АО "МЦСТ"

На правах рукописи

Гимпельсон Вадим Дмитриевич

Сокращение длины критических путей при динамической трансляции двоичных кодов

05.13.11 - "Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей"

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель
к.т.н. Волконский Владимир Юрьевич

Москва – 2018

Содержание:

Введение.

Актуальность работы.

Повышение производительности является одним из главных направлений развития вычислительной техники в течение всей её истории. Во многих отраслях производительность является основным требованием, предъявляемым к вычислительным комплексам. Инвестируются огромные средства в развитие новых архитектур микропроцессоров и вычислительных комплексов.

Одним из подходов к увеличению производительности микропроцессоров является логическое усовершенствование внутренней архитектуры, позволяющее извлекать параллелизм на уровне отдельных операций. Есть разные способы извлечения параллелизма между операциями.

Одним из первых подходов к созданию микропроцессорных архитектур, извлекающих параллелизм на уровне отдельных операций, является так называемая суперскалярная архитектура. В таких микропроцессорах независимость между отдельными операциями (а именно это свойство позволяет распараллеливать код) определяется на аппаратном уровне. Это даёт возможность при неизменной системе команд микропроцессора создавать всё более и более производительные решения за счёт улучшения микроархитектуры конвейера. Однако существенным недостатком суперскалярной архитектуры является то, что анализ, реализованный на аппаратном уровне, является динамическим по своей природе и, следовательно, имеет серьёзные ограничения из-за своей сложности. Это приводит к невозможности проанализировать большое количество операций одновременно, так как усложнения различных анализов влекут за собой нелинейный рост необходимого оборудования и увеличение выделяемой энергии.

Одним из путей устранения описанных ограничений суперскалярной архитектуры является разработка микропроцессорных архитектур с явно выраженной параллельностью на уровне команд. Такие архитектуры в середине 90-х годов получили название *EPIC (Explicitly Parallel Instruction Computing)* [14], [15]. Особенностью таких архитектур является то, что практически вся работа по распараллеливанию на уровне операций перекладывается с аппаратуры на компилятор, а в результирующем коде параллельность между операциями выражается явно. Такой подход избавляет от необходимости аппаратной реализации распараллеливания операций, что позволяет увеличить параллелизм по сравнению с суперскалярными архитектурами.

При разработке новых микропроцессорных архитектур часто приходиться сталкиваться с серьёзным недостатком, препятствующим их быстрому массовому распространению – это

новая несовместимая система команд. Для существующих архитектур создано и отлажено огромное количество программного обеспечения. Это программное обеспечение в лучшем случае придётся перекомпилировать для перехода на новую архитектуру, а в худшем придётся разрабатывать заново, так как исходные тексты программ могут быть не доступны.

Многие разработчики микропроцессоров использовали и продолжают использовать аппаратный метод обеспечения двоичной совместимости со старыми архитектурами [16], [17]. Однако такой подход не всегда достаточно эффективен, так как разработчики вынуждены поддерживать многие свойства ранее разработанных архитектур, и следовательно наследуются недостатки этих архитектур, что приводит к снижению итоговой производительности.

Существует альтернативный способ обеспечения двоичной совместимости, получивший название *двоичной трансляции*. Суть этого метода заключается в программной перекомпиляции (перетрансляции) двоичных кодов платформы, с которой мы хотим обеспечить совместимость (исходная архитектура), в коды новой платформы (целевая архитектура). Достоинствами этого подхода являются высокая эффективность и возможность его применения для широкого класса исходных и целевых архитектур.

Важной особенностью двоичного транслятора является то, что он должен быть динамическим, то есть трансляция (компиляция) кода исходного приложения происходит во время его исполнения. Это связано с тем, что практически любая микропроцессорная архитектура позволяет создавать код динамически и затем исполнять его. Таким образом, время работы приложения под управлением двоичного транслятора складывается из времени работы оттранслированных кодов и времени, затраченным на трансляцию этих кодов. В силу этого на скорость трансляции должны быть наложены очень жёсткие ограничения. С другой стороны для получения быстрого результирующего кода необходимо проводить различные оптимизации, часто тяжеловесные. Как же совместить эти два условия для получения эффективной системы двоичной трансляции?

Хорошо известно, что в большинстве программ код используется неравномерно [49]. Большая часть времени работы проводится в коде, занимающем очень небольшую часть программы. Таким образом, чтобы двоичный транслятор мог обеспечить общую высокую эффективность, сравнимую с кодами, полученными из-под языкового компилятора, он должен включать в себя несколько уровней трансляторов (оптимизирующих компиляторов), различающихся между собой качеством и количеством проводимых оптимизаций кода, а также временем работы. Выбор, каким транслятором оптимизировать данный код, производится на основе профильной информации, которая собирается в процессе работы исходной программы. Чем дольше код работает, тем полезнее будет оттранслировать этот код транслятором, дающим более быстрый код целевой архитектуры. Транслятор верхнего уровня (дающий самый

быстрый код целевой архитектуры), будем далее называть *двоичным оптимизирующим транслятором*.

Двоичный оптимизирующий транслятор с целевой EPIC архитектурой для получения максимально быстрого результирующего кода должен уметь производить основные оптимизации специфичные для этих архитектур: построение предикатного кода, конвейеризация циклов, сокращение критического пути и т.д. Без такого минимального набора оптимизаций невозможно достичь высокой производительности результирующих кодов. В тоже время условия, в которых работает двоичный оптимизирующий транслятор, накладывают серьёзные ограничения и выдвигают определённые требования к алгоритмам, используемым в двоичном оптимизирующем трансляторе. Во-первых, это дополнительные семантические свойства, накладываемые особой спецификой двоичной трансляции. Пожалуй, основным семантическим ограничением является *задача обеспечения точного состояния регистров и памяти исходной архитектуры при возникновении прерываний и исключений*. Вторым важным требованием, предъявляемым к алгоритмам оптимизаций, является уже упомянутая выше, *собственная скорость работы этих алгоритмов*.

Одними из важнейших оптимизаций для архитектур с явно выраженной параллельностью являются оптимизации позволяющие сократить время исполнения за счёт распараллеливания вычислений. Одними из эффективных методов позволяющих распараллелить вычисления являются оптимизации производящие *сокращение длины критического пути* (или просто *сокращение критического пути*) в ациклических областях. Критическим путём называют последовательность связанных (зависимых) операций, таких, что длина вычислений этой последовательности максимальна в данном блоке. Соответственно длина критического пути определяет длину вычислений всего блока, и сократив критический путь мы ускорим вычисления. В рассматриваемых архитектурах имеется большое количество исполняющих устройств, поэтому часто оказывается, что сокращение критического пути за счёт построения одной или нескольких новых операций даёт ускорение результирующего кода.

Отдельно необходимо отметить техники по уменьшению времени выполнения циклов. Здесь применим класс оптимизаций, который получил название “*конвейеризация циклов* методом наложения итераций” или, в англоязычной литературе, “программная *конвейеризация циклов*” (software pipelining). Суть этого класса оптимизаций заключается в том, что возможно совместить вычисление нескольких итераций цикла в одной итерации цикла результирующего кода. В совокупности с методами сокращения критического пути в ациклических областях, специализированные методы интегрирующие конвейеризацию циклов с сокращением критических путей, позволяют получить более быстрый результирующий код.

Необходимость создания быстрых алгоритмов сокращения критического пути и конвейеризации циклов в двоичном оптимизирующем трансляторе, позволяющих максимально использовать возможности архитектуры с явным выраженной параллельностью на уровне команд, и как следствие, достижение эффективной совместимости, определяет актуальность диссертационной работы.

Цель исследования.

Целью диссертационной работы является повышение эффективности двоичного оптимизирующего транслятора за счёт адаптации старых и разработки новых алгоритмов сокращения длины критического пути в циклических и ациклических областях для архитектур с явно выраженной параллельностью на уровне команд. Эти алгоритмы должны учитывать особенности оптимизирующего двоичного транслятора: специфическое окружение и требования скорости работы. В соответствии с этими целями были определены следующие задачи:

- провести анализ современного уровня развития методов сокращения длины критического пути в циклических и ациклических областях, и возможностей их использования в двоичном оптимизирующем трансляторе;
- разработка новых алгоритмов сокращения длины критического пути в ациклических областях;
- разработка новых алгоритмов сокращения длины критического пути в циклических областях основанных на конвейеризации циклов;
- интеграция методов сокращения длины критического пути в ациклических областях с алгоритмом конвейеризации циклов;
- разработка метода интеграции алгоритма сокращения длины критического пути с другими оптимизирующими преобразованиями с целью устранения ситуаций, когда качество некоторой оптимизации зависит от результатов работы алгоритма сокращения длины критического пути, а качество алгоритма сокращения длины критического пути зависит от этой оптимизации;
- обеспечение во всех разрабатываемых алгоритмах семантических требований, накладываемых двоичной трансляцией, и высокой скорости работы всех разрабатываемых алгоритмов;
- реализация указанных алгоритмов в динамическом двоичном трансляторе.

Методы исследования заимствованы из областей системного программирования, технологии компиляции, теории графов, теории алгоритмов. Эффективность предложенных

методов оценивалась путём замера ключевых параметров предлагаемых алгоритмов, а также замеров времени исполнения задач на вычислительном комплексе с микропроцессором “Эльбрус” и на потактной модели микропроцессора Эльбрус-S. Замеры производились на пакетах задач SPEC CPU95 [71] и SPEC CPU2000 [72]. Также для анализа качества результирующего кода использовались горячие участки операционной системы Windows 2000 и горячие участки типичных пользовательских приложений для этой ОС: Microsoft Word, Microsoft Power Point, Internet Explorer и многие другие.

Научная новизна.

Научной новизной обладают следующие результаты работы:

- быстрый алгоритм сокращения длины критического пути в ациклических областях;
- схема взаимодействия алгоритма сокращения длины критического пути в ациклических областях с другими оптимизирующими преобразованиями для преодоления “замкнутого круга”, когда оптимизации являются взаимозависимыми;
- алгоритм разметки времён раннего и позднего планирования на расширенном графе зависимостей;
- алгоритм конвейеризации циклов в динамическом двоичном оптимизирующем трансляторе;
- интеграция алгоритмов сокращения длины критического пути с алгоритмом конвейеризации циклов;
- сформулированы и доказаны теоремы, показывающие корректность и оптимальность предложенных алгоритмов, а также произведена оценка сложности этих алгоритмов.

Результаты работы, выносимые на защиту.

В процессе проведения диссертационного исследования были получены следующие результаты, выносимые на защиту:

- разработка и реализация быстрого алгоритма сокращения длины критического пути в ациклических областях; теорема об оптимальности предложенного алгоритма;
- алгоритм разметки времён раннего и позднего планирования на расширенном графе зависимостей; сформулированы и доказаны теоремы о корректности, оптимальности и сложности предложенного алгоритма;
- алгоритм конвейеризации циклов в динамическом двоичном оптимизирующем трансляторе; сформулирована и доказана теорема о сложности предложенного алгоритма; интеграция методов сокращения длины критического пути в ациклических областях с алгоритмом конвейеризации циклов.

Теоретическая и практическая значимость.

Предложен алгоритм сокращения длинны критического пути в ациклических областях. Предложен алгоритм разметки времён на расширенном графе зависимостей и основе него разработан алгоритм конвейеризации циклов.

Практическая ценность диссертационной работы состоит в том, что на основе предложенных алгоритмов удалось значительно повысить эффективность работы динамического двоичного транслятора для архитектуры “Эльбрус”. Впервые было показано, что аппаратная техника вращающихся регистров может эффективно применяться в двоичном трансляторе. Результаты исследований были реализованы в следующих программных и аппаратных системах:

- динамический двоичный оптимизирующий транслятор уровня всей системы с архитектуры Intel x86 на архитектуру “Эльбрус”, разработанный в АО “МЦСТ”;
- динамический двоичный оптимизирующий транслятор уровня приложений ОС Linux с архитектуры Intel x86 на архитектуру “Эльбрус”, разработанный в АО “МЦСТ”;
- статический оптимизирующий транслятор с архитектуры Intel x86 на архитектуру IPF (Itanium), разработанный в АО “МЦСТ” в рамках совместного проекта с Intel Corporation;
- микропроцессор Эльбрус, разработанный в АО “МЦСТ”, обеспечивающий совместимость с архитектурой Intel x86;
- динамический двоичный транслятор уровня приложений ОС Linux, разработанный в ООО “Эльбрус Технологии”.

Предложенные алгоритмы также могут быть использованы в оптимизирующих компиляторах и двоичных трансляторах для различных архитектур.

Апробация.

Основные результаты диссертационной работы докладывались на следующих научно-технических конференциях и семинарах:

- 5th RISC-V Workshop, Mountain View, CA, November 29-30, 2016.
- Open Conference on Compiler Technologies, Москва, РАН, 2015 г.
- Samsung Compiler Workshop, Москва, Samsung Office, 2014 г.
- На XXXIV Международной молодёжной научной конференции “Гагаринские чтения”, Москва, МАТИ, 2008 г.
- Международная научная конференция, посвящённая 80-летию со дня рождения академика В.А. Мельникова, 2008 г.

- На XXIII научно-технической конференции “Направления развития и применения перспективных вычислительных средств и новый информационных технологий в ВВТ РКО”, Москва, в/ч 03425, 2007 г.
- На XXXIII Международной молодёжной научной конференции “Гагаринские чтения”, Москва, МАТИ, 2007 г.
- На XXI научно-технической конференции войсковой части 03425. Москва, в/ч 03425, 2003 г.
- На семинарах секции программного обеспечения ЗАО “МЦСТ” в 2005-2016 годах.

Публикации.

По теме диссертации опубликовано 13 печатных работ [1]-[13]. Работы [3], [4], [7], [10], [12] опубликованы в изданиях из перечня ВАК. В работе [12] описаны методы сокращения длины критических путей в ациклических областях. Личный вклад автора заключается в разработке и реализации быстрого алгоритма минимизации высоты графа зависимостей. В работах [4] и [8] представлены методы сокращения длины критических путей в циклах. В работах [1] и [2] личный вклад автора заключается в переносе алгоритмов, изложенных в данной диссертационной работе, в динамические двоичные трансляторы из x86 в ARM и из RISC-V в x86. Работа [3] написана совместно с несколькими авторами. Личный вклад автора в этой работе заключается в описании общей схемы функционирования динамического двоичного транслятора, а также описания оптимизирующего двоичного транслятора, из x86 в “Эльбрус”. В работе [7] личный вклад автора заключается в постановке задачи и в разработке методов коррекции профильной информации в случае её не консистентности. В работе [10] личный вклад автора заключается в предложенных методах активации оптимизаций в двоичном трансляторе. Совместная работа [13] посвящена методам обеспечения точного состояния контекста в двоичном трансляторе. Личный вклад автора заключается в реализации и поддержке этих методов в алгоритмах сокращения длины критического пути.

В ходе выполнения работы было получено свидетельство о государственной регистрации программ для ЭВМ [1, см. стр. 157].

Личный вклад автора.

Все представленные в диссертации результаты получены лично автором.

Структура и объём работы.

Работа состоит из введения, четырёх глав, заключения и двух приложений. Основной текст диссертации (без приложений и списка литературы) занимает 147 страницы, общий объем – 166 страниц. Список литературы содержит 113 наименований.

1. Двоичная трансляция и методы сокращения длины критического пути в графе зависимостей

1.1. Двоичная трансляция

В индустрии вычислительной техники постоянно возникает потребность в создании новых микропроцессорных архитектур. Причины этой потребности могут быть самыми различными, начиная от необходимости создания более высокопроизводительных вычислительных комплексов и заканчивая академическими исследованиями. Перед разработчиками встаёт вопрос: как обеспечить новый микропроцессор необходимым программным обеспечением (операционной системой, библиотеками, пользовательскими программами и т.д.).

Первый путь – создать всё программное обеспечение заново или перекомпилировать уже имеющиеся из языков высокого уровня. Однако этот путь может влечь за собой большие ресурсные затраты. Далеко не всегда возможно просто взять и перекомпилировать необходимые программы, в особенности для больших программных систем. Как правило, перевод программного обеспечения на новые вычислительные комплексы требует серьёзной работы, называемой *портированием программного обеспечения*. Например, некоторые части программного обеспечения могут быть написаны на ассемблере и эти части надо полностью переписывать заново. Также в кодах, написанных на языках высокого уровня, могут присутствовать неявные предположения о свойствах архитектуры, для которой они разрабатывались. Например, если программа разрабатывалась для 32-битной архитектуры, то в ней могут присутствовать закладки на то, что адресная переменная имеет размер четыре байта, и при переходе на 64-битную архитектуру это предположение (уже неверное) может привести к ошибке. Возможность возникновения таких ошибок влечёт за собой необходимость полного перетестирования портируемого программного обеспечения. Необходимо проверить весь набор тестов, который запускался перед выпуском финальное версии продукта, а это большие затраты различных ресурсов.

Некоторые части программных продуктов могут быть не доступны в исходных кодах. Например, программа использует библиотеки сторонних разработчиков, которые не распространяются в исходных кодах. В таких случаях надо либо разрабатывать необходимые библиотеки заново, либо заказывать их портирование у авторов. И оба этих варианта требуют дополнительных капиталовложений. Наконец, само используемое программное обеспечение может быть разработано сторонними разработчиками, которые по каким-либо причинам не могут или не будут портировать свои программы.

Второй путь – это поддержка полной аппаратной совместимости новых микропроцессоров со старыми. Здесь имеется два разветвления. Первое – это развитие новых архитектур на базисе старых. Системы команд новых микропроцессоров полностью включают в себя систему команд предыдущих микропроцессоров. Пожалуй, наиболее ярким представителем такого подхода являются микропроцессоры с архитектурой x86. Современные x86 микропроцессоры представляют собой сложнейшие чипы. Для повышения производительности, в этих процессорах, начиная с 90-х годов прошлого столетия, используют так называемую суперскалярную архитектуру. Внутренне строение микропроцессора постоянно изменяется и развивается. Появляется гораздо больше внутренних (скрытых от пользователя) регистров, сложные исходные команды заменяются на несколько менее сложных, но более быстрых внутренних команд и так далее. Для поддержания совместимости с предыдущими микропроцессорами семейства используется аппаратная перекодировка команд во внутренние, для каждого микропроцессора свои, команды. Поскольку перекодировка происходит динамически, во время исполнения кода, это накладывает существенные ограничения на сложность алгоритмов перекодировки, а следовательно накладывает ограничения и на внутреннюю систему команд микропроцессора. Её нельзя сделать принципиально другой, основанной на других принципах, так как процесс перекодировки займёт слишком много времени и это скажется на общей производительности микропроцессора. Также негативно влияет на производительность необходимость поддерживать все команды, которые были использованы в предыдущих версиях микропроцессоров данного семейства. Поддержка этих команд требует дополнительного оборудования, а это не может положительно сказаться на итоговой скорости.

Ещё одним направлением полной аппаратной совместимости является аппаратная реализация старой архитектуры в новой архитектуре, как это сделано в ранних версиях микропроцессора Itanium [16]. В этих микропроцессорах помимо основного ядра с EPIC архитектурой, присутствует также простое x86 ядро, которое и обеспечивает совместимость. Однако производительность такого решения оказалась очень низкой и в дальнейшем для второго поколения микропроцессоров Itanium 2 от этого подхода отказались [38].

Получается, что подходы, описанные выше, обладают рядом недостатков и часто не могут быть использованы для перехода на новые микропроцессорные архитектуры или препятствуют быстрому, дешёвому и эффективному переходу. Эти причины породили ещё один подход к обеспечению совместимости – технологию *двоичной трансляции*. Суть этой технологии заключается в разработке программного обеспечения, которое может коды одной платформы (исходная платформа) исполнять на другой платформе (целевая платформа).

Программное обеспечение, которое умеет исполнять коды одной архитектуры на другой архитектуре, появилось очень давно, можно сказать на заре развития вычислительной техники. Эти решения назывались симуляторами. Ещё в 1973 году в “Искусстве программирования, том 1” Кнут описал такой симулятор. Симуляторы исполняют исходный код последовательно, инструкция за инструкцией, моделируя поведение каждой инструкции исходной платформы, инструкциями целевой платформы. Такой подход является очень медленным[1]. Собственно развитие симуляторов и необходимость сделать их более производительными и вылилось в появления двоичной трансляции. Как правило, двоичным транслятором называют систему, которая умеет транслировать код, то есть по двоичному коду исходной архитектуры строить двоичный код целевой архитектуры, выполняющий в точности такие же преобразования. Также двоичный транслятор умеет переиспользовать уже однажды оттранслированный код. То есть если начнёт исполняться участок исходного кода, который уже оттранслирован, то будет использован именно этот уже оттранслированный код.

В настоящее время область применения двоичной трансляции очень широка. Опишем наиболее важные и интересные из них.

- Полная двоичная трансляция или двоичная трансляция уровня машины. Такие системы используются для обеспечения полной двоичной совместимости. Через двоичный транслятор проходят все коды исходной архитектуры: базовая система ввода-вывода (BIOS), операционная система, драйвера устройств, системное программное обеспечение, пользовательские приложения и т.д. Для пользователя такой системы её существование абсолютно прозрачно, для него вся работа выглядит также, как если бы это происходило на машине с микропроцессором исходной архитектуры. Пользователь может вообще не знать о том, что он работает с микропроцессором другой архитектуры. В качестве примеров микропроцессоров и используемых вместе с ними систем полной двоичной трансляции можно привести микропроцессоры "Crusoe" и "Efficeon" [39] производимых фирмой Transmeta и систему полной двоичной трансляции для них Code Morphing Software [40]. А также отечественные микропроцессоры Эльбрус [44],[45],[46],[47] и систему полной двоичной трансляции для них Lintel [41],[42],[43]. Для обеих, упомянутых выше систем двоичной трансляции, исходной архитектурой является IA-32 [48]. Одним из последних примеров является микропроцессор Denver

[1] Как правило, для моделирования требуется много инструкций целевой платформы. Обычно симуляторы затрачивают на исполнение одной инструкции исходной платформы от десятков до сотен тактов целевой платформы. Это конечно оценка в среднем, так как время симуляции очень сильно зависит от сложности исходной и целевой архитектур.

разработанный компанией Nvidia [112],[113], первое поколение которого выпущено в 2014 года. Исходной архитектурой для Denver является архитектура ARM.

- Двоичный транслятор приложений. Эта группа двоичных трансляторов предназначена для исполнения пользовательских приложений. Операционная система (ОС) для новой архитектуры портируется и двоичный транслятор приложений работает под её управлением. Если в ходе работы ОС происходит запуск приложения в кодах исходной архитектуры, то управления передаётся двоичному транслятору, который уже исполняет данное приложение. Двоичный транслятор может исполнять все непривилегированные команды. Различные системные вызовы исполняются с помощью вызова одного или, если это необходимо, нескольких системных вызовов ОС. Часто для повышения эффективности двоичной трансляции в ОС вносят небольшие доработки. Примерами двоичных трансляторов приложений могут служить IA-32 Execution Layer [49] для исполнения Linux и Windows IA-32 кодов на архитектуре Itanium, а также FX!32 [50],[51],[52] для исполнения Windows IA-32 кодов на архитектуре DEC Alpha.
- Двоичные трансляторы с языков высокого уровня. Двоичные трансляторы с языков высокого уровня схожи с трансляторами приложений описанными выше. Транслятор транслирует код с языка высокого уровня в так называемый bytecode. Этот bytecode не соотвествует никакой реальной архитектуре, реализованной в кремнии. Затем этот байт код с помощью специального программного обеспечения, называемого виртуальной машиной, транслируется в конкретную архитектуру на лету. Наиболее известными представителями являются Java Virtual Machine (JVM) [53], разработанная фирмой Sun Microsystem (сейчас Oracle), Common Language Infrastructure Virtual Execution System (CLI VES) [54] от Microsoft и Dalvik Virtual Machine для операционной системы Android [111]. Отличительной особенностью данных технологий является то, что для новой архитертуры достаточно реализовать транслятор из byte-code и всё существующее программное обеспечение будет работать. Поскольку bytecode не может быть, в силу своей универсальности, просто транслирован в код целевой платформы, то важную роль в производительности bytecode играет эффективный двоичный транслятор, который на выходе будет выдавать качественный результирующий код. Ниже приведены примеры виртуальных машин для bytecode, в которых используют очень развитые двоичные трансляторы: Jalapeno JVM [55],[56], [57], HotSpot JVM [58], JUDO [59].
- Динамическая оптимизирующая трансляция. Такие двоичные трансляторы используются для оптимизации (ускорения) кодов совместимых архитектур. Первый вариант применения – это оптимизирующая трансляции, для которой исходная и целевая архитектура одинаковы. Если программа была скомпилирована без применения высоких

уровней оптимизации или вообще без оптимизаций, то динамическая оптимизирующая трансляции может значительно ускорить исполнение этой программы за счёт применения этих оптимизаций. Такой подход был использован в системе Dynamo[60] для архитектуры PA-RISC, DynamoRio[61] для IA-32, Ispike[62] для Itanuim. Второй вариант применения – это оптимизирующая трансляция, для которой целевая архитектура является развитием исходной архитектуры. Например, в микропроцессоры следующего поколения могут быть добавлены новые векторные инструкции (наподобие MMX и SSE для IA-32 [48]). Использование таких инструкций может значительно ускорить исполнение кода

- Инструментирование и отладочные средства. Часто возникает необходимость получить некоторую информации о процессе исполнения некоторого кода. Это может быть как полная трасса исполнения, так и какая-нибудь специфическая информация, например количество обращений к данной ячейке памяти. Наряду с аппаратной реализацией счётчиков различных событий [17],[48] для этих целей можно использовать двоичную трансляцию. Примерами таких систем являются Shade [63], PIN[104], [105] и valgrind [106], [107].
- Безопасность. Двоичная трансляция также может использоваться для предотвращения различных недоброжелательных атак. Например, для предотвращения атаки с помощью переполнения буфера можно инструментировать код таким образом, чтобы при возврате из процедуры проверялся адрес возврата, и в случае его некорректного значения, программа аварийно завершалась с соответствующей диагностикой. Более подробную информацию по этому вопросу можно почерпнуть из работ [64], [65].
- Виртуализация. Последнее десятилетие активно развивается технология виртуализация, позволяющая на одном компьютере одновременно запускать несколько операционных систем. Для реализации такого запуска на аппаратных платформах без аппаратной поддержки виртуализации или с частичной аппаратной поддержкой виртуализации, одна из операционных систем (host) работает в привилегированном режиме, а остальные (guest) в пользовательском. Из такого разделения возникает необходимость перехватывать привилегированные инструкции guest-системы и моделировать их поведение. Эффективно производить такой перехват можно с помощью двоичной трансляции. Весь привилегированный код guest ОС исполняется под контролем системы виртуализации и, если необходимо, некоторые инструкции транслируются в набор других инструкций, выполняющий эквивалентные действия. Использование двоичной трансляции в системах виртуализации позволяет свести накладные расходы к минимуму [110].

В первых четырёх из описанных выше применений двоичной трансляции крайне важным является не просто транслировать код, но ещё и оптимизировать его. Это связано с тем, что одной из главных задач этих систем является получение высокой производительности.

Для того чтобы двоичный транслятор был эффективным с точки зрения скорости работы транслируемого кода, обычно используется несколько уровней оптимизаторов. Каждый следующий уровень генерирует более оптимальный (с точки зрения времени исполнения) код, но в тоже время затрачивает больше времени на оптимизацию этого кода. Сразу же транслировать код с максимальным уровнем оптимизаций нельзя, так как затраты времени на трансляцию достаточно высоки, а при этом код, который мы оттранслируем, возможно никогда больше не будет исполняться. Для того, чтобы избежать этих, вообще говоря огромных, затрат на трансляцию с большим количеством оптимизаций, при первом исполнении код транслируется очень быстрым транслятором или даже интерпретируется. Но при этом ведётся профилирование этого кода, то есть собирается информация о том, сколько раз исполнилась каждая инструкция. Профилирование ведётся на всех уровнях оптимизаций за исключением самого верхнего (в результате работы которого получается самый быстрый код). На основе профильной информации принимается решение о том, что тот или иной фрагмент кода необходимо перетранслировать с более высоким уровнем оптимизаций [57],[66],[67].

Для того, чтобы двоичный оптимизирующий транслятор был эффективным и оттранслированный код мог по скорости своей работы приближаться к тому же коду, скомпилированному языковым компилятором, необходим мощный оптимизатор верхнего уровня. Этот оптимизатор должен уметь выполнять большинство оптимизаций, которые умеет выполнять языковой компилятор. Однако далеко не каждый алгоритм оптимизации, проводимый оптимизирующим компилятором, может быть просто “скопирован” и перенесён в оптимизатор двоичного транслятора. Это связано со спецификой двоичной трансляции. Пожалуй, главным фактором, ограничивающим алгоритмы, входящие в оптимизирующий двоичный транслятор, является скорость их работы. Эти алгоритмы должны быть очень быстрыми, так как время трансляции кода входит в общее время исполнения программы. Если оптимизатор будет работать очень долго, то весь выигрыш от увеличения скорости работы результирующего кода, может быть перечёркнут временем работы самого оптимизатора. Поэтому многие классические алгоритмы анализа и оптимизаций требуют серьёзной переработки с целью сокращения времени их работы, но при этом необходимо чтобы это не сильно сказалось на качестве результирующего кода.

Ещё одним фактором, из-за которого может потребоваться переработка алгоритмов анализа и оптимизаций, является наличие различных семантических ограничений, введение

которых обусловлено необходимостью повторения точного поведения исходного кода. Ярким примером такого ограничения является проблема восстановления точного контекста[1]. Двоичный транслятор должен быть в состоянии на момент исполнения каждой команды восстановить точный контекст (состояние регистры и памяти) исходного кода, так как в случае возникновения прерывания этот контекст может понадобиться в обработчике прерывания. Такого рода семантические ограничения часто приводят к существенным ограничениям на проводимые оптимизации.

Также в результате разработки двоичного транслятора могут понадобиться специальные оптимизации и алгоритмы, которые либо не нужны, либо их не возможно выполнить в языковых компиляторах. Примерами таких оптимизаций могут служить коррекция профильной информации, использование аппаратной поддержки двоичной трансляции (которое может существенно повысить эффективность результирующих кодов), адаптация к поведению системы[2] и так далее.

Несмотря на упомянутые выше трудности технология оптимизирующей двоичной трансляции позволяет эффективно решить проблему совместимости. Одну из главных ролей в достижении этой цели играет двоичный оптимизирующий транслятор. Однако для достижения высокой эффективности требуется переработка известных алгоритмов анализа и оптимизаций, а также разработка новых алгоритмов анализа и оптимизаций. Это связано с дополнительными требованиями, которые появляются при двоичной трансляции.

1.2. Обзор основных особенностей EPIC архитектур

Архитектуры с явным выраженной параллельностью на уровне команд (EPIC) имеют возможностью исполнять большое количество операций за такт. Несколько операций исполняющихся за один такт будем называть *инструкцией*. В архитектуре “Эльбрус” в одной инструкции можно одновременно исполнить:

---

[1] Это свойство необходимо в системах полной двоичной трансляции и в системе двоичной трансляции приложений. В других применениях двоичной трансляции часто можно опустить это свойство, сославшись на не поддерживаемую функциональность.

[2] Под адапцией к поведению исполняемой программы понимается следующее. В динамическом трансляторе есть возможность применять не всегда корректные преобразования, но при этом необходимо делать динамическую проверку в коде на случай попадания в область некорректного поведения. В случае срабатывания такой проверки динамически производиться перекомпиляция данного участка кода без применения этой оптимизации.

- 6 арифметических операций (из них может быть до 4 чтений из памяти, либо до 2 чтений из памяти и одна запись в память, либо до двух записей в память)[1]
- 3 операции над предикатами
- 1 операцию перехода
- загрузка 4-х 32-битных литерала или 2-х 64-битных

все приведённые классы операций не вытесняют друг друга, то есть в одной команде может одновременно исполниться 6 арифметических операций, 3 операции над предикатами и 1 операция перехода.

В архитектуре Itanium в одной инструкции может исполниться не более шести операций:

- до 6 арифметических операций
- до 2 обращений в память
- до 2 записей в памяти
- до 3 операций перехода
- использовать непосредственно в операции можно только 22-х битные литералы, литералы большой длины необходимо загружать на регистр с помощью специальной операции

Операции работы с предикатами в этой архитектуре явно не присутствуют, но они могут быть смоделированы с помощью специальных арифметических операций сравнения.

Особенностью EPIC архитектур является то, что необходимо явно указывать какие операции будут исполнены в данном инструкции. В силу этого основная работа по достижению высокой производительности ложится на компилятор. Именно компилятор должен расставить операции по инструкциям. Одной из основных задач при компиляции является извлечение максимального параллелизма между операциями. Поскольку может исполняться достаточно много операций одновременно, то, как правило, чем больше параллелизма удастся извлечь, тем быстрее будет исполняться код.

Извлечение параллелизма между операциями настолько важно для высокой производительности EPIC архитектур, что в них реализован целый ряд дополнительных механизмов, помогающих лучше распараллеливать код. Часто оказывается возможным извлечь дополнительный параллелизм из программ путём устранения зависимостей (ограничивающими параллельное исполнение) между операциями. Некоторые зависимости нельзя или невозможно устранить только анализами и оптимизациями имеющимися в компиляторе и для их преодоления в микропроцессорах и вводятся специальные механизмы, которые описаны ниже.

[1] Оценки приведены без учета возможности исполнять упакованные малоформатные операции, двухэтажные вещественные операции и операции асинхронной подкачки данных из памяти

Спекулятивное исполнение [27]. Спекулятивный режим исполнения позволяет исполнять операции заранее, ещё до передачи управления в то место, где исходно располагалась операция, поэтому такую спекулятивность ещё называют спекулятивностью по управлению. Без использования спекулятивности такое предварительное исполнение возможно далеко не всегда. Аргументы операции могут быть корректны только в случае передачи управления в исходный линейный участок. В случае некорректности аргументов операция может выработать прерывание в месте, в котором изначально программа не предполагала получать прерывание, или вообще выработать прерывание, которое никогда не должно вырабатывается исходной программой. Эти эффекты приводят к неопределённому поведению программы или к её аварийному завершению. Иллюстрацией может служить последовательность из проверки указателя на равенство нулю и, в случае неравенства, чтение по этому указателю[1]. Если попытаться осуществить чтение по указателю, до проверки его на равенство нулю, то произойдёт прерывание.[2]

В спекулятивном режиме операция не вырабатывает прерываний. Вместо прерывания в регистр назначения записывается специальный признак, называемый дефектностью, который информирует о том, что операция завершилась аварийно (в не спекулятивном режиме выработалось бы прерывание). Не спекулятивная операция, использующая дефектный аргумент, вырабатывает специальное выделенное прерывание. Спекулятивная операция, использующая дефектный аргумент, вырабатывает дефектность. В случае необходимости в исходном линейном участке можно поставить не спекулятивное использование результата спекулятивной операции, для проявления прерывания.

Предикатное исполнение. Каждая из операций в инструкции может выполняться под некоторым условием. Условие задаётся дополнительным предикатным аргументом, который называется условным аргументом или предикатом операции. Если условный аргумент имеет истинное значение, то операция исполняется так же, как если бы у неё не было предиката. Если условный аргумент ложен, то операция не выполняется и не оказывает никакого эффекта на контекст микропроцессора (регистры, память и т.д.). То есть всё выглядит так, как будто этой операции вообще не было.

Свойство предикатного исполнения позволяет преодолевать ограничения, созданные зависимостями по управлению. В отличие от спекулятивного режима исполнения, зависимости по управлению в этом случае не исчезают, а преобразуются в зависимости по данным. В связи с

[1] Классическая организация работы со списком

[2] В большинстве современных микропроцессоров

этим часто процесс построения предикатного кода называют преобразования зависимостей по управлению в зависимости по данным.

Спекулятивность по данным. Часто зависимостями, ограничивающими параллельность, являются зависимости между операциями записи в память и операциями чтения из памяти, которые потенциально конфликтуют по адресу. То есть для операций не удаётся доказать различие адресов обращений в память по всем возможным путям исполнения. Для возможности исполнения чтения раньше записи в аппаратуре заводится специальный буфер адресов. Также заводятся специальные операции чтения из памяти с занесением в буфер адреса, по которому производится считывание. Для проверки заводятся операции передачи управления в случае отсутствия нужного адреса в буфере. Для всех операций записи в память производится проверка на пересечения адреса, по которому осуществляется запись, с элементами буфера. Все элементы буфера, пересекающиеся с адресом записи, удаляются. В архитектуре “Эльбрус” такой буфер называется DAM (disambiguation access memory). В архитектуре IA-64 – ALAT (Advanced Load Address Table).

Применение спекулятивности по данным осуществляется следующим образом. Операция чтения преобразуется в операцию с занесением в буфер и переносится выше потенциально конфликтующей записи. В месте исходного расположения операции чтения строится операция проверки с передачей управления в специально созданный *компенсирующий код*. В случае, если адреса записи и чтения пересекутся, соответствующий элемент из буфера вычёркивается и операция проверки передаст управление на компенсирующий код. В компенсирующем коде производится перевыполнение операции чтения, и всех операций, зависящих от неё и выполненных до перехода в компенсирующий код. Затем управление возвращается обратно в точку вызова компенсирующего кода.

1.3. Обзор внутреннего представления в компиляторе

1.3.1. Внутреннее представление

Определение. Линейным участком называется блок операций, имеющий только одну точку входа – первую операцию блока и содержащий только одну операцию передачи управления, возможно условную, и эта операция является последней в блоке.

Внутри линейного участка выполнение операций идёт последовательно, без передачи управления, от первой операции до последней операции. Если операция передачи управления безусловная, то из линейного участка возможен только один выход – на целевой линейный участок операции передачи оправления. Если операция передачи управления условная, то возможны два выхода. Первый – на линейный участок, являющийся целевым для операции

передачи управления, в случае, когда условие истинно. Второй – на линейный участок следующий непосредственно за данным, в случае, когда условие ложно. Такой тип выхода называется провалом.

Определение. Графом потока управления (или просто графом управления) называется направленный граф, который представляет все возможные передачи управления между линейными участками. Узлы графа управления соответствуют линейным участкам, а дуги – путям передачи управления.

Узлы и дуги графа управления могут содержать профильную информацию, которая описывает число исполнений линейного участка и число передач управления по дуге.

Определение. Совокупность всех операций, всех линейных участков и графа управления называется промежуточным представлением (Intermediate Representation).

Промежуточное представление служит для отображения всей семантики программы или её части (модуля, процедуры, региона) во время всего процесса работы компилятора. Входные данные компилятора (программы на языке высокого уровня для языкового компилятора, двоичный код для двоичного транслятора) сразу преобразуются в промежуточное представление, и дальнейшая работа идёт уже с ним. Атомарным элементом промежуточного представления является операция. Операция преобразует некоторые входные данные в выходные. В сущности, она соответствует одной операции в микропроцессоре, однако бывают операции, у которых нет аналогов в системе команд микропроцессора, но они служат для отображения дополнительной информации для компилятора.

Определение. Гиперблоком называется подграф графа управления объединённый в один узел с помощью предикатного исполнения. Гиперблок имеет только один внешний вход, и может иметь много выходов.

Определение. Графом потока данных называется надстройка над промежуточным представлением такая, что для каждого аргумента операции отображается, какая операция может выработать этот аргумент. Узлами графа являются аргументы и результаты операций.

Потоковых последователей операции можно разделить на два класса:

- Операция b является *прямым* потоковым последователем операции a, если b может использовать только значение, выработанное a, и это не зависит от пути, по которому мы пришли в a.
- Операция b является *не прямым* потоковым последователем операции a, если рассматриваемый аргумент b может определяться ещё и другими операциями, помимо

a. Значение, используемое в операции b определяется в зависимости от пути в графе управления, по которому мы в неё пришли.

Определение. Компонентой потокового графа называется любой максимально связанный подграф графа потока данных.

При дальнейшем изложении будут использоваться некоторые фрагменты промежуточного представления. Введём следующий формат записи для операций:

```
<имя> [спекулятивность] [[арг.]...] [->результат] [предикат]
```

где

`<имя>` – имя операции. Например, для операции сложения это `ADD` и так далее.

`[спекулятивность]` – признак, что операция исполняется в спекулятивном режиме.

`[[арг.]...]` – аргументы операции. У операции может быть произвольное количество аргументов. Количество аргументов зависит от имени операции. В качестве аргументов может быть константа, обозначаемая своим значением. Также в качестве аргументов могут быть использованы виртуальные регистры. Это объекты, которые используются во внутреннем представлении компилятора. Их свойства соответствуют физическим регистрам, имеющимся в микропроцессоре. Количество виртуальных регистров неограниченно. Виртуальный регистр обозначается `Vsn`, где `n`, натуральное число, его номер, например `Vs15`. Перед генерацией кода отдельной фазой виртуальные регистры распределяются на физические регистры. Также аргументом операции может являться виртуальные предикаты. Их свойства аналогичны свойствам виртуальных регистров. У виртуальных предикатов есть дополнительное свойство, называемое маской. Истинная маска обозначает, что для управления выполнением операции будет взято само значение предиката, а ложная маска означает, что будет взято логическое отрицание значения предиката. Виртуальный предикат обозначается `Pn[M]`, где `n`, натуральное число, его номер, в квадратных скобках описывается маска предиката, символ `T` (от английского true) обозначает истинную маску, символ `F` (от английского false) обозначает ложную маску.

`[->результат]` – результат операции. Результатом операции может являться либо виртуальный регистр, либо виртуальный предикат.

`[предикат]` – управляющий предикат операции (условный аргумент). Предикат, который определяет, выполнится операция или нет.

Для иллюстрации приведём несколько примеров.

```
ADD  Vs2  3  ->  Vs5
```

эта запись означает операцию сложения виртуального регистра под номером два, с константой три и результат записывается в виртуальный регистр под номером пять. Данная операция не является спекулятивной и она выполняется безусловно. Ещё один пример:

```
ADD s Vs20  Vs14  -> Vs8  P3[F]
```

эта запись означает операцию сложения виртуального регистра под номером двадцать с виртуальным регистром под номером четырнадцать, а результат записывается в виртуальный регистр под номером восемь. Операция является спекулятивной. Операция выполняется только, если виртуальный предикат под номером три является ложным.

В Таблица 1 приведено описание операций, которые будут использоваться в дальнейшем изложении, а также время их исполнения в тактах (количество тактов через которое готов результат операции и его можно использовать в других операциях). Время указывается по отношению ко всем потребителям, если не оговорено противное. Выбранные времена не описывают некоторую существующую архитектуру, но они выбраны близкими к тем временам, которые встречаются в реальных архитектурах.

Имя операции	Краткое описание операции	Время исполнения операции в тактах
ADD	сложение двух аргументов	1
SUB	вычитание из первого аргумента второго	1
MOV	пересылка первого аргумента в результат	1
SHL	сдвиг влево первого аргумента на значение второго аргумента	1
OR	побитовое логическое “или” двух аргументов	1
AND	побитовое логическое “и” двух аргументов	1
CMP	сравнение на равенство двух аргументов	2
CMPL	сравнение на меньше двух аргументов	2
MUL	умножение двух аргументов	4

LD	чтение из памяти по адресу равному сумме значений первого и второго аргумента	3[1]
ST	запись в память значения третьего аргумента, по адресу равному сумме значений первого и второго аргументов	0[2]
BRANCH	операция передачи управления на метку	1[3]
LD.lock	операция чтения из памяти плюс занесение в таблицу DAM	3/0[4]
LD.chk	операция проверки в таблице DAM	1

Таблица 1. *Описание операций и времени их исполнения.*

1.3.2. Граф зависимостей

Над операциями промежуточного представления можно построить граф зависимостей.

Определение. Граф зависимостей – это ориентированный ациклический граф, узлами которого являются операции промежуточного представления. Дуга соединяет две операции тогда и только тогда, когда операция – предшественник дуги по некоторой причине должна выполниться не позже операции – последователя дуги. Дуги графа зависимостей также будем называть *зависимостями*.

Все причины появления зависимостей между операциями определяются алгоритмом построения графа зависимостей, который фактически анализирует каждую пару операций на предмет наличия между ними зависимости того или иного типа [18]. Фактически граф зависимостей вводит частичный порядок между операциями. Ниже приведены несколько причин, по которым могут упорядочиваться операции:

---

[1] В современных микропроцессорах время исполнения операции чтения из памяти зависит от того находится ли в данная ячейка данных в кэше или нет, и может колебаться от нескольких тактов до нескольких сотен тактов. Этот фактор достаточно трудно учитывать при проведении оптимизаций и везде далее, если не оговорено противное, будем считать, что читаемая ячейка находится в кэше первого уровня и время исполнения операции равно трём тактам.

[2] Так как операция ST не вырабатывает результата, то исходное определение времени исполнения теряет свой смысл, однако будем считать, что операции чтения из памяти доступно записанное значение в том же такте. Порядок выполнения записей в память и чтений из памяти внутри одной команды определяется порядком устройств на которых исполняются операции.

[3] В следующем такте выполнение продолжается с метки

[4] Время чтения из памяти такое же как и операции LD – три такта; время занесения в таблицу DAM – ноль тактов.

