Н.В. Воронов, В.Д. Гимпельсон, М.В. Маслов, А.А. Рыбаков, Н.С. Сюсюкалов
(ЗАО «МЦСТ»)

Nikita Voronov, Vadim Gimpelson, Maxim Maslov, Aleksey Rybakov, Nikita Syusyukalov

СИСТЕМА ДИНАМИЧЕСКОЙ ДВОИЧНОЙ ТРАНСЛЯЦИИ X86 → «ЭЛЬБРУС»

DYNAMIC BINARY TRANSLATION SYSTEM X86 → «ELBRUS»

Дается описание системы динамической трансляции двоичных кодов архитектуры x86 в архитектуру «Эльбрус». Рассматривается общая схема работы двоичного транслятора, многоуровневая система оптимизаций, технологии сокращения накладных расходов на трансляцию (долговременное хранение кодов и параллельная трансляция). Приводится сравнение производительности с несколькими x86 микропроцессорами.

Ключевые слова: двоичная трансляция, виртуальная машина, микропроцессор Эльбрус.

The article describes dynamic binary translation system developed fortranslation of x86 binary codes to Elbrus architecture. We consider general principles of binary translation, describe our multi-level optimization engine and translation overhead decreasing techniques (long-time translation storage and parallel translation). Finally we investigate performance of Elbrus processor running binary translation system and compare it against several x86 microprocessors.

Keywords: binary translation, virtual machine, co-desing virtual machine, Elbrus microprocessor.

Введение

При создании микропроцессорных архитектур неизменно актуальной является проблема переноса большого количества программного обеспечения, разработанного для

уже выпускаемых микропроцессоров, на новую архитектурную платформу – необходимо либо портировать его, либо создавать заново, что, как правило, нереально. Если же попытаться обеспечить совместимость создаваемой архитектуры с уже существующими, то ее возможности по части внедрения новых идей будут весьма ограничены. Хорошо себя зарекомендовавшим способом решения задачи переноса программного обеспечения на новые архитектуры является технология динамической двоичной трансляции. С ее помощью была реализована совместимость с наиболее распространенной сейчас архитектурой Intel x86 следующих платформ: Itanium фирмы Intel (с помощью программного продукта IA-32 Execution Layer [1]), «Crusoe» и «Efficeon» фирмы Transmeta (с помощью Code Morphing Software, CMS [2]), PowerPC фирмы IBM (с помощью PowerVM Lx86 [3]).

Статья посвящена системе динамической двоичной трансляции из кодов архитектуры x86 в коды архитектуры «Эльбрус» (e2к) [4, 5], описаны общая схема ее работы, различные уровни оптимизаций, реализованных в системе, методы уменьшения накладных расходов на трансляцию. Приведены результаты сравнения производительности микропроцессора «Эльбрус», работающего под управлением системы двоичной трансляции x86 → «Эльбрус», с несколькими микропроцессорами архитектуры x86.

1. Общая структура системы двоичной трансляции

1.1. Общая схема работы

Схема работы системы двоичной трансляции x86 → «Эльбрус» изображена на рис. 1. Исполнение x86 кода начинается в интерпретаторе, который собирает профильную информацию об исполненных инструкциях и в случае превышения порогового количества исполнений запускает трансляция этого x86-кода в коды «Эльбруса». Транслированный код сохраняется в *кэш трансляций*. Во время работы оттранслированного кода ведётся статистика исполнения линейных участков x86-кода, на базе которой строится *профильный граф*. В узле профильного графа хранятся: количество исполнений соответствующего

линейного участка, статистика по переходам (счётчики дуг), информация о специфике инструкций, включенных в данный узел (наличие fp, mmx, sse операций и т.д.). При превышении порогового количества исполнений линейного участка x86-кода, запускается *наборщик регионов*, который выделяет область горячего кода для трансляции оптимизирующими компиляторами, называемую *регионом.*

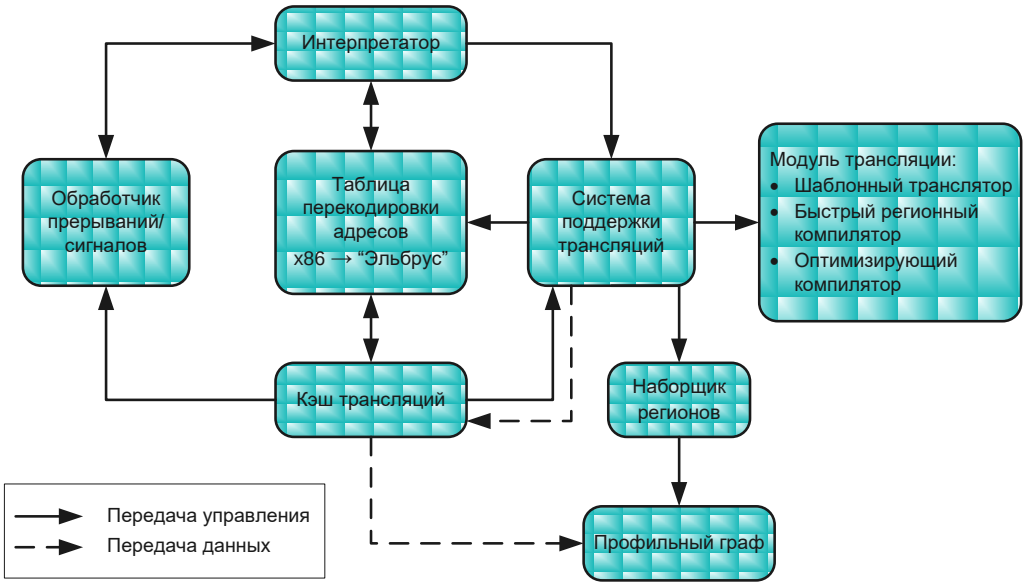

Рис. 1

Общая схема работы системы двоичной трансляции

Для осуществления переходов по x86-адресам между трансляциями используется *таблица перекодировки адресов*, которая хранит соответствие <x86-адрес начала линейного участка → e2k-адрес транслированного кода>. При выходе из трансляции и попытке перейти на определённый x86-адрес проверяется, нет ли в таблице перекодировки адресов соответствующей записи. В случае положительного результата, происходит переход на найдённый вход в трансляцию, иначе запускается интерпретатор.

