Использование шаблонного транслятора в системе двоичной трансляции

Н.В.Воронов, Р.А.Савченко

Введение

ЗАО «МЦСТ» осуществляет разработку микропроцессоров серии «Эльбрус» [7], [8], [9], архитектурно несовместимых с платформой x86 фирмы Intel. Поэтому для поддержки большого количества существующих приложений (распространяющихся в виде двоичного кода) возникает задача программной совместимости с x86. На сегодняшний день одним из наиболее эффективных решений данной проблемы является программное обеспечение, основанное на принципах *двоичной трансляции* [10]. Работа двоичного транслятора заключается в эквивалентном преобразовании двоичного кода *исходной* архитектуры в двоичный код *целевой*. Эквивалентность понимается в том смысле, что в памяти и на регистрах создаётся модель — *контекст* — эмулируемой архитектуры и эта модель (вместе с двоично-транслированным кодом) ведёт себя в точности как программа на x86 процессоре. В контекст входят все x86 регистры (как общего назначения, так и системные), режим работы процессора, память. При трансляции двоичного когда можно проводить оптимизации, повышающие эффективность результирующего кода, но тогда сильно возрастает время трансляции. Решить эту проблему можно, применяя одновременно несколько уровней трансляции (от быстрого, но порождающего плохой код, до медленного, но с наиболее эффективным результирующим кодом) и использовать тяжёлые уровни только для наиболее часто исполняющегося кода [3], [11].

В двоичном оптимизирующем компиляторе, разрабатываемом в «МЦСТ» есть четыре уровня: интерпретатор (1), шаблонный транслятор (2), быстрый оптимизирующий транслятор (3), оптимизирующий транслятор (4). Первый, интерпретатор, в сущности, не является двоичным транслятором, так как, получив x86 команду, сразу сам изменяет контекст (а не генерирует код, который будет его модифицировать). Шаблонный транслятор, получая на вход блоки x86 кода, быстро выдаёт код следующим образом: для каждой x86 инструкции имеется готовый эльбрусовский код — *шаблон*, который с минимальными изменениями прибавляется к трансляции. Оптимизирующие трансляторы сначала переводят x86 код во внутреннее представление, производят оптимизации и затем преобразуют внутреннее представление в результирующий код.

От интерпретации к двоичной трансляции

Для каждой инструкции исходной архитектуры в интерпретаторе есть своя функция, выполняющая над контекстом такое же преобразование (а также проявляющая все исключительные ситуации), которое делает микропроцессор исходной архитектуры во время её исполнения. Чтобы понять, какую функцию вызывать для текущей инструкции, её сначала нужно декодировать. Получается, что для каждой инструкции в интерпретаторе выполняются следующие действия: декодирование, вызов функции, преобразование контекста, возврат. Из них существенным является лишь преобразование контекста. С помощью двоичной трансляции можно ощутимо уменьшить время, затрачиваемое на другие действия.

В простом варианте двоичный транслятор берёт последовательность инструкций и транслирует каждую их них в двоичный код целевой архитектуры (который выполняет эквивалентное преобразование контекста и проявляет исключительные ситуации). Получившимся *двоично-транслированным кодом* можно пользоваться как новой функцией, вызов которой эквивалентен интерпретации группы инструкций. Группа последовательных инструкций называется *базовым блоком*, если в нём не более одной команды перехода, которая, если она есть, стоит последней в блоке. В простом случае двоичный транслятор работает с базовыми блоками.

Транслируется блок не быстрее, чем интерпретируется, но уже оттранслированный блок можно сохранить в *кеше трансляций* и переиспользовать при следующем исполнении оттранслированной группы инструкций. Соответствие <x86 адрес начала базового блока, адрес двоично-транслированного кода> при этом нужно запомнить в *таблице перекодирования адресов*.

В вышеописанной схеме после исполнения двоично-транслированного кода происходит поиск следующего двоично-транслированного кода по таблице перекодирования адресов. Во многих случаях адрес перехода, которым заканчивается базовый блок, является статическим и не меняется в процессе исполнения программы. Для устранения избыточных поисков по таблице, в двоично-транслированном коде сразу делается переход на следующий двоично-транслированный код. Такая оптимизация называется *связывание блоков* [12].

Для достижения более высокой производительности двоичного транслятора необходимо введение нескольких уровней трансляции с различной степенью оптимизации кода. Для того чтобы выделить наиболее часто работающий участки кода, создаётся *профильный граф*. Профильный граф – это структура данных, хранящая информацию о

числе повторений линейных участков. На основании этой информации принимается решение о перетрансляции кода с более высоким уровнем оптимизаций.

