VLIW: старая архитектура нового поколения

Архитектура сверхдлинного командного слова (VLIW — Very Long Instruction Word) берет свое начало от параллельного микрокода, применявшегося еще на заре вычислительной техники, и от суперкомпьютеров Control Data CDC6600 и IBM 360/91. В 1970 году многие вычислительные системы оснащались дополнительными векторными сигнальными процессорами, использующими VLIW-подобные длинные инструкции, прошитые в ПЗУ. Эти процессоры применялись для выполнения быстрого преобразования Фурье и других вычислительных алгоритмов. Первыми настоящими VLIW-компьютерами стали мини-суперкомпьютеры, выпущенные в начале 1980 года компаниями MultiFlow, Culler и Cydrome, но они не имели коммерческого успеха. Планировщик вычислений и программная конвейризация были предложены Фишером и Рау (Cydrome). Сегодня это является основой технологии VLIW-компилятора.

Первый VLIW-компьютер компании MultiFlow 7/300 использовал два арифметико-логических устройства для целых чисел, два АЛУ для чисел с плавающей точкой и блок логического ветвления — все это было собрано на нескольких микросхемах. Его 256bit командное слово содержало восемь 32bit кодов операций. Модули для обработки целых чисел могли выполнять две операции за один такт длиной 130 ns (т.е. всего четыре при двух АЛУ), что при обработке целых чисел обеспечивало быстродействие около 30 MIPS. Можно было также комбинировать аппаратные решения так, чтобы получать из или 256bit, или 1024bit вычислительные машины. Первый VLIW-компьютер Cydrome Cydra-5 использовал 256bit инструкцию и специальный режим, обеспечивающий выполнение команд как последовательности из шести 40bit операций, поэтому его компиляторы могли генерировать смесь параллельного кода и обычного последовательного. Существует мнение, что в то время, как эти VLIW-ВМ использовали несколько микросхем, процессор Intel i860 стал первым VLIW-процессором на одной микросхеме. Однако, i860 можно отнести к VLIW достаточно условно — по сути у него есть всего лишь программно-управляемое спаривание инструкций, в отличие от более позднего программно-неуправляемого, ставшего частью суперскалярных процессоров. В качестве исторической справки также хотелось бы упомянуть компьютеры фирмы FPS (AP-120B, AP-190L и все более поздние под маркой FPS), также основанные на VLIW-архитектуре, которые были в свое время достаточно распространенными и успешными на рынке. Кроме этого, существовали такие «канонические» машины, как М10 и М13 Карцева, а также «Эльбрус-3» — при всем «неуспехе» последнего проекта, он все же явился этапом VLIW. Вообще, быстродействие VLIW-процессора в большей степени зависит от компилятора, нежели от аппаратуры, поскольку здесь эффект от оптимизации последовательности операций превышает результат, возникающий от повышения частоты.

Относительно недавно мы были свидетелями «противостояния» CISC против RISC, а теперь уже намечается новое «сражение» — VLIW против RISC. Строго говоря, VLIW и суперскалярный RISC — никак не антагонисты, ни в коей мере. Справедливости ради необходимо отметить, что последние — это вовсе не «внешнеархитектурное» свойство, а просто некий способ исполнения. Возможно, что в дальнейшем появятся суперскалярные VLIW-процессоры, которые тем самым приобретут, если так можно выразиться, «параллелизм в квадрате» — объединение явного статического параллелизма с неявным динамическим. Но на сегодняшнем этапе развития процессоров нет видимых способов совмещать статическое и динамическое переупорядочивание. Именно поэтому Itanium/Itanium2 сейчас следует рассматривать не столько в контексте сравнения VLIW «против» CISC (и тем более не VLIW «против» O3E), а скорее как «синхронный VLIW» vs «асинхронный (Out-Of-Order) RISC». Ну, и не стоит

Несмотря на то, что архитектура VLIW появилась еще на заре компьютерной индустрии (Тьюринг разработал VLIW-компьютер еще в 1946 году), она до сих пор не имела коммерческого успеха. Теперь Intel воплотила некоторые идеи VLIW в линейке процессоров Itanium. Но значительного повышения производительности и скорости вычислений в системах на базе этих процессоров по отношению к существующим классическим «RISC-inside CISC-outside» архитектурам можно добиться лишь путем переноса интеллектуальных функций из аппаратного обеспечения в программное (компилятор). Таким образом, успех Itanium/Itanium2 определяется в основном программными средствами — именно в этом и состоит проблема. Причем довольно сдержанное отношение индустрии к сравнительно давно существующему Itanium только подтвердило факт ее наличия.