Система поддержки трансляций обеспечивает взаимодействие *модуля трансляции* с остальной системой: запускает наборщик регионов, выбирает уровень трансляции, размещает полученную трансляцию в кэше трансляций, регистрирует глобальные входы в трансляцию в таблице перекодировки адресов.

Работа системы двоичной трансляции может прерываться приходом прерываний (для транслятора уровня всей системы) или сигналов (для транслятора уровня приложений). Обработка этих событий происходит в *обработчике прерываний/сигналов*.

1.2. Трансляторы уровня всей системы и уровня приложений

Были реализованы два подхода к построению системы двоичной трансляции, различающиеся своей применимостью.

Первый подход – система полной двоичной трансляции. Здесь транслятор работает между микропроцессором и запускаемыми на нём x86-кодами. Транслируются коды BIOS, операционной системы, драйверов и прикладных программ. Вычислительный комплекс на базе микропроцессора «Эльбрус» с системой полной двоичной трансляции для пользователя неотличим от вычислительного комплекса на базе x86-микропроцессоров. На рис. 2 изображена схема работы системы полной двоичной трансляции.

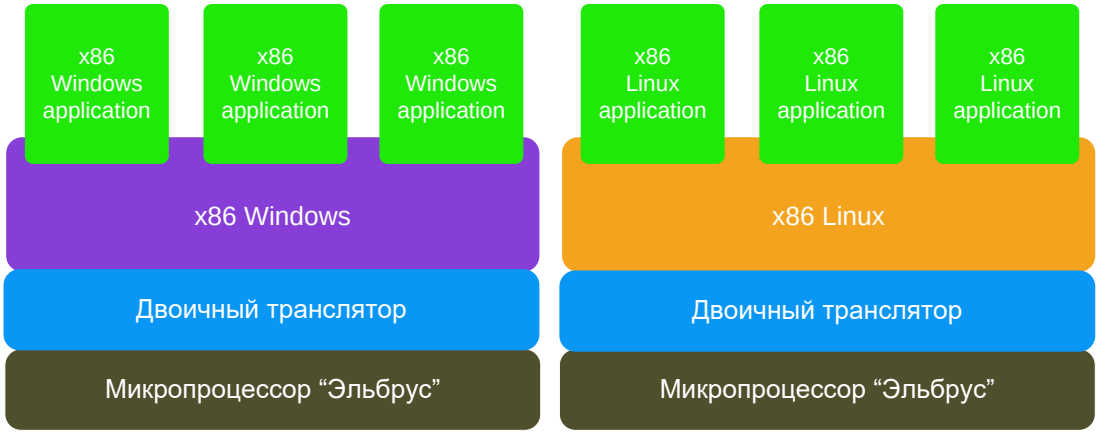

Рис. 2

Схема работы системы полной двоичной трансляции

При втором подходе система двоичной трансляции является обычным Linux-приложением и работает под управлением ОС Linux. Она позволяет запускать Linux-приложения для платформы x86, которые могут работать одновременно с приложениями в кодах платформы «Эльбрус». На рис. 3 изображена схема работы системы двоичной трансляции Linux-приложений.

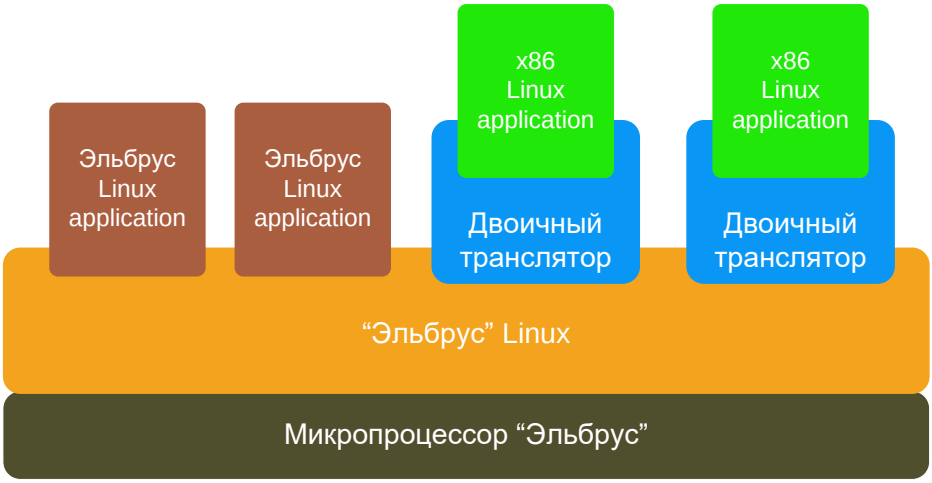

Рис. 3

Схема работы системы двоичной трансляции Linux приложений

2. Многоуровневая система оптимизаций

2.1. Мотивация

Интерпретация не позволяет исполнять код достаточно быстро, поэтому имеет смысл транслировать исходные коды и сохранять их для дальнейшего использования. Для достижения приемлемой скорости выполнения приложений необходимо оптимизировать получаемый код. С другой стороны, трансляция сама по себе занимает значительное время. Рис. 4 иллюстрирует суммарное время, затраченное на исполнение инструкций, выполнившихся меньше заданного количества раз, для приложения Acrobat Reader в двух случаях: исполнение только интерпретатором и исполнение, когда коды сразу транслируются оптимизирующим компилятором. Как видно, накладные расходы на оптимизацию очень велики. Несмотря на то, что оптимизированный код выполняется во много раз быстрее, итоговое время в четыре раза больше времени исполнения в интерпретаторе.