- Потоковые зависимости – это использование результата одной операции в качестве аргумента другой.
- Антизависимости – это переопределение ресурса занимаемого аргументом одной операции, под результат другой операции
- Зависимость по результату или output-зависимость – переназначение ресурса, отданного под результат одной операции, под результат другой.
- Предикатные зависимости – потоковая зависимость в случае, когда аргументом является условный предикат операции
- Зависимость между обращениями в память по потенциально пересекающимся адресам (кроме случая чтение-чтение). Термин потенциально пересекающиеся адреса означает, что не удалось доказать, что адреса не пересекаются по всем путям программы.
- Зависимости по управлению – это зависимости между операциями передачи управления и другими операциями. Например, операция находящаяся в линейном участке должна выполнить до операции передачи управления.

Это далеко не все типы зависимостей, которые могут возникнуть. Набор всех возможных причин, по которым операции необходимо упорядочить, определяется несколькими факторами. В первую очередь архитектура микропроцессора может вводить те или иные ограничения на порядок операций. Например, в архитектурах поддерживающих предикатное исполнение, присутствует особый тип потоковой зависимости – предикатная потоковая зависимость по условному аргументу. Ещё одним фактором, который влияет на набор зависимостей, являются условия, в которых строится граф зависимостей. Под условиями понимается компилятор, производящий построение, и опции, с которыми он запущен. Так, например, специфическими для двоичной трансляции являются зависимости между всеми операциями записи в память.

Граф зависимостей является взвешенным графом. Каждая дуга графа зависимостей имеет *длину*, которая определяет через сколько тактов после того, как выполнился предшественник дуги, может начаться выполнение последователя дуги. Например, большинство операций целочисленной арифметики исполняются за один такт, а операции вещественной арифметики – за несколько тактов. Так как граф зависимостей является ациклическим, найдётся хотя бы одна вершина, в которую не входит ни одна дуга, и хотя бы одна вершина, из которой не выходит ни одной дуги. Добавим к графу две вершины называемые `ENTER` и `END`. Построим дуги, соединяющие `ENTER` со всеми вершинами, в которые не входит дуга, а также дуги, соединяющие все вершины, из которых не выходят дуги, с узлом `END`. В дальнейшем будем рассматривать только такие расширенные графы зависимости.

Определение. Длиной пути в графе зависимостей называется сумма длин всех дуг входящих в этот путь.

Определение. Высотой графа зависимостей называется максимальная длина пути ведущего от ENTER-а к END-у.

Фактически, высота графа зависимостей есть минимальная длительность исполнения данного фрагмента кода микропроцессором заданной архитектуры, сократить которую уже невозможно. Однако, если количество операций, которые могут выполнится в одной инструкции, достаточно велико, то высота графа зависимостей и будет характеризовать то время, за которое исполнится данный код.

Определение. Критическими путями называются пути в графе зависимостей, имеющие максимальную длину.

Определение. Временем раннего планирования узла графа зависимостей называется максимальная из длин всех путей, ведущих от узла ENTER к данному узлу.

Определение. Если взять время раннего выхода из узла вычесть из него длину пути от некоторого узла графа до этого выхода и взять минимум этих величин по всем путям и всем выходам, то полученная величина называется временем позднего планирования этого узла графа.

Время раннего планирования операции соответствует первому такту, в который можно спланировать операцию. Раньше спланировать её нельзя, так как не будут выдержаны задержки в графе зависимостей. Время позднего планирования операции это такой такт, до которого (включительно) необходимо спланировать операцию, чтобы не изменилась общая длина вычислений.

1.3.3. Особенности графа зависимостей в динамическом двоичном трансляторе

При построение графа зависимостей в динамическом двоичном трансляторе на промежуточном представлении полученном из двоичного кода возникает гораздо больше зависимостей, чем при построение графа зависимостей на представлении полученном из языка высокого уровня. Большинство из этих зависимостей ложные, то есть на самом деле этих зависимостей нет, но транслятор не может доказать их отсутствие и поэтому вынужден их строить. Возникновение большего числа зависимостей в двоичном трансляторе является следствием того, что в двоичном коде (а следовательно и в промежуточном представлении полученном из этого кода) содержится гораздо меньше информации о семантике исходной

программы, чем в коде языка высокого уровня. Опишем подробнее основные причины возникновения большего числа зависимостей и типы зависимостей возникающих зависимостей.

Первой причиной является отсутствие в двоичном коде информации об объектах, по которым производятся обращения в память. Рассмотрим пример кода на языке Си:

```
void f(int *a, int *b)
{
  ...
  a[i] = t;
  r = b[i];
  ...
}

void g( )
{
  a = malloc( SIZE);
  b = malloc( SIZE);
  f(a, b);
}
```

Компилятор с языка Си может определить, что в функцию `f` подаются параметры полученные двумя разными вызовами функции `malloc`, и следовательно эти две области памяти не пересекаются. Получив такую информацию компилятор может переставлять между собой обращения к массиву `a` и обращения к массиву `b`. После компиляции рассмотренного фрагмента в двоичный код данная информация теряется. Двоичный транслятор в некоторых случаях может определить, что объекты `a` и `b` получены как результат вызова функции с названием `malloc`, однако даже в этом случае он не может делать никаких предположений относительно её результата – это может быть пользовательская функция, у которой просто название совпадает с общепринятым `malloc`-ом, а результат её совсем другой. Итак в результате отсутствия информации об объектах, по которым производятся обращения в память, в двоичном трансляторе появляется больше зависимостей между обращениями в память.

Перейдём к рассмотрению второй причины. В динамическом двоичном трансляторе время трансляции кода входит в общее время исполнения программы, поэтому оптимизируются только самые горячие участки кода. В силу этого инициализирующие записи в некоторые переменные (регистры, ячейки памяти) могут не попасть в транслируемую область, и

транслятор ничего не будет знать о взаимосвязи между этими переменными. Рассмотрим следующий пример промежуточного представления:

```
MOV  0x80000000  ->  Vs1
MOV  0x80000010  ->  Vs2
...
<много другого кода>
...
ST  Vs1  0   Vs3
LD  Vs2  0  ->  Vs4
```

Если в транслируемую область попали операции ST и LD, но не попали операции MOV, то транслятор не сможет доказать, что операции ST и LD независимы и между ними будет построена зависимость. Итак в результате ограниченности области трансляции в динамическом двоичном трансляторе появляется больше зависимостей между обращениями в память.

Третья причина – маленькое количество регистров. Данное свойство присуще не всем микропроцессорным архитектурам. Однако в рамках данной работы исследование проводилось для двоичного транслятора с исходной архитектурой x86 и результирующей “Эльбрус”, а в архитектуре x86 относительно мало регистров. Маленькое количество регистров приводит к необходимости их часто переиспользовать. Такое переиспользование порождает лишние антизависимости, которых нет по семантике исходной программы. В архитектуре “Эльбрус” гораздо больше регистров, и следовательно нет необходимости в таких антизависимостях и от них можно избавиться.

Четвёртая причина – недостаток информации о времени жизни регистров. В подавляющем большинстве случаев время жизни регистра заканчивается на последнем использовании. Однако во многих архитектурах, в том числе и в x86, есть возможность прочитать значения регистров в обработчике прерывания, в том числе и в обработчике асинхронного прерывания, которое может прийти практически в любой момент. В силу этого свойства время жизни регистра заканчивается только тогда, когда этот регистр будет перезаписан, а не в момент последнего использования. Рассмотрим пример изображённый на Рис. 1 и покажем как недостаток информации о времени жизни регистров может привести к ложным предикатным зависимостям.

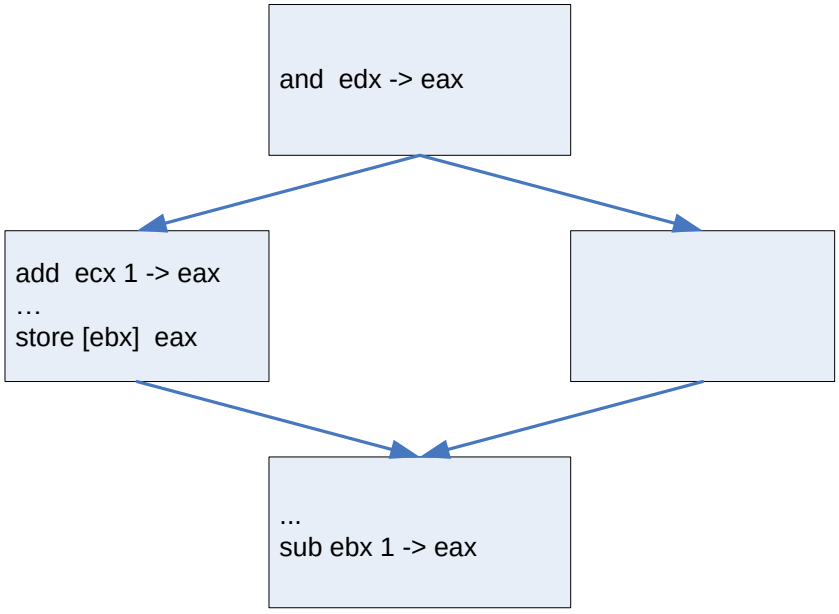

Рис. 1. *Пример как недостаток информации о времени жизни регистров приводит к возникновению ложных предикатных зависимостей.*

Время жизни результата операции `add`, по семантике программы, заканчивает на операции `store`, так как это последнее использование. Однако за счёт того, что регистр может потенциально быть использован в обработчике прерывания необходимо продлить время его жизни до операции `sub`. Заметим также, что до операции `sub` доживает также результат операции `and` (по правой ветви). Предположим теперь, что мы хотим переместить операцию `add` в верхний узел. Если бы нам не надо было продлять время жизни регистра, то можно было бы поступить следующим образом. Заменить результат операции `add` на другой, нигде больше не используемый регистр, а также все использования результата операции `add` заменить на этот новый регистр. После такого преобразования можно перенести `add` в верхний узел не ставя эту операцию под предикат ветвления, при этом бы программа оставалась корректной. Однако в силу того, что результат операции `add` используется в начале нижнего узла, а также в силу того, что в начале нижнего узла регистр eax может быть определён как операцией `add` так и операцией `and`, мы не можем заменить результат `add` на новый регистр. Следовательно при переносе операции `add` в верхний узел её надо поставить под предикат ветвления, создав таким образом “не обязательную” предикатную зависимость.

Выше были описаны четыре причины, по которым при трансляции двоичного кода возникает больше зависимостей, чем в случае компиляции с языка высокого уровня. Это далеко не все причины, однако целью данного раздела является не классификация всех возможных причин и отличий, а демонстрация того, что при трансляции двоичного кода возникает больше

“не обязательных” зависимостей, чем в случае компиляции с языка высокого уровня. Таким образом в динамическом двоичном трансляторе для достижения высокой скорости работы результирующего кода необходимо уделять большое внимание алгоритмам и методам борьбы с ложными зависимостями.

1.4. Ускорение результирующего кода за счёт сокращения длины критических путей

Как уже отмечалось выше, одной из главных задач оптимизирующего компилятора для EPIC архитектур является распараллеливание кода. Часто оказывается выгодным даже построить лишнею операцию, при этом сократив высоту вычислений. Для новой операции найдётся свободное место в одной из инструкций, а высота вычислений уменьшится, и код будет работать быстрее. В двоичном оптимизирующем трансляторе эта задача ещё более актуальна в силу большого количества ложных зависимостей, мешающих распараллеливанию. Существует ряд методов способных сократить длину критических путей. Мы разделим рассмотрение этих методов на два класса. Первый – методы применяемые в ациклических областях, второй – методы применяемые в циклических областях.

1.4.1. Ациклические области

1.4.1.1. Классические оптимизации с точки зрения сокращения длины критических путей

Многие классические оптимизации сокращают длину критических путей, но в силу своей важности и различия между собой получили собственные названия [33], [34]. Примерами могут служить построение предикатного кода (if-conversion), advanced unroll и многие другие. Рассмотрим несколько примеров более подробно.

Первый пример оптимизация основанная на применение различных *алгебраических тождеств*. Рассмотрим следующую последовательность операций:

```
E=0  ENTER
E=0  ADD   Vs1  Vs2 -> Vs3
E=1  SUB   Vs3  Vs2 -> Vs4
E=1  END
```

Она эквивалентна следующей последовательности[1]:

[1] При условии, что регистр `Vs3` больше нигде не используется.

```
E=0  ENTER
E=0  ADD   Vs1  0   -> Vs4
E=0  END
```

Здесь запись `E=1` перед операцией обозначает время раннего планирования операции. В результате такого преобразования, во-первых, уменьшилось количество операций. Во-вторых, уменьшилась и высота вычислений с двух до одного такта.

Часто среди тождественных преобразований выделяют преобразования, которые получили название *понижения силы операций* (operator strength reduction) [22]. Простейшим примером может являться преобразование 2×x в x+x или в x<<1. Развитием этой техники является применение понижение силы операций к индуктивным переменным. Часто в литературе понижением силы операций называют именно оптимизацию индуктивных переменных. Классическим примером применения понижения силы операций к индуктивным переменным является работа с массивом в цикле. Рассмотрим следующий фрагмент кода:

```
K(int * a, int x, int s)
i = 0;
do
{
    a[x+i*s] = 0;
    i++;
} while ( i < 100);
```

промежуточное представление для этого фрагмента будет выглядеть следующим образом

```
MOV    0        -> Vs1
loop:
E=0  MUL    Vs1  s   -> Vs2
E=4  ADD    Vs2  x   -> Vs3
E=5  SHL    Vs3  2   -> Vs4
E=6  ST     a    Vs4  0
E=0  ADD    Vs1  1   -> Vs1
E=1  CMPL   Vs1  100 -> P1
E=6  BRANCH loop           P1[T]
```

```
E=6  BRANCH exit            P1[F]
exit:
```

после применения преобразования получим следующее

```
SHL    x    2   -> Vs5
SHL    s    2   -> Vs6
MOV    Vs5      -> Vs1
MUL    s    400 -> Vs7
ADD    Vs7  Vs5 -> Vs8
loop:
E=0  ST     a    Vs1  0
E=0  ADD    Vs1  Vs6 -> Vs1
E=1  CMPL   Vs1  Vs8 -> P1
E=3  BRANCH loop            P1[T]
E=3  BRANCH exit            P1[F]
exit:
```

в результате высота одной итерации цикла уменьшается с семи до четырёх тактов.

В архитектурах с поддержкой предикатных вычислений имеется возможность *преобразовывать зависимости по управлению в зависимости по данным (предикатные зависимости)* [80]. Такое преобразование может быть осуществлено с помощью техники получившей название if-conversion. Одним из многих положительных эффектов данного преобразования является то, что имеется много методов устранения предикатных зависимостей и следовательно это позволяет устранить исходную зависимость по управлению. Тема построения предикатного кода является достаточно независимой и объёмной. Её обсуждение выходит за рамки этой работы, и в дальнейшем изложении, когда это будет необходимо, мы либо будем считать, что предикатный код уже построен, либо будем просто говорить об алгоритме построения предикатного кода не приводя конкретные детали его реализации. Более подробно эта тема описана в работах [24], [80], [102].

1.4.1.2. Специализированные преобразования для сокращении длины критических путей

Для оптимизаций описанных в предыдущем разделе уменьшение высоты критических путей является одним из действий, можно даже сказать побочным результатом. Помимо него, например, может сокращаться количество операций.

Существует также ряд преобразований единственным эффектом от которых является сокращение длины критических путей. Более того, многие из этих преобразований могут достраивать новые операции, увеличивая их общее количество. Тем не менее эффект от сокращения длины критических путей часто оказывается более весомым и результирующий код работает быстрее. Рассмотрим пример промежуточного представления:

```
E=0  ENTER
E=0  CMP   Vs1  0    ->  P0
E=2  SUB   Vs2  2    ->  Vs3  P0[T]
E=3  AND   Vs3  2    ->  Vs4  P0[T]
E=3  ST    [mem a]  Vs3
E=4  ST    [mem b]  Vs4
E=4  END
```

В данном случае можно избавиться от предикатной зависимости для операции `SUB` проведя следующее преобразование:

```
E=0  ENTER
E=0  CMP   Vs1  0    ->  P0
E=0  SUB   Vs2  2    ->  Vs5
E=2  MOV   Vs5       ->  Vs3  P0[T]
E=2  AND   Vs5  2    ->  Vs4  P0[T]
E=3  ST    [mem a]  Vs3
E=3  ST    [mem b]  Vs4
E=3  END
```

В результате высота вычислений сократилась с пяти тактов до четырёх.

Такие эквивалентные преобразования единственной целью которых является сокращение длины критических путей мы будем называть *преобразованиями, разрывающими зависимости,* или просто *разрывом зависимостей*. Здесь мы не будем рассматривать подробно

способы разрыва зависимостей, а ограничимся лишь этим примером. В последующих главах они будут более подробно описаны.

Методы разрыва зависимостей можно разделить на две группы. Первая – это преобразования, не требующие построения новых операций. Такие преобразования практически всегда дают положительных эффект[1] и их можно применять всегда, когда это возможно. Вторая группа преобразований, разрывающих зависимости – это преобразования требующие построения новых операций. Такие преобразования, в отличие от первой группы, применяемые всегда, когда это возможно, могут оказать отрицательный эффект на скорость работы результирующего кода. Это связно с тем, что количество операций, выполняемых микропроцессором за один такт, ограничено, и построение большого количества новых операций может привести к тому, что потребуется большее количество тактов для исполнения всех операций, чем требовалось изначально. В связи с этим возникает необходимость разработки алгоритмов, которые бы учитывали негативный эффект от разрыва зависимостей и осуществляли бы преобразования, только тогда, когда это необходимо. Такие алгоритмы, которые достигают минимально возможную высоту графа зависимостей, но при этом разрывают не все зависимости, будем называть *алгоритмами минимизации высоты графа зависимостей*.

1.4.2. Циклические области

Перейдём к рассмотрению второго класса методов сокращения длины критического пути – случаю циклических областей, или просто циклов. В силу особой регулярной структуры циклов к ним применим класс оптимизаций получивший название *конвейеризация циклов*. Этот метод является одним из самых эффективных методов увеличения производительности в циклах на одном процессоре[2]. В данном случае остаются актуальными требования и ограничения, вызванные спецификой двоичной трансляции, которые были приведены для ациклических областей, а именно. В циклах также присутствует большое количество зависимостей, которые можно разорвать, поэтому алгоритм конвейеризации должен быть

[1] В случае снятия предиката с операции чтения из памяти возможен негативный эффект связанный с тем, что кэш память используется менее эффективно. Однако предсказать на этапе компиляции влияние этого фактора очень сложно.

[2] Ещё одним эффективным методом увеличения производительности в циклах является распараллеливание цикла на несколько независимых потоков в многоядерных архитектурах. Однако для двоичной трансляции применимость этого метода сильно ограничена. Это связано с необходимость обеспечения точной семантики исходной программы, и следовательно упорядочиванием всех операций имеющих побочный эффект (например записей в память). Однако, если программа изначально уже была распараллелена на несколько потоков, то двоичный транслятор также сможет её исполнять в таком же количество потоков.

интегрирован с техниками разрыва зависимостей. В силу того, что транслятор является динамическим, предъявляются жёсткие требования к скорости работы алгоритма.

1.4.2.1. Основные идеи конвейеризации циклов

Основная идея *конвейеризации циклов* (также называемой *накруткой циклов*) следующая: для увеличения параллельности в вычислениях возможно совместить исполнение нескольких логических итераций цикла в одной физической итерации цикла. То есть операции со следующей итерации могут начать выполняться ещё до того, как предыдущая итерация будет закончена. На Рис. 2 изображён простой цикл подсчёта количества элементов списка. Приведено промежуточное представление и соответствующая циклу расстановка операций по тактам.

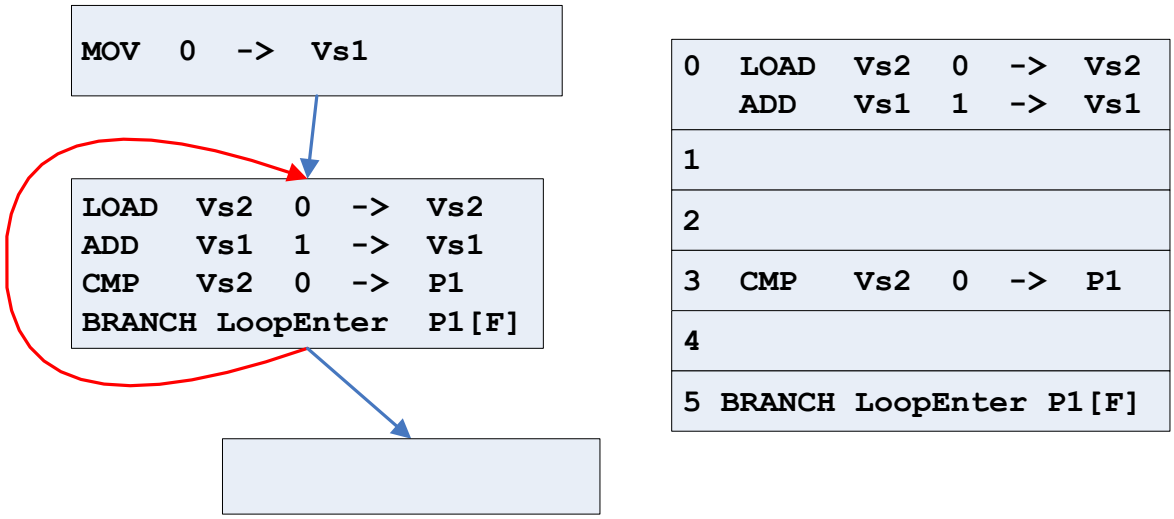

Рис. 2. *Пример цикла до конвейеризации.*

Теперь проведём следующее преобразование. Перенесём операцию `LOAD` вверх с дублированием. Одна копия попадает в верхний узел, вторая проносится по обратной дуге, переводится в спекулятивный режим и становится перед переходом по обратной дуге. Результат преобразования и расстановка операций по тактам приведена на Рис. 3.

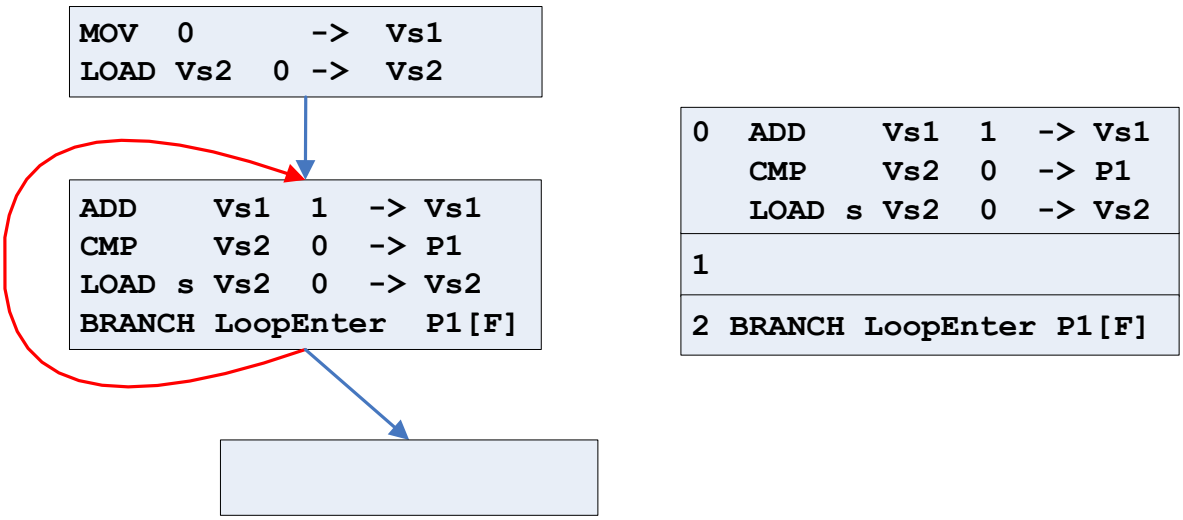

Рис. 3. *Цикл после конвейеризации одной операции.*

Как видно из рисунков, в результате преобразования размер цикла сократился с шести тактов до трёх.

В приведённом примере произведена конвейеризация операции `LOAD`. Фактически, после преобразования, на очередной итерации цикла исполняется операция (`LOAD`) со следующей итерации. Такое наложение итераций и называется конвейеризацией цикла.

1.4.2.2. Аппаратная поддержка конвейеризации циклов

В силу важности и эффективности конвейеризации циклов для получения быстрого результирующего кода, рядом исследователей были предприняты усилия для разработки различных техник аппаратной поддержки конвейеризации. Самым важным среди них является аппарат *вращающихся регистров*.

Пусть в цикле, который мы хотим конвейеризировать, имеются две операции, при этом вторая операция использует результат первой. В такой ситуации существует антизависимость (чтение-запись) идущая от второй операции к первой и реализуемая через одну итерацию. Эта зависимость препятствует переносу первой операции более чем на одну итерацию без переноса второй операции. Избавиться от этой зависимости можно разорвав её с построением новой операции. Однако в таком случае будет увеличиваться количество операций в цикле.

Эффективным методом решения описанной проблемы является техника вращающихся регистров, впервые предложенная в проекте Cydra 5 [96], [76]. Позднее похожий механизм был реализован в микропроцессорах семейства Itanium [16], [17]. Аналогичная аппаратная поддержка имеется и в микропроцессоре “Эльбрус” [44]. Суть механизма вращающихся регистров заключается в том, что часть регистрового файла адресуется не по абсолютным

номерам регистров, а по относительным. В микропроцессоре имеется специальный регистр называемый базой вращающихся регистров. При чтении значения из регистра абсолютный номер вычисляется как сумма базы вращающихся регистров и относительного номера регистра[1]. Также в аппаратуре присутствует специальная операция, которая уменьшает базу вращающихся регистров на определённую величину.

Теперь рассмотрим как можно применять вращающиеся регистры при конвейеризации циклов. Вместе с переходом по обратной дуге ставится операция уменьшения базы вращающихся регистров на единицу. Все используемые регистры в цикле переносятся на вращающиеся регистры. Допустим, у нас имеется пара операций: первая пишет во вращающийся регистр с номером пять, а вторая читает его. При переносе по обратной дуге первой операции необходимо уменьшить на единицу номер регистра в который она пишет (и номера читаемых регистров). Но это реализуется автоматически, так как вместе с переходом выполняется операция уменьшения базы вращающихся регистров. В результате получается, что на каждой новой итерации цикла регистр записываемый первой операцией отличается от регистра читаемого второй и никакой антизависимости не возникает.

Таким образом поддержка в микропроцессоре вращающихся регистров позволяет полностью избавиться от проблемы возникновения ложных антизависимостей между операциями с соседних итераций. Вследствие этого не требуется построение новых операций для разрыва ложных зависимостей и это значительно повышает эффективность конвейеризации циклов.

1.5. Постановка задачи

В рамках работы над двоичным оптимизирующем транслятором из архитектуры x86 в архитектуру Эльбрус была поставлена задача существенного поднятия скорости работы двоично-транслированных кодов до уровня, сравнимого с кодами, полученными языковым компилятором. В результате исследования было выявлено, что при трансляции двоичных кодов возникает большое количество зависимостей между операциями поддающихся разрыву. Была поставлена задача разработки новых более эффективных по качеству результирующего кода и более быстрых по скорости работы методов сокращения длины критического пути в ациклических областях. Также требовалось разработать схему взаимодействия этих методов с другими оптимизирующими преобразованиями. Основная трудность тут заключается в том, что результаты работы алгоритма минимизации критического пути зависят от результатов работы других оптимизирующих преобразований, а результаты работы других оптимизирующих

[1] Естественно все вычисления ведутся по модулю количества вращающихся регистров

преобразований зависят от результатов работы алгоритма минимизации критического пути. Таким образом получается “замкнутый круг”, который необходимо разорвать.

Эффективным способом поднятия производительности в циклических областях для EPIC архитектур является техника конвейеризации циклов. Была поставлена задача разработки нового алгоритма конвейеризации циклов, который бы позволил максимально полно использовать всю ширину целевой архитектуры. Как было сказано выше, при трансляции двоичных кодов возникает большое количество зависимостей поддающихся разрыву. Поэтому важной задачей являлась интеграции алгоритмов разрыва зависимостей и алгоритма конвейеризации циклов.

Одним из важнейших способов поднятия эффективности конвейеризации циклов является использования аппаратной технологии вращающихся регистров. Основной трудностью является то, что ни конвейеризация циклов, ни технология вращающихся регистров никогда ранее не использовались в двоичных оптимизирующих трансляторах. Также необходимо было разработать механизм взаимодействия технологии вращающихся регистров с техникой восстановления точного контекста и предложить аппаратную поддержку повышающую эффективность этого взаимодействия.

При разработке алгоритмов необходимо было использовать положительные результаты уже существующих подходов, а также добиться повышения эффективности за счёт снятия основных ограничений, присущих уже разработанным и используемым методам. Также было необходимо обеспечить высокую скорость работы алгоритмов, так как это является обязательным требованием для динамического двоичного оптимизирующего транслятора. Все разработанные алгоритмы и методы должны удовлетворять всем требованиям предъявляемым к промышленным двоичным оптимизирующим трансляторам: иметь высокую скорость работы, обеспечивать сохранение точной семантики исходной архитектуры, обладать высочайшим уровнем надёжности. Основными этапами решения поставленной задачи являются:

- разработка быстрых алгоритмов минимизации высоты графа зависимостей использующих методы разрыва зависимостей без построения новых операций
- разработка быстрого алгоритма минимизации высоты графа зависимостей использующих методы разрыва зависимостей с построения новых операций и одновременно производящего разрывы всех возможных типов зависимостей
- разработка методов интеграции алгоритмов минимизации высоты графа зависимостей с другими оптимизирующими преобразованиями для повышения эффективности результирующего кода

- разработка алгоритма конвейеризации циклов и использование в нём механизма вращающихся регистров
- интеграция в алгоритм конвейеризации циклов различных методов разрыва зависимостей
- развитие методов аппаратной поддержки для повышения эффективности использования механизма вращающихся регистров в двоичном оптимизирующем трансляторе

1.6. Выводы

1. В данной главе приведено краткое освещение проблематики двоичной трансляции, описаны основные сферы применения этой технологии, а также требования и ограничения, которые необходимо учитывать.
2. Произведён обзор EPIC архитектур и обозначены основные направления оптимизации результирующего кода для этих архитектур.
3. Описаны методы ускорения результирующего кода за счёт сокращения длины критических путей в ациклических и циклических областях.
4. Поставлена задача исследования, которая позволит существенно повысить качество результирующих кодов получаемых оптимизирующим двоичным транслятором, включённым в динамический двоичный транслятор, и следовательно повысить скорость работы программ запущенных под двоичным транслятором.

2. Сокращение длины критических путей в ациклических областях без построения новых операций

2.1. Методы разрыва зависимостей без построения новых операций

Существуют ряд простых методов уменьшения высоты графа зависимостей без дополнительного создания новых операций.

Первый метод разрыва зависимостей, который мы рассмотрим, – это *переименование регистров* [25], [26], [36]. С помощью переименования можно разорвать антизависимости и зависимости по результату. Суть переименования заключается в назначении некоторой компоненте потокового графа нового регистра. Рассмотри пример:

```
E=0  ENTER
E=0  SHL    Vs1  1    -> Vs2
E=1  ADD    Vs2  Vs3  -> Vs4
E=1  SUB    Vs5  Vs6  -> Vs2
E=2  ST     Vs7  0    Vs2
E=2  END
```

В этом примере имеется антизависимость идущая от операции `ADD` к операции `SUB` и получается, что высота вычислений в этом примере равна трём тактам. Если же для результата операции `SUB` произвести переименование, то получим следующее:

```
E=0  ENTER
E=0  SHL    Vs1  1    -> Vs2
E=1  ADD    Vs2  Vs3  -> Vs4
E=0  SUB    Vs5  Vs6  -> Vs9
E=1  ST     Vs7  0    Vs9
E=1  END
```

Теперь уже операция `SUB` не зависит от операции `ADD` и высота вычислений после переименования равна двум тактам.

Ещё одним методом уменьшения высоты графа зависимостей является *использование спекулятивности* [24], [28]. Например, если в обрабатываемом регионе имеется всего одно определение некоторого регистра и это определение стоит под предикатом, то можно снять этот

предикат, но при этом необходимо перевести операцию в спекулятивный режим. Рассмотрим следующий пример:

```
E=0  ENTER
E=0  CMP   Vs1  0   -> P1
E=2  ADD   Vs2  Vs3 -> Vs4  P1[T]
E=3  ST    Vs5  0   Vs4     P1[T]
E=3  END
```

Можно провести следующее преобразование:

```
E=0  ENTER
E=0  CMP   Vs1  0   -> P1
E=0  ADD s Vs2  Vs3 -> Vs4
E=2  ST    Vs5  0   Vs4     P1[T]
E=2  END
```

В результате преобразования высота вычислений уменьшиться на один такт.

Если снять предикат полностью не возможно, то можно использовать некоторый промежуточный вариант, то есть предикат истинный всегда, когда истинен полный предикат, но также ещё истинный в некоторых случаях, когда полный предикат ложен. Наиболее адекватным названием этого метода на русском языке, является *“использование частичных предикатов”*. Оригинальное название – predicate speculation. Этот метод был предложен в работах [24], [35] и является развитием техники применения спекулятивности.

Метод разрыва зависимостей с помощью переименования регистров, по крайней мере, не ухудшает результирующий код. Поэтому его можно и нужно применять всегда, когда это возможно. В связи с этим, в дальнейшем изложении, мы не будем останавливаться на том, как эта оптимизация влияет на другие оптимизации. Также при описании различных алгоритмов мы будем предполагать, что произведено полное переименование, то есть каждой компоненте потокового графа назначен свой уникальный регистр.

Использование спекулятивных вычислений может приносить некоторые ухудшения, связанные с тем, что спекулятивные операции считывания из памяти протравливаю кэш данных. Может оказаться так, что спекулятивное считывание из памяти не будет в дальнейшем использовано, но в тоже время вытеснит из кэша другие нужные данные. Ещё один случай ухудшения может возникнуть, когда результат считывания из памяти используется

спекулятивно. Если в реальном исполнении это использование не должно было исполниться, то возможно получение блокировок для ожидания значения считывания из памяти. Однако заранее факт того, что операция чтения из памяти не попадёт в кэш, выяснить достаточно затруднительно, поэтому, как правило, эту оптимизацию стараются применять, только когда высота вычислений в результате применения сокращается.

2.2. Обзор существующих методов минимизации высоты графа зависимостей

2.2.1. Переименование регистров

Переименование регистров является одним из самых простых методов разрыва зависимостей. Как было сказано выше оно, по крайней мере, не ухудшает результирующий код. Поэтому его можно и нужно применять всегда, когда это возможно. В работе [25] приводится следующий алгоритм переименования регистров. На первом шаге строится граф потока данных. На втором шаге берутся все компоненты потокового графа. На заключительном, третьем шаге, каждой компоненте потокового графа назначается новый виртуальный регистр.

2.2.2. Использование частичных предикатов

Опишем алгоритм использования частичных предикатов предложенный в работе [35]. Алгоритм работает в два прохода снизу вверх. В специальном массиве хранится информация, называемая временем жизни, о том, под каким предикатом жив каждый регистр или ячейка памяти. Первоначально информация о времени жизни доступна только на выходах, затем она вычисляется в процессе обхода для каждой операции. Все операции стоящие под предикатом являются кандидатами для использования частичных предикатов.

На первом проходе условный аргумент операции p заменяется более широким предикатом q. Более широкий предикат означает, что если p истинен, то q тоже истинен. Предикат расширяется настолько насколько это возможно, до тех пор, пока исходная операция не начинает переписывать живой регистр или ячейку памяти. Второй проход наоборот выборочно заменяет более широкий условный аргумент операции на более узкий. Предикат может сузиться частично, либо полностью (то есть до оригинального предиката). Второй проход предназначен для того, чтобы бороться с негативными эффектами на производительность, которые могут возникнуть при расширении предиката.

2.3. Схема работы двоичного транслятора для архитектуры “Эльбрус”

Как уже было обозначено в постановке задачи, важным аспектом разработки новых оптимизаций, является их интеграция с уже имеющимися оптимизациями. Для всех реализованных алгоритмов мы будем приводить их место в цепочке оптимизаций и схему взаимодействия с другими оптимизациями, если она не тривиальна. Чтобы это осуществить необходимо привести схему работы двоичного оптимизирующего транслятора для архитектуры “Эльбрус”[1]. Данная схема также поможет дать более целостное представление о работе транслятора.

Схема работы двоичного оптимизирующего транслятора для архитектуры “Эльбрус” изображена на Рис. 4. Первый шаг – это генерация промежуточного представления из исходного x86 кода. Затем производится ряд оптимизаций на не предикатном представлении. Это различные потоковые оптимизации, такие как применение различных тождественных преобразований, удаление мёртвого кода, вынос инвариантов из циклов, сбор общих подвыражений и так далее. Также производятся различные оптимизации графа управления: дублирование, раскрутка циклов, сведение циклов, удаление избыточных ветвлений и так далее. Также до построения предикатного кода производятся различные преобразования специфичные для двоичной трансляции: перенос в компенсирующий код [74], использование MLT [75].

Далее следует построение предикатного кода. При построении предикатного кода используется техника if-conversion. Затем следует ряд оптимизаций работающих на предикатном представлении. Это и различные классические оптимизации, адаптированные к предикатному представлению (применение тождественных преобразований, удаление мёртвого кода, сбор общих подвыражений и так далее), и адаптированные к предикатному представлению оптимизации, специфические для двоичной трансляции, и оптимизации, которые работают только на предикатном коде[2].

В заключении работы двоичного оптимизирующего транслятора производится планирование промежуточного представления по инструкциям, распределение регистров и генерация кода.

---

[1] Приведённое описание не является подробным и не описывает некоторые важные детали, но его вполне достаточного в рамках этой работы.

[2] Набор оптимизаций, работающих на предикатном представлении, достаточно большой и его описание выходит за рамки этой работы.

Рис. 4. *Схема работы двоичного оптимизирующего транслятора для архитектуры “Эльбрус”*

2.4. Минимизация высоты графа зависимостей без построения новых операций.

2.4.1. Переименование регистров

Рассмотрим технику разрыва зависимостей с помощью переименования регистров.

Компонента потокового графа фактически представляет из себя замкнутый набор аргументов и результатов. Если взять новый регистр и назначить его всем элементам компоненты, то полученное представление будет семантически эквивалентно. Алгоритм нахождения компоненты потокового графа по некоторому результату или аргументу вытекает из определения. Необходимо в компоненту добавить все элементы связанные с текущим, а затем рекурсивно провести ту же процедуру для всех добавляемых элементов.

Предлагаемый алгоритм производит полное переименование регистров. Это означает, что всё, что можно переименовать, переименовывается. Каждая компонента потокового графа получает свой собственный уникальный виртуальный регистр или предикат.

Переименование производится с использованием потокового графа для поиска компонент и с использованием счётчиков использования регистров. Счётчик использования

регистра содержит информацию о том, сколько раз данный регистр используется в качестве аргумента или результата в рассматриваемом промежуточном представлении. Эту информацию можно собрать один раз перед началом работы алгоритма пройдя по всем операциям представления. В процессе работы алгоритма её необходимо корректировать при переименовании очередной компоненты[1]. Счётчики использования регистров позволяют определить, что рассматриваемая компонента уже переименована, и нет необходимости её заново переименовывать. Это позволяет ускорить алгоритм переименования.

Приведём более формальное описания алгоритма переименования регистров. Этот алгоритм был предложен в работе [25]. Ниже мы приводим его модификацию. Концептуально это тот же самый алгоритм, однако он адаптирован к структурам данных используемым в данной работе, а также является более быстрым за счёт использования счётчиков использований регистров.

Алгоритм (переименования регистров).

1. **Цикл** по всем результатам всех операций промежуточного представления и результат является виртуальным регистром или предикатом.
2. Для текущего результата находим всю компоненту потокового графа.
3. Если количество использований регистра соответствующего компоненте больше количества элементов в компоненте, то всем элементам компоненты назначаем новый уникальный виртуальный регистр и корректируем счётчики использований регистров.
4. **Конец цикла.**

Отметим, что на вход региона (часть программы поступающая на вход двоичному компилятору) поступает x86 состояние регистров, а виртуальные регистры заводятся и действуют только в пределах региона. Поэтому для всех использований виртуальных регистров всегда имеется определение и следовательно цикл только по результатам операций является корректным.

[1] В двоичном оптимизирующем трансляторе для архитектуры “Эльбрус” реализована несколько другая схема для работы со счётчиками использования регистра. Информация о количестве использований каждого регистра на протяжении всего процесса трансляции храниться в промежуточном представлении и корректируется при различных преобразованиях представления. Это связано с тем, что эта информация является полезной и для других различных преобразований. Таким образом перед началом работы алгоритма переименования не происходит сбор информации о количестве использований регистра – она уже имеется.

Приведённый алгоритм полного переименования регистров позволяет нам избавиться от ложных антизависимостей и зависимостей по результату, то есть тех зависимостей, которых на самом деле нет в исходной программе и которые возникли лишь в результате переиспользования регистров. Отметим, что таких ложных антизависимостей и зависимостей по результату возникает очень много в процессе трансляции кодов архитектуры x86, так как в этой архитектуре очень мало рабочих регистров и поэтому они очень часто переиспользуются.

Приведённый алгоритм является не улучшаемым в том смысле, что убираются все ложные зависимости, которые можно убрать без построения новых операций. Действительно, если определение и использование одного регистра, или два определения одного регистра не были переименованы, то они находятся в одной компоненте потокового графа. Следовательно они транзитивно (через несколько промежуточных узлов потокового графа) находятся в отношении связанности и если их положить на разные регистры, то полученное представление не будет семантически эквивалентно исходному представлению.

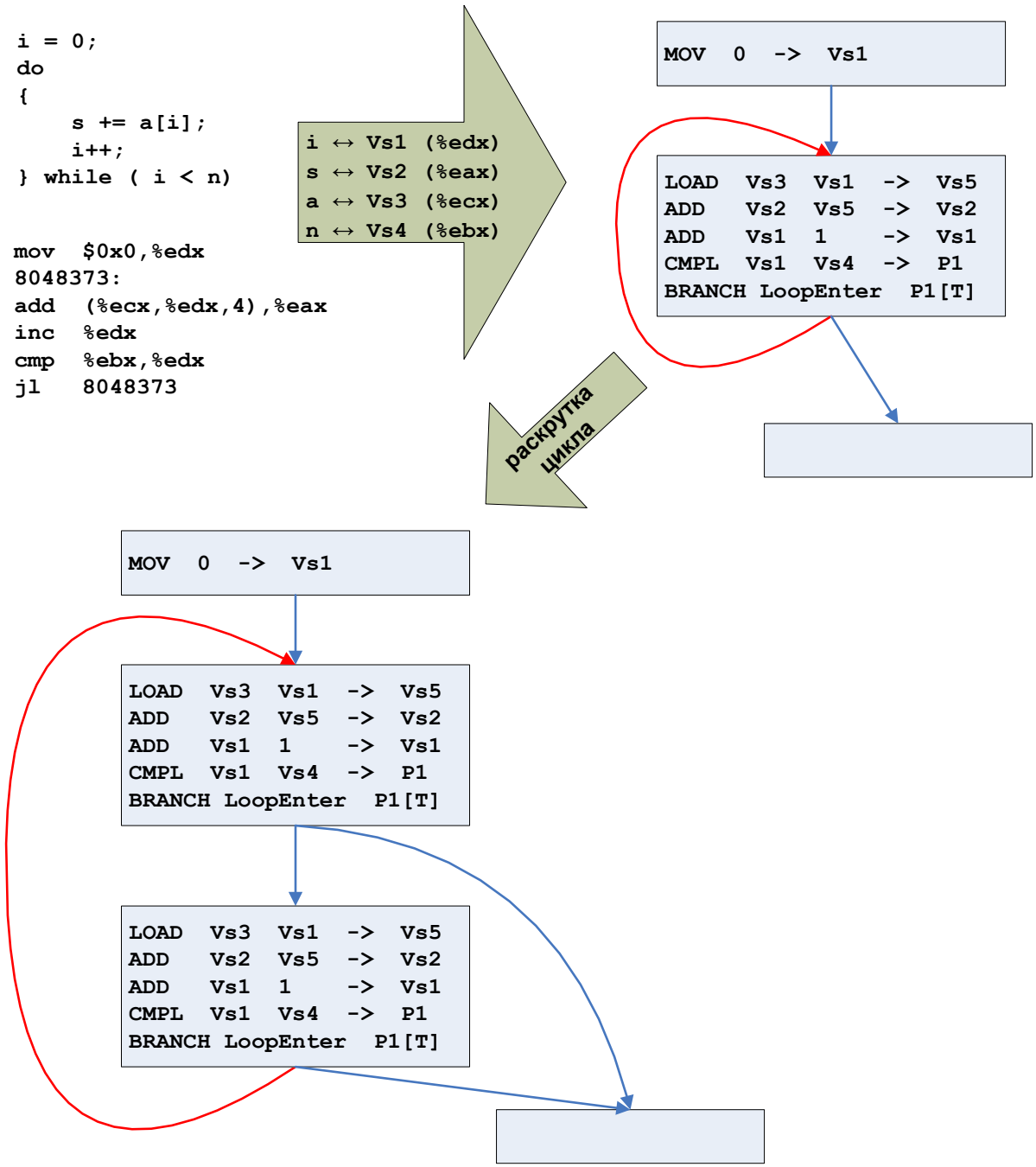

Рис. 5. *Пример возникновения не переименованной компоненты после применения оптимизации “раскрутка цикла”*

Переименование регистров запускается несколько раз в процессе компиляции региона, как на не предикатном представлении, так и на предикатном представлении. Это связано с тем, что некоторые преобразования либо создают новые не переименованные компоненты, либо делят одну компоненту на несколько независимых частей, которые после разделения могут быть переименованы.