EPIC: явный параллелизм команд

Концепция реализации параллелизма на уровне команд (Explicitly Parallel Instruction Computing) определяет новый тип архитектуры, способной конкурировать по масштабам влияния с RISC. Эта идеология направлена на то, чтобы упростить аппаратное обеспечение и, в то же время, извлечь как можно больше «скрытого параллелизма» на уровне команд, используя большую ширину «выдачи» команд (WIW -Wide Issue-Width) и длинные (глубокие) конвейеры с большой задержкой (DPL — Deep Pipeline-Latency), чем это можно сделать при реализации VLIW или суперскалярных стратегий. EPIC упрощает два ключевых момента, реализуемых во время выполнения. Во-первых, его принципы позволяют во время исполнения отказаться от проверки зависимостей между операциями, которые компилятор уже объявил как независимые. Во-вторых, данная архитектура позволяет отказаться от сложной логики внеочередного исполнения операций, полагаясь на порядок выдачи команд, определенный компилятором. Более того, EPIC совершенствует возможность компилятора статически генерировать планы выполнения за счет поддержки разного рода перемещений кода во время компиляции, которые были бы некорректными в последовательной архитектуре. Более ранние решения достигали этой цели главным образом за счет серьезного увеличения сложности аппаратного обеспечения, которое стало настолько значительным, что превратилась в препятствие, не позволяющее отрасли добиваться еще более высокой производительности. EPIC разработан именно для того, чтобы обеспечить более высокую степень параллелизма на уровне команд, поддерживая при этом приемлемую сложность аппаратного обеспечения.

Более высокая производительность достигается как за счет увеличения скорости передачи сигналов, так и благодаря увеличению плотности расположения функциональных устройств на кристалле. Зафиксировав рост этих двух составляющих, дальнейшего увеличения скорости выполнения программ можно добиться в первую очередь благодаря реализации определенного вида параллелизма. Так, параллелизм на уровне команд (ILP — Instruction-Level Parallelism) стал возможен благодаря созданию процессоров и методик компиляции, которые ускоряют работу за счет параллельного выполнения отдельных RISC-операций. Системы на базе ILP используют программы, написанные на традиционных языках высокого уровня, для последовательных процессоров, а обнаружение «скрытого параллелизма» автоматически выполняется благодаря применению соответствующей компиляторной технологии и аппаратного обеспечения.

Тот факт, что эти методики не требуют от прикладных программистов дополнительных усилий, имеет крайне важное значение, поскольку данное решение резко отличается от традиционного микропроцессорного параллелизма, который предполагает, что программисты должны переписывать свои приложения. Параллельная обработка на уровне команд является единственным надежным подходом, позволяющим добиться увеличения производительности без фундаментальной переработки приложения.

Суперскалярные процессоры — это реализации ILP-процессора для последовательных архитектур, программа для которых не должна передавать и, фактически, не может передавать точную информацию о параллелизме. Поскольку программа не содержит точной информации о наличии ILP, задача обнаружения параллелизма должна решаться аппаратурой, которая, в свою очередь, должна создавать план действий для обнаружения «скрытого параллелизма». Процессоры VLIW представляют собой пример архитектуры, для которой программа предоставляет точную информацию о параллелизме — компилятор выявляет параллелизм в программе и сообщает аппаратному обеспечению какие операции не зависят друг от друга. Эта информация имеет важное значение для физического слоя, поскольку в этом случае он «знает» без дальнейших проверок какие операции можно начинать выполнять в одном и том же такте. Архитектура EPIC — это эволюция архитектуры VLIW, которая абсорбировала в себе многие концепции суперскалярной архитектуры, хотя и в форме, адаптированной к EPIC. По сути — это «идеология», определяющая, как создавать ILP-процессоры, а также набор характеристик архитектуры, которые поддерживают данную основу. В таком смысле EPIC похож на RISC: определяющий класс архитектур, подчиняющихся общим основным принципам. Точно также, как существует множество различных архитектур наборов команд (ISA) для RISC, может существовать и больше одной ISA для EPIC. В зависимости от того, какие из характеристик EPIC использует архитектура EPIC ISA, она может быть оптимизирована для различных приложений —

Код для суперскалярных процессоров содержит последовательность команд, которая порождает корректный результат, если выполняется в установленном порядке. Код указывает последовательный алгоритм и, за исключением того, что он использует конкретный набор команд, не представляет себе точно природу аппаратного обеспечения, на котором он будет работать или точный временной порядок, в котором будут выполняться команды. В отличие от программ для суперскалярных процессоров, код VLIW предлагает точный план (POE — Plan Of Execution, схема исполнения создается статически во время компиляции) того, как процессор будет выполнять программу. Код точно указывает когда будет выполнена каждая операция, какие функциональные устройства будут работать и какие регистры будут содержать операнды. Компилятор VLIW создает такой план выполнения, имея полное представление о самом процессоре, чтобы добиться требуемой записи исполнения (ROE — Record Of Execution) — последовательности событий, которые действительно происходят во время работы программы. Компилятор передает POE (через архитектуру набора команд, которая точно описывает параллелизм) аппаратному обеспечению, которое, в свою очередь, выполняет указанный план. Этот план позволяет VLIW использовать относительно простое аппаратное обеспечение, способное добиться высокого уровня ILP. В отличие от VLIW, суперскалярная аппаратура динамически строит POE на основе последовательного кода. Хотя такой подход и увеличивает сложность физической реализации, суперскалярный процессор создает план, используя преимущества тех факторов, которые могут быть определены только во время выполнения.