Шаблонная трансляция

Шаблонный транслятор является вторым после интерпретатора уровнем системы и первым, на котором собственно происходит двоичная трансляция. Интерпретатор в процессе своей работы формирует *трассу* — последовательность исполненных базовых блоков. Когда некоторый линейный участок выполнился несколько раз, от него набирается трасса и запускается шаблонный транслятор, преобразующий команды трассы в код целевой архитектуры. *Входом* в двоично-транслированном коде называется адрес, для которого существует запись в таблице перекодирования адресов. Входом в трассу является начало каждого базового блока. Можно было бы выбрать единицей трансляции базовый блок, но трасса позволяет убрать переходы между последовательными блоками, сохранив возможность начала каждого блока быть входом (как это было бы, если бы блоки транслировались отдельно). В двоично-транслированном коде блоки размещаются последовательно и в переходах нет необходимости.

Генерация кода базового блока происходит следующим образом: вначале создаётся *пролог*, затем каждая инструкция транслируется отдельно, и заканчивается двоично-транслированный код *эпилогом*. Если в блоке была инструкция перехода, то в соответствующий ей код встраивается эпилог, иначе он генерируется отдельно. В прологах и эпилогах увеличиваются счётчики профильного графа (доступ к счётчикам из двоично-транслированного кода осуществляется непосредственно по адресу), делается проверка, не пора ли перетранслировать код с использованием быстрого оптимизирующего транслятора (на основании информации в профильном графе). Также, в прологах 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 том числе при прерываниях

выдавать контекст таким же, каким он был бы в микропроцессоре исходной архитектуры. Для повышения производительности проявление многих прерываний (ошибка страницы, выход за границы сегмента и т.п.) выполняются на аппаратном уровне, а двоичный транслятор следит только за корректностью контекста. Все вычисления в шаблонном двоично-транслированном коде делаются на временных регистрах. Затем производится попытка проявить исключения (например, если команда пишет в память, то двоично-транслированный 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).

На момент трансляции, даже для переходов на константный адрес в общем случае ещё неизвестно, по какому адресу будет двоично-транслированный код, соответствующий данному переходу. Поэтому адреса в двоично-транслированным коде, где прописаны смещения переходов, и соответствующие им x86 адреса сохраняются в отдельной таблице, записи которой называют *релокациями*. Естественно, двоично-транслированный код нельзя исполнять, пока релокации не будут *связаны*. Связыванием называется процесс, в котором по таблице релокаций в двоично-транслированный код прописываются работающие смещения. Для релокаций, цели которых оказались внутри трассы, связывание происходит сразу после завершения трансляции. Оставшиеся релокации сохраняются вместе с трансляцией в кеше трансляций, а связывание осуществляется позже. Если цели релокации нет ни в какой трансляции, то при соответствующем переходе будет запущен интерпретатор.

В шаблонном коде для косвенных переходов (с адресом цели, лежащем на регистре или в памяти) связывание не делается и, в текущей реализации, переход всегда осуществляется с использованием таблицы перекодирования адресов. Следует отметить, что в литературе встречаются вполне применимые к шаблонному транслятору приёмы оптимизации косвенных переходов [2], [4], [5], [6], однако в микропроцессоре «Эльбрус» для этого есть собственный механизм. Использование данной возможности аппаратуры является одним из пунктов дальнейшего развития шаблонного транслятора.

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

инструкции, осуществляется только шаблонным транслятором.

Отладочные технологии

Для отладки и анализа двоично-транслированного кода эффективно использовать технологию статической двоичной трансляции. При этом можно ограничиться лишь пользовательским x86 кодом и обеспечивать трансляцию отдельных приложений. Статический транслятор, в отличие от динамического, работает до запуска приложения. Это накладывает существенные ограничения: не поддерживаются приложения с самомодифицирующимся кодом, приложения, генерирующие код и запускающие его, если происходит переход на не предсказанный адрес, оттранслированным приложением невозможно пользоваться. В рабочих системах статической трансляции для решения всех этих проблем в двоично-транслированную программу встраивается интерпретатор или динамический транслятор [14], [15]. Но, если ограничиться не слишком сложными приложениями, этого можно и не делать.

Статическая трансляция позволяет легко получить и посмотреть (в дизассемблере или отладчике) на двоично-транслированный код. В бинарный файл можно также встроить дополнительную отладочную информацию, которую дизассемблер или отладчик понимают и наглядно отображают. На практике очень полезны метки, показывающие, какому x86 адресу соответствует данный код и какой функцией он был сгенерирован.

