *виртуальной* команды, хранящей все данные о виде команды, не связанные с ее местом в регионе, и являющейся кэширующим контейнером для представления одной и той же команды в разных частях региона в процессе ее подъема к точке планирования. Виртуальная команда вместе с данными о ее конкретной точке нахождения называется *операцией* и является элементом множества доступных команд. Следовательно, при добавлении модулями потоково-нечувствительной информации она хранится как данные виртуальной команды, а потоково-чувствительной – как данные операции.

Необходимость учета различных типов команд, как уже упоминалось, возникает из-за неприменимости преобразований ко всем без исключения командам. Например, переименование регистров применимо только к выражениям, т.к. нужно выделить правую и левую части команды для генерации соотвествуюших пересылок. Аналогично, для некоторых команд нельзя создать компенсационные копии (для ассемблерных команд, атомарных командщ и некоторых других), т.к. невозможно гарантировать, что копия команды будет иметь ту же семантику. Тип команды указывает на применимые к ней преобразования, является потоково-нечувствительной информацией и хранится в виртуальной команде.

Улучшения анализа зависимостей направлены на две компоненты: уточнение зависимости и ускорение анализа. Уточнение необходимо из-за знания того, из какой части команды происходит зависимость, и достигается добавлением *событий* анализатора (найдена зависимость по памяти/регистрам, чтению/записи и т.п.) [4]. Подпиской на события анализатора планировщик может выполнять различные действия – отслеживание частей команд, породивших зависимость, стоимостей зависимостей и т.д. Ускорение анализа получается за счет кэширования наиболее частых результатов анализа – наличия/отсутствия зависимости – битовыми множествами, содержащими идентификаторы соответствующих команд, и за счет поддержки специальных контекстов анализатора, хранящих эффект разбора данной команды [3].

Этап планирования выбранной команды требует поиска всех команд, которые могли быть унифицированы в данную. Изначальный подход селективного планирования [9] уже включает в себя средства для ускорения этого поиска, сохраняя промежуточные множества готовых команд в начале каждого базового блока. Для преобразованных команд этого оказывается недостаточно, так как при поиске необходимо отменять сделанные преобразования ровно в тех точках региона, при протаскивании через которые они были впервые применены. Для ускорения этой части алгоритма мы

разработали механизм кэширования преобразований, который при выполнении преобразования некоторой операции на некоторой команде записывает обе формы операции – изначальную и преобразованную – в вектор *истории преобразований*, сохраняемый для каждой команды из региона. При поиске запланированной операции, зная текущую обрабатываемую команду региона, легко найти в кэше старую версию искомой команды. Тонкостью при реализации кэша является тот факт, что к моменту поиска операции команда, на которой она была изначально преобразована, уже может быть сама запланирована и перемещена со старого места в регионе. Также возможно, что отмену преобразования нужно будет делать по достижении при поиске компенсационной копии команды, на которой была преобразована операция. Поэтому поиск в кэше осуществляется не по идентификатору команды, а по хэшу ее виртуальной команды, который одинаков для всех компенсационных копий одной команды. При создании компенсационных копий векторы историй переносятся на эти копии с оригинальной команды.

Поддержка переименования регистров ядром требует отслеживания регистров, которые не могут быть использованы для переименования при перемещении команды из-за нарушений интервалов жизни регистра, а также из-за ограничений целевой архитектуры, возникающих при планировании уже распределенного на регистры кода. Наиболее важным является первый случай, и для него выполнено два улучшения. Во-первых, частый случай доступности оригинального целевого регистра также кэшируется в потоково-чувствительных данных операции. Во-вторых, для контроля над агрессивностью планировщика ведется учет регистрового давления. В каждой точке региона ядром поддерживается значение текущего регистрового давления, которое изменяется на пути перемещения вверх планируемой команды. Предполагается, что влияние команды на регистровое давление не зависит от ее места в программе (что не совсем точно), а значит, хранится в виртуальной команде. При передвижении команды наверх обновляется значение максимального регистрового давления вдоль соответствующего пути переноса, и, если это максимальное значение превысит текущее сохраненное значение регистрового давления в некоторой точке пути, то оно обновляется. Для ограничения роста давления сохраненное значение (хранящееся отдельно для каждого регистрового класса для большей точности) сравнивается с максимальным числом доступных регистров, и для некоторого значения перемещение команды запрещается.