Примером создания новых не переименованных компонент может служить оптимизация “раскрутка цикла”. На Рис. 5 изображён пример кода на языке C, соответствующее этому фрагменту промежуточное представление и промежуточное представление после применения раскрутки цикла. До применения оптимизации в промежуточном представлении все регистры были переименованы. После применения оптимизации возникло две не связанных компоненты по регистру Vs5 и две компоненты по предикату P1, которые можно переименовать на различные регистры. Возможность переименовать остальные регистры зависит от того используются ли они за циклом или нет.

Также не переименованные компоненты могут возникнуть в результате дублирования управляющего графа. На Рис. 6 изображён управляющий граф и часть промежуточного представления до, и после дублирования узла содержащего операцию `ST`. После дублирования результаты операций `ADD` уже не входят в одну компоненту и их можно переименовать на различные регистры.

Ещё одним плюсом наличия полного переименования регистров является то, что становится возможным применение более быстрых алгоритмов. Например, построение графа зависимостей можно значительно ускорить, если все регистры переименованы. В таком случае возможно для построения зависимостей по регистрам воспользоваться потоковым графом, что значительно ускорит построение зависимостей[1].

Таким образом приведённый алгоритм полного переименования регистров позволяет полностью решить задачу разрыва антизависимостей и зависимостей по результату без построения новых операций. Он является не улучшаемым в том смысле, что убираются все ложные зависимости, которые можно убрать без построения новых операций. Также использование полностью переименованного представления даёт возможность использовать более быстрые варианты некоторых алгоритмов.

[1] Как правило, потоковый граф остаётся корректным вплоть до распределения регистров, так как это сильно упрощает и делает более эффективным само распределение регистров. В двоичном оптимизирующем компиляторе для архитектуры “Эльбрус” потоковый граф на момент построения графа зависимостей является корректным.

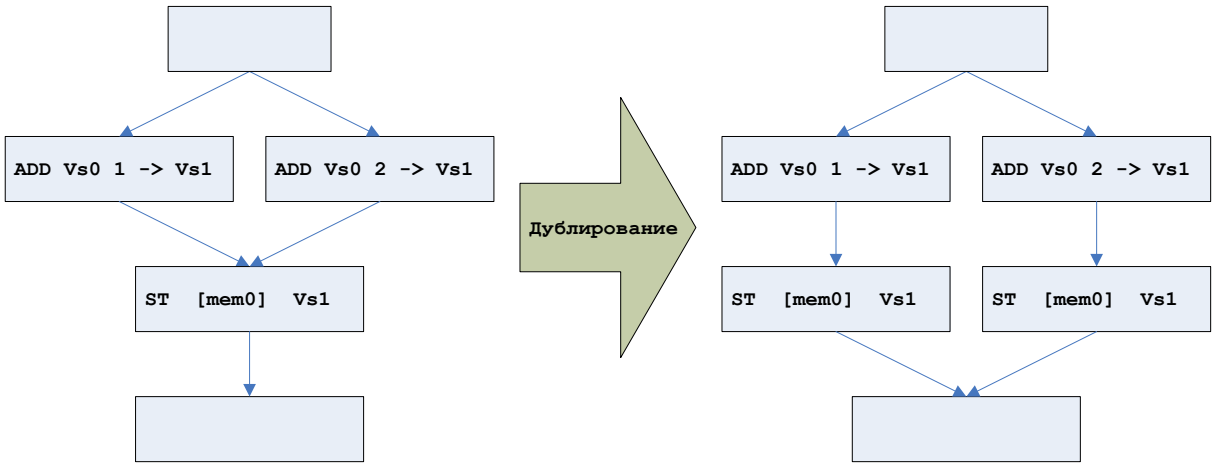

Рис. 6. *Пример возникновения не переименованных компонент после дублирования графа управления.*

2.4.2. Спекулятивность по управлению

Теперь рассмотрим, как разрываются зависимости с использованием спекулятивности по управлению без построения новых операций. Опишем плюсы и минусы раннего применения спекулятивности и позднего применения. Применять спекулятивность нужно как можно раньше по следующей причине: убирание с операции предиката открывает более широкие возможности для применения других оптимизаций. Например, при объединении двух эквивалентных операций (сбор общих подвыражений) стоящих под разными предикатами надо дополнительно построить операцию, которая производит логическое “или” над этими предикатами, и результирующую операцию поставить под этот предикат. В результате получаем два негативных последствия. Во-первых, появляется новая операция над предикатами (получается, что одну операцию удалили, а другую построили). Во-вторых, есть потенциальная возможность увеличения критического пути, так как путь в графе зависимостей идущий от начала узла к условному аргументу результирующей операции увеличивается на время исполнения операции логического “или”.

Ещё одной оптимизацией, для которой важно раннее применение спекулятивности является использование отложенных вычислений для восстановления точного контекста [74]. Технически гораздо тяжелее выполнять эту оптимизацию для операций, которые выполняются под условием и это может негативно сказаться на скорости работы оптимизации.

Основной аргумент в пользу позднего применения спекулятивности является то, что спекулятивные операции чтения из памяти увеличивают нагрузку на подсистему памяти. Однако, как было отмечено ранее, оценить негативный эффект от расширения предиката операции чтения из памяти, достаточно трудно.

Таким образом, получается, что фактов за раннее применение спекулятивности больше, поэтому был выбран именно этот вариант. После построения предикатного кода откладывать применение спекулятивности не имеет никакого смысла, поэтому применение спекулятивности производится сразу же после построения предикатного кода.

Алгоритм применения спекулятивности достаточно простой. Со всех операций, с которых можно снять предикат, он снимается и они переводятся в спекулятивный режим. Ограничения на снятие предиката могут быть следующими:

- Операция не может выполняться в спекулятивном режиме, тогда естественно с неё нельзя снимать предикат.
- Необходимо, чтобы семантика программы осталась корректной после снятия предиката. Например, если есть две операции пишущие в один и тот же регистр и стоящие под антипредикатами, то с них одновременно нельзя снять предикат, так как это нарушит логику программы. При снятии предикатов получится, что после исполнения обеих операций, в регистре всегда будет находиться значение, вырабатываемое последней операцией. Однако по первоначальной логике работы, в регистре может находиться значение обеих операций в зависимости от того пути в программе по которому мы пошли.

Если произведено полное переименование регистров, то достаточным условием того, что с операции можно снять предикат, не нарушив семантику программы, является отсутствие у операции не прямых потоковых последователей. Если все потоковые последователи являются прямыми, то отсюда следует, что отсутствуют операции, которые могут перезаписать значение регистра. Также при снятии предиката операция не может испортить другие компоненты потокового графа, так как всё переименовано. Таким образом, в данном случае можно безопасно снимать предикат. В соответствии с описанным достаточным условием и производится снятие предиката.

Для того, чтобы дополнительно снизить негативный эффект от раннего снятия предиката, связанный с увеличением нагрузки на подсистему памяти, был реализован алгоритм возвращения предикатов операциям (избавления от излишней спекулятивности), который работает ближе к концу работы транслятора. Описание этого алгоритма будет приведено далее в разделе 3.4.

2.4.3. Частичные предикаты

Рассмотрим область управляющего графа предназначенную для слияния в гиперблок. Такую область будем называть *регионом*. Напомним, что это одновходовой подграф управляющего графа, внутри которого отсутствуют циклы. При слиянии региона в гиперблок

необходимо каждую операцию поставить под предикат. С помощью этих предикатов выбираются только те пути управляющего графа, по которым должна исполниться данная операция. Дадим более формальные определения.

Определение. Базовыми предикатами региона будем называть набор предикатов $P_i, i = 1\text{K}\ n$ состоящий из предикатов всех условных операций переходов региона. Если несколько операций перехода стоят под одним и тем же предикатом, то этот предикат несколько раз входит в набор базовых предикатов.

В рассматриваемой нами модели построения предикатного кода принято соглашение, что все операции перехода внутри одного гиперблока параллельны. Также считается, что должна выполниться ровно одна операция перехода в гиперблоке.

Определение. Каждой дуге управляющего графа соответствует операция перехода. Базовый предикат соответствующий этому переходу будем называть *предикатом дуги* и обозначать $P(\textit{дуги})$.

Каждому пути в управляющем графе (путь не обязательно должен заканчиваться выходом из региона) можно сопоставить набор значений базовых предикатов, при котором он реализуется.

Определение. Функцию от базовых предикатов, которая принимает истину только на наборе значений базовых предикатов, при котором данный путь реализуется, будем называть *полным предикатом* этого пути.

Определение. *Полным предикатом* дуги называется дизъюнкция всех полных предикатов путей начинающихся с входа в регион и заканчивающихся данной дугой.

Определение. *Полным предикатом* узла называется дизъюнкция всех полных предикатов путей начинающихся с входа в регион и заканчивающихся в данном узле.

Выражения вычисляющие полные предикаты узлов и дуг могут быть построены следующим образом[1]. Полный предикат дуги равен логическому “и” полного предиката узла, из которого она выходит, и предиката этой дуги. Первый узел имеет тождественно истинный полный предикат. Полный предикат остальных узлов равен логическому “или” полных

[1] Тут необходимо заметить, что существует много различных способов построения выражений вычисляющих полные предикаты дуг и узлов, минимизирующих те или иные характеристики этих выражений. Здесь описывается самый простой метод построения. Некоторые другие методы построения выражений вычисляющих полные предикаты описаны в работах [101], [102] и [103].

предикатов всех дуг входящих в узел. Таким образом обходя регион в порядке топологической сортировки можно построить полные предикаты всех дуг и узлов.

Рассмотрим пример управляющего графа предназначенного для слияния в гиперблок, изображенного на Рис. 7.

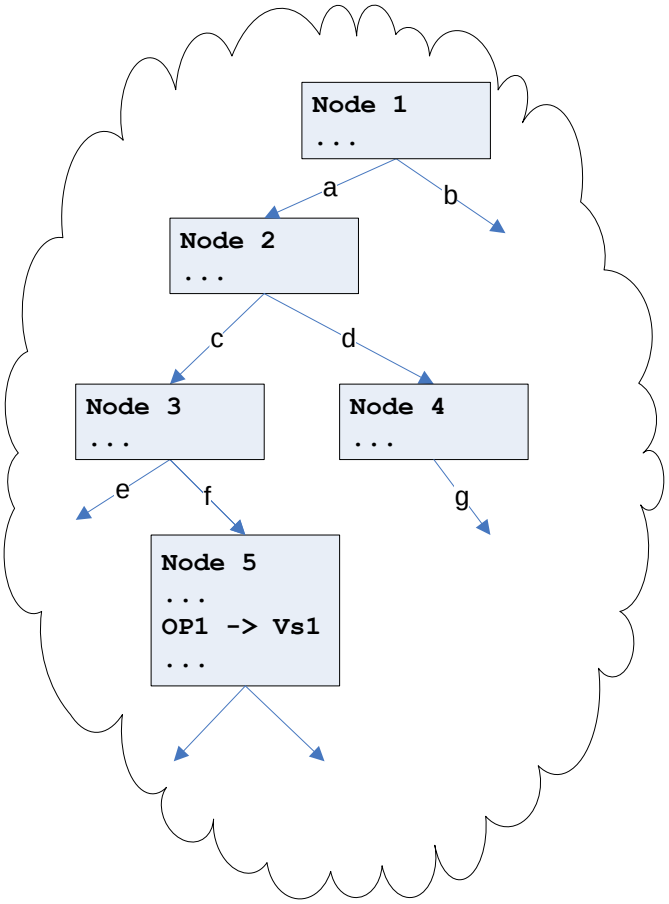

Рис. 7. *Пример управляющего графа для слияния в гиперблок.*

Здесь полный предикат `Node 4` будет равен $P(a)\,\&\,P(d)$, а полный предикат `Node 5` будет равен $P(a)\,\&\,P(c)\,\&\,P(f)$.

Определение. Предикат P является частичным по отношению к полному предикату Q, если $Q \Rightarrow P$.

При слиянии региона в гиперблок, в общем случае, все операции, находящиеся в любом узле A из региона, должны быть в итоговом гиперблоке поставлены под полный предикат узла A. Однако часто оказывается возможным поставить операцию не под полный предикат узла A, а под частичный предикат, или вообще не ставить под предикат. Использование частичного предиката позволяет сократить высоту вычислений предиката и если окажется, что высота

вычислений определяется временем вычисления предиката операции, то это ускорит результирующий код.

Поясним использование термина “частичный предикат”. Полный предикат ограничивает исполнение операции по тем путям, по которым она не должна выполняться. Частичный предикат позволяет операции исполняться на более широком количестве путей, то есть операция не исполняется на меньшем количестве путей. Именно из-за рассмотрения предикатов как ограничителей путей, по которым исполняется операции и были даны названия “полный предикат” (ограничивает исполнение на всех путях на которых не должна выполняться операция) и “частичный предикат” (ограничивает исполнение лишь на части путей на которых не должна выполняться операция).

Под использованием частичных предикатов понимается техника, которая вместо постановки операции под полный предикат, ставит её под частичный предикат. Использование частичных предикатов является по сути тем же применением спекулятивности. Фактически техника, описанная в предыдущем пункте, является частным случаем применения частичных предикатов. В силу сказанного все плюсы и минусы раннего и позднего применения частичных предикатов, такие же, как и для применения спекулятивности (снятия предиката).

Применение частичных предикатов интегрировано с построением предикатного кода. Если операцию возможно поставить под частичный предикат, то она сразу ставится под него, вместо постановки под полный предикат. Аналогично предыдущему пункту здесь встаёт проблема: как не нарушить времена жизни регистров, после расширения предиката.

Основой для принятия решение о том, можно ли построить частичный предикат, и какой построить частичный предикат, принимается на основании информации о жизни регистров на дугах управляющего графа, входящих в регион. Эту информацию можно достаточно просто получить на основе потокового графа, при условии, что все компоненты переименованы.

Нахождение частичного предиката для операции осуществляется следующим образом. Производятся последовательные попытки перенести операцию вверх по управлению в ближайший доминатор (физически операция не переносится, так как в этом нет необходимости, потому что в дальнейшем все операции будут объединены в один гиперблок). Если это возможно, то пытаемся перенести операцию в следующий ближайший доминатор. Перенос в ближайший доминатор может быть невозможен в силу того, что нарушится семантика программы. Это происходит в том случае, если в доминаторе живёт другое определение того же регистра, и если осуществить перенос, то значение регистра будет испорчено перенесённой операцией. Если перенос в ближайший доминатор невозможен, то к операции необходимо добавить предикат (не полный) равный логическому “или” всех входящих в текущий узел дуг. При условии добавления такого предиката операцию можно перенести в узел ближайший

доминатор. Обратимся вновь к Рис. 7 и проиллюстрируем на нём технику построения частичного предиката. Допустим мы хотим построить частичный предикат для `OP1`. Цепочка ближайших доминаторов выглядит следующим образом: `Node 3`, `Node 2`, `Node 1`. Пусть перенос без добавления предиката нельзя осуществить только для пары `Node 3 – Node 2`. Тогда мы будем действовать следующим образом. Сначала пытаемся перенести операцию в `Node 3`. Это возможно. Затем производиться попытка перенести операцию в `Node 2`. Такой перенос не возможен и поэтому к операции необходимо добавить предикат $P(c)$. Далее осуществляется перенос в `Node 1`. Это также возможно без добавления предиката. В итоге получаем, что частичный предикат для `OP1` равен $P(c)$.

Постановка операции под частичный предикат производится только тогда, когда для этого действия не требуется построения новых операции. Это разумное ограничение, так как с одной стороны это упрощает алгоритм определения частичного предиката, а с другой стороны построение новой операции увеличивает давление на ресурсы при планировании, что может негативно сказаться на результирующем коде.

Опишем более подробно работу алгоритма. Алгоритм работает рекурсивно. На каждом шаге определяем можно ли расширить предикат операции до предиката узла, являющегося ближайшим доминатором текущего узла. Если это возможно, то продолжаем рекурсию с доминатором в качестве текущего узла. Если расширять предикат до ближайшего доминатора нельзя, то возможны следующие случаи:

1. Первый – в узел входит несколько дуг. В этом случае операцию необходимо поставить под предикат, который является логическим “или” предикатов (не полных) входящих дуг. Для получения такого предиката необходимо построить новую операцию, и как было сказано выше, случай, когда требуется построение новой операции, мы не обрабатываем. В итоге считается, что в случае, когда нельзя расширить предикат до ближайшего доминатора и, когда в узел входит несколько дуг, постановка под частичный предикат невозможна, и операции ставится полный предикат.
2. Второй случай – в узел входит одна дуга. Ближайшим доминатором является узел предшественник этой дуги. Предикат этой дуги (не полный) запоминается, предикат операции должен включать этот предикат.

Далее рекурсия продолжается. Запомнить несколько предикатов нельзя, так как в этом случае может потребоваться построение новой операции. Если мы хотим запомнить некоторый предикат, и при этом какой-то предикат уже запоминался, то считается, что построение частичного предиката невозможно и операция ставится под полный предикат.

Необходимо заметить, что в некоторых случаях, когда описанный алгоритм не может построить частичный предикат без создания новой операции, это всё-таки возможно за счёт

использования уже имеющийся эквивалентной операции. Однако для упрощения и ускорения алгоритма такая возможность не используется.

Опишем теперь часть алгоритма, в которой определяется можно ли расширить предикат операции с текущего узла до ближайшего доминатора.

Алгоритм (определения возможности расширить предикат операции с текущего узла node до узла – ближайшего доминатора)

1. Находим и заносим в список nodes_list все узлы между узлом node и его ближайшим доминатором
2. **Цикл** по всем узлам из nodes_list; текущий узел cur_node
3. **Цикл** по всем дугам выходящим из cur_node; текущая дуга cur_edge
4. **Если** последователь дуги находится в nodes_list и дуга не обратная, то **продолжить цикл**
5. **Если** по дуге cur_edge живёт исследуемый регистр, то **вернуть** невозможность расширения предиката
6. **Если** по какой-то из дуг в области, начинающейся с последователя cur_edge, до конца сливаемой области вниз, живёт исследуемый регистр, то **вернуть** невозможность расширения предиката
7. **Конец цикла**
8. **Конец цикла**
9. **Вернуть**, что возможно расширение предиката

Здесь необходимо более подробно пояснить пункт 6 приведённого алгоритма. На Рис. 8 изображён пример области для слияния на if-conversion. Следовательно, для этой области будет применяться построение частичных предикатов. Порядок, в котором изначально параллельные узлы управляющего графа (точнее операции из этих узлов) будут стоять в итоговом гиперблоке может быть произвольным. Примеры различного порядка в итоговом гиперблоке приведены на Рис. 9. Определённый порядок задаётся внутренними эвристиками оптимизации if-conversion и вообще говоря никак не зависит от построения предикатов. Как правило, эти эвристики руководствуются профильной информацией, а именно ставят выше узлы управляющего графа с большим числом повторений. Основным аргументом для слияния в таком порядке служит минимизация потерь от появления “невидимых” в момент слияния зависимостей, то есть зависимостей, которых либо нет в момент слияния, либо о которых в момент слияния предлагается что они будут разорваны, но позднее по каким-то причинам их не удалось разорвать. Приведём несколько причин появления “невидимых” зависимостей. Это могут быть

зависимости, которые в итоге не удалось разорвать, например, по причине явной нехватки регистров или по каким-то другим причинам. Ещё одной причиной появления “невидимых” зависимостей может служить сбор общих подвыражений. После объединения нескольких операций в одну все зависимости переносятся на эту одну операцию, что в итоге может привести к замедлению одной из веток. В итоге получается, что порядок узлов управляющего графа в гиперблоке разумнее задавать независимыми эвристиками.

Вернёмся к пояснению пункта 6 приведённого алгоритма. Допустим, мы хотим построить частичный предикат для операции `OP1` из узла `Node 4` на Рис. 8. Ближайшим доминатором для `Node 4` является `Node 1`. При обработке `Node 3` мы будем просматривать все выходящие из него дуги, в том числе дугу идущую в `Node 5`. По этой дуге не живёт регистр `Vs1`. Если бы мы на этом остановились (разрешили расширить предикат до предиката `Node 1`) и не стали бы делать проверки из 6-ого пункта алгоритма, то при выборе порядка слияния было бы дополнительное ограничение: `Node 5` должен обязательно стоять ниже `Node 4`. Если такую зависимость между `Node 4` и `Node 5` при построении гиперблока не ввести, то получится некорректная семантика. Напомним, что в рассматриваемой модели предикатного кода все переходы являются независимыми, следовательно, переход соответствующий выходу из узла `Node 5` в итоговом гиперблоке может опуститься ниже `OP1`. Таким образом, в случае если в коде мы должны пройти через `Node 5`, то сначала `OP2` запишет верное значение, потом `OP1` перепишет его не верным значением (мы же не поставили её под предикат), и только потом выполниться переход, соответствующий выходу из `Node 5` на Рис. 8. В итоге в точке соответствующей цели этого перехода мы получим неверное значение `Vs1`.

Таким образом пункт 6 приведённого алгоритма позволяет в итоге отделить принятие решения о порядке узлов в гиперблоке от построения предикатов.

В принципе возможен вариант, в котором вместо того, чтобы запрещать применение частичных предикатов, можно было бы вводить ограничения на порядок слияния узлов при создании гиперблока. В таком варианте шестой пункт алгоритма в случае обнаружения конфликтующей записи в исследуемой области фиксировал бы некоторый порядок при сливании узлов. Фактически это эквивалентно построение временной дуги управляющего графа между узлами, которые необходимо упорядочить. Однако в таком варианте ещё бы пришлось следить за тем, чтобы не возникало конфликтов в упорядочивании. Например, может оказаться, что с одной стороны необходимо узел A поставить выше узла B, а с другой стороны необходимо B поставить выше A. Если об упорядочивании размышлять как о построении временных дуг, то возникновение конфликта эквивалентно возникновению циклов в сливаемой области.

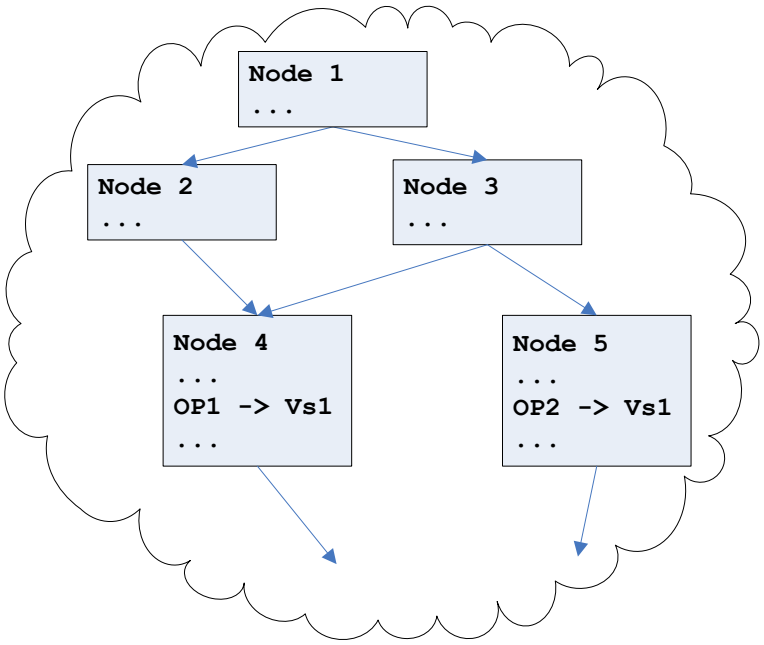

Рис. 8. *Пример области для слияния на if-conversion.*

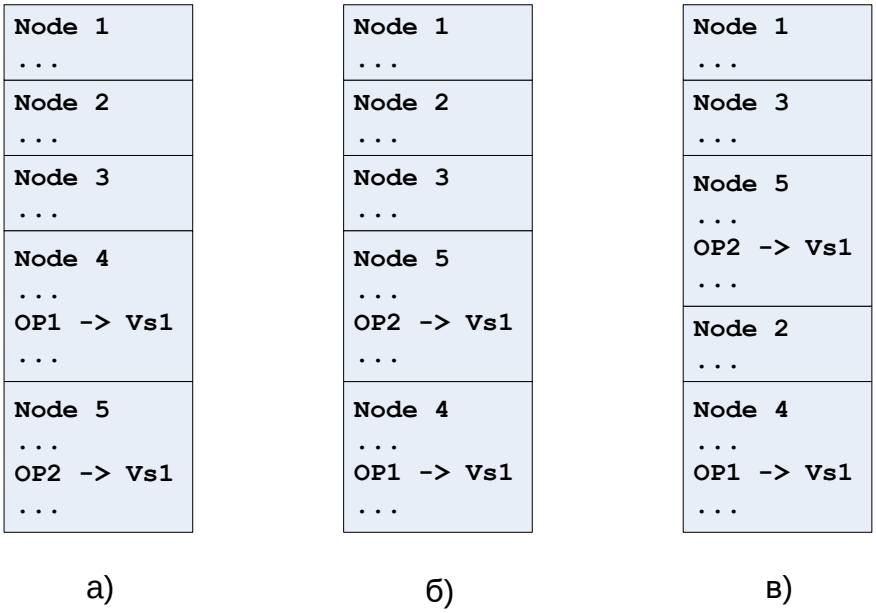

Рис. 9. *Пример различного порядка узлов при создании гиперблока.*

Один из типичных случаев использования частичных предикатов приведён на Рис. 10.

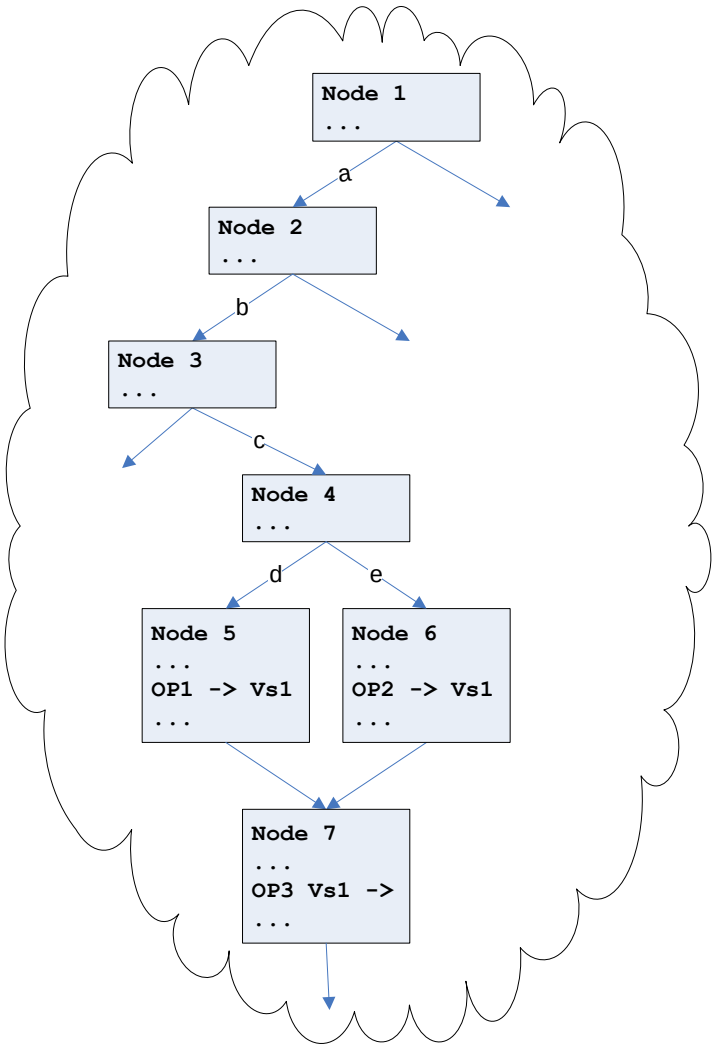

Рис. 10. *Типичный пример использования частичных предикатов*

Здесь имеются две операции `OP1` и `OP2`, результат которых сходится и используется в `OP3`. Регистр `Vs1` больше нигде не определяется и не используется. Полные предикаты для `OP1` и `OP2` имеют вид $P(a)\&P(b)\&P(c)\&P(d)$ и $P(a)\&P(b)\&P(c)\&P(e)$ соответственно. При использовании описанной техники построения частичных предикатов для операции `OP1` построится частичный предикат $P(d)$, а для `OP2` – $P(e)$. В результате время необходимое на вычисления условных аргументов операций `OP1` и `OP2` существенно сократиться.

Описанная техника построения частичных предикатов полностью покрывает алгоритм применения спекулятивности без построения новых операций описанный в предыдущем пункте. То есть со всех операций, с которых можно снять предикат с помощью техники описанной в разделе 2.4.2, можно снять предикат и с помощью техники использования частичных предикатов описанного в этом пункте.

Также отметим, что раннее построение частичных предикатов (при построение предикатного кода) обладает теми же достоинствами и недостатками, что и раннее снятие предиката (эти достоинства и недостатки описаны в разделе 2.4.2).

Описанный алгоритм построения частичных предикатов был реализован в двоичном трансляторе для микропроцессора “Эльбрус”. Применение алгоритма даёт **1-2%** прироста производительности результирующего кода на широком классе задач. При этом замедление времени работы транслятора составляет **0,4%**. Подробные результаты замеров влияния этого алгоритма на качество результирующего кода будут приведены в разделе 2.5.

2.4.4. Схема работы двоичного компилятора с учётом алгоритмов минимизации без построения новых операций

Подытожим всё что было сказано выше про минимизацию критических путей без построения новых операций и опишем место этих алгоритмов в процессе компиляции. На Рис. 11 приведена схема работы двоичного оптимизирующего транслятора с учетом алгоритмов минимизации высоты графа зависимостей без построения новых операций.

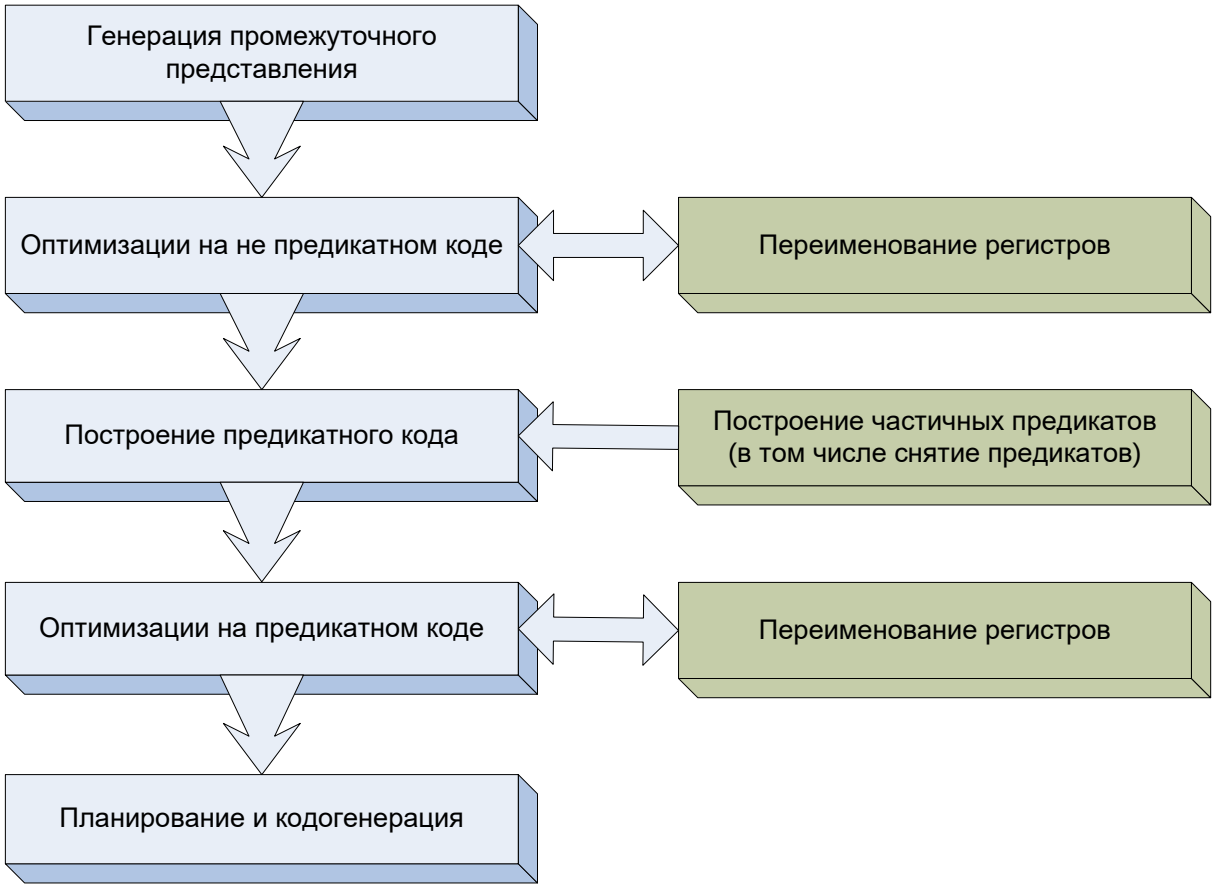

Рис. 11. *Схема работы двоичного оптимизирующего транслятора для архитектуры “Эльбрус” с учётом алгоритмов минимизации высоты графа зависимостей без построения новых операций*

Как указывалось ранее многие оптимизации могут порождать не переименованные компоненты потокового графа, поэтому алгоритм переименования регистров запускается несколько раз за время компиляции как на не предикатном коде, так и на предикатном коде. Построение частичных предикатов происходит сразу же при построении предикатного кода. Если операцию можно поставить под частичный предикат, то она сразу ставится под этот предикат. Как отмечалось, используемая техника построения частичных предикатов также позволяет снять предикат с тех операций для которых это возможно. Такое раннее снятие предиката позволяет более эффективно применяться различным оптимизациям на предикатном коде.

2.5. Экспериментальные результаты

Для анализа эффективности работы алгоритмов использовался статический двоичный оптимизирующий транслятор для микропроцессора “Эльбрус” и потактовый симулятор этой архитектуры. Статический двоичный оптимизирующий транслятор для микропроцессора “Эльбрус” переводит коды приложения архитектуры x86 в семантически эквивалентное приложение в кодах архитектуры “Эльбрус”. Для выявления горячих областей исходного приложения, сбора профильной информации и обнаружения точек входа в коды используется предварительный запуск приложения в со специальным инструментированием для сбора этой информации.

Потактовый симулятор микропроцессора “Эльбрус” моделирует архитектуру с точностью до такта. В нём полностью моделируются все стадии конвейера и регистровый файл. Также моделируется кэш память первого и второго уровня, задержки и темп обращений в память. Отрабатываются все остановки конвейера связанные с неготовностью данных или команд к исполнению.

Для анализа изменения скорости работы результирующих кодов использовались следующие методы:

- Запуск статически скомпилированных кодов задач из пакета SPEC CPU2000 [72] на потактном симуляторе микропроцессора “Эльбрус-С”. Для ссылок на этот метод будем использовать фразу “замеры на симуляторе”
- Технология предсказания времени работы кода на основе планирования. Время работы процедуры вычисляется как

$$\sum_{\substack{\text{по} \\ \text{всем} \\ \text{операциям} \\ \text{передачи} \\ \text{управления}}} C \cdot L$$

где C - счетчик операции передачи управления, а L - расстояние в тактах от начала узла до инструкции в которой спланирован переход. Описанная величина представляет из себя время работы кода с точки зрения транслятора. Здесь могут оказаться неучтёнными задержки связанные с промахами в кэш, блокировками не учитываемыми транслятором и т.д. Однако такой подход позволяет более точно оценить вклад алгоритма в качество результирующего кода. Причина этого в отсутствии шума связанного, например, с плохой предсказуемостью времени обращения в память[1]. В качестве тестов для анализа брались горячие участки задач из пакетов SPEC CPU95 и SPEC CPU2000, а также горячие участки операционной системы Windows и типичных пользовательских приложений для неё. Именно такие участки в первую очередь оптимизируются самым высоким уровнем системы двоичной трансляции. Для ссылок на этот метод будем использовать фразу “предсказание на основе планирования”.

Эксперимент состоит в изучении влияния построения частичных предикатов на качество результирующего кода. Было произведено сравнение времени работы результирующего кода с включённым и выключенным алгоритм построения частичных предикатов описанным в 2.4.3. Результаты приведены на Рис. 12, Рис. 13 и Рис. 14.

[1] Для пояснения предположим, что у нас есть некоторая задача проводящая 90% времени своей работы в ожидании прихода данных из памяти. Допустим мы реализовали некоторую оптимизацию сокращающую время вычислений данной задачи на 10%. В таком случае время работы задачи сократиться всего на 1%. Раньше время складывалось как 0.9+0.1=1.0, после реализации оптимизации 0.9+(0.1-10%)=0.99.

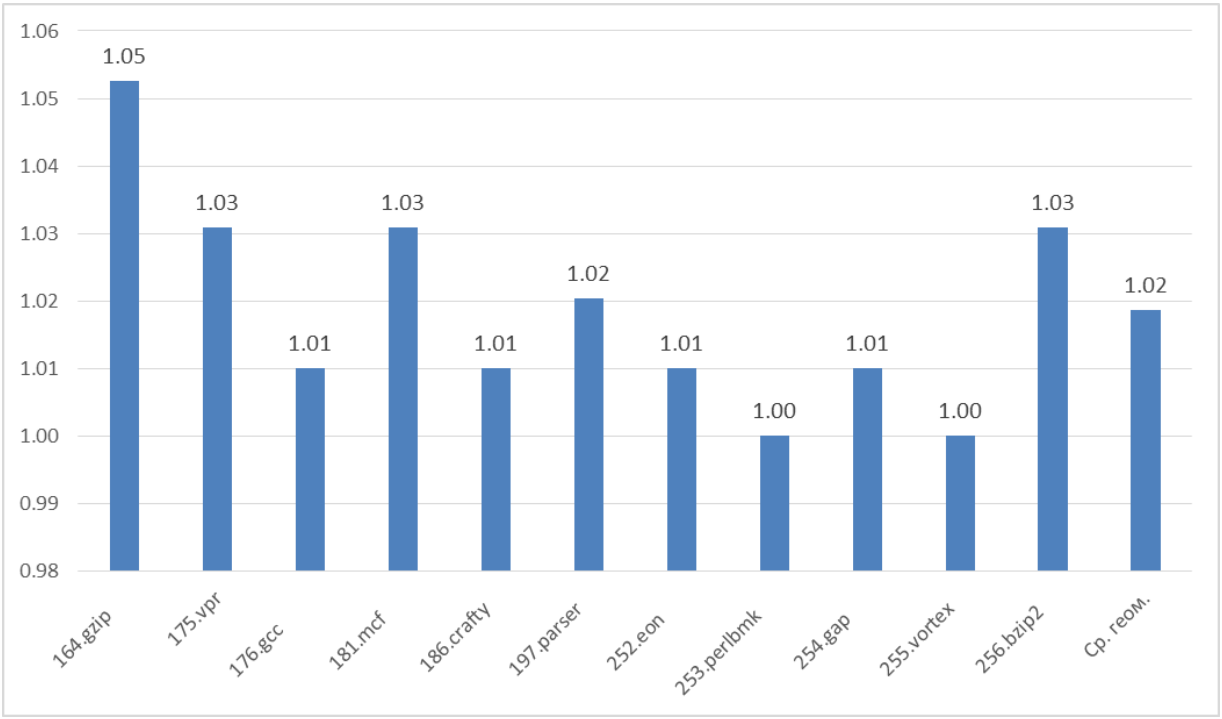

Рис. 12. *Влияние техники построения частичных предикатов на время работы результирующего кода на целочисленных задачах пакета SPEC CPU2000. Замеры на симуляторе.*

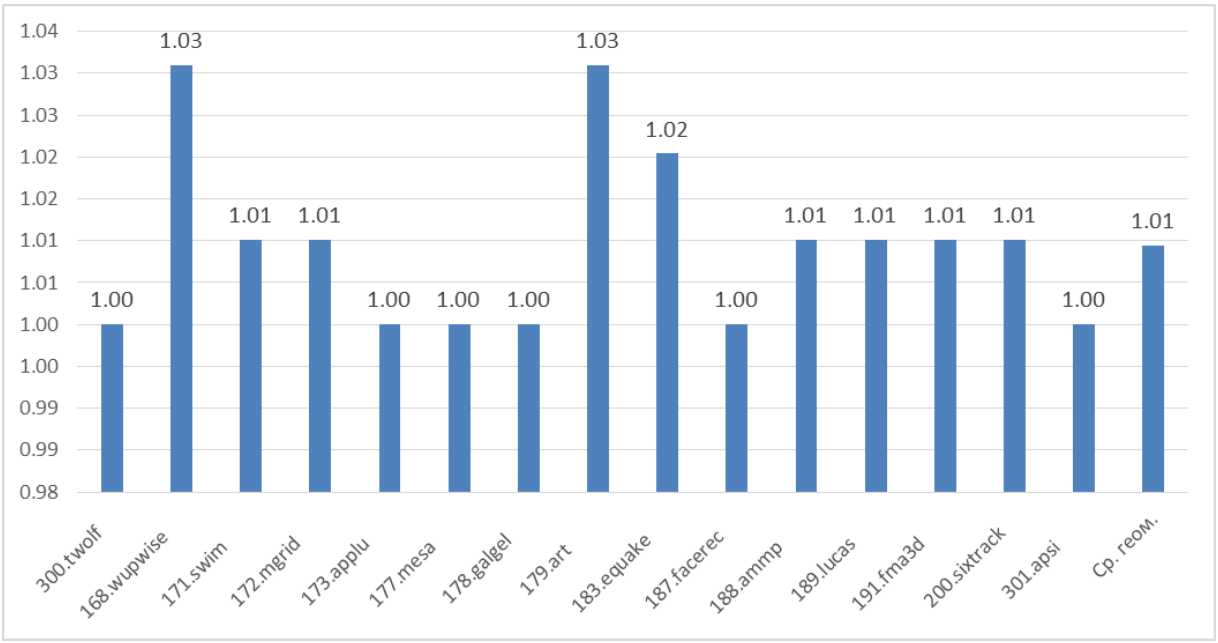

Рис. 13. *Влияние техники построения частичных предикатов на время работы результирующего кода на вещественных задачах пакета SPEC CPU2000. Замеры на симуляторе.*

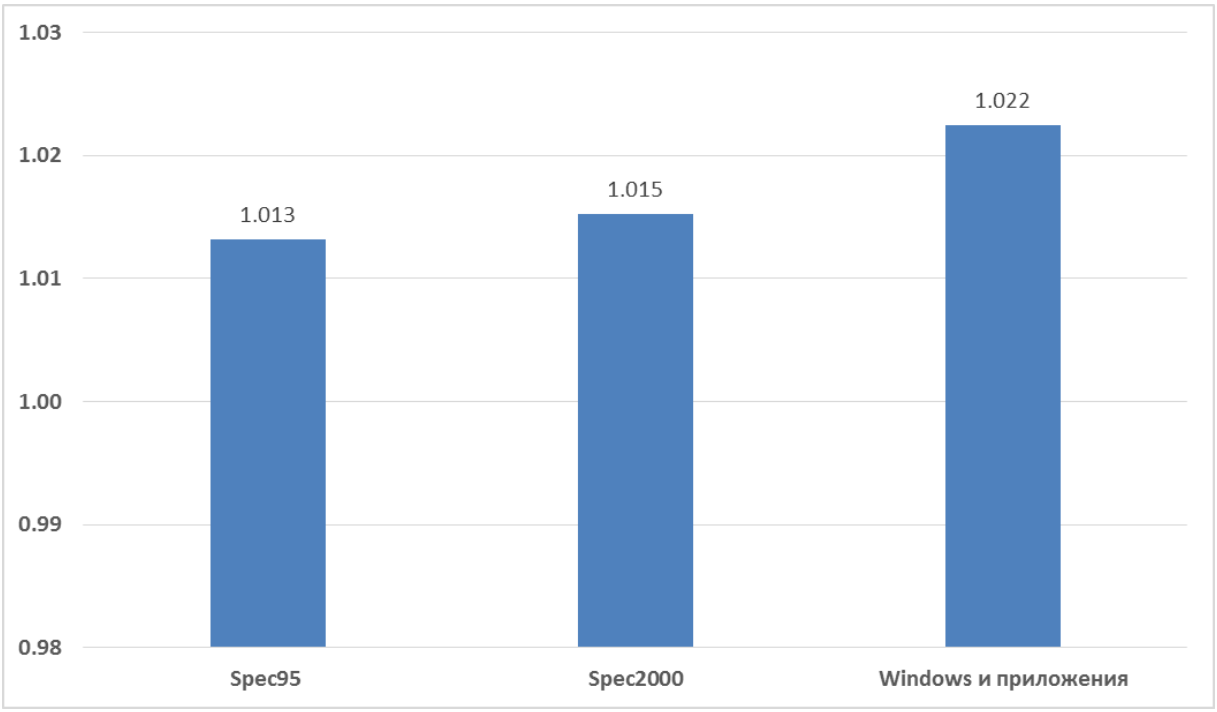

Рис. 14. *Влияние техники построения частичных предикатов на предсказанное по планированию время работы кода для горячих участков SPEC CPU95, SPEC CPU2000, Windows и пользовательских приложений. Предсказание на основе планирования.*

Применения алгоритма даёт **1-2%** прироста производительности результирующего кода. При этом время работы самого алгоритма составляет **0,4%** от общего времени работы транслятора.

