Быстрый регионный компилятор системы двоичной трансляции для архитектуры «Эльбрус»

А. А. Рыбаков, М. В. Маслов

Введение

Технология динамической двоичной трансляции обеспечивает полную совместимость архитектуры «Эльбрус» [1, 2] (называемой в дальнейшем исходной) с архитектурой Intel x86 [4] (называемой в дальнейшем целевой). 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». При этом работа LIntel по трансляции программного обеспечения различных уровней: x86bios, операционной системы, пользовательских приложений, - выполняется незаметно для пользователя.

Основными составляющими LIntel являются интерпретатор, многоуровневый двоичный 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. При этом после каждой исполненной инструкции x86 гарантируется корректное состояние интеловского *контекста*, включающего в себя регистры общего назначения, системные регистры, режим работы процессора, память. Интерпретатор используется, когда требуется моделировать поведение процессора при возникновении исключения или при первом исполнении кода. При этом интерпретатор отличается невысокой скоростью работы. 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*. Далее следует быстрый оптимизирующий компилятор (далее просто *быстрый* компилятор), который применяет к созданному коду некоторый базовый набор оптимизаций (уровень оптимизации O0). Качество кода возрастает, но также возрастает и время трансляции. И, наконец, оптимизирующий компилятор, выполняющий полный набор оптимизаций (далее просто *оптимизирующий* компилятор, или компилятор уровня оптимизации

O1), создавая наиболее эффективный код, но еще больше времени затрачивая на трансляцию (рисунок 1). Из соображений минимизации времени работы системы в целом нужно придерживаться правила – чем больше раз исполняется код, тем транслятор более высокого уровня целесообразно использовать [5]. Во время работы системы переключение между уровнями трансляции происходит динамически.

Рисунок 1. Уровни трансляции

Шаблонный транслятор обрабатывает линейные участки, заменяя инструкции x86 на соответствующие фиксированные наборы команд «Эльбрус» (*шаблоны*). Из последовательности инструкций получается двоично-транслированный код, который сохраняется в *кэше кода* и может быть использован повторно. Шаблонный транслятор не выполняет никаких оптимизаций, и его использование оправдано для линейных участков исходного кода с небольшим числом повторений. Параллельно с исполнением оттранслированного кода проводится мониторинг, направленный на выявление наиболее часто повторяющихся фрагментов кода. По мере превышения отдельными фрагментами кода некоторого предопределенного значения порога повторений на основе этих фрагментов строится *регион* с графом управления, который передается для оптимизации двоичному транслятору следующего уровня – быстрому компилятору.

Быстрый компилятор работает с регионом, применяя базовый набор

оптимизаций, которые при наименьших затратах по времени компиляции дают максимальный прирост производительности результирующего кода. Также в результирующий код внедряются вспомогательные команды, предназначенные для сбора профильной информации. При превышении определенного порога числа повторений кода начинается набор региона для оптимизирующего компилятора (уровень оптимизации O1). На вход оптимизирующему компилятору подается набранный регион вместе с профильной информацией. К этому региону применяется полный набор оптимизаций, учитывающих все возможности аппаратуры, для достижения максимальной производительности. Далее в этой статье речь пойдет о быстром регионом компиляторе.

Общие принципы функционирования быстрого компилятора

Единицей трансляции быстрого компилятора является регион, который представляет собой объединение некоторого числа линейных участков, связанных между собой переходами.