Конвейеризация циклов с точки зрения ядра планировщика из предложенной нами схемы является особым преобразованием, поскольку поддерживается уже имеющимся преобразованием клонирования команд за исключением единственной детали – запретом сбора доступных для планирования команд вдоль обратного ребра региона-цикла. Разрешением сбора команд для планирования в начале цикла, в том числе уже запланированных, достигается конвейеризация. Конечность планирования обеспечивается запретом перехода

фронта планирования на запланированный код и ограничением на максимальное количество раз планирования команды [3].

2.3. Оценка выгоды перемещения при конвейеризации

Особеностью поддержки конвейеризации является трудность оценки влияния конвейеризованных команд на расписание цикла в целом [1]. Из-за важности этой проблемы и отсутствия ее решения в базовом алгоритме [9] мы кратко обсуждаем ее и в настоящей статье. Рассмотрим следующий типичный цикл:

```
Загрузка данных
Использование данных
Переход на начало цикла
```

После планирования загрузок и некоторых использований планировщик может обнаружить «дыры» в расписании, которые не могут быть заполнены командами, находящимися ниже. В этот момент он рассматривает для конвейеризации команды сверху, и если они подходят, то происходит следующее:

```
<-- Уже запланированный -->
<-- Старое место загрузки, «дыра» -->
Использование данных <-- уже запланировано -->
Загрузка <-- здесь был свободный слот -->
Незапланированный код
Переход на начало цикла
```

Был использован свободный слот для конвейеризации загрузки, но при этом создана «дыра» среди уже запланированных команд, которая потенциально могла бы быть заполнена одной из команд, использующих результат загрузки (запланированной или незапланированной).

Другая проблема состоит в том, что команды из начала цикла могут быть выданы слишком рано после конвейеризованных командами в конце цикла, от которых они зависят. В том же примере после конвейеризации загрузки на ее старом месте в случае Itanium возникает команда проверки, которая выполняется сразу после команды перехода на начало новой итерации, что достаточно близко для того, чтобы вызвать задержку. Чтобы решить эту проблему, между этими двумя командами необходимо больше запланировать других команд либо в начале цикла, либо загрузка должна быть помещена перед одной из команд, использующих значение с предыдущей итерации.

Мы пытаемся улучшить конвейеризованное расписание, выполняя дополнительный проход планировщика над конвейеризованным циклом, но на этот раз уже с отключенной конвейеризацией. Такая техника помогает уплотнить конвейеризованное расписание, заполняя появляющиеся «дыры» за

счет увеличения времени компиляции. Для решения второй проблемы мы используем две различные эвристики. Во-первых, в некоторых случаях можно предсказать, произойдет ли задержка на команде проверки. Это происходит, когда спекулятивная загрузка планируется на одном такте с последней командой перехода в цикле. Пусть N – номер этого такта, C – номер такта, на котором планируется команда проверки. Тогда расстояние между загрузкой и проверкой будет C+1 тактов, т.к. один такт необходим для команды перехода. Когда это расстояние больше либо равно L, латентности загрузки, можно выполнять спекуляцию. Во-вторых, мы стараемся увеличить промежуток между загрузкой и проверкой, откладывая планирование этой проверки с помощью уменьшения ее приоритета и давая другим командам больше шансов быть запланированными между загрузкой и проверкой, уменьшая вероятность задержки на команде проверки.

3. Поддержка спекулятивного и условного выполнения

3.1. Спекулятивное выполнение

Модуль поддержки спекулятивного выполнения опирается на теоретические результаты из [6] и использует выполненные улучшения анализа зависимостей, описанные выше и в [3, 5]. Согласно предложенной в [6] модели спекулятивного выполнения необходимо определить легальные возможности для спекулятивного преобразования и вероятности выполнения зависимостей по данным и по управлению, после чего с помощью утверждений 1-3 из [6] можно судить о выгоде каждого конкретного спекулятивного перемещения. Покажем, как в условиях промышленного компилятора можно максимально близко реализовать такой подход.