Одна из целей, которые ставили перед собой при создании EPIC, состояла в том, чтобы сохранить реализованный во VLIW принцип статического создания POE, но и в то же время обогатить его возможностями, аналогичными возможностям суперскалярного процессора, позволяющими новой архитектуре лучше учитывать динамические факторы, традиционно ограничивающие параллелизм, свойственный VLIW. Чтобы добиться этих целей, «идеология» EPIC была построена на некоторых основных принципах.

Первый — это создание плана выполнения во время компиляции. EPIC возлагает нагрузку по созданию POE на компилятор. Хотя, в общем, архитектура и физическая реализация могут препятствовать компилятору в выполнении этой задачи, процессоры EPIC предоставляют функции, которые помогают компилятору создавать план выполнения. Во время исполнения поведение процессора EPIC с точки зрения компилятора должно быть предсказуемым и управляемым. Динамическое внеочередное исполнение команд может «запутать» компилятор так, что он не будет «понимать», как его решения повлияют на реальную запись выполнения, созданную процессором, поэтому ему необходимо уметь предсказывать действия процессора, что еще больше усложняет задачу. В данной ситуации предпочтителен процессор, четко исполняющий то, что ему указывает программа. Суть же создания плана во время компиляции состоит в переупорядочивании исходного последовательного кода так, чтобы использовать все преимущества параллелизма приложения и максимально эффективно тратить аппаратные ресурсы, минимизируя время выполнения. Без соответствующей поддержки архитектуры такое переупорядочивание может нарушить корректность программы. Таким образом, поскольку EPIC возлагает создание POE на компилятор, она должна обеспечивать еще и архитектурные возможности, поддерживающие интенсивное переупорядочивание кода во время компиляции.

Следующим принципом является использование компилятором вероятностных оценок. Компилятор EPIC сталкивается с серьезной проблемой при создании плана выполнения: информация определенного типа, которая существенно влияет на запись исполнения, становится известна только лишь в момент выполнения программы. Например, компилятор не может точно знать какая из ветвей после оператора перехода будет выполняться, когда запланированный код пройдет базовые блоки и какой из путей графа будет выбран. Кроме того, обычно невозможно создать статический план, который одновременно оптимизирует все пути в программе. Неоднозначность также возникает и в тех случаях, когда компилятор не может решить, будут ли ссылки указывать на одно и то же место в памяти. Если да, то обращение к ним должно осуществляться последовательно; если нет, то их можно запланировать в произвольном порядке. При такой неоднозначности часто наиболее вероятен некий конкретный результат. Одним из важнейших принципов EPIC в данной ситуации является возможность разрешения компилятору оперировать вероятностными оценками — он создает и оптимизирует POE для наиболее вероятных случаев. Однако EPIC обеспечивает архитектурную поддержку, такую как спекулятивное выполнение по управлению и по данным (Control and Data Speculation), с тем, чтобы гарантировать корректность программы, даже если исходные предположения были не верны. Когда предположение оказывается неверным, совершенно очевидно падение производительности при выполнении программы. Такой эффект производительности иногда виден на плане программы, к примеру, в тех случаях, когда существует высоко оптимизированная программная область, а код исполняется в менее оптимизированной. Также падение производительности может возникнуть в моменты «остановки» (Stall), которые на плане программы не видны — определенные операции, подпадающие под наиболее вероятный и, следовательно, оптимизированный случай, выполняются при максимальной производительности, но приостанавливают процессор для того, чтобы гарантировать корректность, если возникнет менее вероятный, не оптимизированный случай.

