Оптимизация переходов в системе двоичной трансляции для архитектуры «Эльбрус»

Рыбаков Алексей Анатольевич[1,2]

[1]*ЗАО «МЦСТ», ул. Нижняя Красносельская, д. 35, строение 50, Москва, Россия*
[2]*ООО «Институт электронных управляющих машин им. И. С. Брука», ул. Вавилова, д. 24, Москва, Россия*

rybakov.aax@gmail.com

Аннотация. *В компании ЗАО «МЦСТ» разрабатываются микропроцессоры архитектуры «Эльбрус». Для вычислительного комплекса «Эльбрус» разработана система двоичной трансляции LIntel, позволяющая исполнять приложения Intel x86 на микропроцессорах «Эльбрус». Важной составляющей LIntel является многоуровневый оптимизирующий двоичный транслятор. Команды подготовки переходов в архитектуре «Эльбрус» позволяют распараллеливать исполнение программы по разным ветвям и выполнять переходы за один такт, без потери тактов. Применение оптимизации переходов в двоичном трансляторе позволяет переносить команды подготовки переходов между линейными участками программы, достигая таким образом существенного повышения производительности системы.*

Ключевые слова

Оптимизирующие компиляторы, двоичная трансляция, Intel x86, «Эльбрус», глобальное планирование, промежуточное представление, подготовки переходов.

1 Введение

В течение всего времени развития компьютерной техники проблема скорости выполнения программ не теряет своей актуальности. Для повышения скорости работы приложений создано множество различных микропроцессорных архитектур в совокупности с оптимизирующими компиляторами, которые позвляют использовать особенности данных архитектур.

При появлении новых микропроцессорных архитектур возникает проблема переноса на них ранее созданного программного обеспечения. При этом исходный код того или иного приложения зачастую недоступен, и перекомпилировать его для новой архитектуры не представляется возможным. Технология динамической двоичной трансляции [1] позволяет переносить программное обеспечение путем перевода двоичного кода из набора инструкций исходной архитектуры в набор инструкций целевой архитектуры.

1.1 Особенности архитектуры «Эльбрус»

В компании ЗАО «МЦСТ» разрабатываются микропроцессоры архитектуры «Эльбрус» [2, 3, 6]. Архитектура «Эльбрус» обладает высокой предельной архитектурной скоростью, которая достигается за счет ее многочисленных особенностей.

Основной причиной высокой производительности архитектуры «Эльбрус» является параллелизм исполнения кода на уровне отдельных команд. Это обусловлено тем, что «Эльбрус» является VLIW-архитектурой (very large instruction word), или архитектурой с *широким командным словом* [9], что позволяет кодировать в одной VLIW-команде несколько отдельных операций и выполнять их параллельно. Время выполнения каждой отдельной операции детерминировано, а также известны все задержки, которые должны быть выдержаны между операциями. Используя эту информацию на этапе компиляции, можно эффективно планировать широкие команды, максимально используя вычислительные ресурсы процессора.

В микропроцессоре «Эльбрус» присутствует 6 арифметико-логических устройств (arithmetic logic unit, ALU), четыре из которых могут использоваться для вычислений с плавающей арифметикой. Кроме того, некоторые ари-

фметические операции могут объединяться в так называемые *комбинированные* („двухэтажные“) операции, что позволяет загружать вычислительные ресурсы микропроцессора более эффективно.

Другой особенностью является большой размер *регистрового файла*, организованного в виде процедурных окон. При переполнении регистрового файла автоматически выполняется его откачка, при исчерпании - подкачка.

Наличие *спекулятивного* (упреждающего) режима выполнения операций позволяет выполнять некоторые операции до того, как станет известно, понадобится ли их результат для дальнейших вычислений. Для этого на аппаратном уровне должна быть возможность игнорировать исключения, которые могут возникнуть во время выполнения данных операций. Это может быть, например, обращение в память по неверному адресу. Кроме того, в аппаратуре должны быть предусмотрены случаи работы операций со специальными *диагностическими значениями*, являющимися результатами спекулятивных операций, которые не будут далее использованы в вычислениях. Использование спекулятивного режима исполнения операций позволяет применять некоторые специфические оптимизации, которые могут существенно повысить производительность результирующего кода [10].