2.6. Выводы

1. В данной главе предложен алгоритм переименования регистров, который позволяет сократить высоту графа зависимостей без построения новых операций.
2. Приведена схема применения алгоритма переименования в двоичном оптимизирующем трансляторе.
3. Также в данной главе предложен алгоритм применения спекулятивности и построения частичных предикатов, который позволяет сократить высоту графа зависимостей без построения новых операций. Алгоритм построения частичных предикатов позволяет существенно повысить эффективность предикатного кода.
4. Приведена схема применения и место в цепочке оптимизаций алгоритма построения частичных предикатов.
5. Приведённые экспериментальные результаты показывают высокую эффективность предложенных методов на широком классе задач. Предложенный алгоритм даёт прирост

производительности на **1-2%** При этом замедление времени работы транслятора составляет 0,4% от общего времени трансляции.

3. Сокращение длины критических путей в ациклических областях с построением новых операций

В начале этой главы приводится обзор методов разрыва зависимостей, а также существующих методов минимизации высоты графа зависимостей. Далее описываются различные алгоритмы минимизации высоты графа зависимостей реализованные в двоичном трансляторе для архитектуры “Эльбрус”. Минимизация высоты производится с помощью различных техник разрыва зависимостей. Для ряда алгоритмов приводится их формализация с помощью языка теории графов, доказывается их оптимальность и оценивается сложность. В заключении приводятся результаты экспериментов.

3.1. Методы разрыва зависимостей с помощью построения новых операций

Рассмотрим преобразования, основанные на построении новой операции, с помощью которых можно сократить высоту графа зависимостей.

Первый класс разрываемых зависимостей, который мы рассмотрим, это *антизависимости*. Если аргументы, по которым возникает антизависимость, и предшественника зависимости и последователя, находятся в одной компоненте потокового графа, то данную зависимость невозможно разорвать с помощью переименования регистров (описанного в предыдущей главе). Однако в этом случае можно всё-таки разорвать зависимость с помощью построения операции пересылки. Проиллюстрируем это примером:

```
E=0  ENTER
E=0  CMP    Vs1  0    -> P1
E=0  MUL    Vs2  3    -> Vs3
E=4  ST     Vs6  4     Vs3
E=4  MUL    Vs5  3    -> Vs3  P1[F]
E=8  AND    Vs3  2    -> Vs8  P1[F]
E=9  ST     Vs7  Vs8   Vs3
E=9  END
```

В этом примере имеется антизависимость идущая от операции `ST` по адресу `Vs6+4` ко второй операции `MUL`. Здесь невозможно разорвать зависимость с помощью переименования, так как регистры, через которые осуществляется рассматриваемая зависимость, находятся в одной компоненте потокового графа (первая операция `MUL` и вторая операция `MUL` в зависимости от

значения предиката P1 могут потребляться последней операцией ST по адресу Vs7+Vs8). С помощью построения новой операции пересылки рассматриваемый линейный участок можно преобразовать к следующему виду:

```
E=0  ENTER
E=0  CMP    Vs1  0    -> P1
E=0  MUL    Vs2  3    -> Vs3
E=4  ST     Vs6  4     Vs3
E=2  MUL    Vs5  3    -> Vs9  P1[F]
E=6  MOV    Vs9       -> Vs3  P1[F]
E=6  AND    Vs9  2    -> Vs8  P1[F]
E=7  ST     Vs7  Vs8   Vs3
E=7  END
```

На Рис. 15 а) приведён граф зависимостей, соответствующий представлению до преобразования. Высота графа зависимостей в этом случае равна девяти тактам и данный код может спланировать в десять тактов. На Рис. 15 б) приведён граф зависимостей после разрыва антизависимости. В результате преобразования высота графа сократилась до семи тактов.

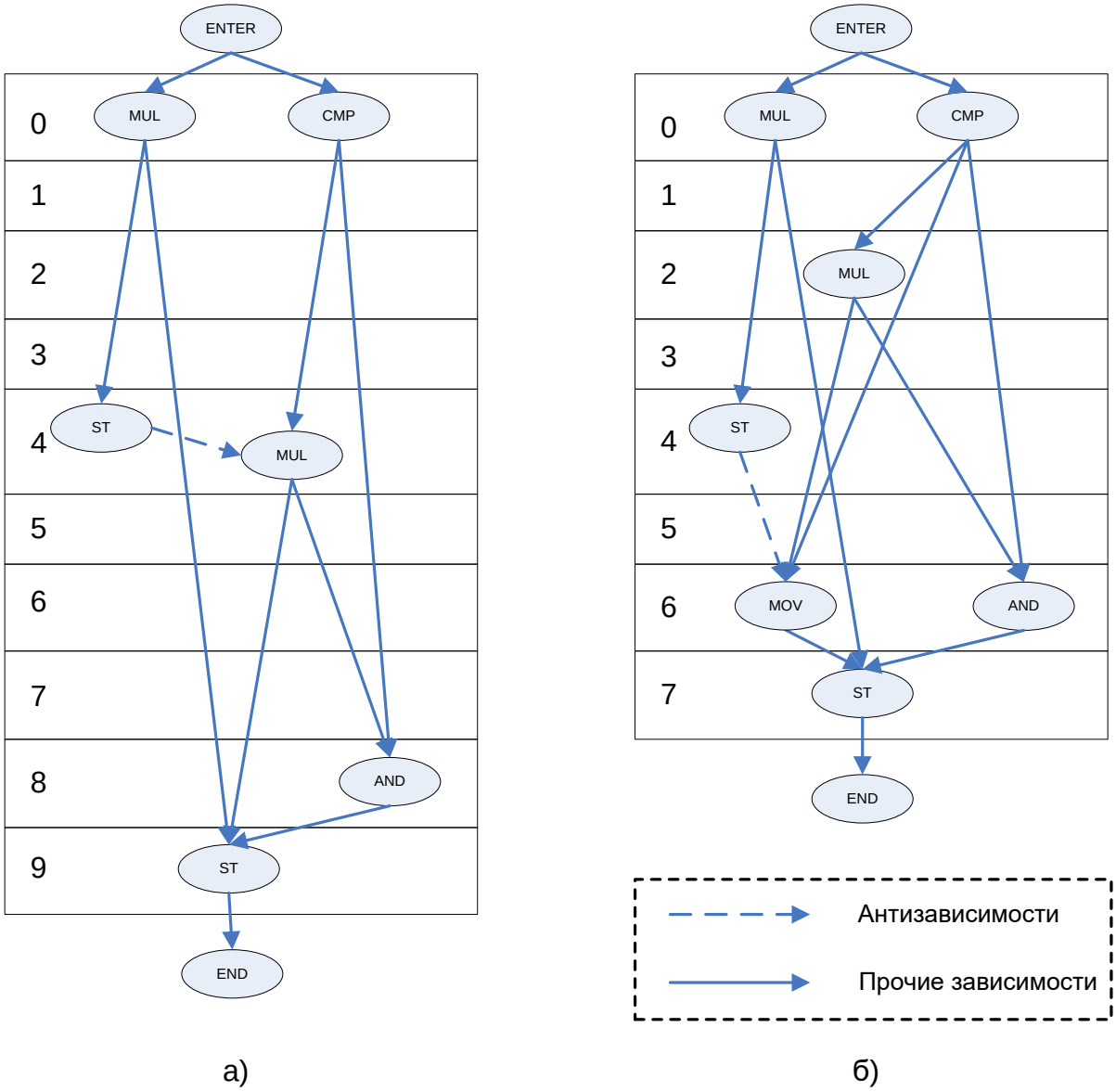

Рис. 15. *Пример разрыва антизависимости. а) Граф зависимостей до разрыва антизависимости; б) Граф зависимостей после разрыва антизависимости*

При построении новой операции пересылки все не прямые потоковые последователи (см. Определение в 1.3) переносятся на неё. В то же время прямые потоковые последователи остаются на исходной операции и потребляют новый регистр. Такое преобразование корректно, так как исходная операция является единственным определением для прямых потоковых последователей, и такое преобразование не изменит поведение кода.

Перейдём к рассмотрению следующего класса разрываемых зависимостей – *зависимости по результату*. Эти зависимости разрываются в точности таким же методом как и антизависимости. Поэтому не будем останавливаться подробно на этом, а приведём лишь небольшой пример:

```
E=0  ENTER
E=0  ADD   Vs1  0  -> Vs2
E=1  ADD   Vs1  1  -> Vs2  P1[T]
E=2  AND   Vs2  3  -> Vs3  P1[T]
E=3  OR    vs3  16 -> Vs4  P1[T]
E=2  SHL   Vs2  2  -> Vs5
E=3  END
```

здесь имеется зависимость по результату между операциями `ADD`. Разорвать её можно с помощью построения пересылки:

```
E=0  ENTER
E=0  ADD   Vs1  0  -> Vs2
E=0  ADD   Vs1  1  -> Vs6  P1[T]
E=1  MOV   Vs6     -> Vs2  P1[T]
E=1  AND   Vs6  3  -> Vs3  P1[T]
E=2  OR    vs3  16 -> Vs4  P1[T]
E=2  SHL   Vs2  2  -> Vs5
E=2  END
```

В результате такого преобразования высота вычислений сократится на один такт.

Следующий класс разрываемых зависимостей это *предикатные потоковые зависимости* (или просто *предикатные зависимости*). Такую зависимость можно разорвать с помощью построения дополнительной операции пересылки под таким же предикатом как и исходная операция. При этом с исходной операции предикат убирается, и она переводится в спекулятивный режим. У исходной операции результат записывается в новый уникальный виртуальный регистр, и все прямые потоковые последователи остаются последователями исходной операции, но уже потребляют новый виртуальный регистр. Не прямые потоковые последователи переносятся на построенную операцию пересылки[1]. Рассмотрим следующий фрагмент промежуточного представления:

```
E=0  ENTER
E=0  ADD   Vs1  Vs2 ->  Vs3
```

[1] Перераспределение прямых и не прямых потоковых последователей происходит также как и в случае построения операции пересылки для разрыва антизависимости

```
E=1  CMP   Vs3  0    ->  P0
E=3  SUB   Vs4  2    ->  Vs5  P0[T]
E=3  SUB   Vs4  4    ->  Vs5  P0[F]
E=4  MUL   Vs5  2    ->  Vs6  P0[T]
E=4  MUL   Vs5  3    ->  Vs6  P0[F]
E=4  ST    [mem a]   Vs5
E=8  ST    [mem b]   Vs6
E=8  END
```

Здесь имеется четыре предикатных зависимости идущих от операции сравнения к операциям вычитания и умножения. Эти зависимости нельзя разорвать без построения новых операций, так как у всех последователей зависимостей имеются не прямые потоковые последователи. Разрыв предикатных зависимостей для этого примера выглядит следующим образом:

```
E=0  ENTER
E=0  ADD   Vs1  Vs2 ->  Vs3
E=1  CMP   Vs3  0    ->  P0
E=0  SUB s Vs4  2    ->  Vs10
E=3  MOV   Vs10      ->  Vs5  P0[T]
E=0  SUB s Vs4  4    ->  Vs11
E=3  MOV   Vs11      ->  Vs5  P0[F]
E=1  MUL s Vs10 2    ->  Vs12
E=5  MOV   Vs12      ->  Vs6  P0[T]
E=1  MUL s Vs11 3    ->  Vs13
E=5  MOV   Vs13      ->  Vs6  P0[F]
E=4  ST    [mem a]   Vs5
E=6  ST    [mem b]   Vs6
E=6  END
```

До разрыва зависимостей длина вычислений равняется девяти тактам. После преобразования длина сокращается до семи тактов.

Существует ещё один способ разрыва предикатной зависимости – это *оптимизация unzipping* [19], [20]. В отличие от выше рассмотренного преобразования данный метод разрыва зависимостей может сократить критический путь даже в том случае, когда у операции нет

прямых потоковых зависимостей. Преобразование выполняется следующим образом. Операция, один из аргументов которой определяется в зависимости от значения некоторого предиката, дублируется. Операция и её копия ставятся под предикаты таким образом, чтобы и операция и её копия потребляли строго одно значение, то есть первоначальный аргумент имеет строго определённого предшественника. Потоковые последователи раздублированной операции становятся теперь не прямыми последователями и потребляют либо результат операции либо результат её копии в зависимости от значения предиката. В результате у исходной операции остаются только прямые потоковые последователи и с этой операции можно снять предикат, методами описанными выше. Для пояснения приведём пример:

```
E=0  ENTER
E=0  CMP   Vs1  0    ->  P1
E=2  ADD   Vs2  1    ->  Vs3  P1[T]
E=2  ADD   Vs2  2    ->  Vs3  P1[F]
E=3  OR    Vs3  Vs4  ->  Vs5
E=3  END
```

В этом примере длина вычислений равна четырём тактам. Время раннего операции сложения определяет готовность её предикатного аргумента. Разорвать эту зависимость с помощью операции пересылки нельзя, так как у операций сложения имеется только не прямой потоковый последователь. В результат применения unzipping-а получается следующее:

```
E=0  ENTER
E=0  CMP   Vs1  0    ->  P1
E=0  ADD s Vs2  1    ->  Vs10
E=0  ADD s Vs2  2    ->  Vs11
E=2  OR    Vs10 Vs4  ->  Vs5  P1[T]
E=2  OR    Vs11 Vs4  ->  Vs5  P1[F]
E=2  END
```

После применения оптимизации высота длина вычислений сократилась до трёх тактов.

Рассмотренное преобразование может быть применено итеративно к другим операциям, последователям раздублированной операции, если они также находятся на критическом пути и критический путь проходит через предикат.

Следующий метод уменьшения критического пути называется *балансировка арифметических выражений* [21]. Этот метод может применяться к последовательности зависимых ассоциативных операций. Изменив порядок вычисления, использую ассоциативность, можно сократить высоту вычислений. Это и называется балансировкой арифметических выражений. Если выражение:

```
(((((X1 + X2) + X3) + X4) + X5) + X6) + X7) + X8
```

вычислять, так же как указан порядок скобок, то есть начиная со сложения `X1` с `X2` затем к результату прибавить `X3`, затем к новому результату прибавить `X4` и так далее, то длина вычислений составит семь тактов. Однако если перегруппировать, использую ассоциативность, следующим образом:

```
((X1 + X2) + (X3 + X4)) + ((X5 + X6) + (X7 + X8))
```

то длина вычислений сократиться до трёх тактов. Заметим, что в этом методе новые операции появляются, однако их общее количество не изменяется. В силу этого данная оптимизация может применяться всегда, поскольку она не ухудшет код[1]. Описанная техника может быть применена и в других случаях, например, с использованием дистрибутивности или закона де Моргана, но при этом общее количество операций будет изменяться.

Программный динамический разрыв зависимостей по памяти впервые был предложен в работе [31]. Суть этого метода заключается во вставлении в код явной проверки адресов с помощью операции сравнения и операции пересылки под предикатом. Ниже приведён пример программного разрыва зависимостей. В примере рассматриваются записи и чтения из памяти размером один байт.

```
E=0  ENTER
E=0  MUL  Vs8  3  ->  Vs3
E=0  MUL  Vs8  5  ->  Vs5
E=4  ST   Vs2  0  Vs3
E=4  ST   Vs4  0  Vs5
E=4  LD   Vs6  0  ->  Vs1
```

[1] В двоичном компиляторе для архитектуры Эльбрус данная оптимизация реализована в качестве отдельной фазы компилятора.

```
E=7  SUB  Vs1  1  ->  Vs9
E=7  END
```

После преобразования получим следующее:

```
E=0  ENTER
E=0  MUL  Vs8  3   ->  Vs3
E=0  MUL  Vs8  5   ->  Vs5
E=0  LD   Vs6  0   -> Vs1
E=4  ST   Vs2  0  Vs3
E=0  CMPE Vs2  Vs6 -> P1
E=4  MOV  Vs3      ->  Vs1  P1[T]
E=4  ST   Vs4  0  Vs5
E=0  CMPE Vs4  Vs6 -> P2
E=5  MOV  Vs5      ->  Vs1  P2[T]  //output зависимость
E=6  SUB  Vs1  1   ->  Vs9
E=6  END
```

В приведённом примере длина вычислений уменьшилась с восьми тактов до семи. В приведённом примере рассмотрен самый простой случай и нам потребовалось построить по две операции, чтобы перенести операцию чтения выше каждой операции записи. Однако, если работа с памятью осуществляется форматами большими, чем один байт, может понадобиться гораздо больше операций, так ячейки памяти с которыми идёт работа могут частично пересекаться. Большое количество новых построенных операций является главным недостатком этого метода. В общем случае, если необходимо разорвать зависимости между n операциями чтения из памяти с m операциями записи в память, потребуется $n \times m$ групп проверок, каждая из которых состоит как минимум из двух операций.

Зависимости между обращениями в память часто создают узкое место в производительности результирующего кода. Программный метод разрыва зависимостей по памяти создаёт много новых операций и это ограничивает возможность его широкого применения. Для более эффективного разрешения конфликтов можно использовать *аппаратный механизм спекулятивности по данным* [30], описанный в разделе 1.2. Приведём пример разрыва зависимости по памяти, без построения компенсирующего кода:

```
E=0  ENTER
```

```
E=0  ADD     Vs1  Vs2  -> Vs3
E=1  SUB     Vs3  4    -> Vs4
E=2  MUL     Vs4  Vs5  -> Vs6
E=6  ST      [mem a]   Vs6
E=6  LD      [mem b]   -> Vs7
E=9  ADD     Vs7  Vs8  -> Vs9
E=9  END
```

если статическим анализом не удалось определить, что адреса `[mem a]` и `[mem b]` независимы, то для корректности результирующего кода упорядочить операцию записи в память и операцию чтения из памяти. В итоге получается, что длина вычислений равна десяти тактам. После применения спекулятивности по данным получится следующий код:

```
E=0  ENTER
E=0  ADD     Vs1  Vs2  -> Vs3
E=1  SUB     Vs3  4    -> Vs4
E=2  MUL     Vs4  Vs5  -> Vs6
E=6  ST      [mem a]   Vs6
E=0  LD.lock [mem b]   -> Vs7
E=6  LD.chk  [mem b]   -> Vs7
E=7  ADD     Vs7  Vs8  -> Vs9
E=7  END
```

задержка от операции `LD.chk` равна одному такту, а операция `LD.lock` не имеет больше никаких зависимостей сверху, и может планироваться в нулевой такт. Так как в примере не предполагается использовать компенсирующий код, то все потоковые зависимости должны быть перенесены на `LD.chk`, который в свою очередь зависит от операции записи в память. Таким образом, в случае если не возникнет конфликта между обращениями в память, длина вычислений сократиться до восьми тактов. Высоту можно ещё уменьшить, если создать компенсирующий код. Это позволит избежать зависимости между `LD.chk` и операцией сложения. Тогда длина вычислений уменьшится до семи тактов, а в компенсирующем коде появиться копия операции сложения, которая будет пересчитывать значение регистра `Vs9`, в случае обнаружения конфликта.

3.2. Обзор существующих алгоритмов минимизации высоты графа зависимостей

3.2.1. Разрыв зависимостей и минимизация высоты графа зависимостей в суперблоках

Первый метод, который мы опишем, был реализован в проекте IMPACT в компиляторе IMPACT-I [27]. Рассматриваемый метод основывается на структуре данных получившей название суперблок (superblock) [26]. Суперблок является развитием понятия трассы [29]. Трасса представляет из себя некоторый путь в управляющем графе. Обычно трассы составляют из наиболее вероятных путей. В трассе могут иметься условные выходы из трассы и переходы из других трасс в середину трассы. Суперблок является трассой без сторонних входов, то есть существует только одна точка входа в суперблок – начало суперблока.

Для того, чтобы можно было перенести операцию выше условного перехода (выхода из суперблока), необходимо чтобы регистр, в который пишет операция, не был жив (не использовался) по этому выходу, так как иначе переносимая операция перезапишет этот регистр и семантика программы изменится. Такой перенос вверх требует использования спекулятивности.

Внутри суперблока отсутствуют не прямые потоковые зависимости, так как есть только один вход в суперблок и нет ветвлений/схождений внутри суперблока. Это позволяет использовать следующую технику переноса операций вверх с построением новых операций. Производится переименование результата операции, то есть в качестве нового результата операции назначается новый уникальный регистр. Если первоначальный регистр, в который пишет операция нужен по какому-то ниже находящемуся выходу из суперблока, то на каждом таком выходе строится пересылка из нового регистра в первоначальный. Такое преобразования является семантически корректным. После того как описанное переименование произведено, новый результат операции не требуется ни по одному из вышележащих выходов (так как операция пишет в новый уникальный регистр) и её можно беспрепятственно перенести вверх. Поскольку суперблоки содержат самые вероятные пути исполнения, накладные расходы на новые операции пересылки допустимы, так как мы ускоряем исполнение более вероятного пути (суперблока) за счёт менее вероятных (выходы из суперблока).

Минимизация высоты графа зависимостей осуществляется отдельной фазой компилятора, которая работает до планирования суперблоков. Это фазой разрываются все зависимости, которые возможно разорвать. В итоге планирование суперблока имеет большую свободу, так как все “лишние” зависимости отсутствуют.

3.2.2. Минимизация высоты графа зависимостей в процессе работы планировщика

Разрывать все зависимости не всегда эффективно, в силу того, что может создаться слишком много новых операций[1]. Поэтому в уже упомянутом проекте IMPACT, при реализации спекулятивности по данным, был использован более осторожный метод минимизации высоты графа зависимостей [32]. Опишем этот алгоритм подробно.

Рассматриваемый алгоритм использования спекулятивности по данным включает следующие шаги:

1. Построение графа зависимостей.
2. Добавление операций check сразу же за каждой операцией чтения из памяти и построение необходимых зависимостей.
3. Для каждой операции чтения удаляются зависимости по памяти.
4. Планируется суперблок и в процессе планирования удаляются все не нужные операции check.
5. Вставляется компенсирующий код.

Предварительная подготовка к планированию, включая построение графа зависимостей, при использовании алгоритма остаётся неизменной. После построения графа зависимостей, операции check вставляются после каждой операции чтения из памяти в суперблоке. Первоначально компенсирующий код для операций check не определён. Операция чтения и операция check связываются зависимостью. В процессе планирования любая операция check должна иметь корректные зависимости, поэтому на неё наследуется часть зависимостей от соответствующей операции чтения из памяти. Все потоковые последователи остаются на исходной операции. Также добавляются зависимости к предыдущей и последующей операциям передачи управления, так как необходимо, чтобы операция check находилась в оригинальном линейном участке[2].

Следующий шаг алгоритма – это удаление зависимостей по памяти. Для каждой операции чтения перебираются все зависимости вверх и удаляются те из них, которые идут к тем операциям записи, для которых не удалось доказать, что они точные (то есть обращения точно идут по пересекающимся адресам). Для каждой операции чтения запоминаются все зависимости, которые были удалены. Для того чтобы смягчить эффект от избыточной

[1] Подробнее о различных недостатках не ограниченного разрыва зависимостей мы будем говорить позже в разделе 3.2.5, специально посвящённом этой теме.

[2] Для схемы планирования гиперблоков это условие трансформируется в наличие точного предиката у операции check

спекулятивности, алгоритм ограничивает количество удаляемых зависимостей для каждого чтения.

Следующий шаг – планирование суперблока. Во время планирования каждой операции чтения из памяти, проверяются все операции записи, зависимость с которыми была разорвана. Если все операции записи в память уже спланированы, то получается, что операция чтения не обогнала ни одну запись в память и применять спекулятивность по данным, не имеет смысла, следовательно, соответствующую операцию check можно удалить. Зависимость, построенная между операцией чтения и операцией check, гарантирует, что операция check не может спланироваться раньше чтения из памяти, таким образом, не потребуется удалять уже спланированную инструкции. Если операция чтения спланировалась раньше, чем некоторая запись, с которой разрывалась зависимость, то операция чтения конвертируется в операцию чтения с занесением в таблицу конфликтов.

В заключении алгоритма создаётся компенсирующий код для тех операций check, для которых это необходимо.

3.2.3. Минимизация высоты графа зависимостей в гиперблоках

Перейдём теперь к рассмотрению алгоритмов минимизации высоты графа зависимостей в гиперблоках. В работе [37] предложен алгоритм основанный на временах планирования операций. Ранним временем планирования операции в данной работе называется максимальная длина пути в графе зависимостей от начала гиперблока до рассматриваемой операции, при этом учитываются только так называемые “настоящие” зависимости[1]. Под настоящими зависимости понимаются только потоковые зависимости и зависимости по обращениям в память. В результате применения алгоритма, все операции могут быть спланированы в своё время раннего планирования. Зависимости разрываются двумя способами. Первый способ это использование спекулятивности в сочетании с переименованием. Этот метод очень похож на технику применения unzipping-a, и фактически является некоторым его обобщением. В отличие от unzipping-a описанного выше, дублируются не только операции у которых имеется два потоковых предшественника для одного аргумента, но также и операции, которые имеют произвольное количество последователей для одного аргумента. Вторым способом разрыва зависимостей является техника сокращения вычисления предикатных выражений. Она основана на особых параллельных операциях логического “и” и логического “или” [99]. Эти

[1] Поскольку имеет место различные определения одного и того же термина, отметим, что во всей работе за исключением этого пункта под временем раннего планирования понимается максимальная длина пути в графе зависимостей от начала гиперблока по всем зависимостям.

операции очень похожи на операции, имеющиеся в архитектуре IA-64 [16][1]. Однако такой метод разрыва зависимостей требует специальных операций, отсутствующих в микропроцессоре “Эльбрус”, поэтому не будем останавливаться на нём подробнее.

Минимизация высоты графа зависимостей с использованием спекулятивности и переименования производится следующим образом. Операции обрабатываются последовательно, начиная с первой операции в гиперблоке. Если это не операция записи в память, то возможно применение спекулятивности. Происходит сравнение времени раннего операции со временем, когда готов условный аргумент. Если операция может быть спланирована раньше (учитывая только “настоящие” зависимости), чем готов её условный аргумент, то предикат с операции снимается и при необходимости производится переименование её результата и дублирование потоковых последователей. Если же предикат готов раньше, чем мы можем спланировать операцию (опять же учитывая только “настоящие” зависимости), то спекулятивность не применяется.

Важным результатом рассмотренной работы и предложенного алгоритма является то, что для всех операций обеспечивается возможность быть спланированными настолько рано насколько это возможно (в терминах статьи – быть спланированными в своё время раннего). Эта идея будет являться одной из ключевых в дальнейшем изложении.

Фактически в работе все рассуждения ведутся для архитектуры с бесконечным количеством исполняющих устройств. Недостатком такого подхода является отсутствие какого-либо контроля над количеством раздублированых операций. А такое неограниченное дублирование может негативно сказаться на качестве результирующего кода. Вообще говоря, рассмотренный алгоритм может привести даже к экспоненциальному росту количества операций на некоторых классах графов управления. Таким образом, пока не учитываются реальные ресурсы имеющиеся в микропроцессоре всё хорошо: зависимости рвутся, критический путь сокращается. Однако во время планирования при превышении ресурсов, когда будут учитываться реальные ресурсы имеющиеся в микропроцессоре, может произойти ухудшение производительности результирующего кода.

3.2.4. Минимизация высоты графа зависимостей с помощью решения задачи целочисленного линейного программирования

В заключение опишем подход изложенный в работах [68], [69]. В этих работах делается попытка наилучшим образом решить задачу глобального планирования операций.

[1] В статье один из разделов посвящён освещению вопроса о сходстве используемых операций с операциями в архитектуре IA-64, а также о возможности применения описанных идей и алгоритмов для этой архитектуры.

Исследования проводятся для микроархитектуры Itanium [16], [17]. Задача глобального планирования переформулируется в терминах комбинаторной оптимизационной проблемы. Затем применяется целочисленное линейное программирование для нахождения оптимального решения. В технику глобального планирования интегрировано применение спекулятивности по управлению и спекулятивности по данным. При этом находится оптимальный с точки зрения производительности вариант применения спекулятивности.

Такой подход является очень эффективным с точки зрения скорости работы результирующего кода. Во-первых, удаётся найти оптимальное решение для проблемы планирования с применением спекулятивности. Во-вторых, техники разрыва зависимостей полностью интегрированы с планированием, что позволяет добиться максимальной эффективности за счёт того, что планирование и разрыв зависимостей знают друг о друге и учитывают друг друга. В-третьих, схема является масштабируемой с точки зрения добавления новых техник разрыва зависимостей. Результаты, приведённые в работе, также впечатляют. Учитывая, что работа по большей части велась, как академическое исследование, на нескольких функциях из целочисленных задач пакета SPEC CPU2000 [72] удалось получиться выигрыш в 20-30%, по сравнению с промышленным компилятором Intel’s Linux compiler for the Itanium [73] с максимальным уровнем оптимизаций.

Однако у этого метода есть существенный недостаток: он работает очень долго, так как в нём используется NP-сложный алгоритм. Для двух из девяти рассматриваемых в [69] функций из целочисленных задач пакета SPEC CPU2000, состоящих из нескольких сотен инструкций, время решения превысило сто секунд. В общем-то, такое большое время работы ожидаемое для выбранного подхода. Но, к сожалению, при всей своей эффективности в качестве создания результирующего кода, такое время работы никак не позволяет использовать данную технику в двоичном оптимизирующем трансляторе (впрочем, массово и в языковом компиляторе тоже).

3.2.5. Проблемы и недостатки существующих методов минимизации высоты графа зависимостей

В двоичном оптимизирующем трансляторе алгоритмы разрыва зависимостей очень важны. Это связано с тем, что в двоичном трансляторе по сравнению с языковым априори и апостериори гораздо больше операций зависят друг от друга. Для этого есть ряд причин:

- В двоичном коде можно использовать лишь ограниченное число рабочих регистров, так как в самом микропроцессоре имеется ограниченное число регистров. Это особо актуально для относительно старых архитектур. Например, в архитектуре x86 имеется всего восемь рабочих целочисленных регистров. Для сравнения коды, написанные на языках высокого уровня, могут иметь фактически не ограниченное число рабочих

переменных. Маленькое количество регистров приводит к необходимости постоянного их переиспользования. Вследствие этого возникает большое количество ложных зависимостей типа чтение-запись и запись-запись.

- Необходимость обеспечения возможности восстановления точного контекста в любой момент времени часто приводит к значительному увеличению времени жизни регистров. В результате часто возникают схождения нескольких определений в потоковом графе, что в свою очередь мешает разрыву зависимостей без построения новых операций.
- Двоичный транслятор имеет гораздо меньше информации об обращениях в память по сравнению с языковым компилятором. Он не видит всей программы, не видит, где выделяется память, к которой обращаются, не может проанализировать переменную, лежащую в стеке, на предмет взятия её адреса и т.д. Все эти техники может эффективно использовать языковой компилятор для доказательства независимости двух обращений в память. Двоичный транслятор лишён этой возможности, в результате чего у него гораздо больше операций обращений в память зависимы между собой, и приходится использовать техники разрыва этих зависимостей с помощью построения новых операций.

Также необходимо заметить, что наряду с сокращением критического пути, разрыв зависимостей может привести к нескольким негативным побочным эффектам:

- Увеличивается количество операций. Это может повлиять и на качество результирующего кода и на время работы транслятора
- Увеличение давления на регистры. Перевод операций в спекулятивный режим и агрессивный перенос их вверх увеличивают давление на регистры. В результате регистров может не хватить и фазы распределения регистров построит операции закачки и подкачки из памяти, что ухудшит качество результирующего кода
- Спекулятивное исполнение операций чтения из памяти приводит к подкачке данных из памяти в кэш. Чтобы данные поместить в кэш необходимо вытолкнуть оттуда другие данные. Если спекулятивность была применена зря, то есть путь управления в программе пошёл таким образом, что спекулятивно прочитанные данные не понадобились, то получится, что данные из кэша были вытеснены зря, что в дальнейшем приведёт к задержкам при чтении этих вытесненных данных.
- Блокировки связанные с ожиданием готовности аргументов при спекулятивном использовании спекулятивного чтения из памяти в случае, если необходимые данные отсутствуют в кэше и эти вычисления в дальнейшем не будут востребованы. То есть,

если мы в ветке, которая в дальнейшем окажется не востребованной, переведём в спекулятивный режим операцию чтения из памяти и использование результата этого чтения, и данные для чтения не окажутся в кэше, то на операции использующей (спекулятивно) эти данные мы остановимся и будем ожидать пока эти данные не придут из памяти.

Таким образом получается, что, с одной стороны, в процессе работы двоичного транслятора зависимостей, которые необходимо разрывать, очень много. Но в тоже время разрыв всех зависимостей может оказать серьёзное негативное последствие на качество результирующего кода. Следовательно необходимо производить разрыв зависимостей достаточно аккуратно. Алгоритмы минимизации высоты графа зависимостей должны стараться не разрывать лишние зависимости, то есть те, разрыв которых не приводит к сокращению критического пути.

Существует ещё одна важная особенность, которую необходимо учитывать при разработке алгоритмов минимизации высоты графа зависимостей. Разрыв зависимостей одним методом может повлиять на разрыв зависимостей другим методом. Например, пусть на некотором варианте промежуточного представления к некоторой операции не имеет смысла применять разрыв антизависимости с помощью построения новой операции, так как в результате критический путь не сократится, так как операция не стоит на критическом пути[1]. Однако, после применения спекулятивности по данным к другим операциям, путь, на котором стояла исходная операция, может стать критическим. Таким образом, становится необходимым, разорвать антизависимость, от разрыва которой мы раньше отказались. Разрыв этой антизависимости может в свою очередь позволить разорвать ещё некоторые зависимости, используя спекулятивность по данным, для которых разрыв раньше был неэффективным и так далее.

Могут также возникнуть случаи, когда сократить высоту графа зависимостей можно только за счёт разрыва нескольких зависимостей различными методами.

Таким образом, получается, что для эффективной работы алгоритм минимизации высоты графа зависимостей должен уметь одновременно разрывать зависимости всеми доступными методами. Также он должен уметь анализировать влияние всех методов разрыва на длину критического пути одновременно. В противном случае может получиться менее производительный результирующий код. Ни в одной из работ, описанных выше, не было

[1] Могут существовать также другие причины, по которым разрыв зависимости не приведёт к сокращению критического пути. Например, время операции определяется другой зависимостью, отличной от той, которую мы хотим разорвать

предложено и реализовано комплексного метода минимизации высоты графа зависимостей, который бы обрабатывал одновременно все типы разрываемых зависимостей.

Ещё одним важным требованием, уже упоминавшимся выше, предъявляемым к алгоритмам, работающим в динамических трансляторах, является скорость их работы. Она должна быть максимально высокой, так как время работы транслятора включается во время работы приложения.

Рассмотренные в разделе 3.2 методы минимизации высоты графа зависимостей не удовлетворяют тем или иным требованиям описанным выше.

Оригинальный алгоритм, используемый в проекте IMPACT, и описанный в 3.2.1 и 3.2.2 применяется для суперблоков, что уже сильно ограничивает область его применения. Также в предложенном подходе разделены применение спекулятивности по управлению и спекулятивности по данным. В принципе небольшие модификации этих алгоритмов позволяют избавиться от этих недостатков. Однако даже при такой модификации разрыв зависимостей во время планирования имеет свои минусы. Во-первых, возможен разрыв лишних зависимостей. Операции с разорванными зависимостям ничто не мешает спланироваться слишком рано, в результате чего для отката разрыва придётся перепланировать код, а это сложное действие как с технической точки зрения, так с точки зрения времени работы алгоритма. Во-вторых, алгоритмы планирования достаточно сложные с точки зрения объема кода их реализующего. Например, планировщик в компиляторе icc для Itanium имеет порядка 45000 строк исходного кода [97]. Планировщик двоичного оптимизирующего транслятора для “Эльбруса” имеет порядка 70000 строк исходного кода. Внедрять в такие сложные алгоритмы ещё дополнительную технику по разрыву зависимостей технологически достаточно сложно. В-третьих, алгоритмы планирования достаточно сложные с точки зрения времени своей работы. В двоичном оптимизирующем трансляторе для архитектуры “Эльбрус” время затрачиваемое на планирования кода составляет порядка 20-25% от общего времени компиляции. Внедрение в алгоритм планирования техники разрыва зависимостей значительно увеличит время его работы.

В работе описанной в 3.2.3 используется идея принятия решения о разрыве зависимостей на основе времён раннего и позднего планирования. Такой подход является достаточно привлекательным: с одной стороны он обеспечивает достаточно хорошее качество разрыва зависимостей (высота графа становиться минимальной при этом разрываются не все зависимости), с другой стороны он является достаточно быстрым. Скорость работы получается за счёт следующих факторов. Алгоритм расчёта времён является линейным по сложности от количества зависимостей. Хорошие, с точки зрения качества результирующего кода, алгоритмы планирования также является, как минимум линейными от количества зависимостей. Но при этом алгоритмы планирования имеют существенно большее время обработки одного шага

(имеют бельшую константу сложности), так как при планировании операции необходимо учесть большое количество различных ограничений на планирование. Недостатком работы является то, что рассматриваются лишь два метода разрыва зависимостей. Также, не смотря на то, что предпринимаются попытки не разрывать лишние зависимости, авторы обращают внимание на то, что алгоритм всё ещё может разрывать много не нужных зависимостей, и тех самым увеличивать дублирование кода, которое может привести к ухудшению результирующего кода.

Метод минимизации высоты графа зависимостей с помощью решения задачи целочисленного линейного программирования, описанный в 3.2.4, даёт очень хороший результат с точки зрения качества результирующего кода. Однако применение его на практике, даже в языковом компиляторе, сильно затруднено в силу очень большой алгоритмической сложности. Когда же речь идёт о двоичном трансляторе, то применение таких алгоритмов просто не возможно.

3.3. Общее описание алгоритма минимизации высоты графа зависимостей основанного на техниках разрыва с построением новых операций

В данном разделе предлагается алгоритм минимизации высоты графа зависимостей, основанный на технике разрыва зависимостей с помощью построения новой операции.

3.3.1. Вводные замечания

Предлагаемый алгоритм минимизации высоты графа зависимостей реализован в виде отдельной оптимизации (фазы) транслятора. Для общего ускорения процесса компиляции в процессе работы алгоритма также производится построение графа зависимостей. В процессе компиляции фаза работает практически перед финальным планированием кода.

В данном алгоритме применяются следующие техники разрыва зависимостей с построением новой операции: предикатные потоковые зависимости, антизависимости, зависимости по результату и техника разрыва зависимостей с использованием спекулятивности по данным. Эти техники разрыва были описаны ранее. Также было опробовано включение техники разрыва зависимостей unzipping. Однако добавление этой техники не дало существенно выигрыша в производительности и она была исключена из финальной версии алгоритма.

Основной аналитической структурой данных являются времена раннего и позднего планирования операций, построенные на основе графа зависимостей. На основе времён принимаются все решения о разрыве той или иной зависимости.

Как уже отмечалось ранее, разрыв зависимостей без построения новых операций можно осуществлять всегда, не беспокоясь о том, что это может привести к увеличению нагрузки на исполняющие устройства и снизит производительность результирующего кода, поскольку не создаётся новых операций. В соответствии с этим, без ограничения общности будем считать, что все разрывы зависимостей, возможные без построения дополнительных операций, уже осуществлены и будем рассматривать только те случаи, для которых необходимо построение новой операции.

Если в операцию входят несколько зависимостей, которые можно разорвать, то их разрыв можно осуществить с помощью построения только одной операции пересылки. Это возможно даже для разрыва зависимостей различных типов. Для пояснения рассмотрим пример:

```
SUB  Vs5    1  ->  Vs1
CMP  Vs2    0  ->  P1
ADD  Vs3  Vs4  ->  Vs5  P1[T]
```

В этом примере операции ADD имеет две входящие разрываемые зависимости: антизависимость от операции SUB, и предикатную зависимость от операции CMP. С помощью построения операции пересылки можно разорвать обе зависимости:

```
SUB  Vs5    1  ->  Vs1
CMP  Vs2    0  ->  P1
ADD  Vs3  Vs4  ->  Vs6
MOV  Vs6       ->  Vs5  P1[T]
```

Все выходящие зависимости из операции, к которой применяется разрыв зависимостей, делятся на два класса. Первый класс это зависимости, которые после разрыва переносятся на новую операцию. Второй класс это зависимости, которые не переносятся, а остаются на операции, к которой применяется разрыв зависимости. В случае разрыва зависимости с помощью построения операции пересылки на новую операцию переносятся выходящие зависимости, которые соответствуют не прямым потоковым последователям.

Для разрыва зависимостей с помощью техники спекулятивности по данным, также все входящие и выходящие зависимости делятся на те, которые переносятся, и те которые не переносятся.

Таким образом, при разрыве зависимости с помощью построения новой операции всегда возможно разделить все входящие и выходящие зависимости на два класса. Первый – это те зависимости, которые переносятся на новую операцию. Второй – это те зависимости, которые никуда не переносятся, а остаются на операции, к которой применяется разрыв. Класс однозначно определяется зависимостью, её типом и свойствами. Описанное выше разделение на классы можно трактовать очень просто. Фактически такое разделение означает, что мы можем скорректировать граф зависимости при разрыве каждой зависимости, и при этом граф зависимостей после преобразования будет семантически корректным. Возможность скорректировать граф зависимостей в свою очередь означает, что мы можем семантически корректно (не изменив поведение программы) применить разрыв зависимости[1]. Если бы мы не могли скорректировать граф зависимостей при разрыве, то мы бы не смогли получить корректный результирующий код. А это означает, что преобразование нельзя применять, так как после такого применения невозможно корректно завершить процесс компиляции. Таким образом разбиение на классы переносимых и не переносимых зависимостей является естественным свойством любой из техник разрыва зависимостей.

3.3.2. Формализация задачи

Формализуем задачу в терминах теории графов. Пусть имеется произвольный граф зависимостей. Пусть задан узел A с хотя бы одной входящей зависимостью, которую можно разорвать. Построим новый узел A'. Все дуги, входящие и выходящие из узла A, перераспределяются между узлами A и A' в соответствии со своим классом, как описано выше. Все дуги, входящие в A, которые относятся к классу разрываемых, переносятся на новый узел. Остальные дуги, которые входят в A, никуда не перемещаются и остаются входными дугами этого узла. Множество дуг, выходящих из узла A, разделятся на два подмножества: дуги, которые после разрыва зависимости по-прежнему выходят из узла A, и дуги, которые после разрыва переносятся на узел A'. Все дуги выходящие из узла A, распределяются между узлами A и A' также в соответствии со своим классом. При разрыве зависимости также необходимо построить дугу с длиной, равной l, между узлами A и A'. Величина l зависит от способа разрыва зависимости и от типа операции к которой применяется разрыв зависимости. Например, для случая разрыва зависимости с помощью построения операции пересылки l равно времени выработки результата операцией, к которой применялся

[1] Необходимо отметить, что здесь возможны некоторые другие варианты. Например, консервативная коррекция или перестроение графа зависимостей (невозможно скорректировать, но возможно перестроить). Приведённые рассуждения необходимо рассматривать как иллюстрацию того факта, что разбиение на описанные классы переносимых и не переносимых зависимостей является в большой степени естественным свойством.

разрыв зависимости. Необходимо, чтобы операция завершилась, прежде чем мы сможем использовать её результат в операции пересылки.

Часть необходимых определений была введена в 1.3.2. Для формулировки алгоритма нам понадобится ещё несколько определений.

Определение. Критическим путём в графе зависимостей называется путь от ENTER-а к END-у такой, что у всех узлов в этом пути времена раннего и позднего планирования совпадают.

Определение. Дуга v, входящая в узел A, называется определяющей для узла A, если

$$early(pred(v)) + delay(v) \geq \max_{\substack{succ(u)=A \\ u \neq v}} \left(early(pred(u)) + delay(u)\right),$$

где $pred(\cdot)$ – предшественник дуги, $succ(\cdot)$ – последователь дуги, $early(\cdot)$ – время раннего планирования узла, $delay(\cdot)$ – длина дуги.

Эта формулировка означает, что фактически время раннего планирования узла задаётся его определяющей дугой.

3.3.3. Формальное описание алгоритма минимизации высоты графа зависимостей

Для описания разработанного алгоритма минимизации высоты графа зависимостей необходимо ввести понятие топологической сортировки.

