М.И. Нейман-заде, В.Ю. Волконский

СРЕДЫ ПРОГРАММИРОВАНИЯ И ОПТИМИЗИРУЮЩИЕ КОМПИЛЯТОРЫ ДЛЯ КОМПЬЮТЕРОВ С МИКРОПРОЦЕССОРАМИ АРХИТЕКТУРЫ «ЭЛЬБРУС»

Аннотация

Дано описание различных систем компиляции кода, входящих в состав программного обеспечения вычислительных комплексов «Эльбрус». Перечислены поддерживаемые языки программирования, продемонстрирована совместимость с распространенными системами программирования. Рассмотрены аппаратные решения для поддержки производительности и использующие их техники оптимизации кода для архитектуры «Эльбрус», применяемые в перечисленных системах компиляции.

Ключевые слова: среды программирования, оптимизирующая компиляция, архитектура VLIW, микропроцессоры «Эльбрус».

Введение

Вычислительные комплексы линии «Эльбрус» на базе универсальных российских микропроцессоров архитектуры «Эльбрус» разрабатываются в АО «МЦСТ» и ПАО «ИНЭУМ им. И.С. Брука» и оснащаются современным программным обеспечением (ПО).

Отличительными особенностями архитектуры «Эльбрус» являются [1]-[4]:

- явный параллелизм исполнения отдельных операций, объединяемых при компиляции программ в так называемые широкие команды;
- средства поддержки эффективной двоичной совместимости с широко распространенной в мире архитектурой Intel x86 на базе динамической бинарной компиляции;
- специальные средства для надежного и безопасного исполнения программ.

Эти архитектурные особенности поддерживаются инфраструктурным программным обеспечением, в первую очередь статическими и динамическими (работающими в процессе исполнения программ) оптимизирующими компиляторами с наиболее распространенных языков программирования высокого уровня (C, C++, Fortran, Java, JavaScript, C#) и с двоичных кодов Intel x86. Именно на компиляторы возлагается значительная часть вопросов производительности в процессе построения оптимизированного кода для архитектуры «Эльбрус», относящейся к классу VLIW-архитектур (Very Long Instruction Word). Важнейшими среди этих вопросов являются:

- обнаружение параллелизма элементарных инструкций кода (операций);
- планирование операций в широкие команды, которые запускаются на исполнение одновременно в одном такте микропроцессора;
- использование различных механизмов аппаратной поддержки производительности.

Технологии оптимизирующей компиляции с аппаратной поддержкой параллелизма и двоичной совместимости позволяют совершенствовать архитектуру и являются основой технологической независимости и информационной безопасности российских высокопроизводительных вычислительных систем линии «Эльбрус».

В первой части статьи будет проведен обзор различных компиляторов и систем программирования, доступных для микропроцессоров «Эльбрус». Вторая часть будет посвящена специфическим вопросам достижения производительности кода при оптимизирующей компиляции для архитектуры «Эльбрус».

Компиляторы, входящие в состав общего программного обеспечения «Эльбрус»

Архитектура «Эльбрус» прошла долгий путь становления в качестве полноценной универсаль-

ной вычислительной платформы. За последние 10 лет она нашла практическое применение в самых различных областях – от встраиваемых систем и настольных ЭВМ до больших высокопроизводительных вычислителей промышленного и научного применения. Обилие и разнообразие ПО, а также интересы потенциальных пользователей потребовали поддержки определенного множества языков программирования.

Наряду с параллелизмом отдельных операций архитектура «Эльбрус» поддерживает параллелизм упакованных данных (векторные операции) и параллелизм потоков управления, которые также требуют поддержки в компиляторах.

Если обратиться к доступным системам программирования с открытым кодом, то интерпретационные языки, такие как Perl, Python, Ruby, потребовали небольших усилий по переносу на платформу «Эльбрус». В то же время компиляционные статические (C, C++, Fortran) и динамические (Java, JavaScript, C#) языки потребовали существенных доработок, прежде всего в части компиляторов, и решение об использовании готовых систем было неоднозначным. Забегая вперед, можно сказать, что в большинстве случаев от открытых решений пришлось отказаться. Рассмотрим теперь более подробно особенности реализации важнейших языковых и двоичных систем.