Наличие *предикатного* (условного) режима исполнения операций позволяет в некоторых случаях избежать создания ненужных операций передачи управления путем планирования в одном линейном участке команд под разными предикатами [11]. Это хорошо видно, например, при слиянии под противоположными предикатами двух ветвей исполнения конструкции if-else.

Предикатные вычисления архитектуры «Эльбрус» отличаются тем, что предикат не является обычной булевой переменной. Кроме значений true и false предикат может принимать диагностическое значение, которое накладывает на операции с предикатами специальное ограничение — операция конъюнкции двух предикатов не является коммутативной. В одной широкой команде может содержаться до трех операций с предикатами, причем аргументом одной предикатной операции может служить результат другой предикатной операции, располагающейся в той же широкой команде.

Специальные операции *подготовки переходов* позволяют распараллелить передачу управления по разным ветвям исполнения и выполнять переходы за один такт. Кроме того, при использовании операций подготовки переходов возможно выполнение предварительной подкачки кода, что приводит к уменьшению задержек подкачки кода, связанных с промахами в кэш инструкций [6]. В одной широкой команде может присутствовать одна команда перехода и одна команда подготовки перехода.

В архитектуре «Эльбрус» реализована аппаратная поддержка *конвейеризации циклов*. С помощью конвейеризации итерация цикла разбивается на несколько последовательных частей (*стадий*), а затем эти стадии планируются так, что одновременно могут исполняться разные стадии сразу нескольких итераций цикла. Для поддержки конвейеризации циклов используются такие аппаратные средства, как переименование регистров путем вращения регистровой базы, счетчик цикла и специальные предикаты, управляющие выполнением отдельных стадий [12, 13]. Если оптимизируемый цикл удовлетворяет некоторым определенным требованиям, то применение конвейеризации позволяет ускорить его в несколько раз. Особенно заметный прирост производительности наблюдается для циклов, работающих с плавающей арифметикой.

Также в архитектуре «Эльбрус» присутствуют специальные команды *асинхронной предварительной подкачки* данных, которые позволяют более эффективно работать с иерархической системой памяти. Команды предварительной подкачки позволяют обращаться за данными в память до того, как результат обращения используется для вычислений, уменьшая таким образом количество задержек по доступу в память, связанных с промахами в кэши разных уровней [14]. В одной широкой команде могут располагаться четыре операции асинхронной загрузки из памяти.

1.2 Система динамической двоичной трансляции LIntel

Технология динамической двоичной трансляции обеспечивает полную совместимость архитектуры «Эльбрус» с архитектурой Intel x86 [7, 8]. Для вычислительного комплекса «Эльбрус» разработана аппаратно поддерживаемая система двоичной трансляции LIntel, которая эмулирует поведение машины x86 путем декодирования инструкций x86 и перевода их в коды архитектуры «Эльбрус».

Основными составляющими LIntel являются интерпретатор, многоуровневый двочный транслятор и система поддержки, обеспечивающая функционирование и целостность всей системы.

Интерпретатор предназначен для пошагового исполнения инструкций x86 с точным их моделированием и обработкой возможных прерываний. Интерпретатор используется, когда требуется моделировать поведение процессора при возникновении исключения или при первом исполнении кода. Если код x86 начинает исполняться часто, то управление от интерпретатора передается многоуровневому двоичному транслятору. Задачей транслятора является создание кода целевой платформы, который может быть сохранен и впоследствии выполнен многократно.

Двоичный транслятор LIntel состоит из трех уровней. Первый уровень представляет шаблонный транслятор [15]. Далее следует быстрый оптимизирующий компилятор (компилятор уровня O0) [16], который применяет к созданному коду некоторый ограниченный набор оптимизаций. Качество кода возрастает, но также возрастает и время компиляции. И, наконец, оптимизирующий компилятор уровня O1, выполняющий полный набор оптимизаций, создает наиболее эффективный код, но еще больше времени затрачивает на трансляцию. Из соображений минимизации времени работы системы в целом нужно придерживаться правила — чем чаще исполняется код, тем более высокого уровня транслятор целесообразно использовать [17]. Во время работы системы переключение между уровнями трансляции происходит динамически.