Определение. Топологической сортировкой узлов графа, называется такая нумерация узлов, при которой для любой дуги номер её предшественника меньше номера её последователя.

Алгоритм топологической сортировки описан в [23]. Везде далее будет рассматриваться его частный случай, предполагающий выполнение дополнительного свойства для нумерации: если в графе разорваны некоторые зависимости, то новый узел, построенный для разрыва данной зависимости, должен иметь номер на единицу больший, чем узел, к которому была направлена эта зависимость. С таким ограничением топологическая сортировка возможна, так как между этой парой узлов есть только один путь, а именно – дуга между ними.

Ниже следует описание предлагаемого алгоритма.

Алгоритм (минимизации высоты графа зависимостей). Суть предлагаемого алгоритма состоит в том, что сначала разрываются все зависимости, которые можно разорвать, а далее восстанавливаются зависимости, разрывы которых неэффективны, то есть те, которые заведомо не уменьшают высоту графа зависимостей.

Для формулировки алгоритма существенна проблема разбиения дуг на классы. Как отмечалось в разделе 3.3.1, способ выполнения этого разбиения определяется причиной, по которой строилась данная зависимость. Отсюда следует, что разбиение проще всего сделать при построении графа зависимостей, так как в процессе его работы известен тип каждой построенной зависимости и можно сохранить о нем информацию. Сложность подобного разбиения на классы пропорциональная количеству дуг в графе зависимостей. В дальнейшем описании алгоритма предполагается, что граф зависимостей уже построен, и разметка дуг на классы задана.

Разработанный алгоритм разрыва зависимостей реализуется в четыре этапа, последовательно реализуемых функциями

```
Break_Dependences( );
Mark_Early_Time_With_Partial_Recover_Dependence( );
Mark_Late_Time( );
Final_Recover_Dependence( ).
```

Ниже с комментариями приводится полный набор функций, занятых в реализации алгоритма.

1. Функция `Break_Dependences( )` для каждого узла графа зависимостей (операции), для которого можно осуществить разрыв зависимости, вызывает функцию `Break_Dependence(node)`. Функция `Break_Dependence(node)` разрывает все зависимости, которые можно разорвать для данной операции, и запоминает их. Её действия поясняются текстом

```
Break_Dependence(node)
{
  В глобальных структурах данных, запоминается, что к узлу node шла
  зависимость, которую разорвали;
  new_node = создаётся_новый_узел;
  for edge in все предшественники node
  {
    if(edge – разрываемая зависимость)
    {
      предшественник(edge) = new_node;
    }
  }
  for edge in все последователи node
  {
    if(edge – переносимая дуга)
    {
```

```
      последователь(edge) = new_node;
    }
  }
}
```

2. Функция Mark_Early_Time_With_Partial_Recover_Dependence( ) производит разметку времён раннего планирования для всех узлов графа зависимостей с одновременным восстановлением части не эффективно разорванных зависимостей, а именно тех зависимостей, которые не являлись определяющими. В результате такого восстановления высота графа зависимостей не увеличивается, так как восстанавливаемая зависимость не определяющая.

```
Mark_Early_Time_With_Partial_Recover_Dependence( )
{
  early_time(begin) = 0;
  for node в графе в порядке топологической нумерации
  {
    max_time = 0;
    for edge in все дуги входящие в node
    {
      pred_node = предшественник(edge);
      max_time = max(max_time,время_раннего(pred_node)+длина(edge));
    }
    время_раннего(node) = max_time;
    if (к node не шла зависимость, которую разорвали
        в функ. “Break_Dependence”)
    {
      continue;
    }
    /* к node шла зависимость, которую разорвали. */
    /* получаем операцию от которой шла зависимость */
    dep_pred = предшественник_разорванной_зависимсти( node);
    /* получаем саму зависимость */
    dep_edge = разорванная_дуга(node);
    early_dep = время_раннего(dep_pred)+длина(dep_edge);
    if (early_dep > max_time)
    {
      continue;
    }else
    {
      /* восстанавливаем зависимость dep_edge */
      Rec_One_Dep(dep_edge);
```

```
    }
  }
}
```

3. Функция Mark_Late_Time( ) производит разметку времён позднего планирования.

```
Mark_Late_Time( )
{
  время_позднего(end) = время_раннего(end);
  for node в графе в порядке обратной топологической нумерации
  {
    min_time = MAX_INT; /* максимальное целое */
    for edge in все дуги выходящие из node
    {
      succ_node = последователь(edge);
      min_time = min(min_time, время_позднего(succ_node)+длина(edge));
    }
    время_позднего(node)= min_time;
  }
}
```

4. Функция Final_Recover_Dependence( ) восстанавливает неэффективные разрывы, оставшиеся невосстановленными в процедуре Mark_Early_Time_With_Partial_Recover_Dependence. Зависимость восстанавливается, если время раннего планирования нового узла не больше времени позднего планирования исходного узла, к которому шла разорванная зависимость.

```
Final_Recover_Dependence( )
{
  for node in последователь разорванных зависимостей в порядке
              топологической нумерации,
      edge in разорванная зависимость
  {
    new_node = новый узел построенный для разрыва зависимости;
    if (время_позднего(node) ≥ время_раннего(new_node))
    {
      /* восстанавливаем зависимость */
      Rec_One_Dep(edge);
      /* необходимо скорректировать времена после
```

```
          восстановления зависимости */
        Correct_Times(node,new_node);
      }
    }
}
```

5. Функция `Correct_Times(node,new_node)` выполняет коррекцию времён раннего планирования после восстановления зависимости в функции `Final_Recover_Dependence`.

```
Correct_Times(node,new_node)
{
  /* корректируются времена раннего планирования */
  delta = время_раннего(new_node)-время_раннего(node);
  node включается в список с номером delta;
  while (есть хотя бы один элемент, хотя бы в одном списке)
  {
    m =  номер максимально по номеру не пустого списка;
    cur_node = первый_элемент_списка_с_номером_m;
    увеличивается время раннего у cur_node;
    for ( succ_cur_node in все последователи cur_node)
    {
      delta_new = для succ_cur_node вычисляется изменение
                  времени раннего;
      list_num = список,_в_котором_находится(succ_cur_node)
      if (list_num == неопределенный_список)
      {
        /* нет ни к каком списке */
        succ_cur_node включается в список с номером delta_new;
      } else if  ( list_num < delta_new)
      {
        succ_cur_node удаляется из списка с номером list_num;
        succ_cur_node добавляется в список с номером delta_new;
      }
    }
  }
}
```

6. Функция `Rec_One_Dep(edge)` выполняет восстановление одной зависимости, а именно - удаляет новый узел, построенный для разрыва зависимости, и переносит все дуги входящие и выходящие из него на текущий узел.

```
Rec_One_Dep(edge)
{
  node = узел_к_которому_шла_зависимость(edge);
  new_node = узел_построенный_для_разрыва(edge);
  удаляется дуга между node и new_node;
  все предшественники и все последователи new_node переносятся на
  node;
  удаляется new_node;
}
```

3.3.4. Доказательство корректности алгоритма

Докажем, что алгоритм коррекции времён планирования (Correct_Times), используемый в алгоритме, описанном в предыдущем разделе, верен.

Утверждение 1. Приведённый алгоритм коррекции времён раннего и позднего планирования (Correct_Times) правильно корректирует времена для узлов с большими топологическими номерами, чем номер стартового узла.

Доказательство. Обозначим стартовый узел алгоритма A, новый узел, построенный для разрыва зависимости – A'. При восстановлении зависимости функцией Final_Recover_Dependence( ) у всех узлов с большим топологическим номером время позднего планирования не изменяется. Действительно, время позднего планирования, в соответствии с определением, определяется длиной максимального пути, ведущего к END, а для узла с большим топологическим номером этот путь не проходит ни через узел A, ни через новый узел A' (вспомним, что мы расширили понятие топологической сортировки и узел A имеет номер на единицу меньший, чем узел A').

Заметим, что если у некоторого узла должно измениться время раннего планирования на величину ε, то у одного из предшественников этого узла время раннего должно измениться не меньше, чем на ε. Отсюда по индукции получаем, что у всех узлов с большим номером в топологической нумерации будет скорректировано время раннего. Утверждение доказано.

3.3.5. Оптимальность алгоритма

Теперь можно показать, что предложенный алгоритм действительно формирует граф зависимостей минимальной высоты (при условии, что применяются только фиксированные типы разрыва зависимостей, которые заданы первоначальной разметкой).

Теорема. Приведённый алгоритм преобразует произвольный граф зависимостей G в граф зависимостей с минимально возможной высотой для этих преобразований.

Доказательство. На первом этапе алгоритма в функции `Break_Dependence` разрываются все зависимости. Покажем, что при восстановлении зависимостей в соответствии с алгоритмом все разрывы, которые увеличивают высоту, будут восстановлены, а те, которые могут уменьшить высоту, останутся разорванными.

Первое восстановление зависимостей происходит при исполнении функции `Mark_Early_Time_With_Partial_Recover_Dependence`. Разделим множество всех путей в графе на два подмножества: подмножество X путей проходящих через узел A (узел к которому шла разрываемая зависимость) или через узел A' (новый построенный узел для разрыва зависимости), и подмножество Y всех остальных путей. Высота графа зависимостей по определению равна:

$$h(G) = \|X \text{ Y } Y\| = \max(\|X\|, \|Y\|),$$

где $\|M\| = \{\max|\Gamma| : \Gamma \in M - \text{множество путей}\}$, а $|\Gamma|$ – в длина пути. В результате наших преобразований величина $\|Y\|$ не изменяется.

Исследуем величину $\|X\|$.

Пусть X_1 - множество всех путей идущих из `ENTER` в узел A,

X_1' - множество всех путей идущих из `ENTER` в узел A',

X_2 - множество всех путей идущих из узла A к `END`,

X_2' - множество всех путей идущих из узла A' к `END`.

Исключим из множеств X_1' и X_2' пути включающие дугу идущую от A к A'. Для высоты графа с разорванной зависимостью имеем неравенство:

$$\|X\| = \max\left(\|X_1\| + \|X_2\|, \|X_1'\| + \|X_2'\|, \|X_1\| + 1 + \|X_2'\|\right)$$

Обозначим через $\overline{X}$ соответствующую величину с неразорванной зависимостью, тогда

$$\|\overline{X}\| = \max\left(\|X_1\|, \|X_1'\|\right) + \max\left(\|X_2\|, \|X_2'\|\right)$$

Рассмотрим ветвь алгоритма в случае $early_p > maxearly$. Это означает, что $\|X_1\| < \|X_1'\|$, то есть

$$\begin{aligned}\|X\| = \max\left(\|X_1\| + \|X_2\|, \|X_1'\| + \|X_2'\|, \|X_1\| + 1 + \|X_2'\|\right) \le \\ \le \max\left(\|X_1'\| + \|X_2\|, \|X_1'\| + \|X_2'\|, \|X_1\| + 1 + \|X_2'\|\right) \le \\ \le \max\left(\|X_1'\| + \|X_2\|, \|X_1'\| + \|X_2'\|, \|X_1'\| + \|X_2'\|\right) = \|X_1'\| + \max\left(\|X_2\|, \|X_2'\|, \|X_2'\|\right) = \\ = \|X_1'\| + \max\left(\|X_2\|, \|X_2'\|\right) = \max\left(\|X_1\|, \|X_1'\|\right) + \max\left(\|X_2\|, \|X_2'\|\right) = \|\overline{X}\|\end{aligned}$$

Таким образом, при разрыве данной зависимости высота графа не увеличилась.

Теперь рассмотрим случай, когда $early_p \le maxearly$, это означает, что $\|X_1'\| \le \|X_1\|$, тогда

$$\|\overline{X}\| = \max(\|X_1\|, \|X_1'\|) + \max(\|X_2\|, \|X_2'\|) = \|X_1\| + \max(\|X_2\|, \|X_2'\|) =$$
$$= \max(\|X_1\| + \|X_2\|, \|X_1\| + \|X_2'\|) = \max(\|X_1\| + \|X_2\|, \|X_1\| + \|X_2'\|, \|X_1\| + \|X_2'\|) =$$
$$= \max(\|X_1\| + \|X_2\|, \|X_1'\| + \|X_2'\|, \|X_1\| + \|X_2'\|) \le$$

$$\le \max(\|X_1\| + \|X_2\|, \|X_1'\| + \|X_2'\|, \|X_1\| + \|X_2'\| + 1) = \|X\|$$

Следовательно, восстановление этой зависимости не увеличивает высоту графа зависимостей. Таким образом, восстановление зависимостей в функции `Mark_Early_Time_With_Partial_Recover_Dependence`, с одной стороны, не восстанавливает зависимости, разрыв которых приводит к уменьшению высоты, с другой стороны, восстанавливает часть зависимостей, которые не уменьшают высоту, причем восстанавливаются все зависимости, которые увеличивают высоту.

Покажем теперь, что восстановления зависимостей в функции `Final_Recover_Dependence` не увеличивают высоту графа. (Для всех зависимостей, разрыв которых увеличивает высоту, то есть восстановление уменьшает высоту, восстановление уже выполнено в функции `Mark_Early_Time_With_Partial_Recover_Dependence`).

Из условия $lateA \ge earlyA'$ следует, что $\|X\| - \|X_2\| \ge \|X_1'\| \Rightarrow \|X\| \ge \|X_2\| + \|X_1'\|$. Так же выполнено условие $\|X_1\| < \|X_1'\|$, ибо в противном случае разрыв этой зависимости был бы восстановлен в функции `Mark_Early_Time_With_Partial_Recover_Dependence`.

Итак:

$$\|\overline{X}\| = \max(\|X_1\|, \|X_1'\|) + \max(\|X_2\|, \|X_2'\|) = \|X_1'\| + \max(\|X_2\|, \|X_2'\|) =$$
$$= \max(\|X_1'\| + \|X_2\|, \|X_1'\| + \|X_2'\|) = \max(\|X_1'\| + \|X_2\|, \|X_1'\| + \|X_2'\|, \|X_1'\| + \|X_2'\|) =$$
$$= \max(\|X_1'\| + \|X_2\|, \|X_1'\| + \|X_2'\|, \|X_1\| + 1 + \|X_2'\|) \le$$

$$\le \max(\|X\|, \|X_1'\| + \|X_2'\|, \|X_1\| + 1 + \|X_2'\|) = \|X\|$$

Таким образом, восстановление зависимостей в функции `Final_Recover_Dependence` не увеличивает высоту графа зависимостей. Следовательно, приведённый алгоритм даёт минимально возможную высоту графа зависимостей для данных преобразований. ■

3.3.6. Оценка сложности алгоритма

Оценим сложность алгоритма минимизации высоты графа зависимостей, предложенного в 3.3.3.

Пусть e – количество дуг в графе зависимостей, m - количество зависимостей, которые можно разорвать.

Первый шаг алгоритма (функция `Break_Dependences`) выполняется за $O(m)$ действий.

Второй шаг (`Mark_Early_Time_With_Partial_Recover_Dependence`) требует обойти все дуги (обойти все узлы и от каждого узла обойти все выходящие из него дуги) – соответственно, имеем сложность $O(e)$. Кроме того, необходимо добавить $O(m)$ на анализ узлов, к которым шла разорванная зависимость. В результате имеем сложность $O(e+m)$.

На третьем шаге (`Mark_Late_Time`) также необходимо обойти все дуги (сложность $O(e)$).

На четвёртом шаге (`Final_Recover_Dependence`) для m разорванных зависимостей в худшем случае может понадобится обойти все узлы, и для каждого узла обойти всех последователей, то есть обойти все дуги, плюс ещё некоторое константное время на добавления, удаления из списков. При этом сложность равняется $P(m)R(e)\cdot const$, где $P(\cdot)$ и $R(\cdot)$ не превосходят $O(\cdot)$. В итоге, в худшем случае сложность алгоритма будет равна

$$O(m)+O(e+m)+O(e)+O(me),$$

Однако, в среднем величины $P(m)$ и $R(e)$ являются достаточно малыми. $P(m)$ мала, так как далеко не для всех зависимости будет откатываться разрыв. Величина $R(e)$ будет мала, так как коррекция времён будет быстро затухать и реально понадобится обойти лишь небольшое количество дуг.

Приведём результаты исследования показывающих скорость алгоритма. Тестовыми задачами, на которых проводилось исследования, является пакет тестов Specperf, разработанный в ЗАО “МЦСТ”, и содержит наиболее значимые фрагменты пакетов тестов SPEC CPU92 [70], SPEC CPU95[71] и SPEC CPU2000[72].

Исследуем величину $R(e)$. На Рис. 16 приведён график распределения величины $R(e)/e$, то есть график функции $F(x)=\{P(R(e)/e<x)\}$, где $P(\cdot)$ – вероятность события указанного в скобках.

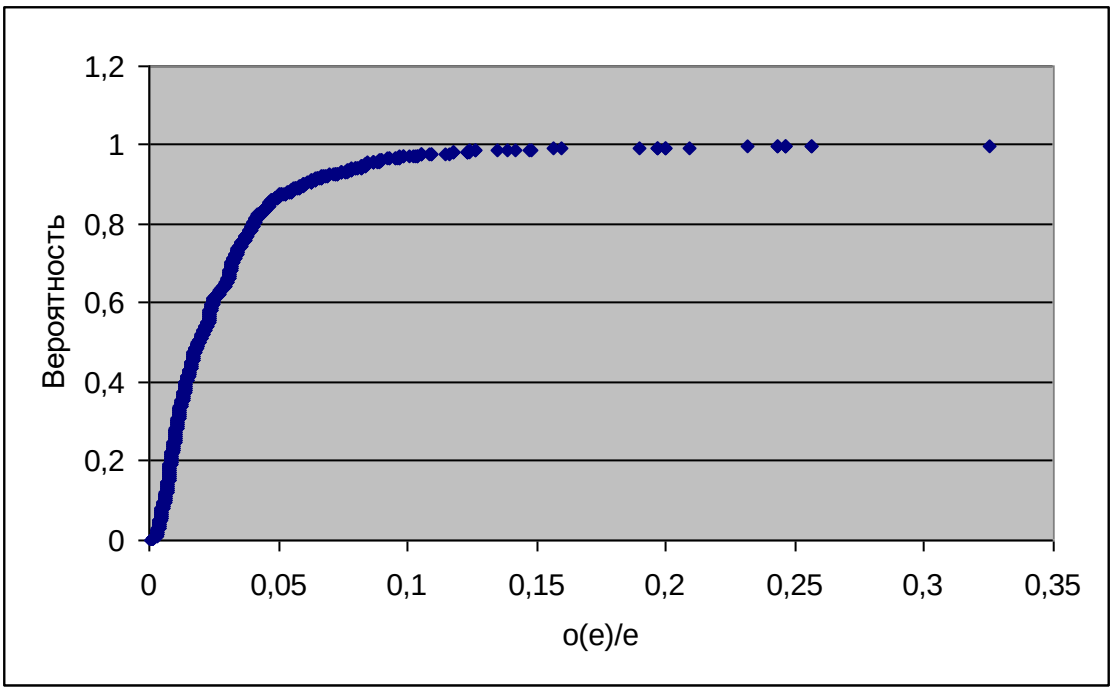

Рис. 16. *Функция распределения величины* $R(e)/e$.

Как видно из графика величина $R(e)/e$ действительно достаточно мала, в среднем её значение составляет 0,02846. Это подтверждает утверждение, сделанное выше, о том, что величина $R(e)$ достаточно мала.

Также интересной может оказаться зависимость величины $R(e)/e$ от e. Плохим случаем может оказаться следующее поведение. При больших e, становится большим и значение $R(e)/e$. На Рис. 17 приведён график отображения e α $R(e)/e$.

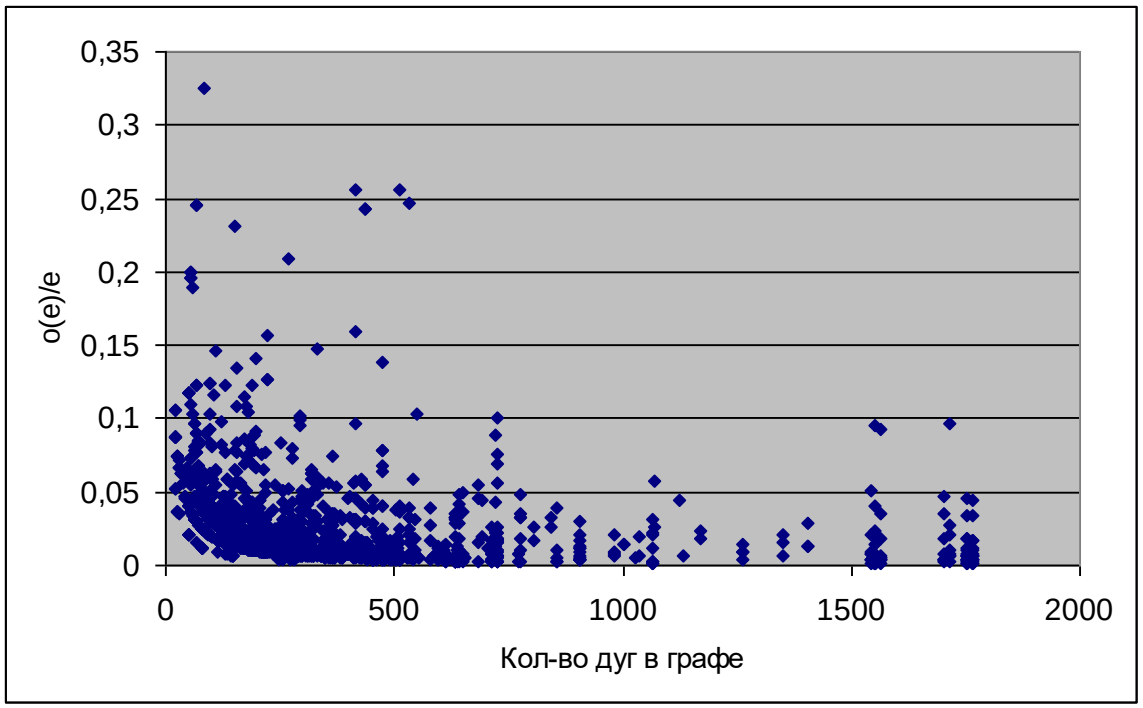

Рис. 17. *Зависимость* $R(e)/e$ *от числа дуг в графе зависимостей.*

Как видно из графика описанного роста не происходит.

Приведённое исследование показывает, что предлагаемый алгоритм действительно обладает хорошими скоростными характеристиками в среднем.

3.4. Избавление от излишней спекулятивности для операций чтения из памяти

Как уже отмечалось ранее снятие предиката у операций чтения из памяти может оказывать негативный эффект на итоговую производительность кода. Это связано с увеличением нагрузки по подсистему памяти: вытеснение данных из кэша, увеличение потока обращений в память и так далее. Предугадать будет ли некоторое обращение в память оказывать негативный эффект достаточно трудно, поэтому после построения предикатного кода производится снятие предиката у всех обращений, для которых это возможно. Для того чтобы сгладить негативный эффект от снятия предиката у операций чтения из памяти был реализован алгоритм, который некоторым операциям чтения из памяти возвращает предикат, если это не увеличивает высоту графа зависимостей.

Не всем операциям чтения из памяти можно вернуть предикат. Это в основном связано с техническими деталями в трансляторе. Некоторые оптимизации, которые производятся после построения предикатного кода, достаточно сильно меняют промежуточное представление. В таких условиях сохранение информации об оригинальном предикате становится сложной

задачей, которая к тому же является ресурсоёмкой по времени работы. Также возникают ситуации, когда предикат операции вернуть можно, однако для этого требуется провести сложные и ресурсоёмкие по времени компиляции преобразования над промежуточным представлением. Примером может служить случай, когда операция, вырабатывающая предикат, в линейке операций стоит ниже, чем операция, которой предикат возвращается. В таком случае необходимо поднять операцию, вырабатывающую предикат, выше операции, которой возвращается предикат, по линейке операций, либо опустить операцию, которой возвращается предикат, ниже в линейке операций. Для осуществления такого преобразования может потребоваться перемещение предшественников или последователей рассматриваемых операций и так далее. Таким образом, для достижения нужного порядка могут потребоваться нелокальные алгоритмы. Мы не будем подробно останавливать на описании того, как сохраняется информация об оригинальном предикате операции, когда возвращение предиката возможно и какие преобразования необходимо произвести для этого, так как это несколько уходит от тематики данной работы. Мы положим, что у нас имеется интерфейс, который возвращает для заданной операции чтения из памяти операцию, вырабатывающую предикат, который можно вернуть рассматриваемой операции чтение, и при этом семантика программы останется корректной.

Возвращение предиката производится, если величина равная времени готовности предиката для использования плюс некоторая небольшая константа, меньше либо равно времени позднего операции. Такое условие означает, что высота графа зависимостей не изменится. Прибавление к времени готовности предиката небольшой константы служит для того, чтобы в случае, если в результате планирования выработка предиката задержится (например, из-за нехватки арифметических устройств для планирования), был некоторый запас, который позволит не увеличится времени исполнения данного гиперблока. Конечно же, данный запас не даёт гарантии, что высота вычислений не увеличится. Однако, при значение константы равном двум тактам, результаты экспериментов показали, что такое преобразования не даёт значительных замедлений на большом наборе тестов, но при этом на некоторых задачах, для которых критична скорость работы с памятью, получился серьёзный выигрыш в скорости их работы.

В случае если время готовности предиката плюс константа оказалось больше, чем время позднего операции, и при этом операция выработки предиката является операцией логического “и” над предикатами, то производится попытка поставить операцию чтения под частичный

предикат. Берётся первый аргумент операции логического “и”[1]. Если он является прямым потоковым последователем (то есть, вне зависимости от пути в программе вырабатывается одной операцией, а не нескольким), то берётся этот потоковый предшественник и производится попытка поставить операцию чтения из памяти под этот предикат. При этом применяются те же эвристики по временам планирования. Если же операцию чтения не удалось поставить и под этот предикат, и операция является логическим “и” двух предикатов, то всё повторяется аналогично.

Приведём формальное описание алгоритма возвращения предиката операции чтения из памяти.

Алгоритм (возвращения предиката операции чтения из памяти)

На вход подаётся `oper` – операция чтения из памяти.

1. Если возвращение предиката возможно, то predct_oper равняется операции вырабатывающей возвращаемый предикат, иначе завершить работу
2. Если время готовности предиката, вырабатываемого predct_oper, плюс CORRECT_TIME_DELAY меньше либо равно времени позднего oper, то вернуть_предикат( ) и завершить работу
3. Если predct_oper является операцией логического “и” и её первый аргумент имеет прямого потокового предшественника pred_predct_oper, то predct_oper := pred_predct_oper и перейти к шагу 2 алгоритма.

В функции `вернуть_предикат( )` производится проверка на возможность продлить время жизни возвращаемого предиката до операции `oper`. Возвращается предикат и корректируется промежуточное представление. Корректируется граф зависимостей. Корректируется времена раннего планирования, так же как это делалось ранее в алгоритме отката разрыва зависимостей.

[1] Операции логического “и” возникают при построении предикатного кода и имеют следующую структуру. Первый аргумент является полным предикатом текущего узла. Второй аргумент является оригинальным предикатом условного перехода. Предикат узла строится по предикатам входящих в него дуг. Таким образом, использование в алгоритме первого аргумента операции логического “и” соответствует попытке поставить операцию под наиболее близкий, но более широкий предикат.

3.5. Схема работы двоичного оптимизирующего транслятора с учётом алгоритмов минимизации высоты графа зависимостей

На Рис. 18 приведена схема работы двоичного оптимизирующего транслятора для архитектуры “Эльбрус” с учётом всех используемых техник минимизации высоты графа зависимостей.

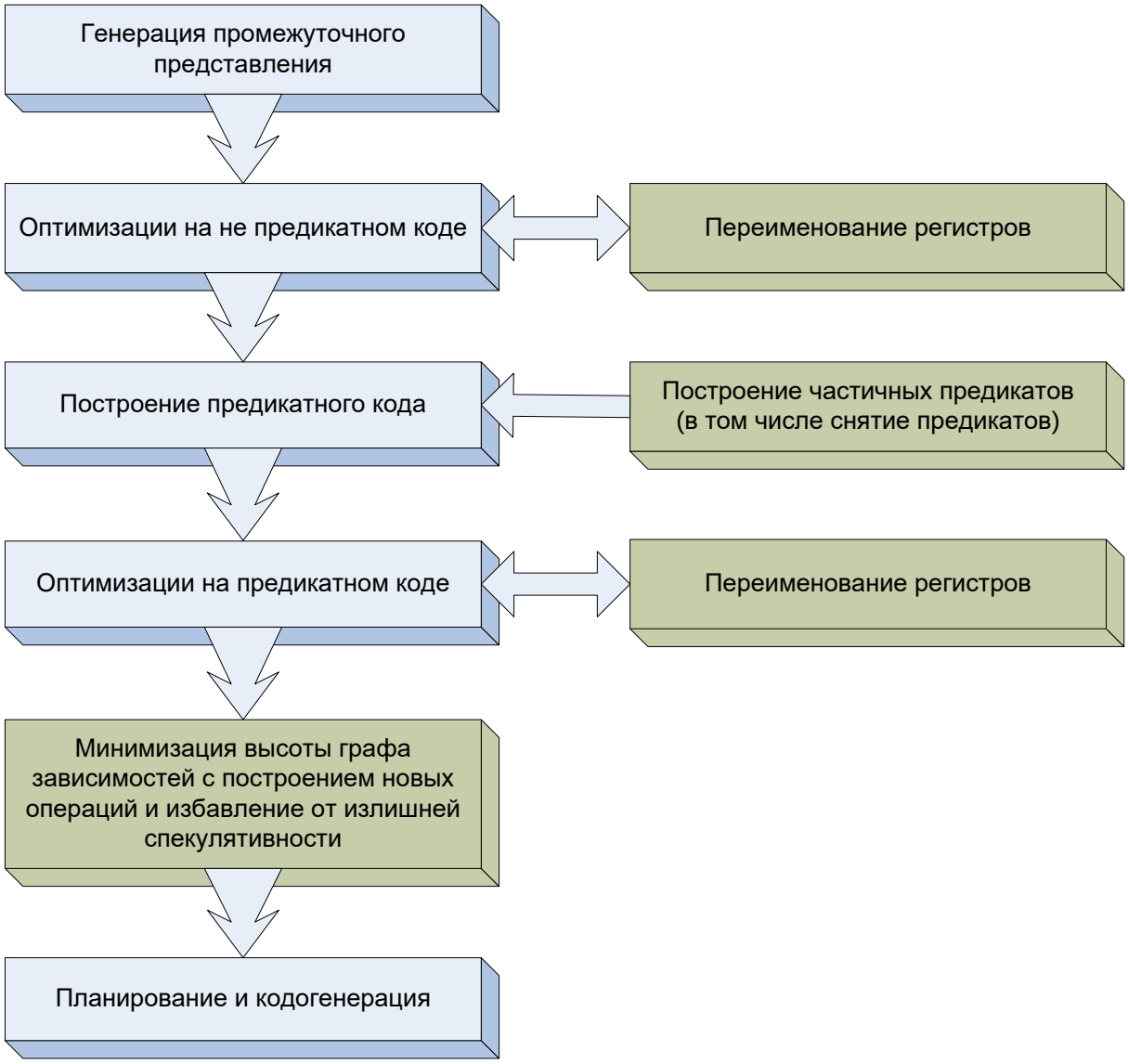

Рис. 18. *Схема работы двоичного оптимизирующего транслятора для архитектуры “Эльбрус” с учётом всех используемых техник минимизации высоты графа зависимостей*

Кроме выделенной цветом новой функциональности по разрыву зависимостей, также был модифицирован алгоритм построения предикатного кода. Остановимся на этом моменте подробнее. Построение предикатного кода, как упоминалось ранее, производиться с помощью алгоритма if-conversion. Алгоритм if-conversion работает пошагово. На каждом шаге

рассматривается некоторое множество узлов и принимается решение будет ли выгоднее (с точки зрения качества результирующего кода) объединить (слить) эти узлы в один гипер-узел[1]. Если такое объединение выгодно, то узлы объединяются в гипер-узел и происходит переход к следующему шагу алгоритма. Принятие решение об объединение узлов основывается на оценке скорости работы слитого и не слитого кодов. В проекте, в рамках которого осуществлялась данная работа, решение принималось на основе результатов планирования[2] слитого и не слитого кодов. Но результаты работы алгоритма минимизации высоты графа зависимостей очень сильно влияют на результаты планирования, а минимизация работает уже после построения предикатного кода. Таким образом в процессе работы алгоритма if-conversion необходимо каким-то образом учитывать, что в дальнейшем будет работать алгоритм минимизации высоты графа зависимостей, так как в противном случае оценки сделанные на этапе if-conversion будут очень неточными. Такой учёт был реализован и суть его состоит в следующем. Все зависимости, которые могут быть разорваны, не учитываются при проведении оценок при слиянии узлов. Таким образом алгоритм if-conversion считает, что все зависимости разорваны, а затем, когда уже будет работать минимизация высоты графа зависимости, те зависимости, которые окажутся на критических путях, будут разорваны.

3.6. Экспериментальные результаты

Для анализа эффективности работы алгоритмов использовался статический двоичный оптимизирующий транслятор для микропроцессора “Эльбрус” и потактовый симулятор этой архитектуры. Статический двоичный оптимизирующий транслятор для микропроцессора “Эльбрус” переводит коды приложения архитектуры x86 в семантически эквивалентное приложение в кодах архитектуры “Эльбрус”. Для выявления горячих областей исходного приложения, сбора профильной информации и обнаружения точек входа в коды используется предварительный запуск приложения в со специальным инструментированием для сбора этой информации.

Потактовый симулятор микропроцессора “Эльбрус” моделирует архитектуру с точностью до такта. В нём полностью моделируются все стадии конвейера и регистровый файл. Также моделируется кэш память первого и второго уровня, задержки и темп обращений в память. Отрабатываются все остановки конвейера связанные с неготовностью данных или команд к исполнению.

---

[1] Напомним, что объединение нескольких узлов в один гипер-узел осуществляется с помощью простановки операций под предикат.

[2] Необходимо заметить, что для обеспечения высокой скорости работы использовался упрощённый алгоритм планирования, который не учитывал некоторые ограничения присущие целевой машине.

Для анализа изменения скорости работы результирующих кодов использовалось следующие методы:

- Запуск статически скомпилированных кодов задач из пакета SPEC CPU2000 [72] на потактном симуляторе микропроцессора “Эльбрус-С”. Для ссылок на этот метод будем использовать фразу “замеры на симуляторе”
- Технология предсказания времени работы кода на основе планирования. Время работы процедуры вычисляется как

$$\sum_{\substack{по\\ всем\\ операциям\\ передачи\\ управления}} C \cdot L$$

где C - счетчик операции передачи управления, а L - расстояние в тактах от начала узла до инструкции в которой спланирован переход. Описанная величина представляет из себя время работы кода с точки зрения транслятора. Здесь могут оказаться неучтёнными задержки связанные с промахами в кэш, блокировками не учитываемыми транслятором и т.д. Однако такой подход позволяет более точно оценить вклад алгоритма в качество результирующего кода. Причина этого в отсутствии шума связанном, например, с плохой предсказуемостью времени обращения в память[1]. В качестве тестов для анализа брались горячие участки задач из пакетов SPEC CPU95 и SPEC CPU2000, а также горячие участки операционной системы Windows и типичных пользовательских приложений для неё. Именно такие участки в первую очередь оптимизируются самым высоким уровнем системы двоичной трансляции. Для ссылок на этот метод будем использовать фразу “предсказание на основе планирования”.

На Рис. 19, Рис. 20 и Рис. 21 приведены результаты исследования эффективности предложенного алгоритма минимизации высоты графа зависимостей с построением новых операций. Сравнивалось времени работы результирующих кодов с включённым алгоритмом и с выключенным алгоритмом.

[1] Для пояснения предположим, что у нас есть некоторая задача проводящая 90% времени своей работы в ожидании прихода данных из памяти. Допустим мы реализовали некоторую оптимизацию сокращающую время вычислений данной задачи на 10%. В таком случае время работы задачи сократиться всего на 1%. Раньше время складывалось как 0.9+0.1=1.0, после реализации оптимизации 0.9+(0.1-10%)=0.99.

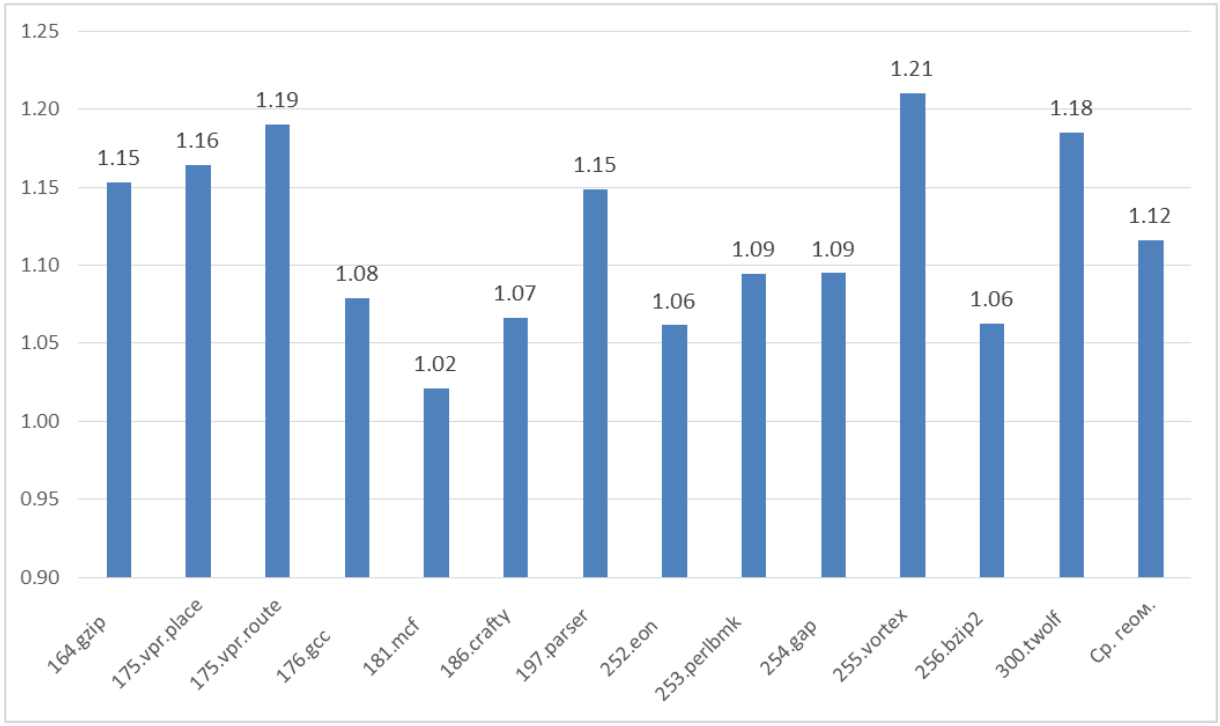

Рис. 19. *Влияние алгоритма минимизации высоты графа зависимостей с построением новых операций на время работы результирующего кода на целочисленных задачах пакета SPEC CPU2000. Замеры на симуляторе.*

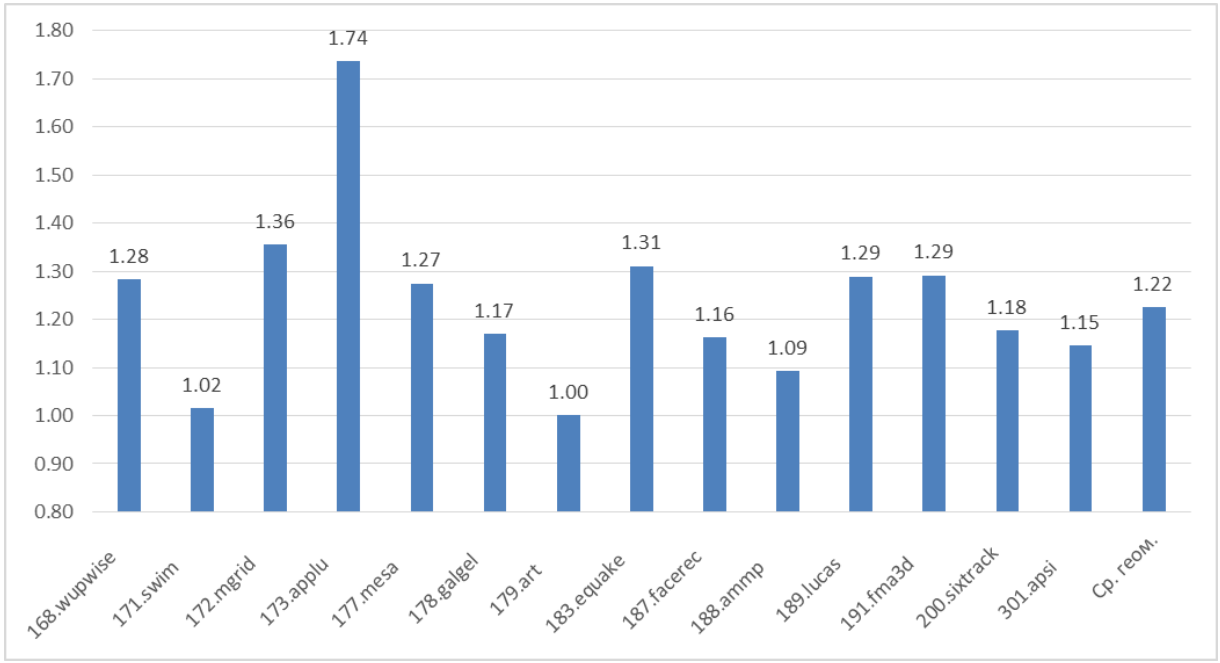

Рис. 20. *Влияние алгоритма минимизации высоты графа зависимостей с построением новых операций на время работы результирующего кода на вещественных задачах пакета SPEC CPU2000. Замеры на симуляторе.*

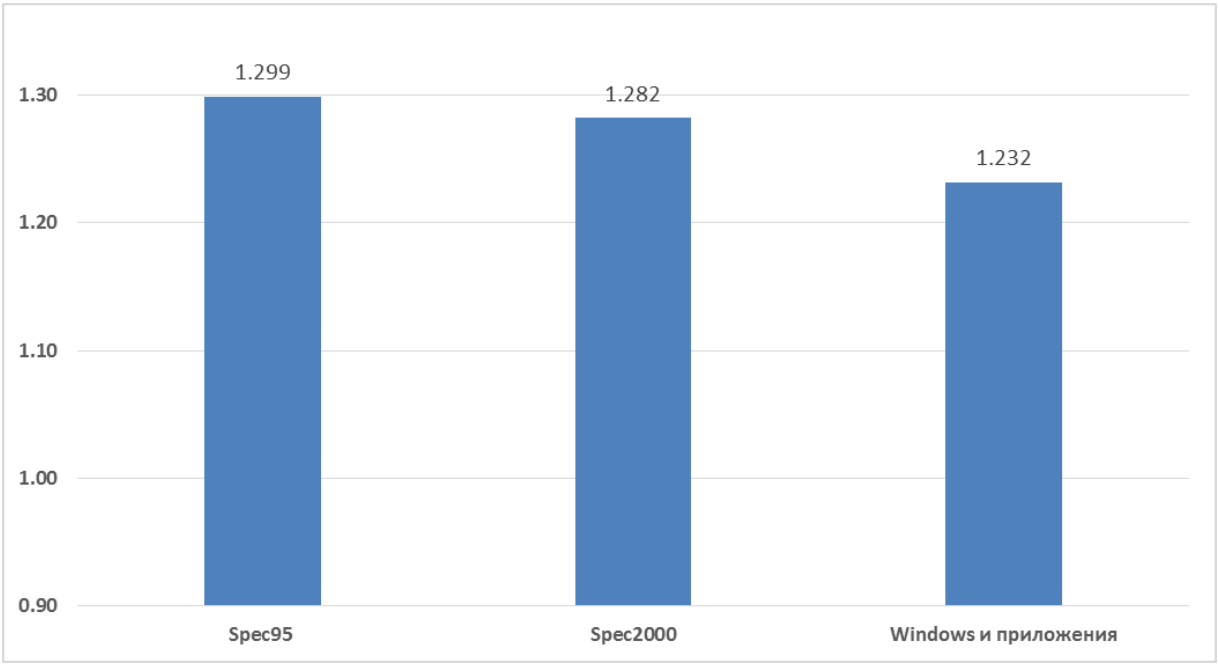

Рис. 21. *Влияние алгоритма минимизации высоты графа зависимостей с построением новых операций на время работы результирующего кода на горячих участков SPEC CPU95, SPEC CPU2000, Windows и пользовательских приложений. Предсказание на основе планирования.*

Также был произведён замер времени затрачиваемое на работу алгоритма минимизации высоты графа зависимостей. Оно составило 3,8% от общего времени трансляции.