Производительность

Проведены сравнения производительности различных уровней трансляции. Измерения производились на нескольких задачах из пакетов spec95 и spec2000 [13] с уменьшенными входными данными. Задачи статически транслировались каждым уровнем в отдельности. Таким образом, каждый запуск отражает работу результирующего кода только для одного уровня трансляции, что позволяет произвести их сравнение. Измерения проводились на потактовом симуляторе микропроцессора «Эльбрус 3S». В таблице 1 приведено сравнение производительности двоично-транслированных кодов для каждого уровня, нормированное относительно четвёртого уровня. Что касается интерпретатора, то измерения при динамической трансляции показывают, что он работает примерно в 30 раз медленнее шаблонного кода.

Помимо времени работы существенной характеристикой системы динамической двоичной трансляции является время, затрачиваемое на трансляцию. В таблице 2 приведено сравнение времени трансляции различными уровнями.

Задача	Шаблонный код	Код 3-го уровня	Код 4-го уровня
099.go_slice	0.23	0.67	1.00
126.gcc_lpriotoize	0.22	0.63	1.00
186.crafty.p4	0.18	0.60	1.00
255.vortex.p4	0.10	0.44	1.00

Таблица 1. Сравнение производительности двоично-транслированных кодов (относительно 4-го уровня)

	Время компиляции (тактов на 1 x86 инструкцию)
Шаблонный транслятор	600
Оптимизатор 3-го уровня	20000
Оптимизатор 4-го уровня	500000

Таблица 2. Среднее время, затрачиваемое на трансляцию одной x86 инструкции различными уровнями трансляции

Список литературы

1. Erik R. Altman, Kemal Ebcioglu, Michael Gschwind, Sumedh Sathaye. Advances and Future Challenges in Binary - Translation And Optimization. // Proceedings of the IEEE. Volume 91, Issue 11. 2001. November. – Pages 1710-1722.
2. Cristina Cifuentes, Brian Lewis, David Ung. Walkabout: a retargetable dynamic binary translation framework. // Sun Microsystems, Inc. Technical Reports; Vol. SERIES13103. 2002.
3. James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Er Klaiber, Jim Mattson. The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. // ACM International Conference Proceeding Series; Vol. 37. 2003. – Pages 15-24.
4. Derek Bruening, Timothy Garnett, Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. // ACM International Conference Proceeding Series; Vol. 37. 2003. – Pages 265-275.
5. K. Scott, N. Kumar, B. R. Childers, J. W. Davidson, M. L. Soffa. Overhead reduction techniques for software dynamic translation. // Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International. 07 June 2004. – Pages 200.
6. Jim Smith, Ravi Nair. Virtual Machines: Versatile Platforms for Systems and Processes.: Morgan Kaufmann, 2005. – 656 pages.
7. А.К.Ким, В.Ю.Волконский, Ф.А.Груздов, М.С.Михайлов, Ю.Н.Парахин Ю.Х.Сахин, С.В.Семенихин, М.В.Слесарев, В.М.Фельдман. Архитектура, программное обеспечение и применения компьютеров серии «Эльбрус». // IV Международная научно-практическая конференция «Современные информационные технологии и ИТ-образование», сборник избранных трудов. 2009. – Стр 53-72.

8. K. Dieffendorf. The Russians Are Coming. Supercomputer Maker Elbrus Seeks to Join x86/IA-64 Melee. // Microprocessor Report, V.13, №.2. February 15, 1999. – Pages 1-7.
9. Boris Babayan. E2K Technology and Implementation. // Parallel Processing: 6th International. Volume 1900 January 2000. – Pages 18-21.
10. Ермолович А.В. Методы повышения производительности двоично-транслирующих систем с аппаратной поддержкой. – Диссертация на соискание ученой степени кандидата технических наук, М., ИМВС РАН, 2003.
11. Волконский В.Ю., Гимпельсон В.Д. Методы определения порогов активизации динамического оптимизирующего транслятора. // Информационные технологии, № 4, 2007 г.
12. F. Bellard. QEMU, a fast and portable dynamic translator. // USENIX 2005 Annual Technical Conference, FREENIX Track. 2005. – Pages 41-46.
13. Standard Performance Evaluation Corporation [Электронный ресурс]. – Режим доступа: http://www.spec.org
14. Cristina Cifuentes, Mike Van Emmerik, Norman Ramsey, Brian Lewis. The University of Queensland Binary Translator (UQBT) Framework. Sun Microsystems Laboratories. 2001.
15. Jiunn-Yeu Chen, Wuu Yang, Tzu-Han Hung, Hong-Men Su, Wei-Chung Hsu. A Static Binary Translator for Efficient Migration of ARM-based Applications // the 6th Workshop on Optimizations for DSP and Embedded Systems (ODES), April 2008