C/C++

Заложенная в архитектуру собственная система команд, не имеющая точных аналогов в мире, требовала разработки компилятора на самых ранних этапах проекта. В первую очередь требовалась поддержка для языков C и C++ с целью сборки операционной системы и основных компонентов общего программного обеспечения (ОПО). Перенос существующих открытых компиляторов оказался затруднительным по ряду причин:

- промежуточное представление (Intermediate Representation, IR) компилируемой программы требовало существенных доработок;
- работа практически всех оптимизирующих преобразований требовала дополнительной адаптации к доработанному IR;
- требовалась реализация множества архитектурно-зависимых оптимизаций.

Такая работа по оценке оказалась не менее сложной, чем разработка собственного компилятора для языков C/C++. Поэтому было решено разработать собственные оптимизаторы и генераторы кода для платформы «Эльбрус», используя в качестве начальной фазы работы с языками продукт компании «Edison Design Group», поставляющей услуги множеству фирм-разработчиков компиляторов.

Первая версия компилятора разрабатывалась вместе с системой команд и архитектурой «Эльбрус» с 1997 года. Текущим итогом разработки явилась выпущенная версия 1.23 компилятора lcc («Elbrus C Compiler») в составе системы программирования, совместимой с широко распространенной системой компиляторов «Gnu Compiler Collection» (gcc) версии 5.5.0.

Отметим основные достижения оптимизирующего компилятора:

- собственное уникальное низкоуровневое промежуточное представление IRE2K компилируемой программы, близкое к системе команд «Эльбрус»;
- собственное высокоуровневое архитектурно-независимое промежуточное представление EIR компилируемой программы, поддерживающее строгую типизацию, унаследованную от языка C;
- более 300 различных фаз компиляции, применяемых к промежуточному представлению, в рамках каждой из которых проводятся либо технологическое или оптимизирующее преобразование представления, либо анализ операций и их зависимостей;
- пять различных последовательностей применения компиляционных фаз для процедур компилируемого приложения и интеллектуальная система собственной разработки, производящая выбор последовательности, наиболее подходящей для оптимизации каждой процедуры;
- разработанные «с нуля» фазы компиляции, максимально использующие все аппаратные механизмы архитектуры «Эльбрус» для обеспечения производительности кода, включая явный параллелизм и использование операций над упакованными данными;
- поддержка специализированной сборки и оптимизации программ для безопасного исполнения, основанного на аппаратном разделении типов (данных и указателей);
- надежность, доведенная до промышленного уровня и позволяющая производить сборку

дистрибутива операционной системы суммарным объемом более 900 млн. строк кода;
- достижение пиковой вещественной производительности микропроцессора на тесте, написанном на чистом языке высокого уровня без использования ассемблера (этот факт традиционно демонстрируется на испытаниях каждой версии микропроцессора);
- производительность на стандартных тестах производительности Spec CPU 2006 и 2017, в среднем равная либо немного превосходящая архитектурную (на приведенной частоте) скорость в сравнении с современными процессорами «Intel x64» с использованием последних версий компилятора gcc;
- поддержка отладочной информации в формате DWARF-2 (привязка к строкам исходного кода возможна только в режиме без оптимизации в силу специфики архитектуры);
- поддержка большинства расширений языка C, принятого в компиляторе gcc;
- поддержка расширения OpenMP языка C для распараллеливания на потоки на уровне стандарта 3.0.

Система программирования (СП), в которую входит компилятор lcc, выполнена на основе системы программирования gcc с целью получения максимальной совместимости, в свою очередь, позволяющей минимизировать затраты на перенос ПО с x86/linux/gcc как для внутренних разработчиков дистрибутива, так и для внешних пользователей. Большая часть компонентов СП перенесена из СП gcc с переделкой архитектурно-зависимой части.

Fortran

Язык Fortran в основном используется для научных расчетов как в академической (например, CERN), так и в прикладной среде (например, разведка полезных ископаемых). Тем не менее для вычислительных комплексов «Эльбрус» он представляет интерес из-за специфики архитектуры – соотношения flop/такт, flop/byte, flop/watt, а также особенностей алгоритмов.