Мы используем описанные выше события анализа зависимостей для маркировки зависимостей как нуждающиеся в спекулятивном выполнении по данным либо по управлению для их преодоления (*статусы* зависимостей). После этого для поддержки целевой архитектуры необходимо уметь отвечать на вопрос, может ли данная команда быть выполнена спекулятивно определенным образом, и если да, то как выглядит ее спекулятивная форма. Так как разные виды спекулятивного выполнения могут быть последовательно применены при проносе команды наверх, то спекулятивный *статус* команды должен быть частью данных операции. Ядро планировщика обеспечивает корректное слияние статусов при слиянии множеств доступных команд вместе с остальными данными операций для того, чтобы при унификации спекулятивной и неспекулятивной команды была сгенерирована правильная команда для планирования. При создании компенсационных копий спекулятивных команд ядром также учитывается ее текущий статус, а на месте оригинальной команды создается соответствующая команда проверки.

Описанные изменения достаточны для выполнения легальных спекулятивных преобразований, но необходимо еще отобрать выгодные преобразования. Для этого те же события анализа зависимостей генерируют не только статусы, но и вероятностные оценки выполнения конкретной зависимостей. Из-за невозможности профилирования зависимостей по данным применяется набор эвристик [5] – различие между прямым доступом к элементам полей структуры и через указатели, различие указателей-параметров функции, различие баз указателей ведет к малой вероятности зависимости. Для зависимостей по управлению используются либо статические вероятности, предсказанные по потоку управления, либо данные профилирования. Аналогично, ядро планировщика обеспечивает слияние вероятностей зависимостей по управлению при перемещении команды наверх.

При планировании спекулятивной команды ее стоимость вместо формул по типу [8] отражается в наборе эвристик сортировки, заключающийся в предпочтении обычных операций спекулятивным, предпочтении спекуляции по данным более «вероятной» (как правило) спекуляции по управлению, сравнении спекулятивных команд одного типа по их вероятности. В условиях промышленного компилятора этих эвристик оказывается достаточно для приближенной оценки выгодности каждого конкретного преобразования, что показывается экспериментальными результатами – можно настроить вероятностные отсечки таким образом, чтобы получить выигрыш в среднем по репрезентативному набору тестов.

Не освещенным ранее вопросом, специфичным для нашего планировщика, является связь между проводимыми преобразованиями команд. Так, спекулятивная команда может быть как перенесена через копию, так и запланирована с созданием компенсационных копий. Для переименования регистров в спекулятивной команде необходимо сначала выполнить переименование, чтобы команда спекулятивной проверки получила правильный номер целевого регистра [4].

3.2. Условное выполнение

Поддержка условного выполнения при планировании команд описана нами в работах [7,8], здесь мы кратко остановимся на вопросах, связанных с использованием разработанной нами схемы планирования для реализации модуля условного выполнения. В отличие от спекулятивного выполнения, где в данных операции нужно сохранять ее спекулятивный статус, для условного выполнения нужно сохранять предикат, которым защищена операция. Далее, определение типов команд и левых/правых частей ядром также учитывает записанный предикат, отбрасывая его по необходимости. Главной особенностью является применение условного выполнения, т.е. наложение предиката, не при проносе команды через другую команду, а при слиянии ядром множеств готовых команд в точках слияния потока управления. В момент слияния ядро определяет, контролируется ли данный условный

переход предикатом, и при положительном ответе передает управление модулю условного выполнения, который накладывает выделенный предикат на те команды из текущего множества готовых команд, которые могут быть переведены в условную форму согласно описанию целевой архитектуры.