когда инициировать каждую операцию и какие ресурсы использовать (в частности, должен существовать способ указать, какие операции инициируются одновременно). В качестве альтернативного решения компилятор мог бы создавать последовательную программу, которую процессор динамически реорганизует с тем, чтобы получить требуемую запись. Но в таком случае цель, сводимая к освобождению аппаратного обеспечения от динамического планирования, не достигается. При передаче POE аппаратному обеспечению крайне важно своевременно предоставить необходимую информацию. Примером этому может служить операция перехода, которая, в случае ее использования, требует, чтобы по адресу перехода команды выбирались с упреждением, заведомо до того, как будет инициирован сам переход. Вместо того, чтобы решение о том, когда это нужно сделать и какой адрес перехода отдавать на откуп аппаратному обеспечению, такая информация в соответствии с основными принципами EPIC передается аппаратному обеспечению точно и своевременно через код. Микроархитектура принимает и другие решения, не связанные напрямую с выполнением кода, но которые влияют на время выполнения. Один из таких примеров — управление иерархией кэш-памяти и соответствующие решения о том, какие данные нужны для поддержки иерархии, а какие следует заменить. Такие правила обычно предусматриваются алгоритмом функционирования контроллера кэша. EPIC расширяет принцип, утверждающий, что план выполнения компилятор создает так, чтобы тоже иметь возможность управлять этими механизмами микроархитектуры. Для этого обеспечиваются архитектурные возможности, позволяющие осуществлять программный контроль механизмами, которыми обычно управляет микроархитектура.

Аппаратно-программный комплекс VLIW

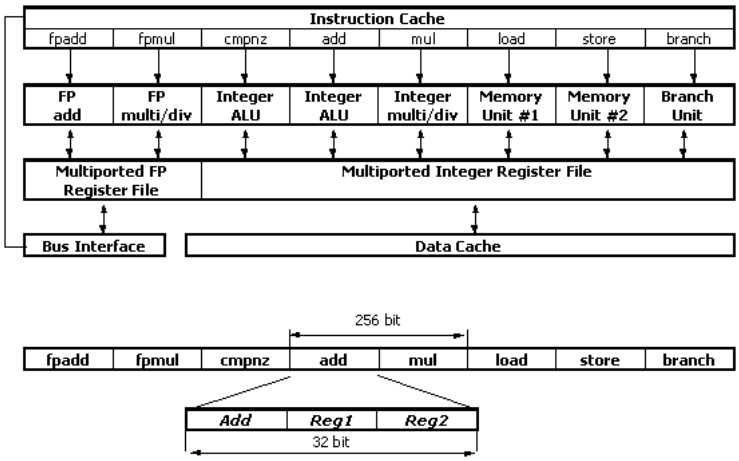
Архитектура VLIW представляет собой одну из реализаций концепции внутреннего параллелизма в микропроцессорах. Их быстродействие можно повысить двумя способами: увеличив либо тактовую частоту, либо количество операций, выполняемых за один такт. В первом случае требуется применение «быстрых» технологий (например, использование арсенида галлия вместо кремния) и таких архитектурных решений, как глубинная конвейеризация (конвейеризация в пределах одного такта, когда в каждый момент времени задействованы все логические блоки кристалла, а не отдельные его части). Для увеличения количества выполняемых за один цикл операций необходимо на одном чипе разместить множество функциональных модулей обработки и обеспечить надежное параллельное исполнение машинных инструкций, что дает возможность включить в работу все модули одновременно. Надежность в таком контексте означает, что результаты вычислений будут правильными. Для примера рассмотрим два выражения, которые связаны друг с другом следующим образом: A=B+C и B=D+E. Значение переменной A будет разным в зависимости от порядка, в котором вычисляются эти выражения (сначала A, а потом B, или наоборот), но ведь в программе подразумевается только одно определенное значение. И если теперь вычислить эти выражения параллельно, то на правильный результат можно рассчитывать лишь с определенной вероятностью, а не гарантировано.

Планирование порядка вычислений — довольно трудная задача, которую приходится решать при проектировании современного процессора. В суперскалярных архитектурах для распознавания зависимостей между машинными инструкциями применяется специальное довольно сложное аппаратное решение (например, в P6- и post-P6-архитектуре от Intel для этого используется буфер переупорядочивания инструкций — ReOrder Buffer, ROB). Однако размеры такого аппаратного планировщика при увеличении количества функциональных модулей обработки возрастают в геометрической прогрессии, что, в конце концов, может занять весь кристалл процессора. Поэтому суперскалярные проекты остановились на отметке 5-6 обрабатываемых за цикл инструкций. На самом же деле, текущие реализации VLIW тоже далеко не всегда могут похвастаться 100% заполнением пакетов — реальная загрузка около 6-7 команд в такте — примерно столько же, сколько и лидеры среди RISC-процессоров. При другом подходе можно передать все планирование программному обеспечению, как это делается в конструкциях с VLIW. «Умный» компилятор должен выискать в программе все инструкции, которые являются совершенно независимыми, собрать их вместе в очень длинные строки (длинные инструкции) и затем отправить на одновременное исполнение функциональными модулями, количество которых, как минимум, не меньше, чем количество операций в такой длинной команде. Очень длинные инструкции (VLIW) обычно имеют размер 256-1024 bit, однако, бывает и меньше. Сам же размер полей, кодирующих операции для каждого функционального модуля, в такой метаинструкции намного меньше.