На ранних этапах проекта использовалась собственная реализация языка, поддерживавшая стандарт Fortran77. В дальнейшем, с выпуском первых версий микропроцессора и появлением приложений пользователей, в компилятор был встроен полноценный конвертор Fortran→C, поддерживающий более современные версии стандарта вплоть до Fortran2008. Вначале конвертор работал как отдельное приложение, и в процессе компиляции скрытым образом участвовал промежуточный файл на C; в дальнейшем конвертор был существенно переработан и переведен на прямую генерацию абстрактного синтаксического дерева. Полученный компилятор доступен в системе с помощью команды lfortran.

Для программ, написанных на языке Fortran, важна возможность их распараллеливания. Компилятор lfortran задействует все виды параллелизма архитектуры «Эльбрус», используя расширения OpenMP для распараллеливания на потоки и библиотеку MPI при распараллеливании для кластерных систем «Эльбрус». Компилятор lfortran используется для сборки тестов производительности SpecCPUfp 2006/2017, для прикладных библиотек BLAS, LAPACK, HDF5, python numpy, а также для ряда задач пользователей. Производительность типичных научных и вычислительных приложений, написанных на Fortran, как правило, не уступает современным универсальным микропроцессорам на приведенной частоте.

Java, JavaScript, C#

Потребность в динамических языках, прежде всего в Java, JavaScript (JS) и C#, неуклонно растет в течение последних двух десятилетий в первую очередь ввиду кросс-платформенной переносимости ПО и удобства написания в условиях современных требований к интерфейсу программ. Переносимость достигается за счет преобразования исходного кода программы в код некоторой виртуальной языковой машины.

Для получения высокой производительности исполнения кодов динамических языков используются так называемые JIT-компиляторы (Just in Time), производящие оптимизирующую компиляцию отдельных процедур или фрагментов кода виртуальной языковой машины в процессе его исполнения. К скорости работы JIT-компиляторов предъявляются высокие требования, так как время их работы зачастую включается во время исполнения программы. В то же время в процессе оптимизации они используют дополнительную информацию (недоступную статическим компиляторам) об исполнении программы в текущем сценарии работы (профиль), которая помогает получать более быстрый код.

Первым динамическим компилируемым языком для вычислительных комплексов «Эльбрус» стал язык Java, реализованный в виде Java Virtual Machine версии 6 в 2013 году на основе открытого проекта OpenJDK. Существенные отличия JIT-компиляторов затруднили использование в этом проекте наработок компилятора lcc, и первая версия была сделана также на базе открытого решения – пары Zero (переносимый интерпретатор Java с «нулевым» присутствием ассемблера) + Shark (компилятор байт-кода Java на основе известной системы трансляции с открытым кодом LLVM). При этом потребовалось адаптировать LLVM к архитектуре «Эльбрус», что удалось сделать с потерей большинства аппаратных решений «Эльбрус» для обеспечения производительности. В дальнейшем состоялся отказ от LLVM и переход на оптимизатор собственной разработки. В его рамках система команд и возможности архитектуры «Эльбрус» задействованы существенно шире, что позволило в несколько раз увеличить производительность приложений. В настоящее время на платформе «Эльбрус» работает и успешно используется Java 8.

Успех JIT-компилятора для виртуальной машины Java позволил использовать его в последующих проектах: в системе Mono, реализующей язык C#, и двух системах (транслятор + система поддержки) JavaScript – SpiderMonkey («Firefox») и V8 («Chrome»). Несмотря на существенные различия этих программных сред, в них на данный момент удается успешно использовать общий оптимизирующий JIT-компилятор. Сам оптимизирующий JIT-компилятор продолжает активно развиваться и совершенствоваться в аспекте получения высокой производительности.

Динамическая бинарная трансляция

Еще одним важным компонентом оптимизирующей компиляции, входящим в состав ОПО «Эльбрус», является система динамической бинарной трансляции кодов x86. Строго говоря, она не является системой программирования и решает принципиально иную задачу – эмуляцию с сопоставимой производительностью вычислительной машины с архитектурой Intel x86.