Таблица 1. Характеристики уровней двочной трансляции.

	Время трансляции	**Качество кода**
Интерпретатор	100	-
Шаблонный транслятор	1200	1/4
Компилятор O0	15000	2/3
Компилятор O1	500000	1

В таблице 1 приведены сравнительные данные по времени компиляции исходного кода x86 различными уровнями траслятора, а также сравнение результирующего кода с кодом, полученным с помощью компилятора O1. В колонке «Время трансляции» приведено количество тактов «Эльбрус», затрачиваемое на трансляцию в пересчете на одну исходную инструкцию x86. Для интерпретатора это означает количество тактов, затрачиваемое на эмуляцию одной инструкции. В колонке «Качество кода» приведено отношение средней скорости работы результирующего кода, полученного с помощью определенного уровня транслятора, к средней скорости работы результирующего кода, полученного с помощью компилятора O1. Для интерпретатора эта характеристика лишена смысла, так как для него отсутствует понятие результирующего кода.

В данной статье речь пойдет о быстром оптимизирующем компиляторе.

2 Алгоритм переноса подготовок переходов между узлами промежуточного представления быстрого компилятора

2.1 Описание быстрого компилятора

Единицей компиляции быстрого компилятора является регион, который представляет собой объединение некоторого числа линейных участков (последовательностей команд с одной точкой входа), связанных между собой переходами. Если некоторый линейный участок оканчивается не командой перехода, то будем говорить, что он оканчивается провалом. В этом случае после исполнения последней команды линейного участка управление перейдет на следующую ячейку памяти. Для своей работы быстрый компилятор использует промежуточное представление, основой которого является граф потока управления. Узлами данного графа являются линейные участки, а ребрами — переходы между ними.

Процесс компиляции региона можно условно разделить на три этапа. На первом этапе для каждого линейного участка осуществляется генерация семантики команд x86. Далее, опираясь на сгенерированное промежуточное представление, последовательно применяется ряд базовых оптимизаций, позволяющих существенно повысить производительность результирующего кода при небольшом увеличении времени компиляции. К наиболее важным оптимизациям относятся слияние суперблоков [18, 19], перенос операций между узлами, устранение избыточных обращений в память [5], разрыв зависимостей по доступу в память [6] и другие. После проведения всех оптимизаций для каждого узла промежуточного представления осуществляется распределение регистров, планирование широких команд и генерация кода архитектуры «Эльбрус». В данной статье рассматривается оптимизация переноса подготовок переходов между линейными участками (узлами графа потока управления).

2.2 Общее описание переноса операций между узлами

При проведении оптимизаций и планировании кода в рамках узла промежуточного представления из-за задержек между операциями возникает простаивание вычислительных ресурсов, обусловленное незанятостью исполнительных устройств в некоторых тактах. Оптимизация переноса критических операций между узлами позволяет удалить операции из одного узла и спланировать их в потенциально свободные места в других узлах.

По своему назначению данная оптимизация близка к шагу глобального планирования [4, 20]. Идея состоит в переносе критических операций, стоящих в начале узла, вверх по всем входящим дугам в предшественники данного

узла. При этом перенесенная операция располагается перед переходом по соответствующей дуге. Цель такого преобразования — попытка загрузить с помощью перемещаемых операций потенциально свободные вычислительные ресурсы.

Рассмотрим пример перемещения операции между узлами (рис. 1). Допустим, принято решение о переносе операции из узла A в узел B. Существует три варианта связи между двумя данными узлами:

1. Узел B доминирует над A, узел A постдоминирует над B (рис. 1, а). Данный случай является самым простым. Операция может быть перенесена из узла A в узел B и помещена перед переходом на узел A.
2. Узел B не доминирует над A (рис. 1, б). Это значит, что существует путь в узел A в обход узла B (пусть такой путь проходит через узел C). В данном случае операция должна быть перенесена не только в узел B, но также в узле C должна быть расположена ее копия.
3. Узел A не постдоминирует над B (рис. 1, б). Это значит, что существует путь из узла B, не проходящий через A (пусть такой путь проходит через узел D). В данном случае необходимо контролировать, чтобы при потоке исполнения по пути $Node\ B \rightarrow Node\ D$ выполнение перенесенной операции не сказывалось на исполнение в узле D и далее.