3.7. Выводы

1. В данной главе предложен быстрый алгоритм минимизации высоты графа зависимостей с построением новых операций. Он позволяет добиться минимально возможной высоты графа зависимостей. Лишние разрывы зависимостей с построением новых операций не производятся.
2. Дано формальное описание этого алгоритма и приведено строгое доказательство его оптимальности. Также произведена оценка сложности этого алгоритма, которая оказалась приемлемой для использования этого алгоритма в динамическом трансляторе.
3. Предложен алгоритм избавления от излишней спекулятивности для операций чтения из памяти. Он с одной стороны не увеличивает критические пути, а с другой позволяет уменьшить поток обращений в память, что позволяет более эффективно использовать кэш.
4. Приведена схема взаимодействия и порядка применения алгоритмов минимизации высоты графа зависимостей с другими оптимизирующими преобразованиями. Это схема позволяет

избавиться от логических циклов возникающих в случае, когда решения по применению тех или иных преобразований зависит друг от друга.

5. Приведённые экспериментальные результаты показывают высокую эффективность предложенных методов на широком классе задач. Предложенный алгоритм минимизации высоты графа зависимостей даёт прирост производительности на **12%** на целочисленных задачах пакета SPEC CPU2000, на **22%** на вещественных задачах пакета SPEC CPU2000 и на **23%** на наборе горячих участков из Windows 2000 и пользовательских приложений. Время работы алгоритма составило 3.8% от общего времени трансляции.

4. Сокращение длины критических путей в циклических областях

В начале этой главы вводятся необходимые определения и производится обзор известных методов конвейеризации циклов. Затем рассматривается вопрос подсчёта ограничений снизу на размер цикла и описываются используемые нами методы оценки этого значения. Далее вводится понятие расширенного графа зависимостей, а также понятия времён раннего и позднего планирования на нём, и приводится алгоритм расчёта этих времён. Доказывается корректность и оптимальность этого алгоритма. Затем описывается алгоритм конвейеризации циклов, основой которого являются времена планирования. Описывается как в алгоритм конвейеризации были интегрированы различные техники разрыва зависимостей. Далее производится оценка сложности алгоритма разметки времён и алгоритма конвейеризации цикла, как с теоретической точки зрения так и с практической. Также приводится описание того, как была решена проблема восстановления точного контекста при использовании вращающихся регистров при работе предложенного алгоритма конвейеризации циклов. В заключение главы приводятся результаты экспериментов.

4.1. Обзор существующих алгоритмов конвейеризации циклов

4.1.1. Основные определения

Введём несколько определений.

Определение. Предциклами называются узлы графа управления, из которых имеется вход в цикл. То есть сам узел не принадлежит циклу, а его последователь принадлежит циклу.

Определение. Постциклами называются узлы графа управления, в которые имеются выходы из цикла. То есть сам узел не принадлежит циклу, а его предшественник принадлежит циклу.

Всегда можно добиться того, чтобы все предциклы имели ровно одну выходную дугу (ведущую в цикл). Для этого достаточно на всех дугах, кроме обратной, ведущих в цикл, вставить пустой узел. Аналогично можно добиться того, чтобы в каждый постцикл входила ровно одна дуга (выход из цикла). Подавляющее большинство алгоритмов конвейеризации обрабатывает так называемые *сводимые циклы*. Это такие циклы, которые имеют единственную точку входа, то есть все входные дуги ведут в один и тот же узел. Для сводимого цикла можно добиться того, чтобы у него имелся единственный предцикл. Для этого достаточно создать новый узел, имеющий одну выходную дугу, последователем которой является точка входа в цикл, а все дуги входящие в цикл перенаправить на этот узел. Большинство алгоритмов

конвейеризации циклов применяются к циклам с одной обратной дугой, поэтому мы при дальнейшем будем рассматривать только такие циклы. Циклы с несколькими обратными дугами путём не сложных преобразований графа управлению могут быть приведены к циклу с одной обратной дугой.

Итак везде в дальнейшем, если это не оговорено специально, мы будем рассматривать циклы имеющие одну точку входа, одну обратную дугу, несколько выходных дуг и один предцикл с одной выходящей дугой. В разделе 4.7 будут описаны преобразованиям управляющего графа, которые позволяют циклы более сложной структуры свести к данному типу циклов.

Определение. Логической итераций цикла называется итерация цикла, которая была в исходной программе. То есть множество всех операций от входа в цикл (цель перехода обратной дуги) до всех обратных дуг.

Определение. Физической итерацией цикла называется итерация цикла, которая получилась в результирующем коде, полученном после трансляции кода.

Определение. Интервалом начала итерации (ИНИ) называется величина, которая указывает на то через сколько тактов запускается новая итерация цикла. То есть каждые ИНИ тактов запускает новая итерация цикла.

Для уже спланированного кода ИНИ равен размеру физической итерации цикла.

4.1.2. Модульное планирование.

Модульное планирование (modulo scheduling) – мощная и важная техника конвейеризации циклов – была впервые предложена в работе [78]. В дальнейшем предложенный алгоритм использовался в качестве исходного в работах [79] и [100]. Модульное планирование работает следующим образом. Сначала производится оценка минимально возможного размера ИНИ[1]. Затем мы пытаемся спланировать цикл в ИНИ тактов. Операции планируются по очереди. Часто используется техника планирования по списку: имеется список готовых к планированию операций и из него выбирается, в соответствии с некоторыми эвристиками, операция и если для неё есть свободное устройство, то она планируется. В модульном планировании все такты равные по модулю ИНИ отождествляются[2]. Например, если ИНИ равно 3, то имеется три группы тактов (0,3,6...), (1,4,7...) и (2,5,8...). Если операция

[1] Например, это можно сделать по ресурсной оценки цикла: если в цикле имеется 10 операций, а целевая архитектура может выполнять до трёх операции за одну инструкцию, то выполнить цикл быстрее, чем за 4 такта не получится. Подробнее об оценках снизу для ИНИ будет ниже, в разделе 4.3.

[2] Собственно из этого свойства и произошло название модульное планирование

планируется в какой-то такт, то она сразу планируется и во все остальные такты группы (точнее говоря её копия на других итерациях цикла). При этом должны быть учтены все ресурсные ограничения и все зависимости на каждом такте группы. В случае, если нам не удалось спланировать какую-то операцию ни в один из тактов, производится откат всего планирования, ИНИ увеличивается на единицу и снова запускается планирование и так до тех пор пока планирование не завершиться успешно.

Первоначальный вариант модульного планирования предложенный в работе [78] имел серьёзное ограничение: он напрямую не применялся к циклам, у которых имелись условные передачи управления в теле цикла. Ряд дальнейших работ был посвящён снятию этого ограничения. В работе [79] предлагается метод называемый иерархическая редукция (hierarchical reduction). В этом методе обе ветви для всех условных переходов[1] сначала планируются независимо. Затем каждое такое спланированное ветвление рассматривается как одна макрооперация требующая ресурсов столько, сколько необходимо для планирования всех операций входящих в это ветвление и производится конвейеризация цикла.

Другой подход к конвейеризации цикла с ветвлениями, называющийся Enhanced Modulo Scheduling (EMS) был описан в работах [82], [83]. Рассматриваются циклы имеющие одну обратную дугу. Для того, чтобы избавиться от передач управления в теле цикла к циклу применяется if-conversion [80] перед проведением модульного планирования. С помощью if-conversion можно преобразовать одновходовую область произвольной топологии управления в один узел. Для этого используется предикатное исполнение и каждая передача управления заменяется на зависимость по данным для предиката. Схема применения EMS к циклу следующая:

1. Применяется if-conversion к циклу, для того чтобы преобразовать его в одноузловой участок.
2. Строится расширенный граф зависимостей.
3. Применяется модульное планирование к телу цикла

Также в работе было предложено применение рассматриваемого алгоритма для архитектур, которые не поддерживают предикатное исполнение. Для этого из предикатного кода восстанавливается не предикатный код с помощью техники reverse if-conversion [81] после модульного планирования. Таким образом, к алгоритму добавляется ещё один пункт:

4. Восстановление явной управляющей структуры кода путём вставления явных условных операций передачи управления.

[1] Ветви – “then” и “else” в терминах языка высокого уровня

Недостатком EMS является то, что всегда необходимо производить полное слияние цикла в один узел. Это может оказаться не всегда эффективным, так как все пути в оригинальном цикле исполняются одновременно и может оказаться, что не хватает исполняющих ресурсов для того, чтобы исполнять одновременно все ветви управления. Это особенно актуально для архитектур у которых количество операций выполняемых за одну инструкцию не очень велико. Ещё одним серьёзным недостатком является то, что во время работы алгоритма не может меняться промежуточное представление (например строится новые операции). В силу этого зависимости не могут быть разорваны во время работы алгоритма, и их необходимо разрывать до начала конвейеризации. Однако в этот момент достаточно трудно определить какие зависимости надо разрывать, а какие нет. Также существенное ограничение вносит тот факт, что обрабатываются только циклы с одной обратной дугой. Если в цикле было несколько обратных дуг и для проведения конвейеризации цикл был преобразован в цикл, имеющий одну обратную дугу, то время работы одной итерации цикла в случае прохода по любой из первоначальных обратных дуг будет одинаковым, и равняется максимальному времени из всех дуг. Таким образом возможны существенные потери в качестве результирующего кода в случае наличия нескольких несбалансированных обратных дуг.

4.1.3. URCR, URPR, GURPR и GURPR*

Рассмотрим ещё одну группу преобразований, которая применяется для конвейеризации циклов. Техника получившая название URCR (UnRolling, Compaction, and Rerolling) является развитием техники планирования трасс (trace scheduling) [85]. Она применяется только к циклам, которые состоят из одного узла управления. Цикл раскручивается[1] на два (тело дублируется дважды), затем операции объединяются вместе и перемешиваются, а затем ищется самый короткий интервал, который содержит все операции цикла. Этот интервал и становиться новым телом цикла.

Алгоритм URPR (UnRolling, Pipelining, and Rerolling) был представлен в работе [86] и является модификацией URCR, которая более точно обрабатывает рекуррентности. Однако по-прежнему алгоритм может применяться только к циклам, состоящим из одного узла управления.

Следующая версия рассматриваемого алгоритма, называемая GURPR (Global URPR), расширяет область применение алгоритма на случай, когда в теле цикла имеется условная передача управления [87].

И наконец в алгоритме GURPR* [84] модифицирована техника работы с условными переходами для повышения эффективности.

[1] Такое преобразование в англоязычной литературе называется unroll

Основным преимуществом этих алгоритмов является их простота и высокая скорость работы. Однако код, который получается в результате их работы, далёк от оптимального варианта. Это происходит из-за неоптимального распределения ресурсов: в процессе работы алгоритма в теле цикла могут быть спланированы лишние операции, которые из финального варианта будут удалены, а они займут некоторые ресурсы, которые фактически не будут использованы. Также негативный фактор в производительность результирующего кода вносит тот факт, что степень конвейеризации (наложения) операций ограничена величиной раскрутки цикла.

4.1.4. Enhanced Pipeline Scheduling

Enhanced Pipeline Scheduling (EPS) для конвейеризации циклов использует техники переноса операций и планирования. Этот алгоритм был предложен в работах [94] и [95]. В EPS используется подход отличный от других техник, основанный на переносе операций между линейными участками. В процессе работы алгоритма операции переносятся вверх через голову цикла. Для этого операция копируется и копия помещается в предцикл, а сама операция перемещается по обратной дуге и ставиться перед переходом по обратной дуге. Фактически получается, что перенесённая операция становиться операцией со следующей итерации. Предцикл и постцикл генерируется автоматически в результате переноса операций и дублирования кода. Длины дуг графа зависимостей более единицы обрабатываются путём вставления фиктивных узлов. Ещё одной важной особенностью алгоритма является то, что на каждом шаге алгоритма мы получаем корректное представление. Если многие другие алгоритмы конвейеризации циклов в случае невозможности произвести какое-то действие (например распределение регистров), полностью отказываются от конвейеризации, то в EPS возможна частичная конвейеризация. То есть мы получаем не идеально конвейеризированный цикл, но тем не менее вариант цикла, который выполняется быстрее первоначального. Также в силу того, что на каждом шаге у нас имеется корректное промежуточное представление и вообще говоря следующий шаг не зависит от предыдущего, то в процессе работы алгоритма у нас есть возможность выполнять различные преобразования, которые могут улучшить результирующий код. В первую очередь это разрыв различных типов зависимостей.

Работа алгоритма ведётся на спланированном коде и перенос операций осуществляется из одной инструкции в другую. Непосредственно сам алгоритм состоит из двух частей. В первой части все операции в цикле переносятся вверх настолько насколько это возможно. Во второй части, первая инструкция цикла дублируется на две копии. Первая копия ставиться перед циклом и образует предцикл, вторая копия переносится в конец цикла и ставиться перед переходом по обратной дуге, фактически эти операции становиться операциями со следующей итерации. Далее работа алгоритма продолжается до тех пор пока все операции из

первоначального тела цикла не получат возможность перенестись. Если цикл имеет несколько обратных дуг, то по каждой дуге переносится отдельная копия операций из первой инструкции. Для поднятия эффективности алгоритма в процессе переноса операций вверх применяется несколько преобразований: разрыв зависимостей с помощью построения операции пересылки и сбор общих подвыражений[1]. В рассматриваемом алгоритме постцикл генерируется в случае, когда некоторый условный выход из цикла в результате переноса проносится вверх мимо других операций. Копии всех операции, мимо которых перенёсся выход, должны быть помещены после выхода.

Главным достоинством этого алгоритма является то, что на каждом шаге алгоритма имеется корректное промежуточное представление. Эта особенность выгодно отличает данный алгоритм от всех других алгоритмов конвейеризации циклов. Данное свойство даёт следующие преимущества. Во-первых, позволяет в процессе работы алгоритма производить различные преобразования, например разрыв зависимостей. Во-вторых, есть возможность получить частично конвейеризованный цикл, в случае нехватки каких-либо ресурсов, например регистров. Ещё одним достоинством данного алгоритма является то, что не происходит значительного роста размера результирующего кода. Новые операции могут появляться только в результате преобразований, количество которых легко поддаётся контролю.

Недостатки алгоритма следующие. Во-первых, не всегда достигается оптимальная величина размера физической итерации. Например, модульное планирование в этом плане показывает более хорошие результаты. Ухудшение качества результирующего кода происходит за счёт того, что некоторые операции могут быть преждевременно перенесены вверх, они займут ресурсы и тем самым помешают другим операциям, более важным с точки зрения конвейеризации, перенестись вверх. Во-вторых, в результате работы алгоритма не может получиться дробное значение размера физической итерации.

4.1.5. Другие алгоритмы конвейеризации

Разработан ряд других алгоритмов конвейеризации циклов. Однако их применение в оптимизирующем двоичном трансляторе затруднительно, так как они не удовлетворяют всем предъявляемым требованиям. Два наиболее известных алгоритма: Perfect pipelining [88], [89] и алгоритм с использований сетей Петри [92], [90], [91] описаны в приложении.

[1] Сбор общих подвыражений является важным преобразование, так как в процессе конвейеризации цикла с несколькими обратными дугами появляется много копий одной и той же операции, которые в дальнейшем можно собрать.

4.1.6. Проблемы и недостатки существующих методов конвейеризации циклов

Из всех описанных выше алгоритмов конвейеризации циклов практическое применение в промышленных компиляторах получили два. Первый – модульное планирование используется в Intel IA-64 Compiler. В [73] можно найти достаточно подробное описание используемого там алгоритма. Вторым алгоритмом является Enhanced Pipeline Scheduling, используемый в Sun Studio Compiler [98].

Модульное планирование в сочетании с вращающимися регистрами является очень хорошим алгоритмом с точки зрения качества результирующего кода. За счёт того, что мы сразу планируем цикл, а лишь потом определяем на сколько итераций прокрутилась та или иная операция, у нас имеется очень много свободы для планирования, а это даёт нам возможность получиться очень плотный код. Также за счёт того, что в случае неудачи при текущем ИНИ мы его итеративно увеличиваем до тех пор пока нам не удастся получить конвейеризированный цикл, алгоритм почти всегда осуществляет конвейеризацию цикла и следовательно увеличивается скорость работы результирующего кода.

Недостатком модульного планирования является его достаточно низкая скорость работы. Как мы уже замечали ранее, правила и алгоритмы планирования для EPIC архитектур достаточно сложные. В данном же алгоритме может потребоваться несколько раз спланировать цикл, прежде чем алгоритм даст финальный результат. Ещё одним недостатком алгоритма является то, что в процессе его работы промежуточное представление цикла должно оставаться неизменным. Из этого свойства следует, что в процессе работы алгоритма нельзя разрывать никакие зависимости путём создания новых операций. В таком случае все зависимости необходимо разорвать до начала работы алгоритма. Однако в этот момент очень трудно определить какие зависимости надо разрывать, а какие нет. Тут можно попробовать следующий вариант. Сначала спланировать цикл без разрыва зависимостей, потом спланировать цикл предварительно разорвав все зависимости и из этих двух вариантов выбрать лучший. Такая техника допустима, когда разрываемых зависимостей достаточно мало, например в языковом компиляторе. Однако в двоичном трансляторе, как уже упоминалось выше, разрываемых зависимостей очень много и вариант, когда мы разрываем либо всё, либо ничего, получается далёким от оптимального.

Алгоритмы описанные в 4.1.3 являются одними из первых подходов к конвейеризации циклов и дают не очень хороший результирующий код. Алгоритм Perfect Pipelining описанный в Приложении А хоть и имеет полиномиальную сложность может приводить к очень большому дублированию кода, и в таких случаях скорость его работы будет низкой. Данный алгоритм обеспечивает очень хорошую конвейеризацию в условиях бесконечных ресурсов, однако в

случае ресурсных ограничений качество кода снижается. Ещё одной проблемой алгоритма является сложность определения момента остановки, то есть момента, когда планирование начинает повторяться. Алгоритм использующий сети Петри и описанный в Приложении Б получает очень хороший результирующий код. Но имеет тот же недостаток, что и Perfect Pipelining: сложность определения эквивалентных состояний. Наконец, все алгоритмы, рассмотренный в этом абзаце, не позволяют в процесс своей работы разрывать зависимости, а следовательно нам придётся либо разрывать их все, либо не разрывать вообще.

Алгоритм EPS, описанный в 4.1.4, является единственным алгоритмом конвейеризации циклов, который на протяжении всей своей работы оставляет тело цикла в корректном состоянии. Именно это свойство выделяет его на фоне других алгоритмов. Имеются два важных следствия это свойства. Во-первых, возможность в процессе работы алгоритма разрывать зависимости. Во-вторых, если даже алгоритм не может по какой-то причине продолжить свою работу, получившейся в данный момент цикл уже является частично конвейеризованным, и скорость его работы будет выше первоначального варианта. Это свойство позволяет алгоритму быть достаточно быстрым и предсказуемым по времени своей работы. Эти два свойства алгоритма EPS являются очень важными и полезными для динамических оптимизирующих трансляторов. Именно поэтому идеи заложенные в EPS были взяты за основу алгоритма конвейеризации циклов предложенном в этой работе. Однако несмотря на все достоинства EPS, предложенные раннее варианты его реализации обладают серьёзными недостатками, в первую очередь связанными с качеством результирующего кода. Выбор операций для переноса по обратной дуге не является оптимальным: могут как переноситься лишние операции, так и не перенестись некоторые операции, которые надо было переносить. Это может приводить как к ухудшению качества результирующего кода, так и к излишнему дублированию операций. Ещё одним недостатком является то, что не обрабатываются зависимости реализуемые более, чем через одну итерацию, что опять же может сказываться на качестве результирующего кода.

4.2. Расширенный граф зависимостей

4.2.1. Расширение графа зависимостей

Для проведения конвейеризации циклов необходимо расширить понятие графа зависимостей на случай *межитерационных зависимостей*. Межитерационная зависимость строится между операциями A и B, если операция B на текущей итерации зависит от операции A на какой либо из предыдущих итераций. Каждой межитерационной зависимости приписывается положительное значение, которое определяет через сколько итераций зависимость реализовывается.

Определение. Количество итераций, через которое реализуется зависимость, будем называть *межитерационным расстоянием* зависимости.

Например, в следующем примере

```
for (i=0;i<1000;i++)
{
  t1 = a[i];
  ...
  a[i+3] = t2;
}
```

чтение из памяти `a[i]` на четвёртой итерации будет обращаться к той же ячейке памяти, что и запись `a[i+3]` на первой итерации. Следовательно от записи к чтению будет идти межитерационная зависимость, которая реализуется через три итерации.

Определение. *Расширенным графом зависимостей* называется граф зависимостей, в котором, помимо обычных зависимостей, присутствуют все межитерационные зависимости.

Если зависимость реализуется через одну или более итераций, то будем называть её *обратной*. Если зависимость реализуется через ноль итераций, то будем называть её *прямой*. Расширенный граф зависимостей, в отличие от обычного графа зависимостей, уже является циклическим.

Расширенный граф зависимостей вообще говоря может содержать бесконечное число дуг, так как у нас имеется новый параметр определяющий через сколько итераций зависимость реализуется и этот параметр может принимать бесконечное число значений. Рассмотрим пример

```
t  = ...
for (i=0;i<1000;i++)
{
  b = ...
  if ( b == 0 )
  {
     t = ...
  }
  ...
```

```
    a[i] = t;
}
```

Здесь операция `a[i] = t` может зависеть от записи в `t` на той же итераций и от записи в `t` с предыдущей итерации, и от записи в `t` произошедшей на две итерации раньше и так далее. То какое значение `t` будет использовано, зависит от значения b, которое в общем случае предсказать невозможно. Все потенциальные зависимости, которые могут возникнуть, должны быть отражены в графе зависимостей, поэтому от записи переменной `t` к операции `a[i] = t` будет идти бесконечное количество зависимостей отличающихся количеством итераций, через которое они реализуются.

Для того, чтобы избавиться от такой бесконечности будем рассматривать минимизированный граф зависимостей. Если между двумя операциями имеется зависимость реализуемая через n итераций, то все остальные зависимости такого же типа реализуемые через $m, m > n$ итераций мы строить и рассматривать не будем. Такая минимизация является семантически корректной, так как если операции будут упорядочены через n итераций, то и через большее количество итераций они тоже будут упорядочены.

4.3. Подсчёт минимального размера высоты цикла

В данном разделе рассматриваются подходы в определению минимального размера физической итерации цикла (высоты цикла). В первом разделе рассматриваются общие идеи и подходы к этому вопросу. Затем дано детальное описание того, как в предлагаемом алгоритме конвейеризации цикла производятся ресурсная оценка и подсчёт максимальной длины реккурентности.

4.3.1. Ограничения снизу на размер физической итерации цикла

Рассмотрим вопрос какова же минимально возможная высота цикла, которую можно получить в результате конвейеризации цикла, то есть ограничения снизу на размер физической итерации. Эта величина может быть ограничена по следующим двум причинам.

- Ограничение по ресурсам. Так или иначе за одну физическую итерацию должны быть исполнены все операции, которые присутствую в цикле. Поэтому цикл не может быть исполнен меньше, чем то количество тактов, за которое смогут исполниться все операции. В простейшем случае, когда все исполняющие устройства однородны минимальная высота цикла по ресурсам может быть выражена следующей формулой

$\left\lceil \frac{N}{w} \right\rceil$[1], где N - количество операций в цикле, а w - максимальное количество операций, которые могут выполниться за один такт.

- Ограничение по длине рекуррентности. В расширенном графе зависимостей возможны циклы, эти циклы называются рекуррентностями. Если имеется некоторая рекуррентность, то это означает, что имеются некоторые зависимости в вычислениях между итерациями цикла. И для того, чтобы начать исполнять операции из рекуррентности со следующей итерации необходимо выполнить все операции из рекуррентности с предыдущей итерации. Пусть L – некоторая рекуррентность. Введём величину $R(L) = \frac{\sum_i l_i}{\sum_i r_i}$, где суммирования ведутся по всем дугам входящим в рекуррентность, l_i – длины дуг, r_i – межитерационные расстояния. Теперь посчитаем максимум по всем рекуррентностям для величин $R(L)$: $R = \max_{L-рекуррентность} R(L)$. Размер физической итерации цикла не может быть меньше величины $R(L)$ для любой рекуррентности. Следовательно он не может быть меньше R. Таким образом величина R является нижней границей для размера физической итерации цикла.

Нахождение ограничения снизу на размер физической итерации является очень важным для ряда алгоритмов конвейеризации циклов. Подсчёт ограничения по ресурсам очень сильно зависит от особенностей архитектуры целевого микропроцессора. Для однородных архитектур эта величина считается достаточно просто: это просто есть количество всех операций делённое на ширину одной инструкции. Архитектура “Эльбрус”, для которой проводилось данное исследование, в большой степени является однородной. Ряд алгоритмов подсчёта ограничения по длине рекуррентности приведён в работе [77]. Следующие два раздела детально описывают используемые алгоритм подсчёта ограничений по ресурсам и алгоритм подсчёта максимальной длины реккуренстности.

4.3.2. Подсчёт ограничения по ресурсам

Была использована следующая схема подсчёта ресурсов необходимых для планирования цикла. Все операции были разбиты на четыре пересекающихся класса: операции требующие АЛУ (арифметическо-логическое устройство), операции обращения в память, вещественные операции и логические операции над предикатами. Для подсчёта ресурсной оценки считается

[1] $\lceil x \rceil$ - ближайшее целое сверху к x

количество операций в каждом классе. Одна операция может увеличить количество операций в разных классах. Также одна операция может увеличить количество операций в некотором классе больше чем на единицу. Например, в архитектуре “Эльбрус” одна инструкция может содержать либо четыре операции чтения из памяти, либо две операции чтения и одна операция записи, либо две операции записи в память. То есть фактически операция записи в память выбивает две операции чтения из памяти. Таким образом при обработке операции записи в память количество операций обращений в память мы увеличиваем на два, а при обработке операции чтения – на единицу. После учёта всех операций цикла, количество операций в каждом классе делиться на максимально возможное количество таких операций в одной инструкции и берётся ближайшее сверху целое к этой величине: получаем значения a_i. Для каждого класса размер цикла не может быть меньше величины a_i, так как по крайней мере для планирования операций из этого класса требуется a_i инструкций. В заключение считаем максимум по всем a_i – это величина и есть финальная ресурсная оценка размера цикла.

Количество классов операций (с описанной выше точки зрения) в архитектуре “Эльбрус” гораздо больше, чем четыре описанных выше. Например, операции деления могут идти только по одной в такт. Однако эти операции являются достаточно редкими. Было исследовано влияние других классов операций на результирующее качество кодов, путём внедрения их в ресурсную оценку. Никакого серьёзного влияния на скорость работы результирующих кодов и на качество ресурсной оценки в результате этого эксперимента обнаружено не было. Поэтому было решено оставить только описанные выше четыре класса.

В процессе работы предлагаемого алгоритма конвейеризации циклов вследствие разрывов зависимостей могут появляться новые операции. Также новые операции появляются вследствие конвейеризации операции стоящей под предикатом[1]. Эти новые операции необходимо учитывать в ресурсной оценке. С предлагаемой схемой это реализуется очень просто. Надо в процессе работы алгоритма хранить количество операций каждого класса. При добавлении новой операции в цикл необходимо увеличить количество операций в соответствующих классах, а затем пересчитать ресурсную оценку.

4.3.3. Подсчёт максимальной длины рекуррентности

Максимальная длина рекуррентности является вторым фактором, который ограничивает высоту цикла.

[1] При переносе операции стоящей под предикатом её нужно поставить также под предикат обратной дуги. Для вычисления финального предиката требуется построить операцию логического “и” над исходным предикатом операции и предикатом обратной дуги.

Определение. *Рекуррентностью* называется цикл без повторяющихся вершин в расширенном графе зависимостей.

Определение. *Длиной рекуррентности* называется величина

$$\frac{\sum length(dep_i)}{\sum iternum(dep_i)}$$

где $length(\cdot)$ – длина зависимости, $iternum(\cdot)$ – межитерационное расстояние зависимости, а суммирование ведётся по всем зависимостям в рекуррентности.

Высота цикла не может быть меньше, чем максимальная длина рекуррентности, так как в противном случае не будут выдержаны задержки между операциями входящими в рекуррентность.

В [108] описан алгоритм поиска максимальной длины рекуррентности. Его сложность составляет $O(n^3 \log n)$, где n количество узлов в графе. Это достаточно много для использования в динамическом трансляторе, поэтому мы использовали не точный алгоритм поиска максимальной длины рекуррентности для сокращения времени работы транслятора. Наш не точный алгоритм обрабатывает не все рекуррентности, поэтому максимальная длина рекуррентности может оказаться больше, чем тот результат который выдаст наш алгоритм. В конце этого раздела приводится сравнение двух алгоритмов как по скорости работы, так и то тому насколько часто в реальных условиях наш алгоритм не находит максимальную длину рекуррентности.

Перейдём к описанию использованного нами алгоритма поиска максимальной длины рекуррентности. Сначала приведём вспомогательный алгоритм поиска длины максимального пути между двумя операциями в обычном (ациклическом) графе зависимостей.

Алгоритм (поиска длины максимального пути между операциями в обычном (ациклическом) графе зависимостей). Пусть имеется представление с построенным графом зависимостей. Пусть имеется две операции x и y. Операция x находится выше в линейке операций, чем операция y. Необходимо найти длину максимального пути между операциями x и y.

Этот алгоритм очень похож на алгоритм разметки времён раннего планирования на обычном графе зависимостей. Сначала помечаются все операции между x и y, и для каждой операции заводится вспомогательное поле, которое будем обозначать time. Алгоритм выглядит следующим образом:

```
MaxPathBetweenOpers( x, y)
{
  x.time = 0;
  oper = NextOper(x)
  for ( ; oper != y; oper = NextOper(oper))
  {
    max_time = 0;
    for edge in все дуги входящие в node
    {
      if ( edge не помечена) continue;
      pred_node = предшественник(edge);
      max_time = max(max_time,время_раннего(pred_node)+длина(edge));
    }
    oper.time = max.time;
  }
  return y.time;
}
```

На выходе алгоритма в поле time операции y, будет лежать искомая длина максимального пути.

Перейдём теперь непосредственно к алгоритму поиска длины максимальной реккурентности.

Определение. *Простой обратной зависимостью* будем называть такую зависимость, у которой межитерационное расстояние равно единице, и предшественник этой зависимости находится в линейке операций ниже последователя зависимости.

Определение. *Простой реккурентностью* называется реккурентность, имеющая одну простую обратную зависимость, а все остальные зависимости должны иметь межитерационное расстояние равное нулю.

Предлагаемый алгоритм не находит самую длинную реккурентность. Он находит самую длинную простую реккурентность.

Алгоритм (поиска длины максимальной простой реккурентности). Задано представление и расширенный граф зависимостей на нём. Необходимо найти длину максимальной простой рккурентности.

Рассмотрим произвольную простую обратную зависимость. Найдём длину максимальной простой реккурентности, содержащей эту зависимость. Делается это следующим образом. Пусть у нас имеется простая обратная зависимость $a \to b$. Тогда длина максимальной

простой реккурентности, содержащей эту дугу, будет равна длине самой зависимости плюс длина максимального пути между операциями b и a, но посчитанной без учёта всех дуг с ненулевым межитерационным расстоянием. Это длина может быть найдена с помощью алгоритма “поиска длины максимального пути между операциями в обычном (ациклическом) графе зависимостей”, в который надо добавить отбрасывание дуг с ненулевым межитерационным расстоянием. Таким образом для заданной простой обратной зависимости, можно найти длину максимальной простой реккурентности, содержащей эту зависимость. Максимизирую эту величину по всем простым обратным зависимостям, мы найдём длину максимальной простой реккурентности.

В работе [109] приведён сравнительный анализ двух алгоритмов: точного алгоритма из [108] и описанного выше алгоритма. Точный алгоритм работает примерно в сто раз медленнее не точного. Эта разница составляет примерно 30% от времени компиляции. В Таблица 2 представлены результаты влияния алгоритма на качество результирующего кода на горячих участках из пакетов тестов SPEC CPU95, SPEC CPU2000, а также Windows 2000.

Пакет тестов	Число горячий участков	Число горячих участков в которых отличаются результаты точного и не точного алгоритмов	Коэффициент улучшения времени работы при использовании точного алгоритма	Среднее значение улучшения времени работы при использовании точного алгоритма
SPEC CPU95	272	6 (2,2%)	1,000-1,007	1,001
SPEC CPU2000	532	19 (3,6%)	1,000-1,034	1,002
Windows 2000	1609	23 (1,4%)	1,000-1,021	1,001

Таблица 2. *Сравнение качества результирующего кода точного и не точного алгоритма подсчёта максимальной длины рекуррентности.*

Таким образом при замедление общего времени компиляции на 30% точный алгоритм даёт лишь 0,1-0,2% ускорения результирующего кода. Отсюда можно сделать вывод, что использование более быстрого, но не всегда точного алгоритма, оправдано.

4.4. Разметка времён раннего и позднего планирования на расширенном графе зависимостей

4.4.1. Времена раннего и позднего планирования на расширенном графе зависимостей

Для описания предлагаемого алгоритма конвейеризации циклов нам понадобятся несколько определений.

По аналогии с временами раннего и позднего планирования на обычном (ациклическом) графе зависимостей введём определения времен раннего и позднего планирования на расширенном графе зависимостей.

Определение. Временами раннего и позднего планирования на расширенном графе зависимостей называется соответствие пар целых неотрицательных чисел и операций промежуточного представления удовлетворяющих следующим соотношениям:

$$(1) \qquad early(op) = \max_{dep_i} \left(early(pred(dep_i)) + length(dep_i) - (time_{bb} + 1) \cdot iternum(dep_i) \right),$$

где

$early(\cdot)$ – время раннего планирования операции

dep_i пробегает всех предшественников в графе зависимостей

$pred(\cdot)$ – предшественник зависимости

$length(\cdot)$ – длина зависимости

$time_{bb}$ – время раннего перехода по обратной дуге

$iternum(\cdot)$ – количество итераций, через которое осуществляется зависимость

$$(2) \qquad late(op) = \min_{dep_i} \left(late(succ(dep_i)) - length(dep_i) + (time_{bb} + 1) \cdot iternum(dep_i), time_{bb} \right),$$

где

$late(\cdot)$ – время позднего планирования операции

dep_i пробегает всех последователей в графе зависимостей

$succ(\cdot)$ – последователь зависимости

$length(\cdot)$ – длина зависимости

$time_{bb}$ – время раннего перехода по обратной дуге

$iternum(\cdot)$ – количество итераций через которое осуществляется зависимость

$$(3) \qquad early(bb) = late(bb) = time_{bb}$$

где bb – переход по обратной дуге.

Дадим некоторые пояснения к этому определению. Величина $time_{bb}+1$ является временем, за которое исполняется одна итерация цикла. Времена раннего и позднего планирования на расширенном графе зависимостей несут смысл аналогичный временам на классическом графе зависимостей. Время раннего планирования является первым тактом, в который возможно спланировать данную операцию. Время позднего является последним тактом, в который необходимо спланировать операцию для того, чтобы одна итерация цикла исполнялась за время $time_{bb}+1$. Если операцию спланировать позже, то переход по обратной дуге получиться спланировать только в такте большем, чем $time_{bb}$. Поясним также значение слагаемого $(time_{bb}+1)\cdot iternum(dep_i)$. Здесь величина $time_{bb}+1$ является высотой цикла, таким образом величина $(time_{bb}+1)\cdot iternum(dep_i)$ является "запасом" времени по данной зависимости, который имеется между операциями, в силу того, что зависимость реализуется через несколько итераций.

Если в графе зависимостей все дуги являются прямыми, то есть $iternum(dep_i)=0$ для всех i, то приведённые выше формулы для времён раннего и позднего планирования на расширенном графе зависимостей совпадают в введённым ранее определением времён раннего и позднего планирования для обычного (не расширенного) графа зависимостей.

4.4.2. Алгоритм разметки времён планирования на расширенном графе зависимостей

В этом пункте приведён алгоритм разметки времён раннего и позднего планирования на расширенном графе зависимостей. Этот алгоритм является главной эвристической составляющей предлагаемого алгоритма конвейеризации циклов. Он является основой для принятия решений.

Предложенный алгоритм разметки времён раннего и позднего планирования на расширенном графе зависимостей является итерационным. Время планирования обратной дуги на каждой итерации алгоритма увеличивается до тех пора, пока не станет возможна корректная разметка времён планирования. Алгоритм состоит из четырёх частей, формальное описание алгоритма разметки приведено ниже.

```
MarkOpersTimesOnAdvDepGraph(cfg_node)
{
  /* 1. Предварительная разметка времени раннего и позднего */
  MarkEarlyTimes(cfg_node);
  MarkLateTimes4BackBranch(cfg_node);

  /* 2. Подсчёт избыточной задержки, образованной обратными
```

```
        дугами в графе зависимостей */
  ConstrCorrectingDep(cfg_node);
  branch_delta = CalcDelta4BackBranch(cfg_node);
  DeleteCorrectingDep(cfg_node);

  /* 3. Увеличение времени перехода по обратной дуге
        до тех пор пока избыточная задержка не станет
        равной нулю */
  while (branch_delta > 0)
  {
    IncBackBranchLateTime(cfg_node);
    MarkLateTimes4BackBranch2(cfg_node);

    ConstrCorrectingDep(cfg_node);
    branch_delta = CalcDelta4BackBranch(cfg_node);
    DeleteCorrectingDep(cfg_node);
  }

  /* 4. Заключительная разметка времён раннего */
  ConstrCorrectingDepFromEnter(cfg_node);
  MarkEarlyTimes(cfg_node);
  DeleteTmpEdge(cfg_node);
}
```

Теперь опишем подробно каждую часть алгоритма.

1. Первая часть алгоритма производит предварительную разметку времён раннего и позднего. Фактически это разметка времён на обычном (не расширенном) графе зависимостей. Приведём формальные описания алгоритмов.

```
MarkEarlyTimes(cfg_node)
{
  early(enter) = 0;
  for oper в порядке прямой топологической нумерации
  {
    max_time = 0;
    for dep in все дуги входящие в oper
    {
      /* Если дуга является обратной, то не обрабатываем её */
      if (dep is back) continue;
      pred_node = pred(edge);
      max_time = max(max_time, early(pred_node) + length(dep));
```

```
    }
    early(oper)= max_time;
  }
}

MarkLateTimes4BackBranch(cfg_node)
{
  late(bb) = early(bb);
  for oper в порядке обратной топологической нумерации
  {
    min_time = MAX_INT; /* максимальное целое */
    for dep in все дуги выходящие из oper
    {
      /* Если дуга является обратной, то не обрабатываем её */
      if ( dep is back) continue;
      succ_node = succ(edge);
      min_time = min(min_time, late(succ_node) - length(dep));
    }
    late(oper)= min_time;
  }
}
```

2. На следующем шаге алгоритма вычисляется избыточная задержка, которая образуется обратными дугами в графе зависимостей. Избыточной задержкой по обратной дуге графа зависимостей будем называть следующую величину

$$(4) \qquad length(dep) - \left(early(succ(dep)) + (time_{bb} + 1) \cdot iternum(dep) - late(pred(dep))\right)$$

Фактически эта величина представляет собой, то количество тактов, на которое длина зависимости превысит фактическое расстояние в тактах между предшественником и последователем зависимости, при условии, что первый спланируется в своё время позднего, а второй в своё время раннего. Говоря другими словами, это нехватка расстояния между операциями до полной длины зависимости в худшем случае. Пока существует хотя бы одна зависимость с положительной избыточной задержкой, данная разметка времён позднего и раннего не удовлетворяет уравнениям (1) и (2), то есть фактически не является разметкой на расширенном графе зависимостей в смысле вышеприведённого определения. Рассмотрим каким образом можно уменьшить избыточную задержку для данной зависимости. Величины $length(dep)$ и $iternum(dep)$ являются постоянными, так как фактически отражают семантику зависимости. Уменьшить же избыточную задержку можно за счёт увеличения времени раннего последователя зависимости, уменьшения времени позднего предшественника зависимости и за

счёт увеличения времени перехода по обратной дуге. Первые два способа являются предпочтительными, так как при таких изменениях времён размер цикла не увеличивается. В то же время при увеличении времени перехода по обратной дуге размер цикла увеличивается. Фактически основная часть алгоритма и основывается на этом замечании. Сначала (на втором шаге алгоритма) производится попытка уменьшения избыточных задержек с помощью первых двух способов. А затем, если всё-таки все избыточные зависимости не исчезли, постепенно отодвигается переход по обратной дуге (третий шаг алгоритма).

Итак, на втором шаге алгоритма происходит попытка уменьшить избыточные задержки за счёт увеличения времён раннего у последователей обратных зависимостей и за счёт уменьшения времени позднего у предшественников обратных зависимостей. Чтобы разметка оставалась корректной в смысле не расширенного графа зависимостей, и чтобы не увеличился размер цикла (время перехода по обратной дуге), время раннего операции можно увеличивать не более её время позднего, а время позднего уменьшать не менее её времени раннего.

Функция `ConstrCorrectingDep` достраивает зависимости от операций к переходу по обратной дуге, тем самым уменьшая время позднего некоторых операций, затем производится разметка времён позднего на не расширенном графе зависимостей, затем опять производиться достроение зависимостей и так до тех пор пока была построена хотя бы одна новая зависимость. При этом зависимости для увеличения времени раннего не строятся, а всегда считается, что время раннего увеличено до максимума, то есть до времени позднего. Формальное описание приведено ниже.

```
ConstrCorrectingDep(cfg_node)
{
  do
  {
    CorrectNodeExtraDelay(cfg_node);
    MarkLateTimes4BackBranch2(cfg_node);
  } while(были построены новые зависимости);
}
```

Функция `CorrectNodeExtraDelay` обрабатывает каждую обратную зависимость. Для текущей зависимости вычисляется избыточная задержка, при этом считается, что у последователя зависимости время раннего увеличено до времени позднего, то есть фактически в формуле (4) $early(succ(dep))$ заменено на $late(succ(dep))$. Другими словами можно сказать, что предполагается в дальнейшем увеличить время раннего последователя зависимости до его времени позднего. После подсчёта такой модифицированной избыточной задержки строится

зависимость от предшественника зависимости к переходу по обратной дуге. Этим действием фактически уменьшается время позднего планирования этой операции. Длина зависимости имеет максимально возможную длину, но при этом она не должна превышать величины избыточной задержки и не должна превышать величину $time_{bb} - early(pred(dep))$, то есть время позднего не должно стать меньше времени раннего.

Функция `MarkLateTimes4BackBranch2` производит разметку времён позднего практически точно так же как функция `MarkLateTimes4BackBranch` за исключением того, что она не вычисляется время позднего перехода по обратной дуге, то есть в ней отсутствует строчка late(bb) = early(bb).

Дадим ещё некоторые пояснения к алгоритму построения корректирующих зависимостей. После построения дополнительных корректирующих зависимостей для уменьшения времён позднего некоторых операций, может уменьшиться время позднего некоторых других операций. В том числе среди этих операций могут оказаться такие, у которых в предшественниках есть обратные зависимости. Однако для этих операций при расчёте избыточной задержки считалось, что у них будет увеличено время раннего до времени позднего, а теперь это стало невозможным, так как у них уменьшилось время позднего. Именно по этой причине достроение корректирующих зависимостей производиться в цикле до тех пор, пока не построится ни одной зависимости. По этой же причине не строятся зависимости увеличивающие времена раннего. Если бы такое происходило, то после достроения корректирующих зависимостей, разметка на не расширенном графе зависимостей могла бы стать корректной только в том случае, если бы был отодвинут переход по обратной дуге. При этом переход может отодвинуться более чем на один такт, а такое отодвижение может оказаться избыточным, так как может оказаться, что переход по обратной дуге достаточно отодвинуть всего на один такт.

Вернёмся к описанию второго шага алгоритма разметки времён на расширенном графе зависимостей. Функция `CalcDelta4BackBranch` рассчитывает максимальную избыточную задержку при этом считая, что время раннего последователя зависимости увеличено до его времени позднего. Вообще говоря, перед этой функцией логически должно быть достроение корректирующих зависимостей для увеличения времени раннего последователей обратных зависимостей. Однако для ускорения алгоритма этого не делается, а как бы “в уме” предполагается, что такое построение произведено. В данный момент такое построение является корректным так как все времена позднего предшественников обратных зависимостей уже уменьшены и время перехода по обратной дуге не может измениться.

Функция `DeleteCorrectingDep` удаляет все дополнительные корректирующие зависимости, которые были ранее построены.

3. Если после уменьшений времён позднего и увеличений времён раннего всё-таки не удалось добиться корректной разметки (это соответствует случаю `branch_delta>0`), то единственным способом добиться корректной разметки остаётся увеличение времени перехода по обратной дуге. Это и делается на третьем шаге алгоритма. Вся функциональность этого шага заключена в цикл, который работает до тех пор пока не удалось достичь корректной разметки. Рассмотрим подробнее этот цикл. Первым делом происходит увеличение времени обратной дуги на один такт – функция `IncBackBranchLateTime`. Затем производиться разметка времён позднего, без пересчёта времени обратного перехода, уже знакомой нам функцией `MarkLateTimes4BackBranch2`. Затем полностью повторяется второй шаг алгоритма. Если после этого опять не удалось добиться корректной разметки, то повторяем всё сначала в цикле.