Эмуляция исполнения кода другого процессора методом пошагового исполнения команд (интерпретацией) приводит к замедлению исполнения на 2 десятичных порядка в случае сопоставимых по сложности архитектур. С целью достижения заявленной цели обеспечения сравнимой архитектурной скорости с x86 над интерпретатором команд надстраивается многоуровневая система поддержки, включающая в себя оптимизирующую компиляцию различной сложности. Реализованная для серии микропроцессоров «Эльбрус» система бинарной компиляции имеет три уровня оптимизации: шаблонный транслятор (второй уровень), легкий оптимизатор (третий уровень), полноценный оптимизатор (четвертый уровень). Для обеспечения качественной и непрерывной работы эмулируемого кода применен ряд программно-аппаратных решений:

- оптимизация третьего и четвертого уровней производится на отдельном ядре микропроцессора;
- в случае одноядерного процессора оптимизация производится дискретным квантованным разделением времени с исполнением кода;
- используется аппаратная трансляция адресов доступа к памяти эмулируемой x86-машины;
- используется аппаратный буфер трансляции адресов исполняемого кода для эффективной поддержки операции передачи управления в эмулируемом коде;
- для ускорения сохранения x86-контекста заведены специализированные регистры;
- поддерживается многопоточная работа кодов x86.

Интересно, что на ранних этапах проекта оптимизатор четвертого уровня имел общие исходные коды с языковым оптимизирующим компилятором lcc, что было продиктовано почти полной идентичностью низкоуровневого промежуточного представления. Однако различия в подходе к отдельным оптимизациям, разные требования к скорости компиляции, отсутствие информации о типах и некоторые другие моменты привели к обособлению и выбору собственного вектора развития оптимизатора в системе бинарной трансляции.

Заметим также, что система динамической бинарной трансляции похожа по свойствам на JIT-компиляторы: требуются автономность, высокая скорость работы и есть возможность профилировать исполняемый код и использовать эту информацию.

В качестве продукта в настоящее время бинарная трансляция доступна в двух видах:

- lintel – полный эмулятор машины x86;
- rtc_opt – эмулятор linux-приложений в кодах x86, работающий под управлением ОС «Эльбрус».

Оба решения обеспечивают совместимость с Intel x86, x64. Архитектурная производительность в сравнении с процессорами Intel достигает в среднем 1,0 на целочисленных задачах и 0,95 на вещественных. Такие характеристики, а также тот факт, что многие прикладные программы часто доступны только в кодах x86, делают систему динамической бинарной трансляции крайне востребованной на практике.

Система динамической бинарной трансляции постоянно совершенствуется и максимально использует аппаратные возможности, появляющиеся в новых версиях микропроцессоров «Эльбрус».

Методы оптимизации кода с аппаратной поддержкой архитектуры «Эльбрус»

С 70-х годов XX века известно, что код программ содержит высокую степень логического параллелизма на уровне элементарных команд (операций). Анализ трасс исполнения различных приложений показал [5], [6], что в теоретических условиях точного предсказания операций перехода, бесконечного количества исполнительных устройств и регистров, а также возможностей по свободному перемещению операций вдоль трассы исполнения лишь в границах готовности аргументов реальные приложения достигают параллелизма свыше 100 команд за такт. Этот факт заставил искать возможности использования такого параллелизма для повышения производительности.

На этом пути были предложены две парадигмы: VLIW и Out-of-Order Superscalar. Парадигма VLIW (явное управление параллелизмом операций) обеспечивает со стороны аппаратуры множество исполнительных устройств и регистров, а задачу поиска параллелизма и параллельного планирования инструкций возлагает на компилятор. Парадигма Out-of-Order Superscalar решает вопросы обнаружения параллелизма в последовательном потоке команд и назначения для них исполнительных устройств и регистров в аппаратуре. С 80-х годов прошлого века оба подхода претерпели существенное развитие. Для парадигмы VLIW в целом и для семейства микропроцессоров «Эльбрус» в особенности результатом развития стал ряд аппаратно-программных решений, позволяющих дополнительно выявить и использовать параллелизм инструкций, не обнаруживаемый либо невозможный к использованию при обычной статической компиляции.

Основными отличительными особенностями оптимизации кода для VLIW являются: повышенные требования к объему оптимизируемого кода (простор для обнаружения параллелизма); потребность в информации о вероятностях ветвлений и траекторий исполнения кода; повышенная чувствительность к блокировкам по неготовности данных, читаемых из памяти; большее количество вычислительных устройств и регистров; требование точности планирования [7]. По этим причинам среди хорошо известных универсальных техник компиляторной оптимизации кода можно выделить несколько, потребовавших особых решений при реализации из-за специфики архитектуры:

- инлайн-подстановка функций [8];
- раскрутка циклов;
- заблаговременная подкачка данных и кода из памяти [9].