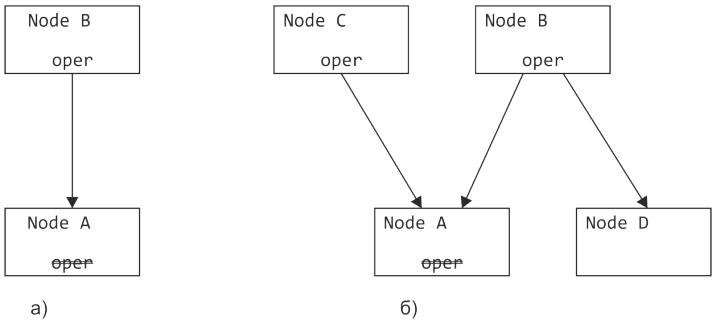

Рис. 1. Перенос операций между узлами.

Чтобы точно определить, является ли операция критической, то есть она задерживает планирование идущих за ней операций, нужно проводить предварительное планирование, что сильно увеличивает время компиляции региона. Поэтому для принятия решения применяются приближенные эвристические оценки. Нужно учитывать и возможный негативный эффект от излишне агрессивного применения оптимизации. Так, если операция должна быть перенесена хотя бы по одной дуге в узел, количество исполнений которого существенно больше количества исполнений исходного узла, то от применения оптимизации к данной операции следует отказаться, так как это приведет к перемешиванию часто исполняемого кода с кодом, исполняемым гораздо реже.

2.3 Описание алгоритма переноса подготовок переходов между узлами

В архитектуре «Эльбрус» для переходов ($BRANCH$) используются специальные регистры, которые содержат адрес перехода и некоторую дополнительную информацию ($CTPR$). Эти регистры также называют станками переходов. Всего таких регистров три. Для инициализации станков используются специальные команды подготовки переходов (CTP). Использование команд подготовок переходов позволяет распараллелить передачу управления по разным веткам. При использовании операций подготовки переходов возможно выполнение предварительной подкачки кода, что приводит к уменьшению задержек подкачки кода, связанных с промахами в кэш инструкций [6]. При планировании команд между переходом и его подготовкой должна быть выдержана определенная задержка. Если эта задержка выдержана, то переход выполняется за один такт, без потери тактов. Однако, если подготовка перехода стоит близко к переходу, то это может привести к потере тактов. Поэтому операции подготовки переходов, использующиеся в архитектуре «Эльбрус», являются подходящими кандидатами для применения оптимизации переноса операций между узлами.

Рассмотрим алгоритм, позволяющий выносить из узлов подготовки наиболее частых переходов. Во время проведения оптимизации доступен профиль выполнения оптимизируемого региона, в частности для каждого узла известно количество его исполнений (счетчик узла).

Из каждого узла будем пытаться вынести подготовку только для одного первого перехода. Из-за небольшого количества станков выносить другие переходы (кроме первого) нецелесообразно.

Узлы промежуточного представления будем обрабатывать последовательно в порядке убывания счетчика, начиная с узла с максимальным счетчиком. Это поможет уменьшить размер кода в первую очередь для самых частых узлов.

При обработке каждого узла последовательность действий следующая:

1. Выполняется поиск первой операции подготовки перехода в узле. Если такая операция отсутствует, то переходим к обработке следущего узла.
2. При переносе операции подготовки перехода вверх по входным дугам в узлы-предшественники возможно возникновение конфликтов по станку. При возникновении конфликта станок переносимой операции должен быть изменен. Для определения доступных номеров станков вычисляется соответственно маска допустимых для использования в подготовке станков. Если данная маска нулевая, значит перенос выполнить не удастся, и нужно переходить к обработке следующего узла.
3. Далее следует проверить, нужно ли проводить переименование станка в операции подготовки перехода (и в соответствующем переходе). Если в маске допустимых станков присутствует номер станка подготовки, то переименование производить не нужно. В противном случае следует выбрать любой номер станка из маски допустимых номеров и выполнить переименование станка в подготовке и соответствующем ей переходе (рис. 2).