Таким образом, необходимо выбрать некоторый порог количества исполнений инструкций в интерпретаторе, после которого исходный код транслируется оптимизирующим компилятором. Из графика, изображенного на рис. 5, видно, что такая двухуровневая схема позволяет достичь существенного уменьшения времени работы.

В системе двоичной трансляции x86 → «Эльбрус» реализована многоуровневая

система оптимизаций. В состав двоичного транслятора включены интерпретатор и три транслятора с различными уровнями оптимизации. Каждый следующий уровень генерирует более эффективный результирующий код. Итоговое количество исполнений участка кода невозможно определить заранее, т.к. это число зависит от логики работы исполняемой программы и исходных данных. Поэтому сначала код транслируется без оптимизаций, а затем, исходя из профильной информации, код перетранслируется более высокими уровнями оптимизаций.

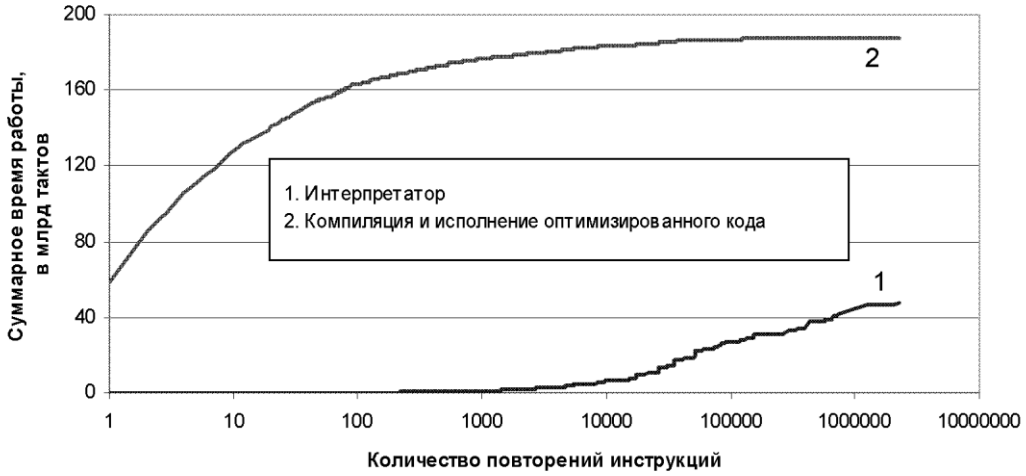

Рис. 4

Сравнение времени работы интерпретатора и оптимизированных кодов (включая время работы оптимизирующего транслятора) на приложении Acrobat Reader

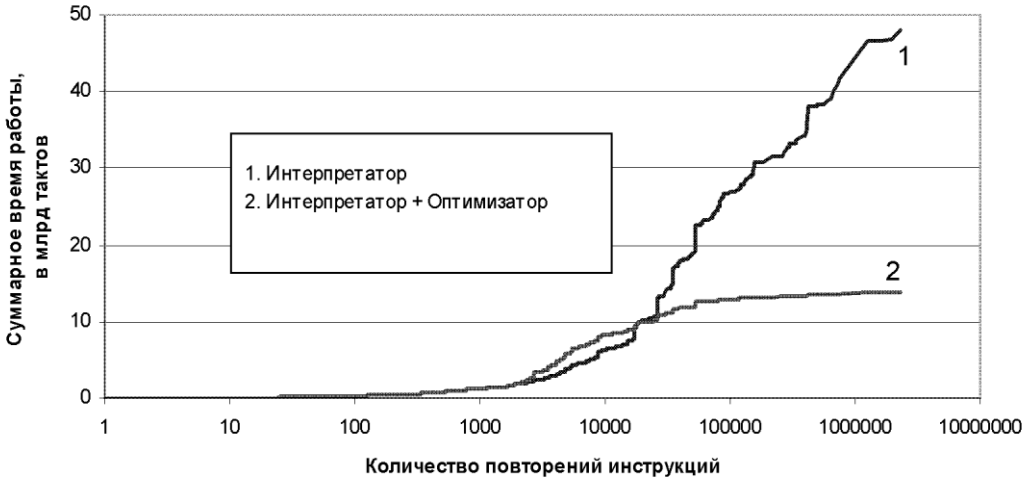

Рис. 5

Сравнение времени работы интерпретатора и двухуровневой схемы интерпретатор – оптимизирующий транслятор (на приложении Acrobat Reader)

В результате использования четырехуровневой схемы достигается заметно более высокая скорость выполнения приложения (рис. 6). Стоит отметить, что, как следует из рис. 7, увеличение скорости работы кодов промежуточных уровней оптимизации заметно влияет на общее время работы приложения. В последующих разделах описаны интерпретатор и три модуля трансляции, реализованные в системе.