Рассмотрим подробнее некоторые специфичные для VLIW-архитектур оптимизационные техники, применяемые в компиляторах для микропроцессоров «Эльбрус».

Статическое планирование кода и распределение ресурсов

Параллелизм на уровне элементарных команд в случае Out-of-Order Superscalar обнаруживается и используется в процессе исполнения кода. Этот подход основан на организованной очереди операций, ожидающих готовности аргументов. При этом все операции в очереди, готовые к исполнению в текущем такте, конкурируют за исполнительные устройства и право исполниться немедленно.

Для VLIW точное планирование операций по исполнительным устройствам производится оптимизирующим компилятором. Для блоков кода строится ориентированный граф зависимостей операций, в котором узлам соответствуют операции, дугам – зависимости, причем каждой дуге приписывается длительность задержки от операции-источника до операции-приемника. Далее операции распределяются по широким командам так, чтобы все длительности задержек между ними были строго выдержаны.

Заметим, что порядок планирования операций, характерный для процессоров Out-of-Order Superscalar, соответствует «жадному алгоритму» планирования фронта готовых операций при обработке линейных участков кода проходом сверху вниз. Преимущество компиляторного подхода в том, что можно использовать различные алгоритмы планирования.

Кроме планирования операций необходимо произвести распределение регистров, т. е. заменить виртуальные регистры, используемые в промежуточном представлении, аппаратными регистрами. Это очень важная функция оптимизации, существенно влияющая на производительность компилируемой программы. На практике распределение регистров возможно как перед планированием, так и после него, причем каждый из подходов имеет свои дефекты. В компиляторе lcc после исследований и экспериментов был реализован комбинированный алгоритм планирования и распределения регистров, показавший преимущество перед раздельными алгоритмами [10].

Слияние кода и спекулятивный заброс операций

Распараллеливание линейного кода на участках между операциями передачи управления не позволяет выявить весь параллелизм программы, так как на них в среднем бывает только пять операций. Чтобы преодолеть это ограничение, требуется объединять участки линейного кода в регионы. Процесс объединения называется слиянием кода и поддерживается в аппаратуре режимами предикатного и спекулятивного исполнений операций.

Предикатный режим исполнения операций позволяет избежать операции передачи управления: условие, по которому выполняется передача управления, преобразуется в предикат, а операции из альтернативных участков кода включаются в широкие команды, но выполняются только при истинном значении предиката.

Спекулятивный режим исполнения, означающий «безопасное предвосхищающее исполнение операции», позволяет выполнять операции до того, как становится известно условие, при котором операция должна выполниться, что сокращает зависимости между операциями и увеличивает параллелизм. Однако заблаговременное исполнение может привести к ошибке (вызвать прерывание), если выполнить операцию альтернативы, которая не должна выполняться. Спекулятивный режим исполнения предотвращает возникновение таких ошибок, превращая ошибочный результат в отложенное прерывание, которое отменяется в момент, когда становится известно, что операция не должна была выполняться.

Этим методом пользуется и парадигма Out-of-Order Superscalar: процессоры этой парадигмы спекулятивно исполняют инструкции участков кода, следующих за цепочкой аппаратно предсказанных переходов. Отличие состоит в том, что спекулятивность для VLIW присутствует в системе команд явным образом.

Отметим, что не каждую операцию можно исполнить спекулятивно: результаты операций записи в память, передачи управления, модификации системных регистров нельзя отменить ввиду утраты старого значения (памяти, регистра IP, системного регистра соответственно).

На использовании предикатного и спекулятивного режимов исполнения операций основаны главные архитектурно-зависимые оптимизации VLIW: слияние кода (для ациклических участков) и программная конвейеризация (для циклов), описанная ниже.