Здесь необходимо сделать небольшое пояснение. Первый вопрос, который приходит на ум при анализе шага три, следующий: а почему бы сразу не увеличить время перехода по обратной дуге на величину `branch_delta`? Это можно сделать, однако результирующая разметка не будет оптимальной, то есть цикл не будет иметь минимальную длину. Это происходит потому, что увеличение времени перехода на один такт, может поглотить избыточную задержку большую, чем один такт. И для того, чтобы результирующая разметка была оптимальной, переход отодвигается по одному такту.

4. Рассмотрим заключительный четвёртый шаг алгоритм. После завершения третьего шага уже корректно размечены времена позднего. На этом шаге необходимо только корректно разметить времена раннего. Функция `ConstrCorrectingDepFromEnter` достраивает зависимости от операции `ENTER` к последователям обратных зависимостей для компенсации избыточной задержки. Затем производиться разметка времён раннего, также как на первом шаге. В заключении удаляются все достроенные вспомогательные зависимости.

4.4.3. Корректность и оптимальность алгоритма.

Теперь покажем, что приведённый в предыдущем пункте алгоритм производит корректную и оптимальную разметку.

Теорема. Приведённый алгоритм разметки времён планирования на расширенном графе зависимостей производит корректную разметку времён.

Доказательство. Рассмотрим равенство (1) и покажем, что

$$(5) \qquad early(op) \geq \max\left(early(pred(dep_i)) + length(dep_i) - (time_{bb} + 1) \cdot iternum(dep_i)\right)$$

Для не обратных зависимостей $iternum(dep_i) = 0$, а время раннего на 4-ом шаге алгоритма как раз и вычисляется по формуле $early(op) = \max\left(early(pred(dep_i)) + length(dep_i)\right)$. Таким образом для не обратных зависимостей неравенство (5) выполняется. Для обратных

зависимостей алгоритм гарантирует, что выражение (4) для каждой зависимости не больше нуля:

$$0 \geq length(dep) - \left(early(succ(dep)) + (time_{bb} + 1) \cdot iternum(dep) - late(pred(dep))\right)$$

$$early(succ(dep)) \geq length(dep) - (time_{bb} + 1) \cdot iternum(dep) + late(pred(dep))$$

Здесь $succ(dep)$ как раз является рассматриваемой операцией. Если учесть, что время позднего всегда не меньше времени раннего, то получаем:

$$early(succ(dep)) \geq length(dep) - (time_{bb} + 1) \cdot iternum(dep) + early(pred(dep))$$

то есть как раз то, что и нужно для доказательства неравенства (5). То, что в неравенстве (5) на самом деле имеет место равенство, следует из следующих соображений: хотя бы в одном из неравенств, которые использовались для доказательства неравенства (5) имеет место равенство, поэтому рассматриваемое неравенство превращается в равенство. Таким образом, исследуемый алгоритм обеспечивает корректную разметку времён раннего в смысле (1).

Корректность разметки времён позднего исследуемым алгоритмом доказывается абсолютно аналогично корректности разметки времён раннего. Таким образом мы доказали, что приведённый алгоритм производит корректную разметку времён раннего и позднего планирования на расширенном графе зависимостей. ■

Теперь докажем оптимальность приведённого алгоритма. Под оптимальностью алгоритма понимается то, что время перехода по обратной дуге (полученное в результате работы алгоритма) является минимально возможным для того, чтобы существовала корректная разметка времён раннего и позднего планирования на расширенном графе зависимостей.

Теорема. Приведённый алгоритм разметки времён планирования на расширенном графе зависимостей производит оптимальную разметку времён.

Доказательство. Доказательство построим следующим образом: покажем, что если в результате работы алгоритма не удалось построить разметку для некоторого значения времени перехода по обратной дуге, то такой разметки не существует.

Вначале заметим, что времена, построенные на первом шаге алгоритма (то есть разметка времён раннего и позднего без учёта обратных дуг), представляют из себя интервалы внутри которых должны лежать финальные времена раннего и позднего. Это следует из того, что фактически такая разметка является максимумом и минимумом в формулах (1) и (2), но только взятыми по подмножеству не обратных дуг. Приведённый алгоритм может остановиться (для заданного значения перехода по обратной дуге) только в том случае, если для компенсации избыточной зависимости необходимо сделать время позднего предшественника зависимости

меньше его времени раннего. Но с другой стороны уменьшение времени позднего у предшественника зависимости нельзя не произвести. Фактически в алгоритме уменьшение времени позднего происходит только тогда, когда его нельзя избежать. В момент уменьшения считается, что у последователя зависимости время раннего будет увеличено до времени позднего, а больше увеличить его нельзя, так как ранее было замечено, что финальные времена должны лежать в интервале первоначальной разметки. Увеличить время перехода по обратной дуге, в рамках текущего доказательства, также нельзя. Поэтому единственным способом получить корректную разметку является уменьшение времени раннего.

Таким образом получается, что алгоритм может не найти корректной разметки для заданного времени перехода, только в том случае, если он захочет уменьшить время позднего и не сможет этого сделать по причине того, что оно станет меньше времени раннего. С другой стороны уменьшение времени позднего происходит только в тех случаях, когда другого способа получить корректную разметку не существует. Следовательно, получается, что если алгоритму не удалось найти корректной разметки, то такой разметки не существует для текущего значения времени перехода по обратной дуге. ■

4.5. Алгоритм конвейеризации циклов.

4.5.1. Описание алгоритма конвейеризации циклов

В этом пункте опишем непосредственно алгоритм программной конвейеризации циклов. Он базируется на переносе операций по обратной дуге. Разметка времён раннего и позднего планирования на расширенном графе зависимостей является основной аналитической информацией используемой при принятии решения о том, какие операции необходимо переносить. Соответственно и алгоритм разметки времён, приведённый выше, также является одним из ключевых.

В приведённом ниже описании алгоритма опущены некоторые технические детали. Например, не описан алгоритм построения расширенного графа зависимостей или не описана коррекция аналитических структур данных транслятора, которая производится при переносе операций. Однако это исключительно технические детали, которые очень сильно зависят от архитектуры для которой делается код, от транслятора и даже от места среди других фаз, в котором происходит конвейеризация. Мы же преследуем цель описать ту часть алгоритма конвейеризации, которая отвечает за получение наиболее эффективного результирующего кода.

Итак, приступим к описанию алгоритма конвейеризации циклов. Алгоритм применяется только к внутренним сводимым циклам. Вначале строится расширенный граф зависимостей на всём множестве операций принадлежащих циклу. Если две операции должны быть упорядочены по какой-то причине, то между ними обязана быть зависимость. При этом если

операции зависят на разных итерациях цикла (например, одна операция вырабатывает результат, который используется на следующей итерации, или некоторое обращение в память происходит по тому же адресу, что и некоторое другое обращение через пять итераций), то между ними также обязана быть зависимость. Для ускорения алгоритма построения графа зависимости некоторые операции со сложным набором зависимостей были исключены из обработки. Если тело цикла содержит такие операции, то конвейеризация к нему не применяется. Такие операции со сложным набором зависимостей достаточно редки, поэтому данное ограничение практически не влияет на качество результирующего кода.

Затем происходит вычисление длины максимальной рекуррентности данного цикла и ресурсная оценка данного цикла методами описанными выше. Результирующий размер цикла не может быть меньше ни размера максимальной рекуррентности, ни ресурсной оценки. Поэтому при достижении размера цикла одной из этих величин алгоритм останавливается.

Следующим шагом берётся минимум из всех времён позднего планирования операций и выбираются операции, у которых время позднего совпадает с минимальным. Эти операции и переносятся по обратной дуге. Под переносом по обратной дуге понимается следующее действие: операция дублируется, сама операция становиться перед переходом по обратной дуге, а копия операции переносится в предцикл. Получается, что в начале цикла в зависимости от того, откуда пришло управление, потребляется либо значение, выработанное самой операцией, либо её копией. Наглядно это преобразование можно представить так: мы берём операцию и начинаем тащить её вверх, когда мы упираемся в схождение образованное обратной дугой и дугой из предцикла, мы дублируем операцию и по каждой из дуг продолжает идти своя копия, в итоге обе операции помещаются в узлы предшествующие соответствующим дугам.

В описанном выше преобразовании переносится самый верхний такт цикла. Опишем смысл этого преобразования. В начале заметим, что данное преобразование не может увеличить длину цикла, так как в начале цикла убирается один такт, а в конец добавляется один такт. Таким образом, данное преобразование не ухудшает ситуацию в цикле. На самом деле обычно в конце цикла не добавляется такт, так как перенесённые операции размешиваются с остальными операциями. Такое преобразование собственно и является наложением итераций цикла или конвейеризацией цикла. После такого переноса получается, что первый такт итерации цикла будет выполнен на предыдущей итерации.

Перенос первых тактов цикла продолжается до тех пор, пока размер цикла не достигнет длины максимальной рекуррентности или пока не будет превышена ресурсная оценка. Вычисление этих величин производится так, как было описано в разделах 4.3.2 и 4.3.3.

Имеется причина, по которой рекуррентность может увеличиться в процессе работы алгоритма. Когда операция переноситься по обратной дуге она должна либо встать под

предикат обратной дуги, либо должна быть переведена в спекулятивный режим. Если операция становиться под предикат, то необходимо построить новую зависимость, а если появляется новая зависимость, то она может образовать новую рекуррентность более длинную, чем все имеющиеся ранее. Точно такая же ситуация образуется, когда переносимая операция уже стоит под предикатом и строится новая операция логического “и” двух предикатов для формирования предиката для перенесённой операции. В таких случаях пересчитывает длина максимальной рекуррентности.

4.5.2. Разрыв зависимостей в процессе работы алгоритма конвейеризации циклов

Как уже было сказано выше, очень важным преимуществом предложенного алгоритма для двоичного транслятора, является то, что в алгоритм можно интегрировать разрыв зависимостей. Опишем как была реализована эта интеграция.

При построении расширенного графа зависимостей все зависимости, которые можно разорвать, помечаются как разрываемые. При разметке времён раннего и позднего планирования разрываемые зависимости не учитываются. Соответственно и решение о переносе тех или иных операций принимается без учёта этих зависимостей. Если в список операций для переноса попала операция, у которой есть входящая разрываемая зависимость реализуемая через ноль итераций и предшественник этой зависимости не попал в список переносимых операций, то эта зависимость разрывается и в таком случае перенос становиться семантически корректным и его можно осуществить. При разрыве зависимости корректируется граф зависимостей и признаки разрываемых зависимостей.

В алгоритм были интегрированы следующие типы разрывов зависимостей с созданием новых операций: разрыв зависимостей типа чтение-запись, запись-запись, разрыв предикатный зависимостей и разрыв зависимостей с использованием спекулятивности по данным.

4.5.3. Оценка сложности алгоритма конвейеризации

В этом пункте мы оценим сложность алгоритма конвейеризации. Для оценки будем пользоваться следующими величинами:

n – количество операций в цикле

e – количество зависимостей в цикле

$l_{\max}$ – максимальная длина зависимости в цикле

r – максимальное количество операций в одной инструкции

m – высота цикла без учёта обратных дуг, то есть время раннего планирования перехода по обратной дуге при разметке времён раннего без учёта обратных дуг плюс один

Заметим, что n и e являются переменными и зависят от рассматриваемого цикла. Оценка сложности будет как раз вести как функция от этих величин. l_{max} и r ограничены некоторыми константами, которые однозначно определяются характеристиками микропроцессорной архитектуры. m потребуется для промежуточных оценок.

4.5.3.1. Оценка сложности алгоритма разметки времён планирования

Вначале оценим сложность алгоритма разметки времён на расширенном графе зависимостей. На первом шаге алгоритма производится разметка времён раннего и позднего планирования на обычном графе зависимостей. Сложность обоих этих алгоритмов составляет $O(e)$.

Теперь рассмотрим второй шаг алгоритма. Функция `ConstrCorrectingDep` содержит цикл. Тело цикла состоит из двух функций `CorrectNodeExtraDelay` и `MarkLateTimes4BackBranch2`. `CorrectNodeExtraDelay` обрабатывает все обратные зависимости, следовательно её сложность не больше, чем $O(e)$. Функция `MarkLateTimes4BackBranch2` имеет сложность $O(e)$. Таким образом сложность тела цикла составляет $O(e)$. Теперь оценим количество итераций цикла. Для этого нам потребуется доказать несколько утверждений.

Утверждение 1. Если $n > (m + l_{\max}) \cdot r$, то дальнейшую конвейеризацию цикла производить бессмысленно, так как достигнута ресурсная оценка.

Доказательство. Если высота цикла по разметке без обратных дуг равна m, то высота цикла с учётом обратных дуг будет не больше, чем $m + l_{\max}$. Получить корректную разметку на расширенном графе зависимостей из первоначальной разметки без учёта обратных дуг можно увеличив время перехода по обратной дуге на $l_{\max}$. Такая разметка будет корректна так длинны всех дуг не превышают $l_{\max}$ и следовательно ни одна из зависимостей не будет иметь избыточной задержки. В тоже время в силу ресурсных ограничений цикл не может быть спланирован в меньше, чем n/r тактов. Поэтому, если $m + l_{\max} < n/r$, то мы уже достигли ресурсной оценки и дальше конвейеризацию цикла продолжать не имеет смысла. Последнее неравенство эквивалентно $n > (m + l_{\max}) \cdot r$, которое и указано в формулировке утверждения. ■

Из данного утверждения следует, что в случае, когда $n > (m + l_{\max}) \cdot r$ не имеет смысла продолжать перенос операций. Поэтому нет необходимости запускать разметку времён планирования на расширенном графе зависимостей в случае выполнения данного неравенства. Для того, чтобы проверить данное неравенство, достаточно в алгоритме конвейеризации перед

запуском разметки на расширенном графе зависимостей произвести разметку времён раннего без учёта обратных дуг и проверить данное неравенство. Если оно выполнено, то алгоритм конвейеризации завершает свою работу. Сложность этого действия составляется $O(e)$. Таким образом в дальнейшем изложении всегда будем считать, что при разметке времён планирования выполняется неравенство

$$(6) \qquad n \le (m + l_{\max}) \cdot r$$

Также, без ограничения общности, можно считать, что

$$(7) \qquad m \ge l_{\max} + 1$$

так как в противном случае, в силу неравенства (6), количество операций в цикле ограничено константой и следовательно алгоритм разметки времён также будет работать за константное время.

Вернёмся к рассмотрению цикла в функции `ConstrCorrectingDep`. Пусть количество итераций этого цикла равняется K. Рассмотрим следующие множества: $A_i = \{$множество всех операций, у которых изменилось время позднего на i-той итерации цикла$\}$.

Утверждение 2. $|A_i| \ge \left\lceil \frac{m}{l_{\max}} \right\rceil$ для всех $i = 1\text{K}\ K-2$.

Доказательство. Рассмотрим i-ю итераций цикла в функции `ConstrCorrectingDep`, при этом $i \le K-2$. После i-й итерации цикла будет ещё как минимум две итерации, или, что тоже самое, мы не завершим цикл на $i+1$ итерации. Это в свою очередь означает, что на i-й итерации изменилось(уменьшилось) время позднего по крайней мере у одной операции такой, что её время позднего до изменения было меньше $l_{\max}$. Если бы это было не так, то на $i+1$ итерации не потребовалось бы строить ни одной новой зависимости, и эта итерация была бы последней. Таким образом по крайней мере у одной из операций с временем позднего меньшим $l_{\max}$ изменилось время позднего на i-й итерации, назовём эту операцию a. Заметим, что в функции `CorrectNodeExtraDelay` изменяется время позднего только у операций с временем позднего больше, чем $m - l_{\max}$. Назовём множество этих операций B. Для того, чтобы изменилось время позднего у a, необходимо, чтобы от a был путь к операций b, такой, что $b \in B$, при этом у всех операций на этом пути было изменено время позднего. Таким образом у нас имеется путь от a к b, такой что у каждой операции из этого пути изменилось время

позднего. Длина этого пути больше, чем $m - 2l_{\max}$, а количество операций больше, чем $\frac{m}{l_{\max}} - 1$, отсюда и следует, что $|A_i| \geq \left[\frac{m}{l_{\max}}\right]$. ■

Утверждение 3. Время позднего операции в функции `ConstrCorrectingDep` не может измениться более чем $l_{\max}$ раз.

Доказательство. Если время позднего некоторой операции изменилось $l_{\max}$ раз, то это означает, что оно уменьшилось на $l_{\max}$. Следовательно, после изменения времени позднего $l_{\max}$ раз, время позднего перехода по обратной дуге минус время позднего этой операции не меньше, чем $l_{\max}$. Но в таком случае избыточная задержка по любой дуге выходящей из операции будет не больше нуля (это следует из определения избыточной задержки), и следовательно больше время позднего операции изменяться не будет. ■

Теперь рассмотрим следующее выражение: $\sum_{i=1}^{K} |A_i|$. В силу Утверждения 2 имеем:

$$(8) \qquad \sum_{i=1}^{K} |A_i| > \sum_{i=1}^{K-1} |A_i| \geq (K-2) \cdot \left[\frac{m}{l_{\max}}\right] \geq (K-2) \cdot \left(\frac{m}{l_{\max}} - 1\right)$$

С другой стороны в силу Утверждения 3:

$$(9) \qquad \sum_{i=1}^{K} |A_i| \leq n \cdot l_{\max}$$

Объединяя (8) и (9), получаем

$$n \cdot l_{\max} \geq \sum_{i=1}^{K} |A_i| \geq (K-2) \cdot \left(\frac{m}{l_{\max}} - 1\right)$$

$$(10) \qquad n \geq \frac{(K-2) \cdot (m - l_{\max})}{l_{\max}^2}$$

Теперь объединим неравенства (6) и (10):

$$\frac{(K-2) \cdot (m - l_{\max})}{l_{\max}^2} \leq n \leq (m + l_{\max}) \cdot r$$

$$K \le r \cdot l_{\max}^2 \cdot \frac{m + l_{\max}}{m - l_{\max}} + 2 \le r \cdot l_{\max}^2 \cdot \frac{m + l_{\max}}{m - l_{\max}} + 2 \le r \cdot l_{\max}^2 \cdot (2l_{\max} + 1) + 2$$

Учитывая неравенство (7) максимум функции $\frac{m + l_{\max}}{m - l_{\max}}$ будет достигаться в точке $m = l_{\max} + 1$, поэтому

$$K \le r \cdot l_{\max}^2 \cdot \frac{m + l_{\max}}{m - l_{\max}} + 2 \le r \cdot l_{\max}^2 \cdot (2l_{\max} + 1) + 2$$

$$(11) \qquad K \le 2 \cdot r \cdot l_{\max}^3 + r \cdot l_{\max}^2 + 2$$

Таким образом из формулы (11) следует, что количество итераций цикла в функции `ConstrCorrectingDep` ограничено константой. Учитывая то, что тело цикла имеет сложность $O(e)$, вся функция `ConstrCorrectingDep` имеет также сложность $O(e)$.

Ограничение (11) имеет скорее теоретическое значение, чем практическое, так как величина $2 \cdot r \cdot l_{\max}^3 + r \cdot l_{\max}^2 + 2$ очень велика. Для условий, в которых применялся данный алгоритм в худшем случае она составляет почти 100000, а это на порядки больше чем допустимое количество операций в цикле. Даже в случае, когда максимальная длина зависимости в цикле равна 3-м тактам (3-х тактной задержкой в архитектуре “Эльбрус” обладает обращение в память) величина $2 \cdot r \cdot l_{\max}^3 + r \cdot l_{\max}^2 + 2$ равняется 379, а это сравнимо с допустимым количеством операций в цикле. Для того, чтобы оценить реальную ситуацию, было подсчитано количество итераций цикла в функции `ConstrCorrectingDep` на горячих участках задач из пакета SPEC CPU95 и SPEC CPU2000, а также на Windows 2000 (всего более 2300 горячих участков). Среднее число повторений данного цикла составило 1.825. Максимально число повторений – 12. Таким образом среднее число повторений этого цикла меньше двух раз, что является очень хорошим показателем.

Заканчивая оценку второго шага алгоритма оценим сложность функции `CalcDelta4BackBranch`. Эта функция проходит по всем обратным дугам и следовательно её сложность не выше, чем $O(e)$. Таким образом сложность второго шага алгоритма составляет $O(e)$[1].

[1] Заметим, что без использования неравенства (6) и соответствующего дополнения к алгоритму конвейеризации, количество итераций цикла в функции `ConstrCorrectingDep` может составить $O(n)$. Таким образом, в общем случае, сложность второго шага алгоритма составит $O(ne)$.

Теперь оценим сложность третьего шага алгоритма. Рассмотрим внешний цикл третьего шага. На каждой итерации цикла происходит увеличение времён перехода по обратной дуге на единицу. Таких увеличений потребуется не больше, чем $l_{\max}$, так как, если взять разметку времён без учета обратных дуг и в этой разметке увеличить время перехода по обратной дуге на $l_{\max}$, то получиться корректная разметка на расширенном графе зависимостей (более подробно эти рассуждения проводились при доказательстве Утверждения 1). Таким образом количество итераций внешнего цикла на третьем шаге алгоритма ограничено константой.

Рассмотрим тело цикла третьего шага алгоритма. Функция `IncBackBranchLateTime` работает за константное время. Сложность `MarkLateTimes4BackBranch2`, `ConstrCorrectingDep` и `CalcDelta4BackBranch` как было показано ранее составляет $O(e)$, сложность `DeleteCorrectingDep` также не превышает $O(e)$. Таким образом сложность третьего шага алгоритма $O(e)$.

Было также подсчитано количество итераций внешнего цикла (если в цикл не разу не заходили, то количество итераций считаем равное нулю) в третьем шаге алгоритма на горячих участках задач из пакета SPEC CPU95 и SPEC CPU2000, а также на Windows 2000 (всего более 2300 горячих участков). Среднее число повторений данного цикла составило 0.302. Максимально число повторений – 12.

Оценим сложность последнего четвёртого шага алгоритма. Все три функции на этом шаге осуществляют обход по дугам и следовательно их сложность не превышает $O(e)$.

Таким образом мы установили:

Теорема. Сложность приведённого алгоритма разметки времён раннего и позднего планирования на расширенном графе зависимостей составляет $O(e)$.

4.5.3.2. Оценка сложности алгоритма конвейеризации циклов

Теорема. Сложность приведённого алгоритма конвейеризации циклов составляет $O(e)$.

Доказательство. Длина максимального пути в цикле из n операций не может быть больше, чем $n \cdot l_{\max}$. С другой стороны в цикле из n операций ограничение по ресурсам не меньше n/r, то есть одна итерация цикла не может исполняться быстрей чем n/r. Таким образом самый длинный путь разбивается не более чем на $\frac{n \cdot l_{\max}}{n/r} = l_{\max} \cdot r = K$ частей. Первую часть необходимо конвейеризировать $K-1$ раз, вторую $K-2$ раза и так далее. Таким образом нам необходимо произвести процедуру переноса группы операций не более чем $\frac{K(K-1)}{2}$, раз.

Так как величина K является константой, то и число производимых переносов группы операций также является константой. Самой сложной частью в переносе группы операций является разметка времён планирования, сложность которой $O(e)$, следовательно и сложность всего алгоритма конвейеризации составляет $O(e)$. ■

4.6. Аппаратная поддержка обеспечения точного контекста при использовании вращающихся регистров

4.6.1. Обеспечение точного контекста

Как уже отмечалось раньше, двоичный транслятор должен обеспечивать точный контекст в случае возникновения прерывания. То есть, при возникновении прерывания мы должны восстановить ровно такое состояние контекста, которые должно было быть в исходном коде. Под контекстом понимается все регистры и вся память исходной архитектуры. Сложность этой проблемы заключается в том, что для эффективной оптимизации кода для EPIC архитектуры требуется агрессивное перемешивание операций. В архитектуре x86 очень многие операции могут вызвать прерывания, например, все операции с вещественной арифметикой и все операции обращения в память. Для обеспечения точного контекста можно все операции вызывающие прерывание производить последовательно. Однако такой подход на корню погубит всю производительность, так как ни о каком агрессивном перемешивании не может идти и речи.

Опишем подход к обеспечению точного контекста в двоичном оптимизирующем трансляторе для архитектуры “Эльбрус”. Весь контекст делиться на две части: переименованный и не переименованный. Переименованный контекст не поддерживается в корректном состоянии все время, но при этом в случае возникновения прерывания он может быть полностью восстановлен по заранее сохранённой информации. Не переименованный контекст должен быть всегда точным и последовательным. Все операции меняющие не переименованный контекст идут в той же последовательности, что и в исходном коде. Обычно к переименованному контексту относят наиболее часто используемые регистры, а к не переименованному все остальное, в том числе и всю память[1].

Итак все операции изменяющие не переименованный контекст идут в строгом порядке. Вместе с каждой такой операцией ставиться специальная операция, называемая контрольная

[1] Что относить к переименованному контексту, а что нет, является внутренним соглашением двоичного транслятора. Более того это разделение может часто меняться в процессе работы для повышения эффективности кода. Даже часть обращений в память может быть перенесена в переименованный контекст, только необходимо обеспечить, чтобы в случае возникновения прерывания, всё корректно восстановилось.

точка (КТ), которую мы будем обозначать SRP (Save Recovery Point). КТ выполняется или не выполняется в зависимости от того выполняется или не выполняется операция изменения не переименованного контекста. Для упрощения реализации КТ в аппаратуре она может применяться только с операциями записи в память. Остальные изменения не переименованного контекста выполняется строго последовательно с предварительным сведением контекста[1]. КТ является местом, к которому мы можем откатиться в случае возникновения прерывания, и в этом месте мы можем полностью восстановить контекст. Это место соответствует состоянию перед операцией исходного кода непосредственно следующей за операцией записи в память. Для каждой КТ сохраняется вся необходимая информация для восстановления переименованного контекста и алгоритм его восстановления. Поскольку между двумя КТ нет изменений не переименованного контекста, а в случае отката к КТ восстанавливается весь переименованный контекст, то такая схема обеспечения точного восстановления контекста является корректной.

Реализована КТ одним битом в инструкции, то есть она либо есть в команде либо нет, и она не выталкивает другие команды и не требует дополнительных ресурсов. В случае успешного прохождения операции записи в память стоящей в одной команде с КТ, КТ регистрируется. Физически это означает запись адреса текущей команды в специальный регистр RPR. Для адреса каждой команды, в которой имеется КТ в специальных таблицах запоминается алгоритм восстановления переименованного контекста из рабочих регистров. Транслятор обеспечивает то, что эти регистры не будут перезаписаны пока не зарегистрируется новая КТ. В случае возникновения прерывания, происходит чтения значения из регистра RPR, поиск в таблицах соответствующего алгоритма восстановления контекста, и непосредственно само восстановление переименованного контекста. Так как за одну инструкцию в микропроцессоре Эльбрус могут исполняться две операции записи, то в регистре RPR заведён специальный бит, который отражает какая из операций записи успешно завершилась. Соответственно команда SRP выставляет этот бит нужным образом.

4.6.2. Взаимодействие схемы восстановления точного контекста с механизмом вращающихся регистров

Использования вращающихся регистров вносит новые трудности в использующуюся нами схему обеспечения точного контекста. Рассмотрим следующий пример:

```
loop:
```

[1] Заметим, что изменения такого не переименованного контекста крайне редкие операции и данное упрощения не оказывает сколько-нибудь значительного влияния на качество результирующего кода.

```
...
SRP  P1[T]
...
OP1
...
BRANCH loop P2[T]
```

Предположим, что в приведённом выше цикле операция `OP1` вызвала прерывание и при этом зарегистрированной контрольной точкой оказалась `SRP  P1[T]`. Однако в таком случае неизвестно на какой итерации была зарегистрирована эта КТ: на текущей или на предыдущей. А от этого в свою очередь зависит с каких регистров необходимо восстанавливать переименованный контекст, так как при переходе по обратной дуге произошло продвижение базы вращающихся регистров и нумерация регистров изменилась.

В принципе установить на какой итерации произошла регистрация КТ можно по значениям предикатов, но для этого необходимо продлить время жизни условного аргумента КТ до всех достижимых вниз КТ. Однако было предложено и реализовано более простое, элегантное и эффективное решение этой задачи. При регистрации КТ происходит, кроме запоминания адреса текущей команды, ещё и сохранение текущей базы вращающихся регистров. В таком случае при возникновении прерывания достаточно сначала восстановить сохранённую базу вращающихся регистров, а уже затем производить восстановление переименованного контекста. При таком подходе основной код не теряет эффективности. При восстановлении переименованного контекста требуется сделать лишь одну операцию чтения и одну операцию записи в системный регистр, что лишь незначительно замедляет этот процесс.

Рассмотрим ещё одну проблему, с которой приходиться сталкиваться при взаимодействии схемы восстановления точного контекста с механизмом вращающихся регистров. Заметим, что аргументы КТ должны доживать до всех достижимых вниз КТ для того, чтобы в любой момент можно было восстановить контекст. Предположим, что количество итераций цикла, которые могут выполняться без регистрации КТ, неограниченно. В таком случае, если КТ имеет аргумент расположенный на вращающемся регистре, то нам понадобиться бесконечное количество поколений данного регистра. Но это обеспечить невозможно, так как в микропроцессоре имеется лишь ограниченное количество регистров.

Первым выходом из этой ситуации может служить правило не использовать вращающиеся регистры в качестве аргументов таких КТ. Такой подход не очень эффективен, так как большинство вычислений в конвейеризированном цикле выполняется на вращающихся

регистрах и придётся строить дополнительные пересылки в обычные регистры для формирования аргументов КТ.

Более эффективным подходом, который был нами и реализован, является обеспечение фронта контрольных точек на каждой итерации цикла. Наличие фронта означает, что на каждой итерации цикла будет зарегистрирована по крайней мере одна КТ. В таком случае потребуется лишь ограниченное количество поколений для аргументов КТ, и описанная выше проблема решается. Фронт в каждом конвейеризированном цикле обеспечивается достаточно просто. Если изначально в цикле нет фронта, то в его голове строиться КТ, которая и обеспечивает фронт. Так как голова цикла доминирует все узлы цикла, то на каждой итерации будет зарегистрирована по крайней мере одна КТ.

4.7. Некоторые обобщения

4.7.1. Использование конвейеризации для циклов с несколькими обратными дугами

Везде ранее речь шла о циклах с одной обратной дугой. В этом разделе будет показано как можно с помощью описанных ранее алгоритмов производить конвейеризацию циклов с несколькими обратными дугами.

Для циклов с несколькими обратными дугами можно провести преобразование графа управления и свести цикл с несколькими обратными дугами к циклу с одной обратной дугой. На Рис. 22 приведён пример такого преобразования. На Рис. 22 а) изображён исходный цикл с двумя обратными дугами. В начале строиться пустой предцикл – узел 3 на Рис. 22 б). Затем все обратные дуги, кроме одной самой вероятной, перенаправляются на новый предцикл. После такого преобразования у исходного цикла остаётся лишь одна обратная дуга, а все остальные обратные дуги трансформируются в выходы из цикла. Таким образом цикл стал удовлетворять предусловиям нашего алгоритма конвейеризации и к нему применяется конвейеризация. После окончания работы алгоритма конвейеризации цикл преобразовывается к исходному виду. Узел 3 (который стал не пустым после работы алгоритма конвейеризации) дублируется. Все обратные дуги перенаправляются на копию и копия прокручивается по обратным дугам. Результат данного преобразования представлен на Рис. 22 в).

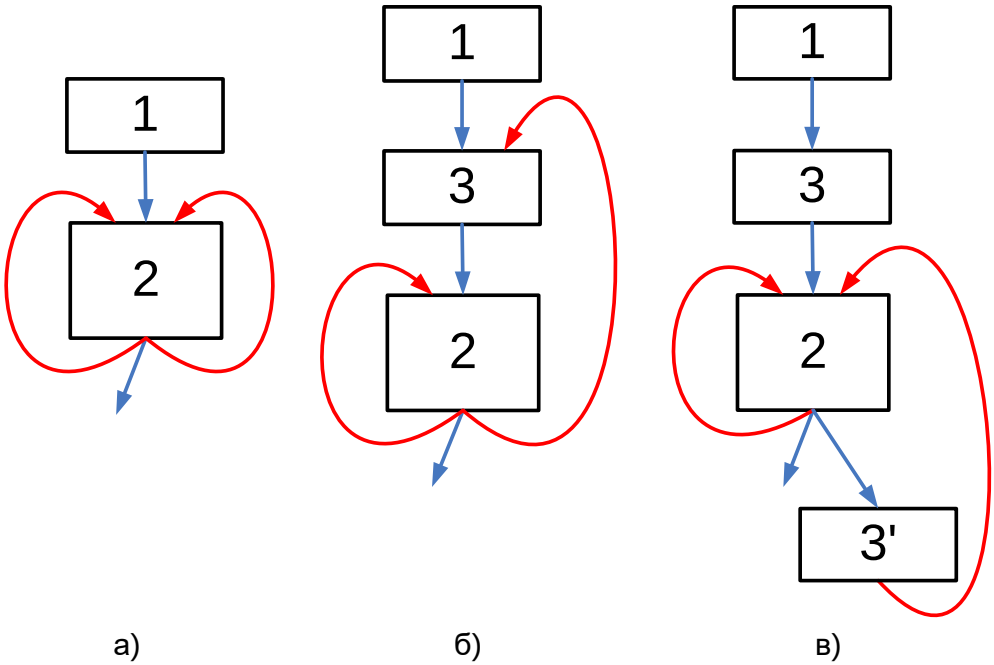

Рис. 22. *Преобразование цикла с несколькими обратными дугами к циклу с одной обратной дугой.*

Используя описанное преобразование можно с помощью представленного нами алгоритма конвейеризации, конвейеризировать циклы с несколькими обратными дугами. Заметим, что многие другие алгоритмы конвейеризации циклов также работают с циклами имеющими одну обратную дугу, а для конвейеризации циклов с несколькими обратными дугами используются подобные преобразования управляющего графа.

4.7.2. Использование конвейеризации для внешних циклов

В этом разделе мы рассмотрим как можно с помощью предложенных методов конвейеризировать не самые внутренние циклы. Как и в предыдущем разделе мы с помощью преобразований управляющего графа преобразуем несколько вложенных циклов в один внутренний цикл с несколькими обратными дугами. Пример такого преобразования представлен на Рис. 23. На Рис. 23 а) изображены два исходных вложенных цикла. Для того, чтобы получить один цикл необходимо раздублировать все узлы между входом во внешний цикл до входа во внутренний (в рассматриваем примере это узлы с номерами 1, 2 и 3) и “прокрутить” копии обратной дуге. Результат преобразования приведён на Рис. 23 б).

Необходимо заметить, что мы описали не общий вид преобразования нескольких вложенных циклов в один цикл с несколькими обратными дугами. В таком виде преобразование невозможно например для не сводимых циклов. Нашей целью было проиллюстрировать возможность такого преобразования, и как следствие возможность применения описанных ранее алгоритмов конвейеризации циклов.

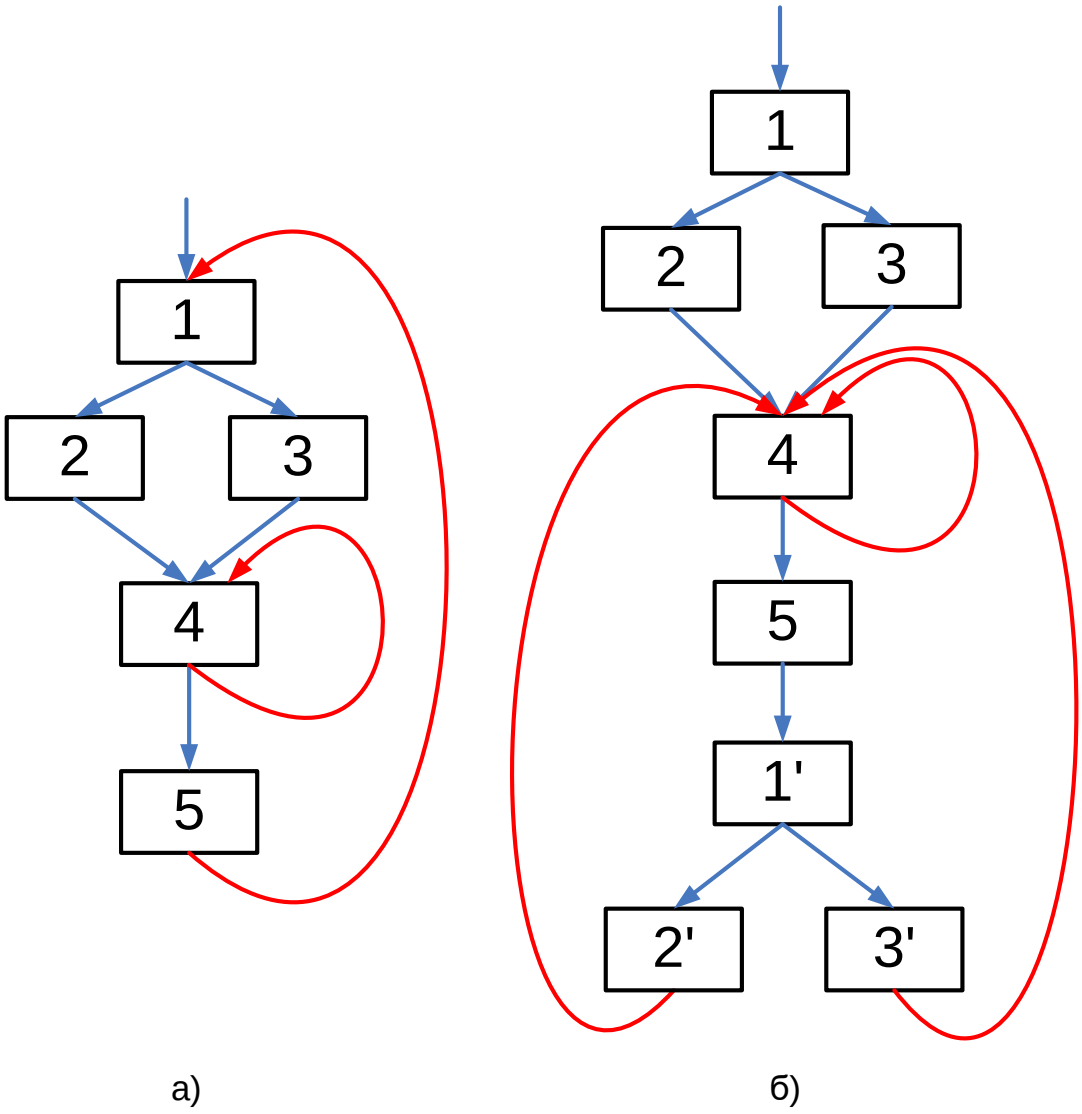

Рис. 23. *Преобразование нескольких вложенных циклов в один цикл с несколькими обратными дугами*

В силу сказанного выше имеется возможность преобразовать несколько вложенных циклов в один цикл с несколькими обратными дугами. В свою очередь (как было показано в предыдущем разделе) цикл с несколькими обратными дугами может быть преобразован к циклу с одной обратной дугой и следовательно к нему может применяться предложенный алгоритм конвейеризации циклов. Таким образом у нас имеется возможность (предварительно проведя не сложные преобразования управляющего графа) конвейеризировать не только самые вложенные, но и внешние циклы.

4.8. Экспериментальные результаты

Для анализа эффективности работы предложенного алгоритма конвейеризации циклов использовались те же методы исследований, что и в предыдущих главах.

На Рис. 24, Рис. 25 и Рис. 26 приведено сравнение времени работы результирующих кодов с включенных и выключенным алгоритмом конвейеризации.

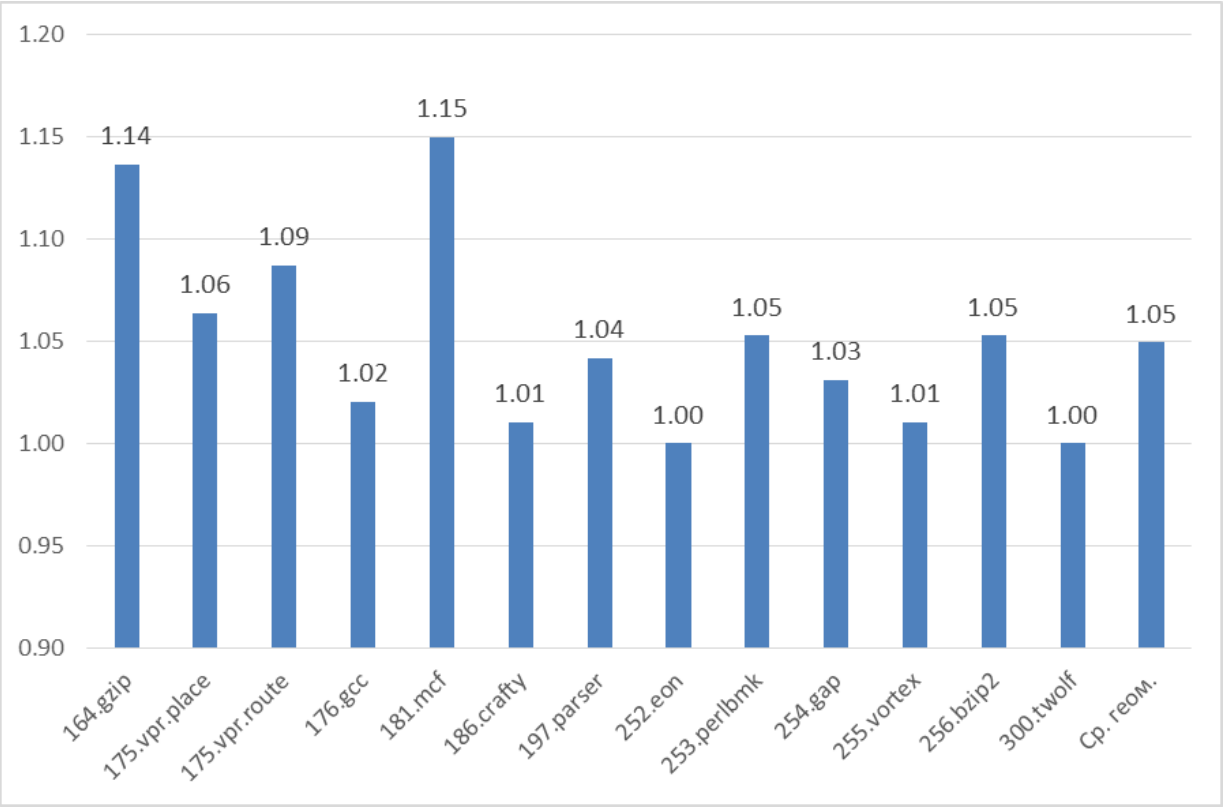

Рис. 24. *Влияние алгоритма конвейеризации циклов на время работы результирующего кода на целочисленных задачах пакета SPEC CPU2000. Замеры на симуляторе.*

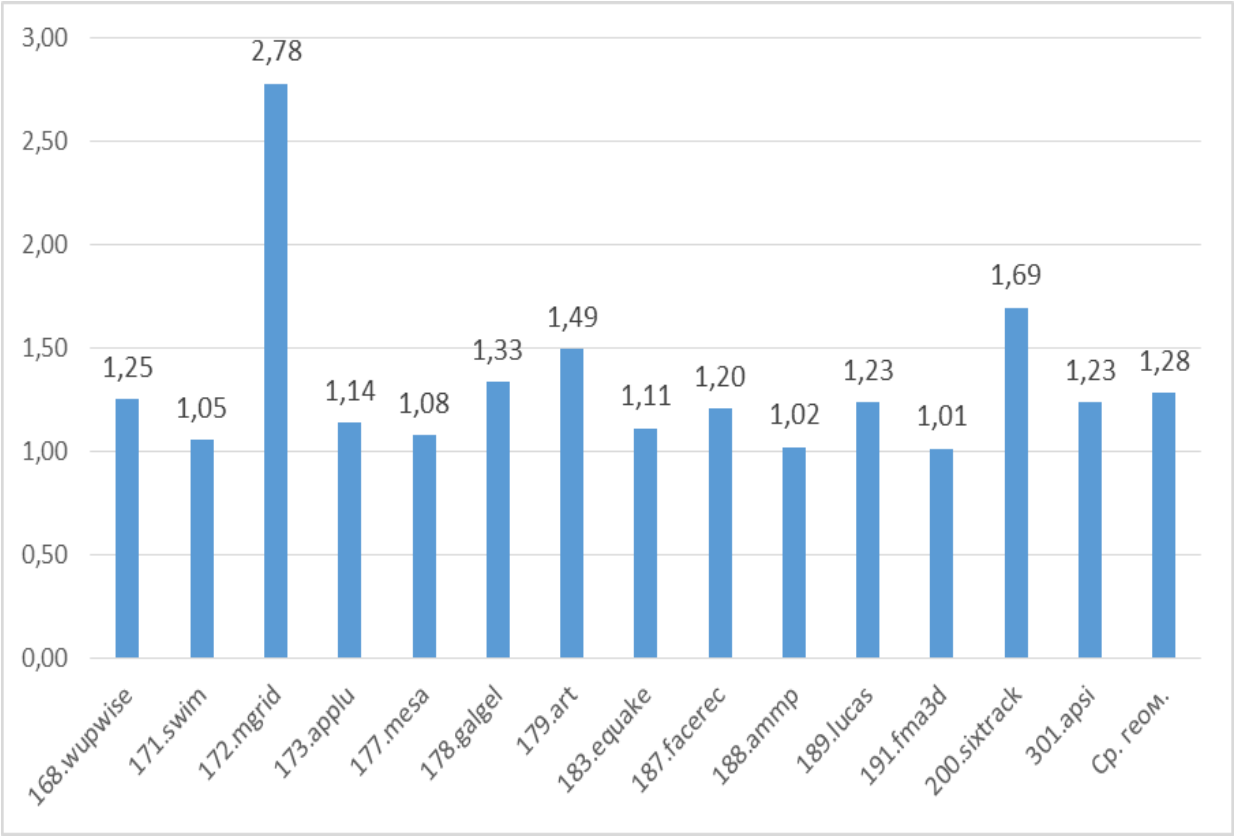

Рис. 25. *Влияние алгоритма конвейеризации циклов на время работы результирующего кода на вещественных задачах пакета SPEC CPU2000. Замеры на симуляторе.*

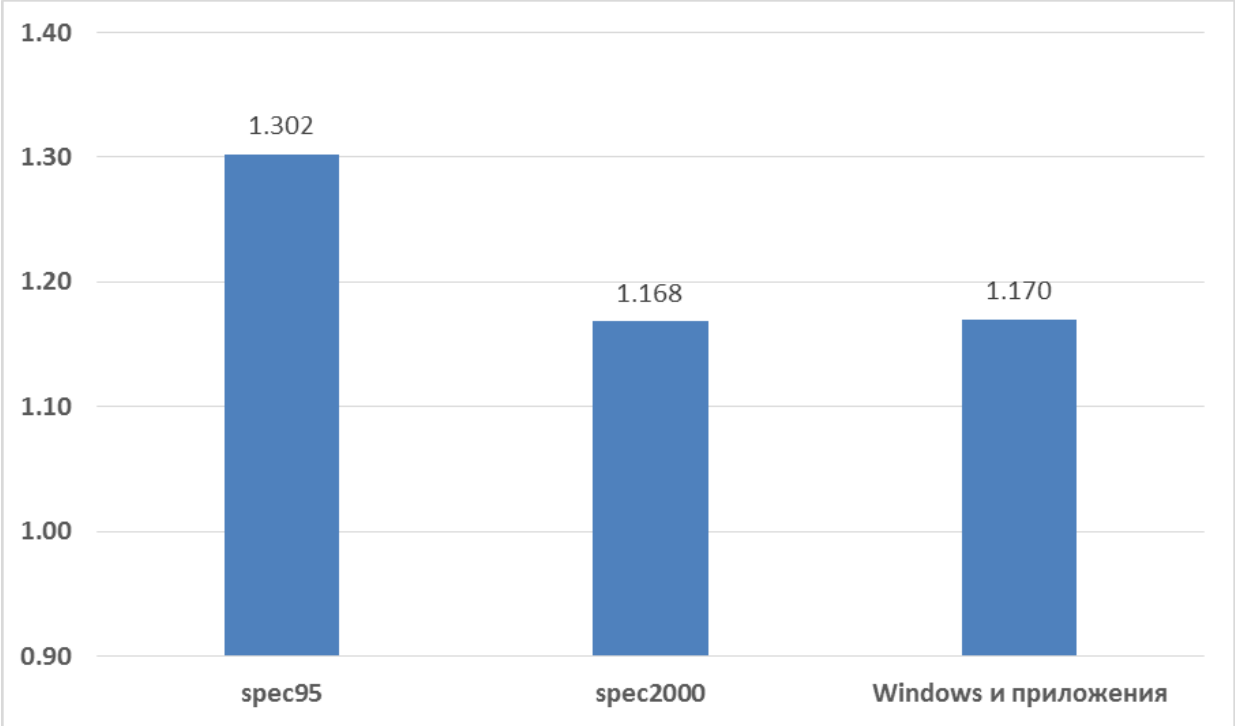

Рис. 26. *Влияние алгоритма конвейеризации циклов на время работы результирующего кода на горячих участков SPEC CPU95, SPEC CPU2000, Windows и пользовательских приложений. Предсказание на основе планирования.*

Также был произведён замер времени затрачиваемое на работу алгоритма конвейеризации циклов. Оно составило 3.5% и 10% от общего времени трансляции на целочисленных и вещественных задачах соответственно. На вещественных задач время работы выше, так как на них конвейеризация может активнее применяться и, как видно из предыдущих замеров, даёт больший прирост качества результирующих кодов.