При поиске запланированной команды и ее перемещении наверх к точке планирования (этап 2b-iii в разделе 2) необходимо, как и для других преобразований, отменять условное выполнение, т.е. удалять контролирующий предикат соответствующих команд при спуске через команду условного перехода. Для этого преобразование в форму с условным выполнением записывается в истории преобразований команды, при этом в качестве команды, на которой произошло преобразование, записывается соответствующий условный переход, что позволяет автоматически отменить его при спуске через этот переход. Кроме того, нужно преобразовывать текущую операцию в условную форму при подъёме через условный переход, если выбранная команда также была преобразована на этом условном переходе. Чтобы решить эту задачу, мы разработали механизм повторения преобразований команд, также использующий историю преобразований. Этот механизм позволяет повторно произвести все преобразования, произошедшие над выбранной для планирования командой в процессе вычисления множества доступных команд, над искомой командой в процессе подъёма от найденной первоначальной команды. Наконец, при поиске команды при спуске через условный переход, контролируемый предикатом искомой команды, необходимо продолжать поиск только на одной ветви, соответствующей этому предикату: так как перенос команды через переход с получением условной формы не изменяет условий, при которых эта команда будет выполнена, то на ветви с обратным предикатом искомая команда никогда не будет выполнена, поэтому на этом пути нельзя удалять такие же команды, если они там есть.

Как и для спекулятивного выполнения, вопрос о комбинации условного выполенния с другими преобразованиями разрешается положительно – можно переименовывать команду в условной форме, а также дополнительно преобразовывать в спекулятивную форму и выполнять подстановку через копии. Отслеживание предиката команды также позволяет более точно оценивать интервал жизни ее целевого регистра (подробнее в [7]). Наиболее интересным результатом является использование условного выполнения вместо спекулятивного при конвейеризации циклов – это делает возможной конвейеризацию доступов в память на архитектурах, не поддерживающих необходимое для этого спекулятивное выполнение (ARM).

4. Экспериментальные результаты

Мы выполнили запуск стандартного промышленного набора тестов SPEC CPU 2000 [11] на архитектуре Intel Itanium с использованием компилятора GCC версии 4.6, в котором присутствует реализация нашего алгоритма планирования. По результатам запуска версии компилятора с поддержкой

спекулятивного выполнения и всех изменений раздела 2, реализованных в первую очередь, пакет тестов SPEC FP ускорился на 3.5% [1], отдельные тесты – до 10.9%. Это показывает хорошие результаты выявления параллелизма, обычно присутствующего на тестовых программах с плавающей точкой. Результаты на SPEC INT в среднем схожи со старым планировщиком, также улучшенным нами в результате работ [14], так как для т.н. целочисленных тестовых программ основное содержание параллелизма достается имеющейся по результатам [14] поддержкой спекулятивности. Затраты на время компиляции составляют 15-18%, что является нормальным показателем для высокого уровня оптимизации на платформе с явно выраженным параллелизмом, и эти результаты позволили включить разработанный планировщик команд по умолчанию для компилятора GCC на платформе Itanium для уровня оптимизации –O3, после его включения в основную ветвь компилятора, начиная с версии 4.4.

При запуске версии компилятора с поддержкой условного выполнения и результатов по улучшению анализа указателей из [5] средние результаты тестов SPEC FP улучшились в среднем на 4%, один тест сработал некорректно (в последних версиях поддержки условного выполнения данный тест был исправлен). При этом существенным достижением являлось отсутствие тестов, на которых производительность заметно падает, но при этом максимальное ускорение снизилось с 10.9% до 8.9%.

При тестировании разработанного планировщика на наборе программ (пакеты SQLite, CLucene, matmult, x264, Evas) на архитектуре ARM со всеми улучшениями, описанными в разделах 2 и 3, были получены ускорения от 1% до 3% в среднем по тестам этих программ. Наибольшое ускорение составляло до 20%, но при этом наблюдались и падения производительности в 5-8%. В целом, аккуратной настройкой планировщика можно добиться стабильного среднего ускорения для набора тестов, но из-за ограниченного параллелизма платформы ARM агрессивное планирование может ухудшить отдельные тесты, и требуется взвешенный подход при использовании планировщика для каждой конкретной программы на ARM.