Слияние кода происходит в пределах выбранных регионов оптимизируемой процедуры. В регион слияния включаются операции из участков со схожей частотой исполнения для того, чтобы в исполняемом коде было как можно меньше операций, результат которых не будет востребован впоследствии. Информация о частоте доступна оптимизации либо за счет динамического профиля исполнения (JIT-компиляторы, бинарный транслятор), либо путем статического предсказания вероятностей ветвления. Эффект по увеличению производительности от слияния кода превышает 40 % в среднем по ациклическим процедурам. Подробнее алгоритм набора регионов и слияния кода в компиляторах описан в [11], [12].

Динамический разрыв зависимостей между операциями обращения в память

При обращении в память по адресам, вычисляемым в процессе исполнения программы, между операциями записи и считывания появляются зависимости, которые являются истинными только при пересечении диапазонов их адресов. Чаще всего такие зависимости оказываются ложными (т. е. запись и считывание выполняются по не

пересекающимся диапазонам адресов), но при компиляции их приходится учитывать и отказываться от параллельного исполнения таких операций. Одним из возможных решений является добавление в код явных операций проверки пересечения диапазонов и команды условного перехода, выбирающего по результату сравнения одну из двух версий кода: быструю, в которой операции чтения и записи исполняются независимо друг от друга, либо медленную, в которой сохранен исходный порядок операций. Такое решение, называемое динамическим разрывом зависимости по памяти, позволяет немного ускорить исполнение, но отягощено накладными расходами в виде добавочных операций.

Специальный аппаратный механизм DAM (DisAMbiguation – снятие противоречия) позволяет осуществлять динамический разрыв с существенно меньшими накладными расходами. Для этого команда считывания данных раздваивается. Первое, спекулятивное, считывание выполняется без учета зависимости с предшествующими ей операциями записи, но в специальном буфере отмечаются пересечения с ними по адресам. Второе считывание сохраняется на исходном месте в коде для проверки того, что в буфере не появилось отметок о пересечении. Если такие отметки появились, операция считывания повторяется.

Схожий механизм реализован в последних поколениях Out-of-Order Superscalar с поправкой на то, что он не присутствует в системе команд явно и используется динамическим аппаратным планировщиком.

Итоговый эффект ускорения кода от использования механизма DAM составляет ~5 % для языкового компилятора и более 10 % для бинарного [13]. Последний факт обусловлен тем, что языковому оптимизирующему компилятору удается разорвать часть конфликтующих операций обращения к памяти за счет проводимых анализов.

Программно-аппаратная конвейеризация циклов

Техника программной конвейеризации циклов известна с начала 80-х годов прошлого века [14]. Суть этой техники сводится к тому, что каждая следующая итерация цикла не дожидается завершения выполнения предыдущей, а спекулятивно начинает исполняться до ее завершения с определенным сдвигом. Такой подход дает заметное ускорение исполнения цикла. Если для последовательного исполнения итераций цикла требуется $n \cdot T$ тактов (n – число повторений цикла; T – число тактов, необходимых для выполнения одной итерации), то для выполнения конвейеризированного цикла потребуется $(n-1) \cdot II + T$ тактов (II – интервал, на который сдвигается следующая итерация по отношению к предыдущей). Получается, что первая итерация цикла завершается через T тактов, а каждая последующая – через II тактов после предыдущей, за счет чего ускорение исполнения при больших значениях n составляет ~ T / II раз.

Аппаратная поддержка программной конвейеризации циклов в процессорах «Эльбрус» дает возможность:

- выделения области регистрового файла для циклического переименования регистров, на которых размещаются совмещенные итерации цикла; это позволяет сделать размер интервала между итерациями цикла меньше, чем длительность отдельных исполняемых в цикле операций;
- циклического переименования предикатных регистров, аналогичного возможности циклического переименования числовых регистров;
- включения в тело цикла участка «разгона» конвейера (пролог), используя специальные предикаты пролога для отмены операций с побочным эффектом во время его исполнения.

Алгоритм эффективного планирования конвейеризированного цикла довольно сложен и трудоемок; он требует решения таких проблем, как нехватка физических регистров, необходимость подбора эффективной дистанции от операций чтения до использования их результатов. Алгоритм для компилятора lcc более подробно описан в [15].

Эффект повышения производительности кода от конвейеризации циклов может достигать более 20 раз на специфичных примерах. Оптимизация является важнейшей для научно-вычислительных приложений, в которых на исполнение простых циклов с большим числом итераций приходится большая доля времени исполнения.