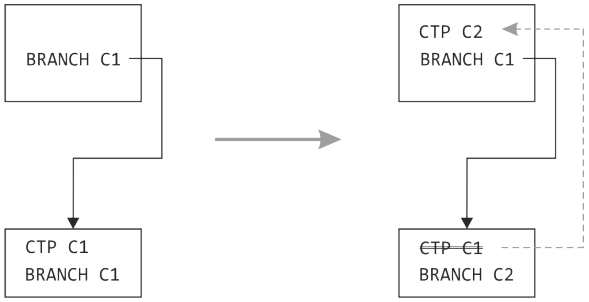

Рис. 2. Переименование станка перехода.

4. Осуществляем перенос копий операции подготовки вверх по всем входным дугам. Сама же операция подготовки перехода удаляется из узла.

Рассмотрим некоторые особенности описанного алгоритма. Так как при переносе подготовки перехода возможно возникновение конфликтов по станку, то необходимо обнаружить и разрешить данные конфликты. Конфликты по станку бывают двух типов.

Пусть переносимая операция подготовки перехода использует некоторый станок $C1$. Если данная подготовка должна переноситься по дуге, переход по которой также выполняется по станку $C1$, то имеем конфликт первого типа (рис. 3, а).

Конфликт второго типа менее очевиден и связан с тем, что в одном такте может быть спланирована одна операция перехода и одна операция подготовки перехода. Пусть переносимая операция подготовки перехода использует станок $C1$. Пусть она переносится из узла $Node\ A$ в узел $Node\ B$ по дуге, переход по которой осуществляется по другому станку — $C2$. Если в узле $Node\ B$ ниже перехода по дуге $Node\ B \rightarrow Node\ A$ присутствует другая подготовка, использующая станок $C1$, то она может быть спланирована с одном такте с данным переходом. В этом случае имеем конфликт второго типа (рис. 3, б).

Все возможные конфликты по станкам по всем входящим в узел дугам должны быть учтены при переносе подготовки перехода в маске доступных станков, иначе может возникнуть ошибка исполнения.

Заметим также, что оптимизация использует некоторые эвристики, которые позволяют повысить производительность результирующего кода. Например, перенос подготовок для маловероятных переходов зачастую оказывается бесполезным, так что в некоторых случаях от него можно отказаться. С другой стороны, переименование станков может привести к возникновению двух подряд идущих переходов по одному и тому же станку. Это гарантированно ухудшает результирующий код, так что переименования станков в этом случае также желательно избегать.

Рассмотрим еще одну особенность промежуточного представления, которую следует учитывать во время применения переноса подготовок. Если узел промежуточного представления оканчивается операцией условного перехода, то в случае невыполнения условия перехода осуществляется переход по провалу (рис. 4, а). Если в узле,

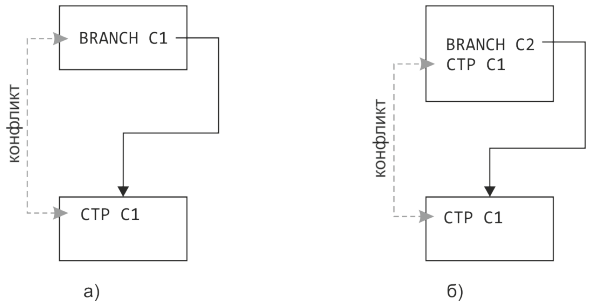

Рис. 3. Конфликты по станкам.