Логический слой VLIW-процессора

Процессор VLIW, имеющий схему, представленную ниже, может выполнять в предельном случае восемь операций за один такт и работать при меньшей тактовой частоте намного более эффективнее существующих суперскалярных чипов. Добавочные функциональные блоки могут повысить производительность (за счет уменьшения конфликтов при распределении ресурсов), не слишком усложняя чип. Однако такое расширение ограничивается физическими возможностями: количеством

количества функциональных блоков. К тому же компилятор должен распараллелить программу до необходимого уровня, чтобы обеспечить загрузку каждому блоку — это, думается, самый главный момент, ограничивающий применимость данной архитектуры.

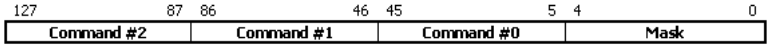


Эта гипотетическая инструкция имеет восемь операционных полей, каждое из которых выполняет традиционную трехоперандную RISC-подобную инструкцию типа <регистр приемника> = <регистр источника 1> — <операция> — <регистр источника 2> (типа классической команды MOV AX BX) и может непосредственно управлять специфическим функциональным блоком при минимальном декодировании.

Для большей конкретности рассмотрим кратко IA-64 как один из примеров воплощения VLIW. Со временем эта архитектура способна вытеснить x86 (IA-32) не только на рынке, но вообще как класс, хотя это уже удел далекого будущего. Тем не менее, необходимость разработки для IA-64 весьма сложных компиляторов и трудности с созданием оптимизированных машинных кодов может вызвать дефицит специалистов, работающих на ассемблере IA-64, особенно на начальных этапах, как самых сложных.

Наиболее кардинальным нововведением IA-64 по сравнению с RISC является явный параллелизм команд (EPIC), вносящий некоторые элементы, напоминающие архитектуру сверхдлинного командного слова, которые назвали связками (bundle). Так, в обеих архитектурах явный параллелизм представлен уже на уровне команд, управляющих одновременной работой функциональных исполнительных устройств (или функциональных модулей, или просто функциональных устройств, ФУ).

В данном случае связка имеет длину 128bit и включает в себя 3 поля для команд длиной 41bit каждое, и 5-разрядный слот шаблона. Предполагается, что команды связки могут выполняться параллельно разными ФУ. Возможные взаимозависимости, препятствующие параллельному выполнению команд одной связки, отражаются в поле шаблона. Не утверждается, впрочем, что параллельно не могут выполняться и команды разных связок. Хотя, на основании заявленного уровня параллельности исполнения, достигающего шести команд за такт, логично предположить, что одновременно могут выполняться как минимум две связки.



Шаблон указывает какого непосредственно типа команды находятся в слотах связки. В общем случае однотипные команды могут выполняться в более чем одном типе функциональных устройств. Шаблоном задаются так называемые остановки, определяющие слот, после начала выполнения команд которого инструкции последующих полей должны ждать завершения. Порядок слотов в связке (более важные справа) отвечает и порядку байт (Little Endian), однако данные в памяти могут располагаться и в режиме Big Endian (более важные слева), который устанавливается специальным битом в регистре маски пользователя.

Вращение регистров является в некотором роде частным случаем переименования регистров, применяемого во многих современных суперскалярных процессорах с внеочередным спекулятивным («умозрительным») выполнением команд. В отличие от них, вращение регистров в IA-64 управляется программно. Использование этого механизма в IA-64 позволяет избежать накладных расходов, связанных с сохранением/восстановлением большого числа регистров при вызовах подпрограмм и возвратах из них, однако статические регистры при необходимости все-таки приходится сохранять и восстанавливать, явно кодируя соответствующие команды.

подмножество и определяет исключительность IA-64. Все подобные команды можно подразделить на команды работы со стеком регистров, целочисленные команды, команды сравнения и работы с предикатами, команды доступа в память, команды перехода, мультимедийные команды, команды пересылок между регистрами, «разные» (операции над строками и подсчет числа единиц в слове) и команды работы с плавающей запятой.

Аппаратная реализация VLIW-процессора очень проста: несколько небольших функциональных модулей (сложения, умножения, ветвления и т.д.), подключенных к шине процессора, и несколько регистров и блоков кэш-памяти. VLIW-архитектура представляет интерес для полупроводниковой промышленности по двум причинам. Первая — теперь на кристалле больше места может быть отведено для блоков обработки, а не, скажем, для блока предсказания переходов. Вторая причина — VLIW-процессор может быть высокоскоростным, так как предельная скорость обработки определяется только внутренними особенностями самих функциональных модулей. Привлекает и то, что VLIW при определенных условиях может реализовать старые CISC-инструкции эффективнее RISC. Это потому, что программирование VLIW-процессора очень напоминает написание микрокода (исключительно низкоуровневый язык, позволяющий всесторонне программировать физический слой, синхронизируя работу логических вентилей с шинами обмена данными и управляя передачей информации между функциональными модулями).