Предподкачка линейно-регулярных данных в цикле

Проблема неравномерности доступа к памяти является весьма острой для всех современных микропроцессоров и для семейства архитектур

VLIW в особенности. Неготовность результата операции чтения из памяти заблокирует на Out-of-Order Superscalar-процессоре только зависимую цепочку операций, а на VLIW-процессоре – весь конвейер.

В рамках Out-of-Order Superscalar проблема задержек исполнения команд из-за промахов в кэш решается с помощью сложных и во многом автономных аппаратных устройств в рамках кэш-памяти, которые обнаруживают регулярные паттерны изменения адресов и производят предварительную подкачку данных из оперативной памяти в кэши данных различных уровней.

Механизм программно-аппаратной подкачки данных по линейно изменяющимся адресам в специальный буфер для предварительной подкачки массивов APB (Array Prefetch Buffer), реализованный в процессорах серии «Эльбрус», является уникальной альтернативой такому подходу. Он требует серьезной поддержки со стороны компилятора и применим в циклах к операциям чтения, работающим по адресам, линейно зависящим от номера итераций, например arr[i] в цикле вида for(;;i++) или *p++ в цикле while().

Поддержка со стороны компилятора проводится в рамках нескольких фаз и для каждого цикла заключается в следующем:

- обнаружение чтений с линейно изменяющимся адресом;
- объединение соседних по памяти адресов в пакеты для уменьшения числа подкачиваемых массивов;
- поиск наиболее эффективного разбиения аппаратного буфера на множество цикловых буферов для отдельных массивов и распределение специализированных регистров APB между разными массивами;
- подготовка кода асинхронной программы, которая осуществляет подкачку данных в APB;
- замена операций чтения из памяти (load) на операции пересылки данных из буфера для соответствующего массива (mova);
- инициализация регистров, хранящих информацию о массивах, и запуск асинхронной программы в коде перед циклом.

Ускорение исполнения кода от применения APB на отдельных циклах может достигать более десяти раз за счет того, что:

- операции mova не занимают место арифметико-логических операций, что позволяет конвейеризировать циклы более эффективно;
- устраняются блокировки конвейера, вызванные длительностью исполнения операций чтений.

Глобальное планирование подготовок переходов

Исполнение связной последовательности линейных участков кода для процессоров Out-of-Order Superscalar имеет существенную аппаратную поддержку в виде предсказателя адресов-назначений для операций перехода, что позволяет набирать множество команд для их спекулятивного исполнения. Неправильно предсказанный переход приводит к опустошению буфера спекулятивно исполненных команд и заметному снижению производительности. Можно сказать, что эффективность работы аппаратного предсказателя переходов является для процессоров Out-of-Order Superscalar одним из важнейших определяющих факторов.

Как уже отмечалось, для семейства архитектур VLIW в большей степени важно иметь информацию о траекториях исполнения кода на этапе компиляции (предсказывать либо получать за счет профилирования при динамической оптимизирующей компиляции), поэтому потребность в предсказателе переходов носит менее острый характер.

В таких условиях важно исполнять переходы в спланированном коде без блокировок, что обеспечивается в архитектуре «Эльбрус» с помощью заблаговременно спланированных операций подготовки переходов. Поскольку каждый подготовленный переход использует специальный регистр подготовки, перед компилятором возникает задача глобального планирования операций подготовки переходов и распределения регистров подготовки для минимизации расстояния от входов в регион планирования до операций переходов.

Эффективность такого программного решения сравнима по производительности с хорошо работающим предсказателем переходов, а для плохо предсказываемых переходов в теле циклов оно обладает безусловными преимуществами.

Заключение

Компиляторы, доступные в дистрибутиве ОС «Эльбрус», предоставляют средства разработки на современных версиях основных распространенных языков: C, C++, Fortran, Java, JS, C#.

Аппаратные возможности используются компиляторами в полном объеме, автоматически и позволяют достигать высокой производительности исполнения программ.

Высокий уровень совместимости с распространенными компиляторами и системами программирования делает процесс переноса общего и прикладного ПО на платформу «Эльбрус» весьма эффективным [16].