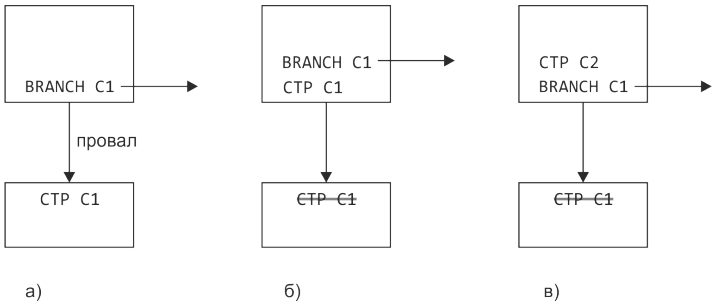

Рис. 4. Перенос подготовки по провалу.

на который осуществляется данный переход по провалу, присутствует подготовка, которая может быть перенесена, и станок этой подготовки совпадает со станком условного перехода, которым оканчивался исходный узел, то перенесенная подготовка не сможет подняться выше условного перехода (рис. 4, б).

Так как изначально узел оканчивался условным переходом, от которого не зависят никакие операции, то этот переход практически гарантированно находится на критическом пути и планируется в последней широкой команде узла. Добавление же в узел новой команды, зависящей от данного перехода, увеличивает длину критического пути, тем самым ухудшая планирование. Для обхода этой ситуации в случае переноса подготовки перехода по провалу требуется занести ее перед предыдущим переходом по тому же станку и применить переименование станков (рис. 4, в).

2.4 Оценка сложности алгоритма

Для оценки сложности алгоритма обозначим граф потока управления (control flow graph, CFG) через G. Пусть $\nu = \nu_G$ — количество его узлов, $\varepsilon = \varepsilon_G$ — количество ребер.

Так как оптимизация применяется к узлам в порядке уменьшения их счетчиков, то перед ее применением необходимо произвести сортировку узлов графа. Сложность данного действия равна $O(\nu \log \nu)$.

Для нахождения первой подготовки в каждом узле CFG необходимо обойти операции каждого узла, начиная с первой операции и заканчивая первой подготовкой. Количество действий, необходимое для этого, пропорционально общему количеству операций в графе, или $O(l\nu)$, где l — среднее число операций в узле.

Для каждой подготовки для каждой входной дуги необходимо выполнить анализ конфликтов по станкам. Так как всего подготовок, которые могут быть перенесены, $O(\nu)$, а среднее количество входных дуг в узел равно $\frac{\varepsilon}{\nu}$, то нужно проверить на конфликты $O(\varepsilon)$ входных дуг. Для проверки на конфликт по станкам необходимо обойти все операции от соответствующего перехода и до конца узла (количество действий пропорционально l). С учетом этого, общее количество действий, нужное для проверки конфликтов, равно $O(l\varepsilon)$.

Количество действий, требуемое для самого переноса операций подготовок, равно $O(\varepsilon)$.

Суммируя все необходимые для проведения оптимизации действия, получим итоговую оценку сложности алгоритма:

$$T(\nu, \varepsilon, l) = O(\nu \log \nu + l(\nu + \varepsilon)). \quad (1)$$

2.5 Результаты

Оценка эффективности описанного алгоритма производилась путем запуска исполняемых файлов x86 задач из пакета SPEC CINT2000 [21], транслированных с помощью быстрого компилятора, на симуляторе микропроцессора «Эльбрус». Для оценки эффективности алгоритма переноса подготовок переходов между узлами выполнялся подсчет общего количества подготовок переходов, являющихся первыми в узлах промежуточного представления, а также тех из них, к которым применилась оптимизация.

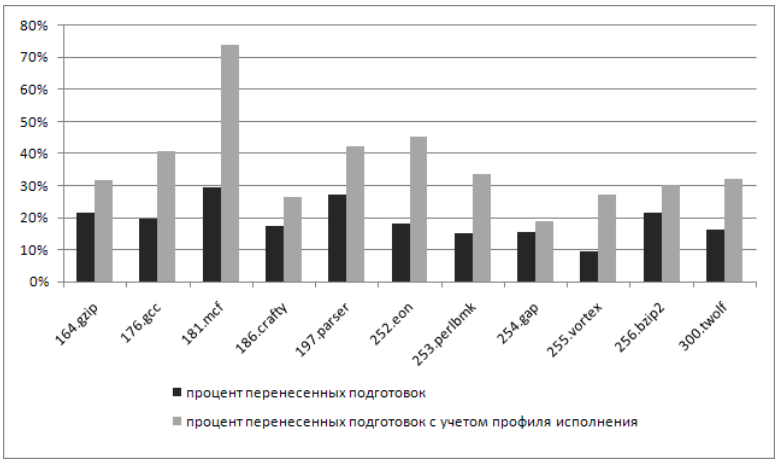

Рис. 5. Процент перенесенных подготовок на задачах SPEC CINT2000.