Процесс компиляции региона можно условно разделить на три этапа. На первом этапе в каждом узле графа управления осуществляется генерация *семантики* команд x86 данного линейного участка. При этом последовательность команд x86 переводится в последовательность команд промежуточного представления компилятора. Одновременно проводятся простейшие оптимизации, не требующие сложного анализа. Далее, опираясь на сгенерированное промежуточное представление, последовательно применяется ряд базовых оптимизаций, позволяющих повысить производительность результирующего кода. К наиболее важным оптимизациям относятся слияние суперблоков [12], перенос операций между узлами [6], устранение избыточных обращений в память [6], разрыв зависимостей по доступу к памяти [3, 7] и другие. После проведения всех оптимизаций осуществляется распределение регистров, планирование широких команд и генерация кода архитектуры «Эльбрус». При реализации функциональности быстрого регионного компилятора зачастую применяются загрубленные консервативные алгоритмы, которые не позволяют достигать максимальной производительности результирующего кода. Однако использование таких упрощенных алгоритмов приводит к существенному уменьшению времени, затрачиваемого на трансляцию, и возможности использования облегченных структур данных промежуточного представления, что приводит к уменьшению времени компиляции региона. Такой компромисс ведет к эффективному снижению суммарного времени работы системы двоичной трансляции

на недостаточно горячих регионах.

Ограничения на время трансляции

Для уменьшения времени компиляции региона в быстром компиляторе не используются целые классы оптимизаций. Существенно ограничивается применение оптимизаций, работающих за пределами одного линейного участка, так как это требует создания и поддержания глобальных (в масштабе региона) структур данных. Отдельно следует упомянуть о классе цикловых оптимизаций [6]. Архитектура «Эльбрус» включает в себя множество аппаратных решений, которые позволяют на полную мощность использовать явный параллелизм на уровне команд и достигать максимальной производительности при работе в циклах. К аппаратным возможностям, направленным на повышение производительности циклов, можно отнести механизм наложения итераций цикла, реализуемый с помощью использования вращающихся регистров [8, 13], или асинхронную подкачку данных [14]. Быстрый регионный компилятор не применяет цикловые оптимизации, так как он в большей мере ориентирован на обработку регионов, состоящих из относительно небольших линейных участков с обилием команд передачи управления.

Для определения порядка выполнения операций, а также величины задержек между ними, требуется построение зависимостей между операциями. Быстрый компилятор не применяет оптимизации, требующие для своей работы создания зависимостей между операциями, находящимися в разных узлах промежуточного представления, а также любые оптимизации, требующие перестроения уже созданных зависимостей. Полностью зависимости используются только для планирования широких команд и строятся непосредственно перед ним. До этого в рамках каждого линейного участка сохраняется жесткая линейка операций, в которой в общем случае изменение порядка следование операций недопустимо.

Основной набор оптимизаций

Далее рассматриваются оптимизации, используемые в быстром компиляторе, применение которых обеспечивает основную долю прироста производительности результирующего кода.

Упрощение арифметических вычислений

На этапе генерации семантики аргументы и результаты создаваемых операций помещаются в специальные структуры внутреннего представления, позволяющие проводить легкие упрощения и удаление избыточных вычислений (peephole) в рамках одного линейного участка. К таким преобразованиям относятся статическое вычисление

константных выражений, сбор общих подвыражений и уменьшение количества вычислений путем применения математических тождеств [6].

Слияние узлов

Изначально линейные участки промежуточного представления могут оканчиваться либо прямой передачей управления (что соответствует simple-узлу с одним выходом), либо переходом по условию (соответствует if-узлу с двумя выходами). Если узел заканчивается косвенным переходом, то он трактуется как простой узел с выходом из региона. Одной из наиболее важных оптимизаций является слияние последовательной цепочки узлов (*суперблока*) в один узел с множественными выходами (if-conv – if nodes conversion) [12]. Такое преобразование обеспечивается поддержкой *предикатных* и *спекулятивных* вычислений в архитектуре «Эльбрус» [10]. Суперблоком будем называть такую последовательность узлов, в которой каждый узел кроме первого имеет ровно одну входную дугу, и предшественником по этой дуге является предыдущий узел последовательности. Сливаемые цепочки узлов выбираются таким образом, чтобы для любых двух последовательных узлов вероятность перехода по дуге между ними была больше вероятности альтернативного перехода. Таким образом, обеспечивается слияние наиболее вероятных путей передачи управления в графе. Кроме того, есть другие небольшие технические ограничения на узлы суперблока (например, отсутствие операций, которые не могут быть поставлены под предикат). Если решение о слиянии принято, линейные участки суперблока объединяются в один линейный участок, в котором операции каждого узла ставятся под условие (предикат) достижимости из первого узла суперблока. Линейный участок, полученный в результате слияния, также будем называть суперблоком. Дуги, соответствующие переходам между узлами сливаемой цепочки, удаляются, в результате образуется один узел с несколькими выходами.