В те времена, когда память для ПК была дорогостоящей, программисты экономили ее, прибегая к сложным инструкциям процессора x86 типа STOS и LODS (косвенная запись/чтение в/из памяти). CISC реализует такие инструкции, как микропрограммы, зашитые в постоянную память (ROM) и выполняемые процессором. Архитектура RISC вообще исключает использование микрокода, реализуя инструкции чисто аппаратным путем — фактически, инструкции RISC-процессора почти аналогичны микрокоду, используемому в CISC. VLIW делает по-другому — изымает процедуру генерирования микрокода из процессора (да и вообще стадии исполнения) и переносит его в компилятор, на этап создания исполняемого кода. В результате эмуляция инструкций процессора x86, таких как STOS, осуществляется очень эффективно, поскольку процессор получает для исполнения уже готовые макросы. Но вместе с тем, это порождает и некоторые трудности, поскольку написание достаточно эффективного микрокода — невероятно трудоемкий процесс. Архитектуре VLIW может обеспечить жизнеспособность только «умный» компилятор, который возьмет эту работу на себя. Именно это обстоятельство ограничивает использование вычислительных машин с архитектурой VLIW: пока они нашли свое применение в основном в векторных (для научных расчетов) и сигнальных процессорах.

Принцип действия VLIW-компилятора

Вновь вспыхнувший в последнее время интерес к VLIW как к архитектуре, которую можно использовать для реализации вычислений общего назначения, дал существенный толчок развитию техники VLIW-компиляции. Такой компилятор упаковывает группы независимых операций в очень длинные слова инструкций таким способом, чтобы обеспечить быстрый запуск и более эффективное их исполнение функциональными модулями. Компилятор сначала обнаруживает все зависимости между данными, а затем определяет, как их развязать. Чаще всего это делается путем переупорядочивания всей программы — разные ее блоки перемещаются с одного места в другое. Данный подход отличается от применяемого в суперскалярном процессоре, который для определения зависимостей использует специальное аппаратное решение прямо во время выполнения приложения (оптимизирующие компиляторы, безусловно, улучшают работу суперскалярного процессора, но не делают его «привязанным» к ним). Большинство суперскалярных процессоров может обнаружить зависимости и планировать параллельное исполнение только внутри базовых программных блоков (группа последовательных операторов программы, не содержащих внутри себя останова или логического ветвления, допустимых только в конце). Некоторые переупорядочивающие системы положили начало расширению области сканирования, не ограничивая ее базовыми блоками. Для обеспечения большего параллелизма VLIW-компьютеры должны наблюдать за операциями из разных базовых блоков, чтобы поместить эти операции в одну и ту же длинную инструкцию (их «область обзора» должна быть шире, чем у суперскалярных процессоров) — это обеспечивается путем прокладки «маршрута» по всей программе (трассировка). Трассировка — наиболее оптимальный для некоторого набора исходных данных маршрут по программе (для обеспечения правильного результата гарантируется не пересечение этих данных), т.е. маршрут, который «проходит» по участкам, пригодным для параллельного выполнения (эти участки формируются, кроме всего прочего, и путем переноса кода из других мест программы), после чего остается упаковать их в длинные инструкции и передать на выполнение. Планировщик вычислений осуществляет оптимизацию на уровне всей программы, а не ее отдельных базовых блоков. Для VLIW, так же как и для RISC, ветвления в программе являются «врагом», препятствующим эффективному ее выполнению. В то время как RISC для прогнозирования ветвлений использует аппаратное решение, VLIW оставляет это компилятору. Сам компилятор использует информацию, собранную им путем профилирования программы, хотя у будущих VLIW-процессоров предполагается небольшое аппаратное расширение, обеспечивающее сбор для компилятора статистических данных непосредственно во время выполнения программы, что

базовый блок, затем повторяет этот процесс для всех других возникших после этого программных веток, и так до самого конца программы. Он также умеет делать при анализе кода и другие «интеллектуальные шаги», такие как развертывание программного цикла и IF-преобразование, в процессе которого временно удаляются все логические переходы из секции, подвергающейся трассировке. Там, где RISC может только просмотреть код вперед на предмет ветвлений, VLIW-компилятор перемещает его с одного места в другое до обнаруженного ветвления (согласно трассировке), но предусматривает при необходимости возможность отката назад, к предыдущему программному состоянию. Формально ничего не мешает это же сделать и RISC процессору, просто соотношение «цена/эффективность» оказывается слишком высоким. Соответствующее аппаратное обеспечение, добавленное к VLIW-процессору, может оказать определенную поддержку компилятору. Например, операции, имеющие по несколько ветвлений, могут входить в одну длинную инструкцию и, следовательно, выполняться за один машинный такт. Поэтому выполнение условных операций, которые зависят от предыдущих результатов, может быть реализовано программным способом, а не аппаратным. Цена, которую приходится платить за увеличение быстродействия VLIW-процессора, намного меньше стоимости компиляции — именно поэтому основные расходы приходятся на сами компиляторы.