На рис. 5 приведены данные о процентной доле перенесенных подготовок переходов в рассматриваемых задачах. Характеристики, указанные в столбцах «процент перенесенных переходов» (T) и «процент перенесенных подготовок с учетом профиля исполнения» (T_ω), вычислялись по следующим формулам:

$$T = \frac{|V'|}{|V|},\ T_\omega = \frac{\sum_{v \in V'} \omega(v)}{\sum_{v \in V} \omega(v)}, \tag{2}$$

где V — множество узлов, содержащих подготовки, V' — множество узлов, из которых удалось вынести подготовку, $\omega(v)$ - счетчик узла v.

Для того, чтобы оценить общий эффект от использования переноса подготовок между узлами, были произведены запуски задач, откомпилированных базовым компилятором в следующих режимах:

1. Без использования переноса подготовок между узлами. Результаты запусков в данном режиме приняты за эталон.
2. С использованием переноса подготовок между узлами.

На рис. 6 можно видеть прирост производительности в районе $0.2\% - 2.8\%$ в зависимости от задачи. Особенно заметный прирост производительности наблюдается на задаче 176.gcc, так как эта задача характеризуется сильно разветвленным управлением и небольшой длиной линейных участков.

Оптимизации переноса подготовок переходов между узлами промежуточного представления применяется в рамках общей оптимизации переноса операций. Кроме подготовок переходов могут быть перенесены также операции чтения из памяти. Поэтому подсчитать точное время, затрачиваемое компилятором на перенос именно подготовок переходов, довольно затруднительно. Но примерно это время можно оценить в 0.5% от времени работы быстрого компилятора.

3 Заключение

Технология двоичной трансляции в современном мире является востребованной, так как позволяет исполнять двоичный код одной архитектуры на процессорах других архитектур. При этом при создании кода целевой платформы использование оптимизирующей компиляции способно существено увеличить работу приложения. Важным моментом являтся создание адаптивных многоуровневых оптимизирующих двоичных трансляторов, позволяющих регулировать линейку и агрессивность применяемых оптимизаций в зависимости от частоты исполнения

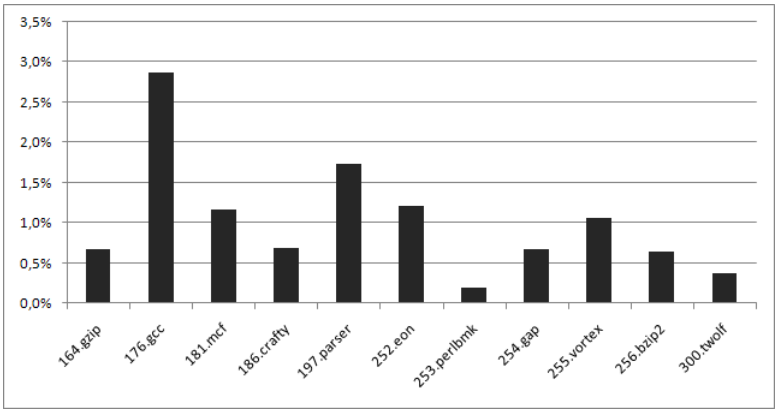

Рис. 6. Результаты измерения производительности на задачах SPEC CINT2000 при использовании переноса подготовок переходов.