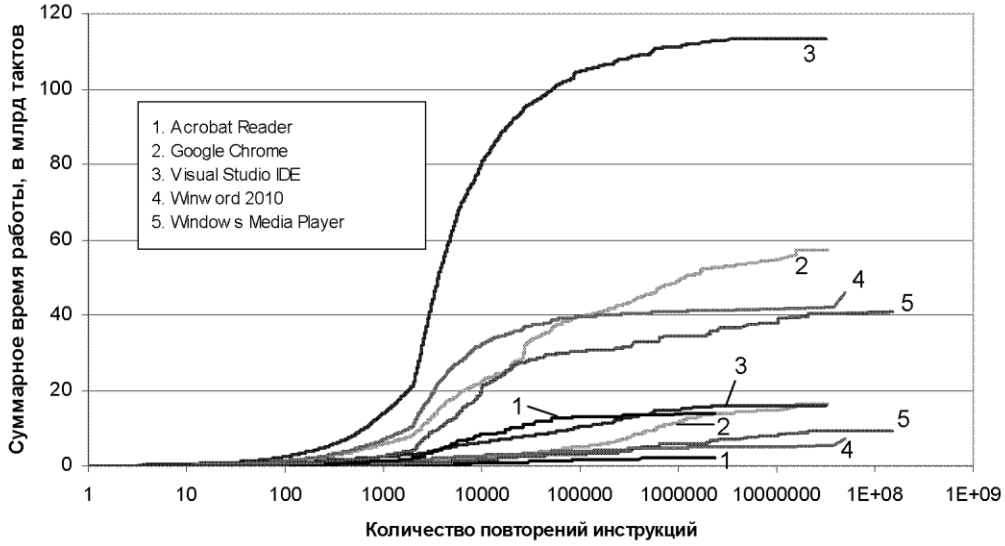

Рис. 6

Сравнение двухуровневой и четырехуровневой схем работы транслятора (последняя всегда быстрее)

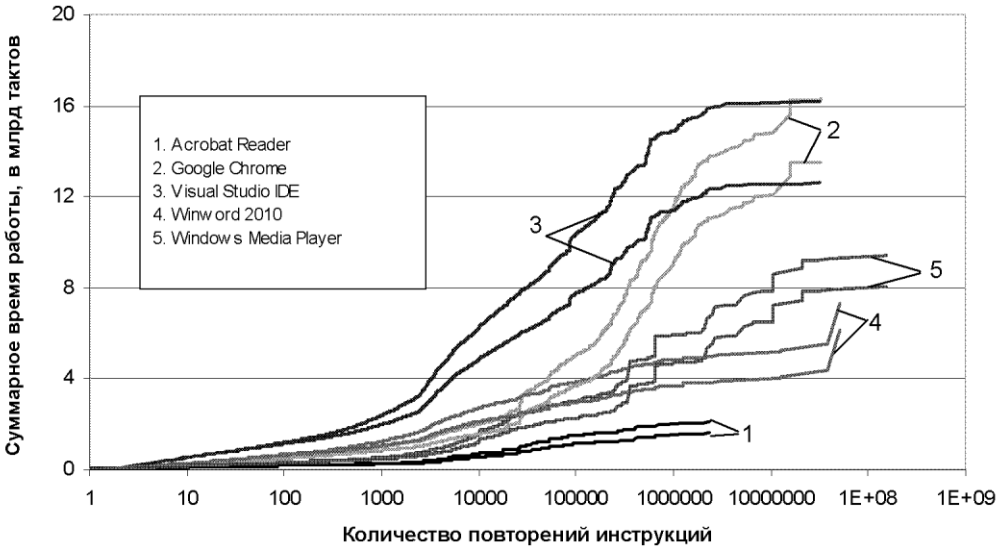

Рис. 7

Ускорение приложения за счет улучшения характеристик трансляторов промежуточных уровней оптимизации

2.2. Интерпретатор

Интерпретатор, по сути, не будучи транслятором, является базовым уровнем многоуровневой системы двоичной трансляции. Он последовательно декодирует инструкции x86 и на основе извлечённой информации вызывает функцию, выполняющую над контекстом (регистры, память и т.д.) такое же преобразование, что и исходная инструкция, затем исполняется следующая инструкция. Состояние контекста перед выполнением каждой x86 инструкции является таким же, как и в исходной архитектуре, причем как внутри функций изменения контекста, так и между исполнением инструкций, интерпретатор проявляет все необходимые исключительные ситуации.

Интерпретатор не генерирует двоичной код целевой архитектуры, поэтому, даже если транслируемый код передаст управление на уже исполненный ранее интерпретатором участок кода, то декодирование и исполнение этих инструкций начнётся заново. В связи с этим интерпретация кода в десятки раз медленнее, чем исполнение оттранслированного кода, но, тем не менее, она существенно быстрее трансляции без применения оптимизаций. Поэтому использование интерпретатора оправдано для инструкций, которые исполняются небольшое число раз.

Помимо эмуляции x86-инструкций интерпретатор набирает *трассу* – последовательность исполненных им линейных участков, идущих в порядке возрастания адресов. Для каждой трассы запоминаются адрес её начала и количество вхождений. Вхождением считается передача управления x86-программы на адрес начала трассы (т.е. переход на адрес внутри трассы вне процедуры набора вызовет создание новой трассы). После того как количество исполнений трассы превышает некоторый порог, запускается процедура трансляции данной трассы неоптимизирующим двоичным транслятором – *шаблонным транслятором*.