VLIW: обратная сторона медали

Тем не менее, при реализации архитектуры VLIW возникают и другие серьезные проблемы. VLIW-компилятор должен в деталях «знать» внутренние особенности архитектуры процессора, опускаясь до устройства самих функциональных блоков. Как следствие, при выпуске новой версии VLIW-процессора с большим количеством обрабатывающих модулей (или даже с тем же количеством, но другим быстродействием) все старое программное обеспечение может потребовать полной перекомпиляции. Производители VLIW-процессоров обрекли себя на, как минимум, не уменьшение ширины пакета, хотя бы ради того, чтобы «старые» программы могли гарантированно исполнятся на новых устройствах. Например, на настоящее время существуют пакеты по 8 команд, а следующая версия не может иметь в реализации всего шесть функциональных устройств даже ради двух-трех- кратного прироста по частоте. Кроме того, что программа, скомпилированная для восьмиканального VLIW, не сможет без специальных дорогостоящих (и в плане сложности, и в плане производительности) аппаратных решений исполняться на шестиканальной архитектуре, придется радикально переписывать и компилятор. С этой точки зрения представляется разумным использование Intel трехкомандного слова в системе IA64 — такое, на первый взгляд, неудобное ограничение позволяет в будущем довольно свободно варьировать число исполнительных устройств в процессорах IA64. И если при переходе с 386 на процессор 486 производить перекомпиляцию имеющегося ПО было совершенно ненужно, то теперь придется. В качестве одного из возможных компромиссных решений предлагается разделить процесс компиляции на две стадии. Все программное обеспечение должно готовиться в аппаратно-независимом формате с использованием промежуточного кода, который окончательно транслируется в машинно-зависимый код только в процессе установки на оборудовании конечного пользователя. Пример такого подхода демонстрирует фонд OSF со своим архитектурно-независимым форматом ANDF (Architecture-Neutral Distribution Format). Но кроссплатформенное программное обеспечение пока что не оправдывает когда-то возлагавшихся на него радужных надежд. Во-первых, оно все равно требует наличия портов, которые «объясняют» компилятору в каждом конкретном случае что следует делать при компиляции данной программы именно на этой платформе (и даже именно на этой ОС). Во-вторых, кроссплатформенное ПО пока отнюдь не является «Speed Demon», а даже наоборот, как правило работает медленнее, чем написанное под конкретную платформу аналогичного класса приложения.

Другая трудность — это по своей сути статическая природа оптимизации, которую обеспечивает VLIW-компилятор. Трудно предугадать как, например, поведет себя программа, когда столкнется во время компиляции с непредусмотренными динамическими ситуациями, такими как ожидание ввода/вывода. Архитектура VLIW возникла в ответ на требования со стороны научно-технических организаций, где при вычислениях особенно необходимо большое быстродействие процессора, но для объектно-ориентированных и управляемых по событиям программ она менее подходит, а ведь именно такие приложения составляют сейчас большинство в сфере Информационных Технологий. Остается труднопредполагаемой проверка, что компилятор выполняет такие сложные преобразования надежно и правильно. Напротив, Out-Of-Order RISC-процессоры вполне способны «адаптироваться» под конкретную ситуацию самым выгодным образом.

Сейчас уже стало очевидным, что разработчики современных быстродействующих VLIW-процессоров начинают отходить от своей первоначальной затеи с чистой VLIW-архитектурой. Intel, по крайней мере, сохранила семейству Itanium возможность исполнять классический x86-код, правда, судя по всему, удалось ей это сделать ценой колоссальных потерь производительности при работе процессора в таком режиме. Можно предположить, что процессорный гигант сдал бастион «чистого VLIW» под нажимом гигантов ПО. Однако решение сложной задачи обеспечения взаимодействия аппаратного и программного обеспечения в архитектуре VLIW требует серьезных предварительных исследований.

правило, если требуется грамотно распараллелить программу (к примеру, для введения в нее SMP-оптимизированных участков), делается это по старинке — «руками», причем руками весьма и весьма квалифицированных программистов. К тому же давно известен немалый список задач, поддающихся распараллеливанию с громадными усилиями или даже не поддающихся вообще.