5. Заключение

В данной статье был рассмотрен алгоритм планирования, созданный на базе селективного планировщика, который позволяет эффективно выявлять и использовать параллелизм на уровне команд с помощью набора преобразований, важнейшими из которых является переименование регистров, спекулятивное и условное выполнение, конвейеризация циклов. Выполненные нами улучшения алгоритма направлены, во-первых, на поддержку современных архитектур типа Itanium и ARM, а во-вторых, на возможность применения подхода к реальным программам в условиях промышленного компилятора GCC. Экспериментальные результаты показывают, что реализованные преобразования позволяют использовать

планирование после распределения регистров с мощью, близкой к алгоритмам, работающим до распределения регистров, с приемлемыми затратами по времени компиляции, которые позволили включить планировщик в основную ветвь компилятора GCC, начиная с версии 4.4.

Список литературы

[1] Arutyun Avetisyan, Andrey Belevantsev, and Dmitry Melnik. GCC instruction scheduler and software pipelining on the Itanium platform. 7th Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC-7). Boston, MA, USA, April 2008. http://rogue.colorado.edu/EPIC7/avetisyan.pdf

[2] Andrey Belevantsev, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik, Dmitry Zhurikhin. An interblock VLIW-targeted instruction scheduler for GCC. In Proceedings of GCC Developers' Summit 2006, Ottawa, Canada, June 2006, pp.1-12.

[3] Andrey Belevantsev, Maxim Kuvyrkov, Alexander Monakov, Dmitry Melnik, and Dmitry Zhurikhin. Implementing an instruction scheduler for GCC: progress, caveats, and evaluation. In Proceedings of GCC Developers' Summit 2007, Ottawa, Canada, July 2007, pp. 7-21.

[4] Andrey Belevantsev, Dmitry Melnik, and Arutyun Avetisyan. Improving a selective scheduling approach for GCC. GREPS: International Workshop on GCC for Research in Embedded and Parallel Systems, Brasov, Romania, September 2007. http://sysrun.haifa.il.ibm.com/hrl/greps2007/

[5] Dmitry Melnik, Sergey Gaissaryan, Alexander Monakov, Dmitry Zhurikhin. An Approach for Data Propagation from Tree SSA to RTL. GREPS: International Workshop on GCC for Research in Embedded and Parallel Systems, Brasov, Romania, September 2007.

[6] А.А.Белеванцев, С.С.Гайсарян, В.П.Иванников. Построение алгоритмов спекулятивных оптимизаций. Журнал Программирование, N3 2008, с. 21-42.

[7] Д. Мельник, А. Монаков, А. Аветисян. Поддержка команд с условным выполнением в селективном планировщике команд. Труды ИСП РАН, том 21, 2011 г.

[8] Dmitry Melnik, Alexander Monakov, Andrey Belevantsev, Tigran Topchan, Mamikon Vardanyan. Improving Selective Scheduler Approach with Predication and Explicit Data Dependence Support. 8th Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC-8), 2010.

[9] Soo-Mook Moon and Kemal Ebcioglu. Parallelizing Nonnumerical Code with Selective Scheduling and Software Pipelining. ACM TOPLAS, Vol 19, No. 6, pages 853-898, November 1997.

[10] GCC, GNU Compiler Collection. http://gcc.gnu.org

[11] SPEC CPU 2000. http://spec.org/cpu2000/

[12] Vladimir Makarov. The finite state automaton based pipeline hazard recognizer and instruction scheduler in GCC. In Proceedings of GCC Developers' Summit, Ottawa, Canada, June 2003.

[13] А. Белеванцев, Д. Журихин, Д. Мельник. Компиляция программ для современных архитектур. Труды Института системного программирования РАН, том 17, 2009 г. стр. 31-50.

[14] Andrey Belevantsev, Alexander Chernov, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik. Improving GCC instruction scheduling for Itanium. In Proceedings of GCC Developers' Summit 2005, Ottawa, Canada, June 2005.