оптимизируемого кода. Типичным контекстом для работы быстрого компилятора системы двоичной трансляции для архитектуры «Эльбрус» является код с не слишком большой частотой исполнения и с очень разветвленным управлением. Таким образом, важной частью быстрого компилятора является оптимизация переходов. Алгоритм переноса подготовок переходов между узлами промежуточного представления направлен на оптимизацию переходов и позволяет добиться заметного улучшения производительности результирующего кода.

Список литературы

[1] E. R. Altman, D. Kaeli, Y. Sheffer: Welcome to the Opportunities of Binary Translation. *Computer*, Vol. 33, No. 3, March 2000, P. 40-45.

[2] B. Babayan: Main principles of E2K architecture. *Free Software Magazine*, Vol. 1, No. 2, Feb. 2002.

[3] B. Babayan: E2K Technology and Implementation. *Parallel Processing: 6th International*, Vol. 1900/2000, Jan. 2000, P. 18-21.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ulman: Compilers: Principles, Techniques, and Tools. *Prentice Hall*, 2nd edition, Sep. 2006.

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

[6] В. Ю. Волконский: Оптимизирующие компиляторы для архитектуры с явным параллелизмом команд и аппаратной поддержкой двоичной совместимости. *Информационные технологии и вычислительные системы*, 3/2004.

[7] K. Diefendorf: The Russians Are Coming: Supercomputer Maker Elbrus Seeks to Join x86/IA-64 Melee. *Microprocessor report*, vol. 11, num. 2, Feb. 15, 1999.

[8] Intel Corporation: Intel IA-32 Architecture Software Developer's Manual. // Vol. 1-3, 2003.

[9] Joseph A. Fisher: Very Long Instruction Word Architectures and the ELI-512. *in Proceedings of 10th International Symposium on Computer Architectures*, IEEE. Jun. 1983, P. 140-150.

[10] А. А. Белеванцев, С. С. Гайсарян, В. П. Иванников: Построение алгоритмов спекулятивных оптимизаций. *Журнал Программирование*, 3/2008, С. 1-22.

[11] В. Ю. Волконский, С. К. Окунев: Предикатное представление как основа оптимизации программы для архитектур с явно выраженной параллельностью. *Информационные технологии*, №4, апрель 2003, С. 36 - 45.

[12] А. Ю. Дроздов, А. М. Степаненков: Технология оптимизации цикловых участков процедур в компиляторах для архитектур с аппаратной поддержкой конвейеризации циклов. *Информационные технологии и вычислительные системы*, 3/2004.

[13] В. Д. Гимпельсон: Конвейеризация циклов в двоичном динамическом транслляторе. *Вопросы радиоэлектроники*, выпуск 3, 2009.

[14] А. Б. Галазин, А. В. Грабежной: Эффективное взаимодействие микропроцессора и подсистемы памяти с использованием асинхронной предварительной подкачки данных. *Информационные технологии*, номер 5, 2007.

[15] Н. В. Воронов, Р. А. Савченко: Использование шаблонного транслятора в системе двоичной трансляции. *V Международная научно-практическая конференция «Современные информационные технологии и ИТ-образование»*, сборник избранных трудов, 2010, Р. 491-497.

[16] А. А. Рыбаков, М. В. Маслов: Быстрый регионный компилятор системы двоичной трансляции для архитектуры «Эльбрус». *V Международная научно-практическая конференция «Современные информационные технологии и ИТ-образование»*, сборник избранных трудов, 2010, Р. 436-443.

[17] В. Ю. Волконский, В. Д. Гимпельсон: Методы определения порогов активизации динамического оптимизирующего транслятора. *Информационные технологии*, номер 4, 2007.

[18] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, Roger A. Bringmann: Effective Compiler Support for Predicated Execution Using the Hyperblock. *in Proceedings of the 25th International Symposium on Microarchitecture*, Dec. 1992, P. 45-54.

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

[20] H. Zhou, M. D. Jennings, T. M. Conte: Tree Traversal Scheduling: A Global Scheduling Technique for VLIW/EPIC Processors. *The 14th Annual Workshop on Languages and Compilers for Parallel Computing (LCP'01)*, LNCS 2624, Springer Verlag, Aug. 2001, P. 223-238.

[21] Standard Performance Evaluation Corporation, www.spec.org