Перенос операций между узлами

Другим преобразованием, выходящим за рамки одного линейного участка, является перенос критических операций между узлами (code motion) [6]. Идея состоит в выносе критических операций, стоящих в начале узла, вверх по всем входящим дугам в предшественники данного узла. Чтобы точно определить, является ли операция критической, то есть она задерживает планирование идущих за ней операций, нужно проводить предварительное планирование, что сильно увеличивает время трансляции 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, счетчик которого существенно больше счетчика исходного узла, то от применения code motion к данной операции следует отказаться, так как это приведет к перемешиванию часто исполняемого кода с кодом, исполняемым гораздо реже.

Удаление избыточных операций чтения

Удаление избыточных операций чтения (rle – redundant loads elimination) [6] применяется в быстром компиляторе в упрощенной форме в рамках одного суперблока, если для двух операций обращения в память, вторая из которых – операция чтения, удается достоверно определить, что адреса этих двух обращений в память и форматы обращения совпадают. В данном случае, второе обращение (чтение) является избыточным, содержимое памяти уже известно.

Удаление мертвого кода

Непосредственно перед планированием широких команд производится построение зависимостей между операциями, что необходимо для планирования, а также происходит подсчет количества использования виртуальных регистров, что нужно для осуществления распределения виртуальных регистров на физические. Накопленная информация позволяет отследить операции, вырабатывающие результат в регистры, количество использований которых равно нулю. Такие операции являются мертвым кодом и удаляются (dce – dead code elimination) [6].

Динамическое разрешение конфликтов по доступу к памяти

Еще одной важной оптимизацией, производимой параллельно с планированием, является динамическое разрешение конфликтов по доступу к памяти (dam – memory access disambiguation), поддержанное аппаратно в архитектуре «Эльбрус» [7]. Данное преобразование позволяет поднимать операции чтения из памяти выше возможно конфликтующих с ними операциями записи в память, когда статически невозможно определить ни пересечение, ни независимость адресов обращения в память. При этом мониторинг возникновения конфликта осуществляется аппаратно. В случае возникновения конфликта по обращению в память, когда стоящая ниже операция записи изменяет содержимое памяти, считанное занесенной вверх операцией чтения, данное значение должно быть считано повторно. Для оценки эффективности этого преобразования необходимо наличие зависимостей между операциями и вычисленных задержек. По этим

задержкам сравнивается время получения результата рассматриваемой операции чтения с применением dam и без него.

Планирование

Решающую роль в достижении эффективности результирующего кода играет явный параллелизм архитектуры «Эльбрус» на уровне команд. Возможность исполнения в рамках одной широкой команды до шести арифметических инструкций за один такт открывает большие возможности для генерации высокопроизводительного кода. Задачу компоновки широких команд решает механизм планирования, совмещенный с распределением виртуальных регистров на физические. Важность качества планирования команд иллюстрируется тем фактом, что непосредственно перед ним в рамках линейного участка происходит единственное на всем протяжении процесса трансляции региона построение графа зависимостей между операциями. Построение всех зависимостей является тяжеловесной процедурой, занимающей около 30% времени трансляции, однако это предпочтительней, чем потери производительности кода, которые могли бы возникнуть вследствие загрубления алгоритма планирования. Данные о задержках между операциями, вычисляемые на основании зависимостей, позволяют выделять критические цепочки операций, планирование которых осуществляется с максимальным приоритетом. Это приводит к более плотному заполнению широких команд, то есть к более эффективному использованию вычислительных ресурсов.