4.9. Выводы

1. В данной главе приводится описание алгоритма конвейеризации циклов. Данный алгоритм обладает высокой скоростью работы, что позволяет его использовать в динамических оптимизаторах.
2. Описан алгоритм разметки времён раннего и позднего планирование. Информация о временах планирования является основной аналитической информацией для проведения конвейеризации цикла.
3. Доказана корректность и оптимальность алгоритма разметки времён планирования.
4. Проведена оценка сложности алгоритма разметки времён планирования и алгоритма конвейеризации циклов как с теоретической, так и с практической точек зрения.
5. Описан эффективный метод интеграции различных техник разрыва зависимостей в алгоритм конвейеризации циклов.
6. Описаны эффективные алгоритмы учёта ограничений на минимальный размер цикла: учёт ресурсов и вычисление максимальной длины рекуррентности.
7. Приводиться описание аппаратного решения, которое позволяет эффективно использовать технику вращающихся регистров совместно с техникой восстановления точного контекста при конвейеризации циклов в двоичных оптимизирующих трансляторах.
8. Приведённые экспериментальные результаты показывают высокую эффективность предложенных методов на широком классе задач. Предложенный алгоритм конвейеризации циклов даёт прирост производительности на **5%** на целочисленных задачах пакета SPEC CPU2000, на **28%** на вещественных задачах пакета SPEC CPU2000 и **17%** на наборе горячих участков из Windows 2000 и пользовательских приложений. Время работы алгоритма составило 3.5% и 10% от общего времени трансляции на целочисленных и вещественных задачах соответственно.

Заключение.

В диссертационной работе рассмотрены вопросы увеличения скорости работы результирующего кода с помощью сокращения длины критических путей в динамическом двоичном оптимизирующем трансляторе для современных микропроцессоров основанных на архитектуре с явно выраженной параллельностью на уровне команд. Проведён анализ известных методов сокращения длины критического пути как для циклического так и для ациклического случаев, а также присущие им недостатки возникающие при использовании этих методов в динамическом трансляторе. В ходе исследования был разработан ряд новых алгоритмов позволяющих существенно повысить качество результирующего кода.

Также в работе рассматриваются не только отдельно взятые алгоритмы, но и их место в общей цепочке оптимизаций, их взаимодействие с другими оптимизациями. Такое рассмотрение и учёт других преобразований даёт возможность применять более эффективные алгоритмы как по качеству результирующего кода, так и по скорости работы.

В процессе исследования и в ходе решения поставленных задач были получены следующие **основные результаты**:

1. Разработан и реализован алгоритм построения частичных предикатов. Это алгоритм может быть естественным образом встроен в стандартную схему построения предикатного кода, что безусловно является его достоинством. Использование данного алгоритма позволило повысить качество результирующего кода на **1-2%** при этом замедление времени трансляции составляет **0,4%**.
2. Разработаны и реализованы алгоритмы переименования регистров и применения спекулятивности по управлению.
3. Разработан и реализован алгоритм минимизации высоты графа зависимостей в ациклических областях с целью сокращения длины критических путей. Данный алгоритм обладает высокой скоростью работы, что позволяет его использовать в динамических оптимизирующих системах. Использование данного алгоритма позволило повысить качество результирующего кода на целочисленных задачах пакета SPEC CPU2000 на **12%**, а на вещественных задачах пакета SPEC CPU2000 на **22%**. Время работы алгоритма составило 3.8% от общего времени трансляции. Дано строгое описание это алгоритма, доказана его корректность и оценена сложность.
4. Разработан и реализован алгоритм разметки времён раннего и позднего планирования на расширенном графе зависимостей. Доказана его корректность и оптимальность. Произведена оценка сложности алгоритма как с теоретической, так и с практической точек

зрения. Проведённое исследование показало, что предложенный алгоритм обладает высокой скоростью работы и может быть использован в динамических оптимизирующих системах.

5. На базе алгоритма разметки времён раннего и позднего планирования на расширенном графе зависимостей был разработан и реализован алгоритм конвейеризации циклов с целью сокращения длины критических путей в циклах. Разметка времён служит основной аналитической информацией для определения того, какие операции следует переносить. Использование данного алгоритма позволило повысить качество результирующего кода на целочисленных задачах пакета SPEC CPU2000 на **5%**, а на вещественных задачах пакета SPEC CPU2000 на **28%**. Время затрачиваемое на работу алгоритма конвейеризации циклов составило 3.5% и 10% от общего времени трансляции на целочисленных и вещественных задачах соответственно.
6. Разработан и реализован эффективный метод интеграции различных техник разрыва зависимостей, сокращающих длину критического пути, в предложенный алгоритм конвейеризации циклов. Разрыв зависимостей во время работы алгоритма конвейеризации цикла, позволяет существенно повысить его эффективность.
7. Разработаны и реализованы эффективные (по соотношению качества работы к скорости работы) алгоритмы учёта ограничений на минимальный размер цикла: учёт ресурсов и вычисление максимальной длины рекуррентности.
8. Суммарное повышение скорости работы результирующего кода от всех методов предложенных в работе составило **19%** и **58%** на целочисленных и вещественных задачах из пакета SPEC CPU2000 соответственно.

Представленные в диссертационной работе алгоритмы и подходы были разработаны и реализованы в рамках следующих проектов:

- динамический двоичный оптимизирующий транслятор уровня всей системы с архитектуры Intel x86 на архитектуру Эльбрус, разработанный в АО “МЦСТ”;
- динамический двоичный оптимизирующий транслятор уровня приложений ОС Linux с архитектуры Intel x86 на архитектуру Эльбрус, разработанный в АО “МЦСТ”;
- статический оптимизирующий транслятор с архитектуры Intel x86 на архитектуру IPF (Itanium), разработанный в АО “МЦСТ” в рамках совместного проекта с Intel Corporation;
- динамический двоичный транслятор уровня приложений ОС Linux, разработанный в ООО “Эльбрус Технологии”.

Использование этих методов позволило существенно поднять эффективность двоичного транслятора, а их опытная эксплуатация позволила сделать вывод об их пригодности к практическому использованию. Успешное использование представленных методов в нескольких двоичных трансляторах для различных микропроцессорных архитектур показывают их универсальность.

Направления дальнейших исследований по теме диссертационной работы включают:

- Реализацию предложенных алгоритмов сокращения длины критического пути для других микропроцессорных архитектур, в том числе не EPIC.
- Реализацию предложенных алгоритмов в языковых компиляторах. Оценка их эффективности.

Список литературы.

[1] Vadim Gimpelson, Anatoly Konuhov, “Running Intel on ARM Servers: Tips and Tricks of Optimizations”, ARMTechCon 2013, Santa Clara CA, 29 Oct – 31 Oct 2013. 9 pp.

[2] Vadim Gimpelson, Anatoly Konuhov, “x86 to ARM Binary Translator”. ARMTechCon 2012, Santa Clara CA, 30 Oct – 01 Nov 2012. 11 pp.

[3] Н.В. Воронов, В.Д. Гимпельсон, М.В. Маслов, А.А. Рыбаков, Н.С. Сюсюкалов ”Система динамической двоичной трансляции x86 → «Эльбрус»”. Вопросы радиоэлектроники, серия ЭВТ, выпуск 3, 2012. С. 89-108.

[4] Гимпельсон В.Д. “Конвейеризация циклов в двоичном динамическом трансляторе”. Вопросы радиоэлектроники, выпуск 3, 2009 г. С. 67-78.

[5] Гимпельсон В.Д. “Статистический метод определения времени начала оптимизаций в динамического оптимизирующем трансляторе”. Международная научная конференция, посвящённая 80-летию со дня рождения академика В.А. Мельникова. Сборник докладов, 2009 г. С. 135-137.

[6] Гимпельсон В.Д. “Сокращение длины критического пути циклических и ациклических участков в динамическом двоичном оптимизирующем трансляторе для архитектуры “Эльбрус”. Научные труды XXXIV Международной молодёжной научной конференции “Гагаринские чтения”, Москва, МАТИ, 2008 г. 1 сс.

[7] Загребин А.А., Гимпельсон В.Д. “Проблема восстановления профильной информации по неполным исходным данным в динамическом двоичном компиляторе”. Информационные технологии, Приложение, №11, 2008. С. 21-26.

[8] Гимпельсон В.Д. “Оптимизация циклов методом наложения итераций в динамическом трансляторе для архитектуры “Эльбрус”. Научные труды XXXIII Международной молодёжной научной конференции “Гагаринские чтения”, Москва, МАТИ, 2007 г. 1 сс.

[9] Гимпельсон В.Д. “Сокращение длины критического пути в циклах в оптимизирующем динамическом двоичном трансляторе”. Сборник тезисов XXIII научно-технической конференции войсковой части 03425. Москва, в/ч 03425, 2007 г. 2 сс.

[10] Волконский В.Ю., Гимпельсон В.Д. “Методы определения порогов активизации динамического оптимизирующего транслятора”. Информационные технологии, № 4, 2007 г. С. 32-41.

[11] Гимпельсон В.Д. “Статистически оптимальное время начала оптимизаций в динамическом двоично-оптимизирующем комплексе”. Высокопроизводительные вычислительные системы и микропроцессоры: сборник трудов ИМВС РАН, Выпуск № 9, 2006 г. С. 38-48.

[12] Волконский В.Ю., Гимпельсон В.Д., Масленников Д.М. “Быстрый алгоритм минимизации высоты графа зависимостей”, Информационные технологии и вычислительные системы, №3, 2004 г. С. 102-116.

[13] Масленников Д.М., Василец П.С., Гимпельсон В.Д., Матвеев П.Г., Муслинов Р.Г. “Программно-аппаратный метод обеспечения точного состояния контекста при прерываниях в двоично-оптимизированном коде”. Сборник тезисов XXI научно-технической конференции войсковой части 03425. Москва, в/ч 03425, 2003 г. 1 сс.

[14] D. I. August, K. M. Crozier, J. W. Sias, P. R. Eaton, Q. B. Olaniran, D. A. Connors, and W. W. Hwu. The IMPACT EPIC 1.0 Architecture and Instruction Set reference manual: Technical Report IMPACT-98-04 / IMPACT, University of Illinois, Urbana, IL, February 1998.

[15] M. S. Schlansker, B. R. Rau. EPIC: An Architecture for Instruction-Level Parallel Processors: Technical Report HPL-1999-111 / Compiler and Architecture Research Hewlett-Packard Laboratories, Palo Alto, February 2000.

[16] Intel Corporation. Intel Itanium Architecture Software Developer’s Manual. Volume 1-3. Oct. 2002.

[17] Intel Corporation. Intel Itanium 2 Processor. Hardware Developer’s Manual. Volume 1-3. Jul. 2002.

[18] Muchnick S. S. Advanced compiler design and implementation. Morgan Kaufmann Publishers, 1997.

[19] Critical path optimization-unzipping: United States Patent 6,564,372 / B. A. Babaian, S. K. Okunev, V. Y. Volkonsky. Appl. No.: 504630; Filed: February 15, 2000; Pub.: May 13, 2003. 4 pp.

[20] Волконский В.Ю, Окунев С.К. Оптимизация критического пути на предикатном представлении программы. Информационные технологии, № 9. Москва, сентябрь 2003.

[21] Michael Schlansker and Vinod Kathail. "Critical Path Reduction for Scalar Programs". Proceedings of the 28th International Symposium on Microarchitecture. November, 1995.

[22] F. E. Allen, John Cocke, and Ken Kennedy. Reduction of operator strength. In Steven S. Muchnick and Neil D. Jones, editors, Progmm Flow Analysis: Theory and Applications, pages 79-101. Prentice-Hall, 1981.

[23] Касьянов В.Н., Евстигнеев В.А. "Графы в программировании: обработка, визуализация и применение". СПб.: БХВ-Петербург, 2003.

[24] S. A. Mahlke, et al. Effective compiler support for predicated execution using the hyperblock. Proceedings of the 25th Annual International Symposium on Microarchitecture (1992), 45-54.

[25] Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B., and Wolfe, M. 1981. Dependence graphs and compiler optimizations. In proceedings of the 8th ACM Symposium on Principles of Programming Languages (Jan.), 207-218.

[26] W. W. Hwu, et al. The superblock: an effective technique for VLIW and superscalar compilation. The Journal of Supercomputing 7, 1/2 (1993), 229-248.

[27] Chang, P. P., Mahlke, S. A., Chen, W. Y., Warter, N. J., and Hwu, W. W. 1991. IMPACT: An architectural framework for multiple-instraction-issue processors. In Proceeding of the 18th International Symposium on Computer Architecture (May), 266-275.

[28] Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu, B. Ramakrishna Ran, Michael S. Schlansker. 1992. Sentinel scheduling for VLIW and superscalar processors. In Proceeding of the 5th International Conference on Architectural Support for Programming languages and Operating Systems (Oct.).

[29] Joseph A. Fisher. Trace Scheduling: A technique for global microcode compaction // Transactions on Computers, IEEE. - V. C-30. - July, 1981. - P. 478-490.

[30] W. Y. Chen, Data Preload for Superscalar and VLIW Processors. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[31] A. Nicolau. Run-time disambiguation: coping with statically unnpredictable dependencies. IEEE Transactions on Computers, vol. 38, pp. 663-678, May 1989.

[32] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, Wen-mei W. Hwu. Dynamic Memory Disambiguation Using the Memory Conflict Buffer. ASPLOS, 1994.

[33] M. Schlansker and V. Kathail. Acceleration of algebraic recurrences on processors with instruction level parallelism. In Proceedings of LCPC-6, pp. 406-429, 1993.

[34] Michael Schlansker, Vinod Kathail, Sadun Anik. Height Reduction of Control Recurrences for ILP Processors. In Proceedings of MICRO 27, 1994.

[35] M. S. Schlansker, S. A. Mahlke, and R. Johnson. “Control CRP: A branch height reduction optimization for EPIC architectures,” in Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, pp. 155–168, May 1999.

[36] D. J. Kuck. The Structure of Computers and Computations, volume 1. John Wiley and Sons, New York, NY, 1978.

[37] L. Carter et al., Predicated Static Single Assignment, in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 1999.

[38] Baraz L. et al, IA-32 Execution Layer: a Two Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-based Systems. Proceedings of the 36th International Symposium on Microarchitecture, 2003.

[39] Klaiber, A., "The Technology Behind Crusoe Processors". Transmeta Corporation white paper, January 2000

[40] Dehnert J.C., Grant B.K., Banning J.P., Johnson R., Kistler T., Klaiber A, and Mattson J. "The transmeta code morphing software: using speculation, recovery and adaptive retranslation to address real-life challenges". Proceedings of the International Symposium on Code Generation and Optimization, 2003.

[41] Рожков С.А. Технология двоичной совместимости программно-аппаратных средств // Программные продукты и системы, №1, 1999.

[42] Рожков С.А. Программные методы исполнения двоичных кодов на основе аппаратной поддержки – Диссертация на соискание ученой степени кандидата технических наук, М., НИИ “Вычислительные Технологии”, 1999.

[43] Ермолович А.В. Методы повышения производительности двоично-транслирующих систем с аппаратной поддержкой. – Диссертация на соискание ученой степени кандидата технических наук, М., ИМВС РАН, 2003.

[44] Волконский В.Ю., Оптимизирующие компиляторы для архитектур с явным параллелизмом команд и аппаратной поддержкой двоичной совместимости. // Журнал “Информационные технологии и вычислительные системы” 3/2004, М.:УРСС, 2004.

[45] Boris Babayan. E2K Technology and Implementation. // in Proceedings of the Euro-Par 2000 - Parallel Processing: 6th International. - Volume 1900 / 2000. – January, 2000. – P. 18-21.

[46] М. Кузьминский. Отечественные микропроцессоры: Elbrus E2K // Открытые системы, № 05-06, 1999. – С. 8-13.

[47] K. Dieffendorf. The Russians Are Coming. Supercomputer Maker Elbrus Seeks to Join x86/IA-64 Melee. Microprocessor Report, V.13, №.2. – February 15, 1999. - P. 1-7.

[48] Intel Corporation. IA-32 Intel Architecture Software Developer’s Manual. Volume 1-3. Jun. 2005.

[49] Baraz L. et al, "IA-32 Execution Layer: a Two Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-based Systems". Proceedings of the 36th International Symposium on Microarchitecture, 2003.

[50] Hookway, R., and Herdeg, M., "DIGITAL FX!32: Combining Emulation and Binary Translation". Digital Technical Journal, Vol. 9, No. 1, August 28, 1997, pp. 3-12.

[51] Chernoff, A. et al, "FX!32 A Profile-directed Binary Translator". IEEE Micro, March/April 1998, pp. 56-64.

[52] Paul J. Drongowski, David Hunter, Morteza Fayyazi, David Kaeli. "Studying the Performance of the FX!32 Binary Translation System". Proceeding of the 1st Workshop on Binary Translation, Oct. 1999.

[53] T. Lindholm, F. Yellin, “The Java Virtual Machine Specification”, 2nd ed., Addison-Wesley, Reading, MA. 1999.

[54] D. Box. “Essential .NET , Volume 1: The Common Language Runtime”, Addison-Wesley, Reading, MA. 2002.

[55] B. Alpern, C. R. et al. The Jalapeno virtual machine. IBM Systems Journal, 39(1),2000.

[56] M. G. Burke et al. The Jalapeno dynamic optimizing compiler for Java. In ACM 1999 Java Grande Conference, pages 129-141, June 1999.

[57] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, Peter F. Sweeney, “Adaptive Optimization in the Jalapeno JVM”, Proceedings of the ACM SIGPALN Conference on Object-Oriented Programming systems, Languages, and Applications, pp. 47-65, Oct 2000.

[58] M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM ’01), pp. 1-12, Apr. 2001.

[59] M. Cierniak, G.Y. Lueh, and J.M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 13-26, Jun. 2000.

[60] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, “Dynamo: A Transparent Dynamic Optimization System”, Proceedings of International Symposium on Programming Language Design and Implementation, pp. 1-12, Jun. 2000.

[61] Derek Bruening, Timothy Garnett, Saman Amarasinghe, “An Infrastructure for Adaptive Dynamic Optimization”, Proceedings of the 1st International Symposium on Code Generation and Optimization, pp. 265-275, Mar. 2003.

[62] Chi-Keung Luk, Robert Muth, Harish Patil, Robert Cohn, Geoff Lowney, “Ispike: A Post-link Optimizer for the Intel Itanium Architecture”, Proceedings of the International Symposium on Code Generation and Optimization, 2004.

[63] Bob Cmelik, David Keppel, “Shade: A Fast Instruction-Set Simulator for Execution Profiling”, 1994.

[64] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In 11th USENIX Security Symposium, 2002.

[65] Dino Dai Zovi. Security applications of dynamic binary translation. B.S., Computer Science, University of New Mexico, 2002.

[66] Shiliang Hu, Efficient Binary Translation In Co-Designed Virtual Machines. PhD thesis, University of Wisconsin, Madison, 2006.

[67] Волконский В.Ю., Гимпельсон В.Д. Методы определения порогов активизации динамического оптимизирующего транслятора. Информационные технологии, № 4, 2007.

[68] Sebastian Winkel. Optimal Global Scheduling for Itanium Processor Family. In Proceedings of the EPIC-2 Workshop, Istanbul, November 2002.

[69] Sebastian Winkel. Optimal Global Instruction Scheduling for the Itanium Processor Architecture. PhD thesis, Naturwissenschaftlich-Technischen Fakultäten der Universität des Saarlandes, Saarbrücken, September, 2004.

[70] SPEC CPU92 Benchmark. www.spec.org.

[71] SPEC CPU95 Benchmark. www.spec.org.

[72] SPEC CPU2000 Benchmark. www.spec.org, 2000.

[73] Carole Dulong, Rakesh Krishnaiyer, Dattatraya Kulkarni, Daniel Lavery, Wei Li, John Ng, and David Sehr. An Overview of the Intel ® IA-64 Compiler. Intel Technology Journal, (Q4), 1999.

[74] Щербинин С. Использование отложенных вычислений для оптимизации двоичного кода. Выпускная квалификационная работа на соискание степени магистр. ФТРК МФТИ, Москва, 2005.

[75] Муслинов Р.Г. , Масленников Д.М. Методы оптимизации работы с памятью в двоичном трансляторе “Эльбрус-3М”. Сборник тезисов XXI научно-технической конференции войсковой части 03425. Москва, в/ч 03425, 2003г.

[76] J. C. Dehnert, P. Y.-T. Hsu, and J. P. Bratt. Overlapped Loop Support in the Cydra-5. In Proceedings of the Third International Conference on Architectural Support for Prograramming Languages and Operating Systems, pages 26-38, Boston, MA, April 1989.

[77] Allan, V. H., Jones, R. B., Lee, R. M., and Allan, S. J. 1995. Software pipelining. ACM Comput. Surv. 27, 3 (Sep. 1995), 367-432.

[78] Rau, B. R. and Glaeser, C. D. 1981. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. SIGMICRO Newsl. 12, 4 (Dec. 1981), 183-198.

[79] Lam, M. 1988. Software pipelining: an effective scheduling technique for VLIW machines. SIGPLAN Not. 23, 7 (Jul. 1988), 318-328.

[80] Allen, J. R., Kennedy, K., Porterfield, C., and Warren, J. 1983. Conversion of control dependence to data dependence. In Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (Austin, Texas, January 24 - 26, 1983). POPL '83. ACM, New York, NY, 177-189.

[81] Warter, N. J., Mahlke, S. A., Hwu, W. W., and Rau, B. R. 1993. Reverse If-Conversion. SIGPLAN Not. 28, 6 (Jun. 1993), 290-299.

[82] Warter, N. J., Haab, G. E., Subramanian, K., and Bockhaus, J. W. 1992. Enhanced modulo scheduling for loops with conditional branches. SIGMICRO Newsl. 23, 1-2 (Dec. 1992), 170-179.

[83] N. J. Warter and W. W. Hwu, Enhanced modulo scheduling. Tech. Rep. CRHC-92-11, Center for Reliable and High-Performance Computing. University of Illinois, Urbana, IL, November 1992.

[84] Su, B. and Wang, J. 1991. GURPR*: a new global software pipelining algorithm. In Proceedings of the 24th Annual international Symposium on Microarchitecture (Albuquerque, New Mexico, Puerto Rico). MICRO 24. ACM, New York, NY, 212-216.

[85] Su, B., Ding, S., and Jin, L. 1984. An improvement of trace scheduling for global microcode compaction. In Proceedings of the 17th Annual Workshop on Microprogramming International Symposium on Microarchitecture. IEEE Press, Piscataway, NJ, 78-85.

[86] Su, B., Ding, S., and Xia, J. 1986. URPR – An extension of URCR for software pipelining. SIGMICRO Newsl. 17, 4 (Dec. 1986), 94-103.

[87] Su, B., Ding, S., Wang, J., and Xia, J. 1987. GURPR – a method for global software pipelining. In Proceedings of the 20th Annual Workshop on Microprogramming (Colorado Springs, Colorado, United States, December 01 - 04, 1987). MICRO 20. ACM, New York, NY, 88-96.

[88] Aiken, A. and Nicolau, A. 1988. Optimal loop parallelization. In Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (Atlanta, Georgia, United States, June 20 - 24, 1988). R. L. Wexelblat, Ed. PLDI '88. ACM, New York, NY, 308-317.

[89] Aiken, A. and Nicolau, A, 1988. Perfect pipelining: A new loop optimization technique, In Proceedings of the 1988 European Symposium on Programming. Springer Verlag Lecture Notes in Computer Science, #300 (Atlanta, GA, March), 221-235.

[90] Allan, V. H., Rajagopalan, M., and Lee, R. M. 1993. Software Pipelining: Petri Net Pacemaker. In Proceedings of the IFIP Wg10.3. Working Conference on Architectures and Compilation Techniques For Fine and Medium Grain Parallelism (January 20 - 22, 1993). M. Cosnard, K. Ebcioglu, and J. Gaudiot, Eds. IFIP Transactions, vol. A-23. North-Holland Publishing Co., Amsterdam, The Netherlands, 15-26.

[91] Rajagopalan, M. and Allan, V. H. 1994. Specification of software pipelining using Petri nets. Int. J. Parallel Program. 22, 3 (Jun. 1994), 273-301.

[92] Gao, G. R., Wong, Y., and Ning, Q. 1991. A timed Petri-net model for fine-grain loop scheduling. In Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (Toronto, Ontario, Canada, June 24 - 28, 1991). PLDI '91. ACM, New York, NY, 204-218.

[93] Котов В.Е. Сети Петри. – М.: Наука. Главная редакция физико-математической литературы, 1984. – 160 с.

[94] Ebcioğlu, K. 1987. A compilation technique for software pipelining of loops with conditional jumps. In Proceedings of the 20th Annual Workshop on Microprogramming (Colorado Springs, Colorado, United States, December 01 - 04, 1987). MICRO 20. ACM, New York, NY, 69-79.

[95] Ebcioğlu, K. and Nakatani, T. 1990. A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture. In Selected Papers of the Second Workshop on Languages and Compilers For Parallel Computing (Urbana, Illinois, United States). D. Gelernter, A. Nicolau, and D. Padua, Eds. Pitman Publishing, London, UK, 213-229.

[96] Rau, B. R., Yen, D. W., Yen, W., and Towie, R. A. 1989. The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-Offs. IEEE Computer 22, 1 (Jan. 1989), 12-35.

[97] Winkel, S. 2007. Optimal versus Heuristic Global Code Scheduling. In Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture (December 01 - 05, 2007). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 43-55.

[98] http://developers.sun.com/solaris/articles/perfoptions.html. Опция компиляции -Qeps включает Enhanced Pipeline Scheduling, July 2007.

[99] Vinod Kathail, Mike Schlansker, and Bob Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, Hewlett-Packard Laboratories, February 1993.

[100] Останевич А.Ю. Планирование операций при генерации кода для архитектур с явно выраженным параллелизмом – Диссертация на соискание ученой степени кандидата технических наук, М., НИИ “Вычислительные Технологии”, 1999.

[101] Tu, P. and Padua, D. 1995. Efficient building and placing of gating functions. In Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (La Jolla, California, United States, June 18 - 21, 1995). PLDI '95. ACM, New York, NY, 47-55.

[102] J. C. Park and M. S. Schlansker. On predicated execution. Tech. Rep. HPL-91-58, Hewlett Packard Laboratories, Palo Alto, CA, May 1991.

[103] Дроздов А. Ю., Новиков С. В., Шилов В. В. Эффективный алгоритм преобразования потока управления в поток данных. Информационные технологии. № 2. 2005. Приложение. С. 24-31.

[104] Reddi, V. J., Settle, A., Connors, D. A., and Cohn, R. S. 2004. PIN: a binary instrumentation tool for computer architecture research and education. In Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st international Symposium on Computer Architecture (Munich, Germany). WCAE '04. ACM, New York, NY, 22.

[105] Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In Proceedings of the 2005 ACM SIGPLAN Conference on Programming

Language Design and Implementation (Chicago, IL, USA, June 12 - 15, 2005). PLDI '05. ACM, New York, NY, 190-200.

[106] Nethercote, N. and Seward, J. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (San Diego, California, USA, June 10 - 13, 2007). PLDI '07. ACM, New York, NY, 89-100.

[107] N. Nethercote. Dynamic Binary Analysis and Instrumentation. PhD thesis, University of Cambridge, United Kingdom, November 2004.

[108] Lawler, E. Combinatorial Optimization: Networks and Matroids. ISBN: 0-03-084866-0. Holt, Rinehart and Winston. 1976.

[109] Филиппова В. Поиск максимальной длины рекуррентности в графе зависимостей. Выпускная квалификационная работа на соискание степени бакалавр. ФТРК МФТИ, Москва, 2009.

[110] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. SIGOPS Oper. Syst. Rev. 40, 2006.

[111] Google, “What is Android?”. http://developer.android.com/guide/basics/what-is-android.html.

[112] L. Gwennap, Nvidia's First CPU Is a Winner, Microprocessor Report, 18th August 2014.

[113] D. Boggs, G. Brown, B. Rozas, N. Tuck and K S Venkatraman. Nvidia's denver processor. 2014 IEEE Hot Chips 26 Symposium (HCS). IEEE, Cupertino, CA, USA, August 2014.

Свидетельства о государственной регистрации программы для ЭВМ.

[1] Гимпельсон В.Д., Маслов М.В., Рыбаков А.А., Воронов Н.В., Садовников О.А., Айрапетян Р.Б., Савченко Р.А., Крылов С.М., Анисимов А.Б., Фомин А.А. «Eltechs ExaGear». Свидетельство о государственной регистрации программы для ЭВМ №2014611961 от 14.02.2014.

Список иллюстраций.

Список таблиц.

Приложение А. Описание алгоритма конвейеризации циклов Perfect Pipelining

Perfect pipelining был впервые предложен в работах [88] и [89]. Этот метод сочетает в себе техники перемещения кода и планирования. В некотором смысле perfect pipelining похож на технику методы изложенные в разделе 4.1.3: цикл также подвергается раскрутке для того, чтобы найти и сформировать физическую итерацию. Однако этот метод является более сложным и эффективным. Perfect pipelining может применяться к циклам содержащим несколько линейных участков, а также позволяет исполнять одну логическую итерацию за дробное число тактов. Исторически этот алгоритм реализовывался для эффективной конвейеризации циклов на новых, на тот момент, архитектурах с возможностью исполнять несколько операций условного перехода за один такт.

Работа алгоритма начинается с глобального переноса вверх операций в цикле. Это делается для того, чтобы ускорить время работы алгоритма. После этого цикл раскручивается на потенциально бесконечное число итераций. Это означает, что мы считаем, что раскрутили цикл настолько раз, насколько нам потребуется в ходе дальнейшей работы алгоритма. Результат раскрутки планируется в предположении наличия бесконечного количества ресурсов. Это делается потому, что идея алгоритма заключается в беспрепятственном переносе операций. В отличие от варианта с ограниченными ресурсами на процесс планирования не оказывает влияния ранее спланированные операции и не возникает ситуации, когда мы не можем спланировать операцию в нужный такт только из-за того, что все ресурсы в этом такте заняты, хотя, возможно, некоторые спланированные операции можно было безболезненно разместить в другом такте, то есть это жадный алгоритм.

Теперь опишем как происходит выбор физической итерации. Состояние на каждом шаге планирования[1] при наличии бесконечных ресурсов можно описать конечным множеством. Оно описывает информацию, на основании которой будет принято решение о том, какие операции будут спланированы в следующей инструкции. Состояние описывается множеством операций, которые можно спланировать в текущем такте, а также множеством операций, которые ждут планирования[2] с указанием сколько тактов ещё необходимо ждать. Если два состояния одинаковы, то планирование с этого момента пойдёт одинаково, то есть планирование начнёт повторять себя. Набор операций между двумя одинаковыми состояния планирования и берется за новую физическую итерацию цикла. Так как множество состояний планирования конечно, то

[1] Шагом здесь называем планирование (заполнение) одной инструкции

[2] Операции, предшественники которых уже спланированы ранее, но ещё от какого-либо предшественника полностью не выдержана задержка

рано или поздно состояние повторится и физическая итерация будет найдена. Однако, найденная таким образом физическая итерация, может быть функционально (семантически) не эквивалентна исходному циклу. Приведём пример. Пусть цикл состоит из трёх операций. На Рис. 27 а) приведён граф зависимостей для этих операций. Длины всех зависимостей равны единице. На Рис. 27 б) приведено планирование, которое получается в результате работы алгоритма perfect pipelining. Начиная с четвёртой инструкции планирование повторяется группами по две инструкции. Однако в получаемой физической итерации операция под номером 1 будет содержаться в двух копиях, с двух последовательных физических итераций. В тоже время операции под номерами 2 и 3 будут содержаться по одному разу, с одной физической итерации. Таким образом получилось, что одна физическая итерация содержит различное количество логических итераций для разных групп операций. В такой ситуации потребуется построить постцикл часть которого должна выполниться динамическое число раз, зависящее от количества итераций выполненных в цикле. Такое развитие событий несколько отходит от самой идеи конвейеризации цикла и может приводить к серьёзным потерям производительности. Для решения описанной проблемы в рассматриваемом алгоритме реализована точная проверка на функциональную эквивалентность найденной физической итерации.

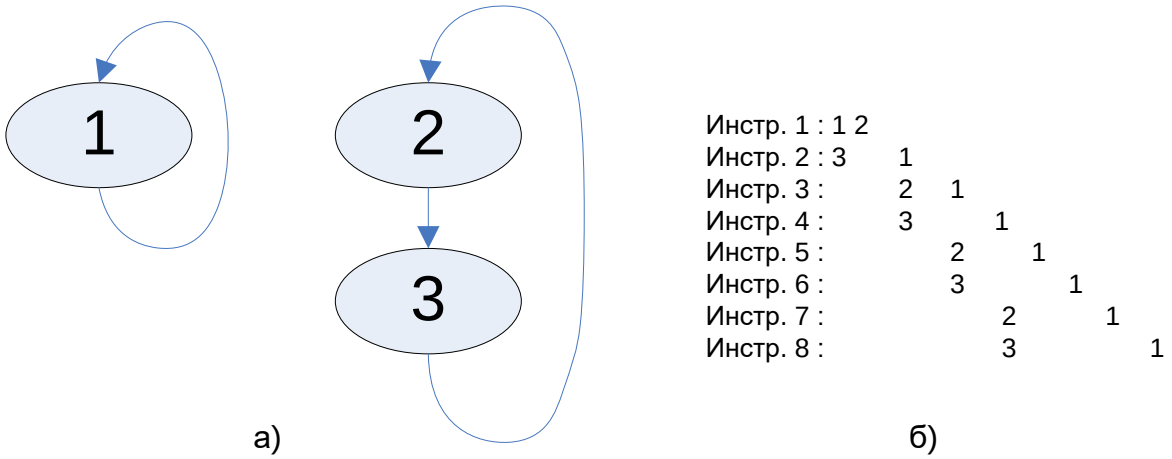

Рис. 27. *Пример, когда найденная в результате работы алгоритма perfect pipelining физическая итерация не является корректной.*

Для решения проблемы функциональной не эквивалентности физической итерации исходному циклу вводиться ограничение на расстояние между итерациями рассматриваемое планировщиком. Вводиться число s, которое говорит о том, что пока все операции с итерации i не спланируются, не начинают планироваться операции с итерации $i+s$.

Рассматриваемый алгоритм позволяет естественным способом получить дробное значение тактов необходимое на исполнение одной логической итерации цикла. Это получается за счёт того, что одновременно планируются сразу несколько итераций цикла. Дробное значение числа логических итераций выполняемых за одну физическую можно добиться и в описанном ранее алгоритме модульного планирования, предварительно раскрутив цикл. Однако предсказать заранее на сколько именно итераций необходимо раскрутить цикл является достаточно трудной задачей.

Недостатками этого алгоритма являются, во-первых, потенциально большое дублирование кода, которое возникает из-за раскрутки цикла. Такое дублирование приводит как к замедлению скорости работы алгоритма за счёт того, что необходимо спланировать большое количество операций, так и может привести к потерям в скорости работы результирующего кода за счёт нехватки кэша инструкций. Во-вторых, алгоритм никак не учитывает ресурсные ограничения имеющиеся в реальных микропроцессорах. В следствии этого, когда надо будет распределить имеющиеся ресурсы, возникнут потери в качестве результирующего кода. Ещё одним недостатком можно считать достаточно сложный алгоритм определения одинаковых состояний.

Приложение Б. Описание алгоритма конвейеризации циклов с использованием сетей Петри

Алгоритм конвейеризации циклов, основанный на математическом аппарате сетей Петри [93], был предложен в работе [92], а затем в работах [90] и [91] были сняты многие ограничения присущие первоначальному алгоритму.

С помощью этого алгоритма можно получить дробное количество логических итераций выполняемых за одну физическую. Также алгоритм не накладывает никаких ограничений на длины задержек и количество итераций через которое реализуется зависимость. По своей мощности алгоритм сравним с алгоритмом Perfect Pipelining, изложенном в предыдущем пункте.

Сеть Петри это четвёрка $G(P,T,A,M_0)$. Тройка P,T,A образуют двухдольный граф. Множество P и T – это вершины графа, называемые местами и переходами соответственно, а множество A – множество дуг. Каждая дуга идёт либо из P в T, либо из T в P. Разметкой сети M называется функция $M: P \to N \cup \{0\}$. Последний член четвёрки M_0 является некоторой разметкой: $M_0 : P \to N \cup \{0\}$ и называется начальной разметкой сети.

Функционирование сети Петри описывается формально с помощью множества последовательных срабатываний. Переход $t \in T$ может сработать, если каждое входное место этого перехода имеет ненулевую разметку. Срабатывание перехода t изменяет разметку таким образом, что разметка каждого его входного места уменьшается на единицу, а разметка каждого его выходного места увеличивается на единицу.[1]

Дальнейшее функционирование сети Петри однозначно определяется текущей разметкой. Если в какой-то момент разметка повторится, то с этого момента начнёт повторяться и всё поведение сети.

С помощью сетей Петри можно моделировать расширенный граф зависимостей. Каждая операций графа зависимостей представляется переходом. Места показывают готовы ли все предшественники операции. Каждой дуге (a,b) расширенного графа зависимостей с длиной равной единице сопоставляется место и пара его инцидентных дуг (первая дуга от перехода соответствующего операции a к месту, вторая от места к переходу соответствующего операции b). Зависимости длины больше единицы можно моделировать с помощью вставления

[1] Здесь представлено определения некоторого подкласса сетей Петри называемого ординарными сетями Петри. С общем случае каждая дуга сети имеет ещё неотрицательную кратность. Переход в таком случае может совершиться только при условии, что в каждом из предшествующих мест значение разметки не меньше кратности дуги. Во все места – последователи сработавшего перехода к разметке добавляется кратность дуги.

нескольких фиктивных узлов на дугу. Таким образом без ограничения общности можно считать, что все зависимости имеют длину не большую единицы. Зависимости длины ноль также требуют специальной обработки. Для всех узлов, у которых есть входная дуга нулевой длины, осуществляется специальный промежуточный этап: если это возможно, то реализуется срабатывание этих переходов.

В исходный граф зависимостей добавляются дуги для того, чтобы сделать его сильно связанным. Это необходимо для того, чтобы более короткие рекуррентности не обгоняли более длинные рекуррентности.

Теперь опишем как выбирается начальная разметка. Каждая дуга, которая идёт с предыдущей итерации, готова к исполнению. Следовательно, места на дугах, представляющих межитерационные зависимости, должны быть помечены. Значение разметки на некотором месте равно количеству итераций, через которое реализуется соответствующая зависимость. Если между операциями a и b есть зависимость, которая реализуется через d итераций, то это означает, что на первых d итерациях цикла операция b не зависит от операции a, поэтому можно разрешить ей первоначально исполниться d раз без зависимости от операции a.

После того как граф зависимостей был модифицирован строится соответствующая ему сеть Петри. В каждый момент времени осуществляется группа переходов. Эта группа формирует один такт планирования. Как только разметка в какой-то момент времени повторяется, это означает, что физическая итерация найдена. Она формируется из всех операций, которые были спланированы между повторяющимися состояниями.

Теперь рассмотрим вопрос о том, как в рассматриваемой модели выдерживаются ограничения на ресурсы. Для каждого ресурса создаётся место p. Это месту присваивается столько ресурсов, сколько их имеется в наличии в одном такте. Например, если у нас имеется шесть равнозначных исполняющих устройств, то необходимо создать место с начальной разметкой равной шести. Использование ресурсов контролируется тем, что все операции нуждающиеся в ресурсе циклически связываются с местом соответствующим этому ресурсу. Если некоторая операция использует некоторый ресурс, то существует путь ведущий из места соответствующего ресурсу в переход представляющий операцию и обратно. Так как переход не может быть выполнен, если разметка места нулевая, то при отсутствии ресурсов переход не будет выполнен (операция не будет спланирована). Дуга ведущая от перехода к месту соответствующему ресурсу, позволяет вернуть ресурс после его использования. Таким образом возможно моделировать регулярные ресурсы. Для более сложных типов ресурсов в приведённых работах также имеются модели для их описания. Однако мы здесь не будем подробнее останавливаться на этом.

Описанный алгоритм даёт хороший результат. Он хорошо формализован и достаточно гибок для описания различных видов ограничений. Его недостатком является то, что во время работы алгоритма не должно меняться промежуточное представление. Следовательно во время работы алгоритма нельзя разрывать зависимости, и это можно делать только до начала конвейеризации цикла. Однако в этот момент достаточно сложно определить какие зависимости нужно разрывать, а какие нет. Также существенное ограничение вносит тот факт, что обрабатываются только циклы с одной обратной дугой. Если в цикле было несколько обратных дуг и для проведения конвейеризации цикл был преобразован в цикл имеющий одну обратную дугу, то время работы одной итерации цикла в случае прохода по любой из первоначальных обратных дуг будет одинаковым, и равняется максимальному времени из всех дуг. Таким образом возможны существенные потери в качестве результирующего кода в случае наличия нескольких несбалансированных обратных дуг.