2.3. Шаблонный транслятор

Шаблонный транслятор [6] принимает на вход трассу, полученную интерпретатором, и транслирует каждую инструкцию трассы в двоичный код целевой архитектуры, который выполняет эквивалентное преобразование контекста и проявляет исключительные ситуации. Оттранслированным кодом можно пользоваться как новой функцией, вызов которой эквивалентен интерпретации трассы. Результирующий код помещается в *кэш трансляций*, и в случае, когда управление транслируемого x86-приложения попадёт на начало любого из линейных участков трассы, управление будет передано на начало соответствующего ему оттранслированного кода.

Шаблонный транслятор обрабатывает каждый линейный участок отдельно, несмотря на то, что на вход ему подаётся трасса. Генерация кода участка происходит следующим образом: сначала генерируется *пролог*, затем отдельно транслируется каждая инструкция x86, и заканчивается код *эпилогом*. Если в блоке была инструкция перехода, то соответствующий ей код встраивается в эпилог, иначе он генерируется отдельно. В прологах и эпилогах увеличиваются счётчики профильного графа (доступ к ним из оттранслированного кода осуществляется непосредственно по адресу), делается проверка, не пора ли запустить оптимизирующий компилятор для данного кода, проверяется наличие отложенных прерываний.

Каждая инструкция транслируется отдельно и обычно набирается из нескольких готовых шаблонов (подготовка операндов, операция, запись результатов). Каждый шаблон представляет собой двоичный код целевой архитектуры с некоторыми пустыми местами – параметрами шаблона. Это могут быть номера регистров, константы, смещения. Для каждой команды в процессе работы определяется, какие параметры нужно вставить в шаблон. Временные регистры, хотя и являются параметрами шаблонов, в рамках одной инструкции распределены статически.

Задача двоичного транслятора усложняется необходимостью корректно поддерживать системный код, в т.ч. выдавать контекст таким же, каким он был в микропроцессоре

исходной архитектуры. Для повышения производительности проявление многих прерываний (ошибка страницы, выход за границы сегмента и др.) выполняются на аппаратном уровне, а двоичный транслятор следит только за корректностью контекста. Все вычисления в оттранслированном шаблонном коде делаются на временных регистрах. Затем производится попытка проявить исключения (например, если команда пишет в память, то оттранслированный код пишет в память на данном шаге) в том же самом порядке, в котором их проявил бы процессор исходной архитектуры, и только после этого обновляются глобальные регистры (например, Instruction Pointer).

Шаблонный транслятор является важным звеном для обеспечения семантической целостности всей системы в силу того, что в оптимизирующих компиляторах некоторые инструкции не реализованы. Это обусловлено сложностью семантики (оптимизации над ней не приносят какого-либо ощутимого эффекта) или тем, что не выдерживаются предположения, позволяющие произвести необходимые оптимизации. Трансляция базовых блоков, содержащих подобные инструкции, осуществляется шаблонным транслятором.

Как уже было сказано, в шаблонном трансляторе производится подсчёт количества исполнений линейных участков, и в случае превышения им некоторого порога управление передается на *наборщик регионов*. Задачей наборщика регионов является набор региона – подграфа профильного графа, который будет передан для трансляции быстрому региональному компилятору. Т.к. регион, в отличие от трассы, может иметь произвольную конфигурацию, этот компилятор имеет возможность для оптимизаций между линейными участками, что даёт выигрыш в качестве результирующего кода. Набор региона осуществляется следующим образом. От узла, на котором сработала проверка на превышение порога, во все стороны по управлению производится добавление линейных участков. Очередной линейный участок выбирается исходя из его количества исполнений – первыми добавляются участки с максимальным счётчиком. Набор регионов заканчивается в том случае, когда превышается количество допустимых инструкций в регионе, либо когда все участки, кан-

дидаты на добавление, имеют недостаточно большой счётчик.

2.4. Быстрый регионный компилятор

Функциональность

Быстрый компилятор [7] предназначен для оптимизации недостаточно горячих регионов. Он включает в себя урезанный набор базовых оптимизаций, которые позволяют достичь существенного прироста производительности результирующего кода при наименьших затратах по времени компиляции.

При реализации функционала быстрого компилятора не обязательно использовать оптимальные алгоритмы. Зачастую применяются достаточно грубые консервативные алгоритмы и эвристики, позволяющие применять оптимизации в общем виде, без обработки редких частных случаев. Кроме того, можно использовать облегченные структуры данных, на поддержке которых дополнительно экономится время компиляции. Такой компромисс между скоростью компиляции региона и качеством результирующего кода позволяет достичь оптимального суммарного времени работы всей системы двоичной трансляции.

Из-за ограничений по времени компиляции в быстром компиляторе не используются целые классы оптимизаций. Например, отсутствуют оптимизации циклов, т.к. быстрый компилятор ориентирован, в первую очередь, на работу с регионами, состоящими из относительно небольших линейных участков с обилием команд передачи управления. Существенно ограничены оптимизации, связанные с преобразованиями, выходящими за пределы одного линейного участка, т.к. это приводит к необходимости поддержки глобальных (в масштабе региона) структур данных. В течение всего процесса оптимизации региона вплоть до этапа планирования в рамках каждого линейного участка сохраняется жесткая линейка последовательности операций, в которой в общем случае изменение порядка операций недопустимо.