Производительность

Наиболее важные оптимизации в быстром регионном компиляторе выполняются в следующей последовательности: peephole, if-conv, code motion, rle, dam. На потактном симуляторе микропроцессора «Эльбрус-3s», подключая последовательно по одной оптимизации из приведенного списка, получены сравнительные данные по времени трансляции и времени выполнения результирующего кода. В качестве тестового пакета использовались наиболее показательные процедуры, выделенные из задач тестовых пакетов Spec92 и Spec95 [11].

В колонке «скорость трансляции» для конкретной оптимизации указана разница (в процентах) между средним временем трансляции, полученным при включении рассматриваемой оптимизации и всех оптимизаций, стоящих до нее, и средним временем трансляции, полученным при отключении всех оптимизаций. Аналогично, в колонке «ускорение результирующего кода» указана разница между средним временем исполнения кода, полученным при включении всех оптимизаций до рассматриваемой включительно, и средним временем

исполнения результирующего кода, полученным при отключении всех оптимизаций.

Оптимизация	Скорость трансляции	Ускорение результирующего кода
peephole	+ 0.6 %	+ 1.0 %
if-conv	- 4.3 %	+ 25.4 %
code motion	- 7.8 %	+ 32.0 %
rle	- 7.9 %	+ 35.5 %
dam	- 8.3 %	+ 41.0 %

Таблица 1. Влияние отдельных оптимизаций на время трансляции и время работы результирующего кода.

Список литературы

1. K. Dieffendorf. The Russians Are Coming: Supercomputer Maker Elbrus Seeks to Join x86/IA-64 Melee. // Microprocessor report, vol. 11, num 2, Feb. 15, 1999.
2. Boris Babayan. E2K Technology and Implementation. // Parallel Processing: 6th International Volume 1900 January 2000. – Pages 18-21.
3. Волконский В. Ю., Гимпельсон В. Д., Масленников Д. М. Быстрый алгоритм минимизации высоты графа зависимостей. // Информационные технологии и вычислительные системы, номер 3, 2004.
4. Intel Corporation. Intel IA-32 Architecture Software Developer's Manual.
5. Волконский В. Ю., Гимпельсон В. Д. Методы определения порогов активизации динамического оптимизирующего транслятора. // Информационные технологии, номер 4, 2007.
6. S. Muchnick. Advanced Compiler Design and Implementation. // Morgan Kaufmann Publishers, 1997.
7. Волконский В. Ю. Оптимизирующие компиляторы для архитектуры с явным параллелизмом команд и аппаратной поддержкой двоичной совместимости. // Информационные технологии и вычислительные системы, номер 3, 2004.
8. Дроздов А. Ю., Степаненков А. М. Технология оптимизации цикловых участков процедур в компиляторах для архитектур с аппаратной поддержкой конвейеризации циклов. // Информационные технологии и вычислительные системы, номер 3, 2004.
9. Волконский В. Ю., Окунев С. К. Оптимизация критического пути на предикатном представлении программы. // Информационные технологии, номер 9. Москва, сентябрь 2003.
10. Волконский В. Ю., Окунев С. К. Предикатное представление как основа оптимизации программы для архитектур с явно выраженной параллельностью. // Информационные технологии, номер 4, апрель 2003.
11. http://www.specbench.org/
12. 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.
13. Гимпельсон В. Д. Конвейеризация циклов в двоичном динамическом

трансляторе. // Вопросы радиоэлектроники, выпуск 3, 2009.
14. Галазин А. Б., Грабежной А. В. Эффективное взаимодействие микропроцессора и подсистемы памяти с использованием асинхронной предварительной подкачки данных. // Информационные технологии, номер 5, 2007.