Качество компиляторов по надежности позволяет успешно компилировать с оптимизацией и исполнять сотни миллионов строк в составе дистрибутива ОС «Эльбрус», а также эффективно и надежно исполнять в кодах Intel x86, x64 операционные системы Windows XP, Windows 7, Linux с множеством проприетарных приложений и прикладные программы под управлением ОС «Эльбрус».

Реализованные системы оптимизирующей компиляции кода позволили достигнуть намеченной цели и сделать вычислительные системы на микропроцессорах «Эльбрус» конкурентоспособными в самых различных прикладных сферах.

Все это дает основание рассматривать «Эльбрус» как платформу для эффективного импортозамещения в сфере универсальных вычислительных устройств.

Список литературы:

1. *Ким А.К., Фельдман В.М.* Вопросы создания суперЭВМ на основе архитектурной платформы Эльбрус // Приборы. 2009. № 1. С. 36-46.
2. *Babayan B.A.* Main principles of E2k architecture // Free Software Magazine. 2002. Vol. 1. № 2.
3. *Кузьминский М.* Отечественные микропроцессоры: Elbrus E2k // Открытые системы. 1999. № 05-06.
4. *Ким А.К., Перекатов В.И., Ермаков С.Г.* Микропроцессоры и вычислительные комплексы семейства «Эльбрус». – СПб.: Питер, 2013. 272 с.
5. *Postiff M.A., Greene D.A., Tyson G.S., Mudge T.N.* The Limits of Instruction Level Parallelism in SPEC95 Application. – INTERACT-3 at ASPLOS-VIII, 1998.
6. *Lee H., Wu Y., Tyson G.S.* Quantifying Instruction Level Parallelism Limits on an EPIC Architecture. ISPASS, 2000. PP. 21-27.
7. *Волконский В.Ю., Брегер А.В., Бучнев А.Ю., Грабежной А.В., Ермолицкий А.В., Муханов Л.Е., Нейман-заде М.И., Степанов П.А., Четверина О.А.* Методы распараллеливания программ в оптимизирующем компиляторе // Вопросы радиоэлектроники. Сер. ЭВТ. 2012. Вып. 3. С. 63-88.
8. *Ермолицкий А.Е., Нейман-заде М.И., Четверина О.А., Маркин А.Л., Волконский В.Ю.* Агрессивная инлайн-подстановка функций для VLIW-архитектур // Труды ИСП РАН. 2015. Т. 27. Вып. 6.
9. *Волконский В.Ю., Грабежной А.В., Муханов Л.Е., Нейман-заде М.И.* Исследование влияния подсистемы памяти на производительность распараллеленных программ // Вопросы радиоэлектроники. Сер. ЭВТ. 2011. Вып. 3. С. 22-37.
10. *Иванов Д.С.* Распределение регистров при планировании инструкций для VLIW-архитектур // Программирование. 2010. № 6. С. 74-80.
11. *Волконский В.Ю., Гимпельсон В.Д., Масленников Д.М.* Быстрый алгоритм минимизации высоты графа зависимостей // Информационные технологии и вычислительные системы. 2004. № 3.
12. *Дроздов А.Ю., Новиков С.В.* Исследование методов преобразования программы в предикатную форму для архитектур с явно выраженной параллельностью // Компьютеры в учебном процессе. 2005. № 5. С. 91-99.
13. *Гимпельсон В.Д.* Конвейеризация циклов в бинарном динамическом трансляторе // Вопросы радиоэлектроники. Сер. ЭВТ. 2009. Вып. 3.
14. *Rau B.R., Glaeser D.C.* Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing / Proc. Fourteenth Annual Workshop on Microprogramming, October 1981. PP. 183-193.
15. *Дроздов А.Ю., Степаненков А.М.* Технология оптимизации циклов для архитектуры с аппаратной поддержкой конвейеризации // Информационные технологии и вычислительные системы. 2004. № 3. С. 52-62.
16. *Шалаев М.А.* Сборка компонентов программного обеспечения вычислительных комплексов семейства «Эльбрус» // Вопросы радиоэлектроники. 2017. № 3. С. 39-43.

Мурад Искендер-оглы Нейман-заде,
канд. физ.-мат. наук, начальник отделения,
Владимир Юрьевич Волконский,
канд. техн. наук, начальник отделения,
ПАО «ИНЭУМ им. И.С. Брука»,
г. Москва,
e-mail: muradnz@mcst.ru