Процесс компиляции региона быстрым компилятором условно можно разделить на три этапа. На первом из них, *генерации семантики,* осуществляется перевод кода x86 в промежуточное представление компилятора. При этом каждая инструкция x86 переводится в соответствующую ей последовательность команд промежуточного представления компилятора. Одновременно с генерацией семантики выполняется множество мелких преобразований, не требующих сложного анализа. Второй этап заключается в последовательном применении к сгенерированному промежуточному представлению основных *оптимизаций*. На третьем этапе, *планировании кода*, осуществляется распределение регистров, планирование широких команд и генерация результирующего кода. Также во время планирования реализованы некоторые оптимизации, требующие для своей работы информацию о временах планирования операций.

Оптимизации, обеспечивающие основной прирост производительности

Оптимизации описаны в той последовательности, в которой они применяются.

На этапе генерации семантики используются специальные структуры виртуальных объектов, с помощью которых проводится оптимизация *удаления избыточных вычислений (peephole)*. На самом деле, это не отдельная оптимизация, а набор, состоящий из множества простых частных правил, которые применяются в рамках генерации команд промежуточного представления. К удалению избыточных вычислений можно отнести такие преобразования как вычисление константных выражений, сбор общих подвыражений, удаление дублирующих операций, применение математических тождеств и др.

Слияние узлов (if-conversion) является одним из важнейших преобразований, при котором могут объединяться различные линейные участки. Изначально каждый линейный участок промежуточного представления имеет один или два выхода. Оптимизация находит такие последовательности узлов, в которых каждый узел, кроме первого, имеет ровно один вход, причем предшественником по входящей дуге является предыдущей узел последовательности. Такой подграф называется *суперблоком*. Суперблок можно объединить

в один узел с использованием предикатного исполнения. Суперблоки выбираются таким образом, чтобы вероятность перехода от любого линейного участка суперблока на следующий была выше вероятности выхода из суперблока. Таким образом, осуществляется слияние наиболее вероятных путей исполнения.

Еще одним преобразованием, выходящим за рамки одного линейного участка, является перенос операций, стоящих на критическом пути, между узлами *(code motion)*. Его идея заключается в том, чтобы перенести критическую операцию, стоящую в начале узла, вверх по входным дугам во все узлы-предшественники. Для того чтобы определить, стоит ли операция на критическом пути, требуется выполнить предварительное планирование команд. Т.к. это слишком дорогая процедура, используется эвристическое решение, заключающееся в том, что некоторые операции (например, операции чтения из памяти), стоящие в начале узла, всегда считаются критическими. Проблемой является и то, что нельзя знать наверняка, что перенесенная в другой узел операция не приведет к негативному эффекту. Для принятия решения о переносе используется другая эвристика, запрещающая перенос операций в узлы, частота исполнения которых намного больше частоты исполнения исходного узла.

Одной из важных особенностей архитектуры «Эльбрус» является явный параллелизм на уровне команд. Широкая команда «Эльбрус» позволяет исполнять до шести арифметических операций за один такт, что существенно повышает производительность кода. Компоновка операций промежуточного представления в широкие команды осуществляется с помощью механизма *планирования*. На этом этапе впервые происходит построение полного графа зависимостей в рамках линейного участка, а также на основе информации о задержках между операциями происходит вычисление времен раннего и позднего планирования операций. Это позволяет выделить цепочки критических операций, планирование которых должно происходить с максимальным приоритетом для повышения производительности результирующего кода.

На этапе планирования осуществляется и ряд других оптимизаций. *Удаление избыточных операций чтения (redundant loads elimination)* позволяет удалить операцию чтения из памяти, если удалось доказать, что в данном линейном участке уже встречалась команда обращения в память по тому же адресу и того же формата, и после нее не было конфликтующей операции записи в память. *Удаление мертвого кода (dead code elimination)* позволяет удалять операции, у которых нет использований. *Динамический разрыв зависимостей по доступу в память (memory access disambiguation)* позволяет поднимать операцию чтения выше операции записи, даже если не удалось определить, конфликтуют ли эти операции. Мониторинг конфликта по доступу в память между двумя этими операциями выполняется аппаратно, и в случае его возникновения чтение из памяти повторяется.

В результирующий код, создаваемый быстрым компилятором, также как и в шаблонный код, добавляются профилирующие вставки, собирающие информацию о частоте исполнения линейных участков. Если частота исполнения превышает определенный предел, то запускается наборщик регионов. Набранный регион транслируется оптимизирующим регионным компилятором. Наборщик регионов для оптимизирующего компилятора в целом похож на наборщик регионов для быстрого регионного компилятора. Однако имеются и отличия – регион может иметь больший размер; наборщик пытается сделать регион более структурированным: если имеется некоторая процедура в исходном коде, то делается попытка полностью включить её в регион, т.к. это даёт больше возможностей для оптимизаций.

2.5. Оптимизирующий регионный компилятор

Оптимизирующий регионный компилятор является транслятором самого высокого уровня. В результате его работы получается максимально эффективный результирующий код, хотя при этом затрачивается много времени на саму трансляцию. По своему внутрен-

нему устройству этот компилятор похож на классические языковые оптимизирующие компиляторы. Основными отличиями являются, во-первых, жесткие ограничения на скорость компиляции, поэтому самые затратные по времени работы алгоритмы либо урезаны, либо не используются; во-вторых, при двоичной трансляции появляются новые семантические ограничения, накладываемые семантикой исходных двоичных кодов. Примером такого семантического ограничения является задача обеспечения точного контекста x86-ой машины при возникновении прерываний и исключений.

Подробное описание двоичного оптимизирующего компилятора слишком объёмно и выходит за рамки этой статьи. Здесь мы ограничимся лишь перечислением наиболее важных оптимизаций и ссылками на более подробную информацию.

Основными оптимизациями, проводимыми оптимизирующим регионным компилятором, являются: конвейеризация циклов [8], if-conversion [10], глобальное планирование, сокращение длины критических путей [9], удаление избыточных обращений в память [11], различные цикловые оптимизации (unroll, peeling, nesting, tail duplication), operation strength reduction, оптимизации, основанные на тождественных преобразованиях, prefetch. Кроме того, важную роль играют распределение регистров и планирование результирующего кода.

2.6. Механизм связывания трансляций

Рассмотрим ещё одну очень важную оптимизацию, которая может применяться к любым оттранслированным кодам и, по сути, является оптимизацией между различными трансляциями. На момент трансляции, даже для переходов по константному адресу, в общем случае неизвестно, по какому адресу будет располагаться оттранслированный код, соответствующий цели данного перехода. Поэтому адреса в оттранслированном коде, где прописаны смещения переходов, и соответствующие им x86-адреса сохраняются в отдельной таблице, записи которой называют *релокациями*. Естественно, оттранслирован-

ный код нельзя исполнять, пока релокации не будут связаны. Связыванием называется процесс, в котором по таблице релокаций в целевой код прописываются работающие смещения. Изначально в инструкции переходов прописывается смещение на функцию поиска по таблице перекодировки адресов. Однако, во многих случаях адрес перехода, которым заканчивается базовый блок, является статическим и не меняется в процессе исполнения программы. Для устранения избыточных поисков по таблице при последующей трансляции соответствующих участков кода в переходы перезаписываются смещения следующего оттранслированного кода. Такая оптимизация называется *связыванием блоков*.

2.7. Показатели производительности и времени трансляции для различных уровней системы

Для того чтобы дать представление о том, как меняется качество результирующего кода, и как соотносятся между собой времена компиляции для различных уровней, ниже приводятся результаты измерений этих величин. Измерения производились на нескольких задачах из пакетов SPEC 95 [12] и SPEC 2000 [13] с уменьшенными входными данными.

Для измерения качества результирующего кода задачи статически транслировались каждым уровнем в отдельности. Таким образом, каждый запуск отражал работу результирующего кода только для одного уровня трансляции, что позволило произвести их сравнение. Измерения проводились на потактовом симуляторе микропроцессора «Эльбрус-S». В табл. 1 приведено сравнение производительности полученных целевых кодов для каждого уровня, нормированное относительно оптимизирующего регионного компилятора. В табл. 2 приведены времена трансляции одной инструкции различными уровнями трансляции.

Таблица 1

Сравнение производительности целевых кодов

Задача	**Шаблонный код**	**Быстрый регионный компилятор**	**Оптимизирующий регионный компилятор**
099.go_slice	0,35	0,68	1,00
126.gcc_lpriotoize	0,32	0,64	1,00
186.crafty.p4	0,26	0,67	1,00
255.vortex.p4	0,15	0,42	1,00

Таблица 2

Среднее время, затрачиваемое на трансляцию одной x86 инструкции различными уровнями трансляции

	Время компиляции (тактов на одну x86 инструкцию)
Интерпретатор[1]	100
Шаблонный транслятор	650
Оптимизатор 3-го уровня	15500
Оптимизатор 4-го уровня	500000

3. Методы уменьшения накладных расходов на трансляцию

Несмотря на использование нескольких уровней, на трансляцию кодов тратится существенное время. Рассмотрим методы, позволяющие его сократить.

3.1. База кодов

Первым способом уменьшения накладных расходов является переиспользование ранее транслированных кодов. Для этой цели в системе двоичной трансляции предусматривается возможность долговременного хранения транслированного кода в энергонезависимой памяти (например, жёсткий диск) – *базе кодов* – и его переиспользования после перезагрузки системы [14].

Транслированный оптимизирующим регионным компилятором код сохраняется в базе кодов. Кроме этого, в базу записывается также соответствующий x86-код. Для обес-

[1] Для интерпретатора указано время на интерпретацию одной инструкции.

печения быстрого поиска регионов в базе кодов используется хэш-таблица.

Перед трансляцией очередного региона оптимизирующим компилятором производится его поиск с помощью хэш-таблицы базы кодов. Если регион найден, производится дополнительная проверка – сравнение соответствующего x86-кода в оперативной памяти и в базе кодов. При совпадении трансляция не производится, а регион из базы кодов помещается в кэш регионов для дальнейшего использования.

3.2. Трансляция в параллельном потоке

Другим способом сокращения накладных расходов на трансляцию является выполнение трансляции оптимизирующим компилятором на отдельном процессоре [15]. В этом случае компиляция проводится одновременно с выполнением кодов. В такой схеме после отправки транслирующему процессору информации о компилируемом участке кода выполнение этого кода может быть продолжено с текущим уровнем оптимизаций. Как только компиляция заканчивается, транслирующий процессор сообщает об этом выполняющему процессору, который заменяет предыдущий, менее оптимизированный вариант кода, только что скомпилированным вариантом.

Выполнение трансляции на отдельном процессоре позволило ускорить время первого запуска приложений (кэш трансляций пустой) из пакета SPEC2000 в среднем на 6% [14].

4. Экспериментальные результаты

В заключение приведём результаты сравнения производительности системы полной двоичной трансляции, работающей на микропроцессоре «Эльбрус-S» (частота 500 МГц) с двумя x86-микропроцессорами: Pentium-M (частота 1000 МГц) и Atom D510 (частота 1660 МГц). Сравнение проводилось на пакете тестов SPEC 2000. На рис. 8 и 9 приведены результаты целочисленных и вещественных задач, соответственно.

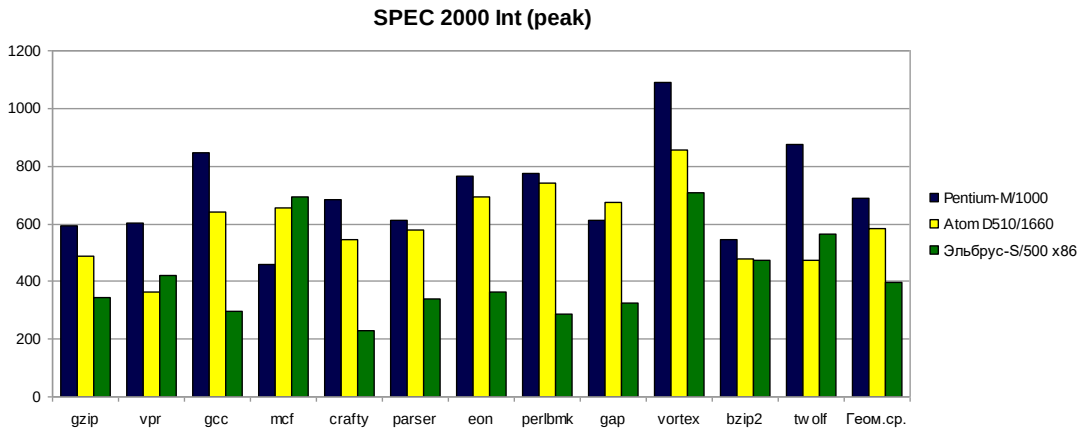

Рис. 8

Результаты сравнения производительности на пакете SPEC 2000 Int

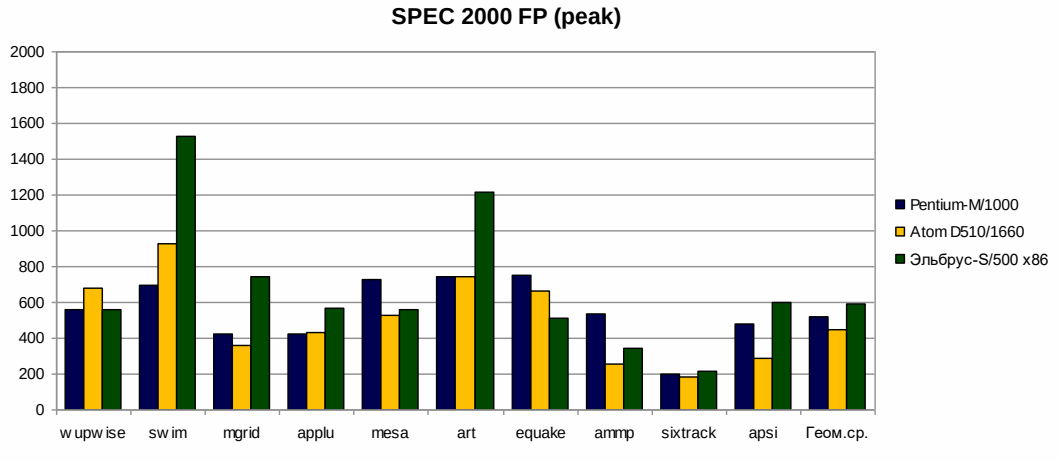

Рис. 9

Результаты сравнения производительности на пакете SPEC 2000 FP

Данные для микропроцессора Pentium-M были взяты с официального сайта SPEC. Результаты на микропроцессорах Atom и «Эльбрус» получены авторами, при этом для обоих измерений брались одинаковые коды. Система двоичной трансляции x86 → «Эльбрус» работала со всеми описанными в данной статье технологиями и была собрана оптимизирующим языковым компилятором с высоким уровнем оптимизаций.

Литература

1. 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.

2. 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.

3. http://www.ibm.com/developerworks/linux/lx86/

4. Волконский В.Ю. Оптимизирующие компиляторы для архитектур с явным параллелизмом команд и аппаратной поддержкой двоичной совместимости. – «Информационные технологии и вычислительные системы», 2004, №3.

5. 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.

6. Воронов Н.В., Савченко Р.А. Использование шаблонного транслятора в системе двоичной трансляции // Научные труды V Международной научно-практической конференции «Современные информационные технологии и ИТ-образование», 2010.

7. Рыбаков А.А., Маслов М.В. Быстрый регионный компилятор системы двоичной трансляции для архитектуры «Эльбрус» // Научные труды V Международной научно-практической конференции «Современные информационные технологии и ИТ-образование», 2010.

8. Гимпельсон В.Д. Конвейеризация циклов в двоичном динамическом трансляторе. – «Вопросы радиоэлектроники», серия ЭВТ, 2009, вып. 3.

9. 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», М., МАТИ, 2008.

10. Дроздов А.Ю., Новиков С.В., Шилов В.В. Эффективный алгоритм преобразования потока управления в поток данных. – «Информационные технологии», Приложение, 2005, №2, с. 24-31.

11. 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. М., в/ч 03425, 2003.

12. SPEC CPU 95 Benchmark. www.spec.org.

13. SPEC CPU 2000 Benchmark. www.spec.org, 2000.

14. Roman Sokolov. Advancing Persistent Binary Translated Code Caching, in Proceedings of the XLVIII Annual MIPT Scientific Conference. Dolgoprudny, Russia. November 2005, vol.1, p. 51-52.

15. Roman A. Sokolov, Alexander V. Ermolovich Background Optimization in Full System Binary Translation // Spring/Summer Young Researchers' Colloquium on Software Engineering, 2011.