Как пример, можно вспомнить знаменитый Crusoe от Transmeta. Принцип его работы фактически состоит в динамической перекомпиляции под VLIW-архитектуру уже скомпилированного x86 кода. Однако, если даже в создании эффективных VLIW-компиляторов для языков высокого уровня разработчики сталкиваются с такими большими сложностями, то что можно сказать о «компиляторе» весьма хаотичного и непредсказуемого машинного кода x86, тем более, что он зачастую опять-таки оптимизирован при компиляции, но в расчете на совсем другие архитектурные особенности.

Реально же Crusoe применяется только в режиме эмуляции x86, хотя принципиальных ограничений на эмуляцию любого другого кода нет. При этом все программы и сама операционная система работают поверх низкоуровневого программного обеспечения, называемого морфингом кода (Code Morphing) и ответственного за трансляцию x86-кодов в связки, имеющие размер 128bit. Crusoe использует также собственную терминологию, тонко определяющую уровень логического слоя архитектуры, в которой связки называются молекулами (Molecule), а 32bit подкоманды, располагаемые в связке, — атомами (Atom). Каждая молекула содержит два или четыре атома. Для совместимости с форматом команд, если при двоичной трансляции не удается заполнить все «атомные» слоты молекулы, в незаполненные поля должна вставляется пустая подкоманда NOP, указывающая на отсутствие операции (No OPeration). В результате, благодаря применению двух типов молекул, в каждой из них оказывается не более одной NOP. Всего же за такт могут исполняться до четырех подкоманд VLIW. Среди важных архитектурных особенностей VLIW-ядра Crusoe выделяются относительно короткие конвейеры: целочисленный на семь стадий и на десять с плавающей запятой. Теоретически данный процессор может применяться для эмуляции различных архитектур, однако ряд особенностей его микроархитектуры нацелен на эффективную эмуляцию именно x86 кода. Принципиально важным отличием является практически отсутствие потерь производительности в условиях примерно равной частоты процессоров при x86-эмуляции. Сначала устройство декодирует x86-последовательность в режиме интерпретации «байт за байтом», однако если код выполняется несколько раз, то механизм морфинга транслирует его в оптимальную последовательность молекул, а результат трансляции кэшируется для повторного использования.

Можно сказать, что Transmeta попыталась не то, что опередить время, а даже «перескочить через голову» уже и так опередившей время технологии. В принципе же то, что сделала Transmeta — просто фантастическое техническое достижение. По сути дела, продемонстрирована работающая в реальном времени технология динамической компиляции кроссплатформенного ПО. Если так работает система, переводящая «на лету» x86 код во внутреннее представление, то остается только догадываться как бы она работала с первоначально ориентированной для нее программой. Таким образом, бинарная совместимость возможна и довольно эффективна.

С другой стороны, даже сам Гордон Мур уже заявил, что рост частоты процессоров в ближайшее время скорее всего перестанет подчиняться сформулированному им эмпирическому правилу, и существенно замедлится. В этой ситуации производительность классических систем рискует довольно скоро упереться в «тупик» физических ограничений, что в еще большей степени подхлестнет развитие альтернативных повышению частоты способов увеличения быстродействия, одним из которых является разработка принципиально новых и хорошо распараллеливающихся алгоритмов, что и позволит проявить себя VLIW во всей красе.

Вместо заключения

За последние годы сравнение достоинств VLIW и суперскалярных архитектур было основной темой в дискуссиях специалистов по реализации параллелизма на уровне команд (ILP). Сторонники той и другой концепции сводят это обсуждение к противопоставлению простоты и ограниченных возможностей VLIW, и динамическим возможностям суперскалярных систем. Но, как уже было замечено ранее, такое сравнение в корне неверно. Совершенно очевидно, что оба подхода имеют свои достоинства и говорить об их альтернативности не уместно. Создание плана выполнения во время компиляции существенно для обеспечения высокой степени распараллеливания на уровне команд, даже для суперскалярного процессора. Также ясно и то, что во время компиляции существует неоднозначность, которую можно разрешить только во время исполнения, и для решения этой задачи процессор требует наличия динамических механизмов. Сторонники EPIC согласны с обеими этими позициями — различие только в том, что EPIC предоставляет эти механизмы на уровне архитектуры так, что компилятор может управлять такими динамическими механизмами, применяя их выборочно где это возможно. Столь широкие возможности помогают компилятору использовать правила управления этими механизмами более оптимально, чем это позволяет аппаратура.

Основные принципы EPIC, наряду с возможностями самой архитектуры, которая их поддерживает

прикладных областях. Говоря более конкретно, IA-64 — это пример того, как принципы EPIC могут применяться к вычислительными системам общего назначения — области, где совместимость объектного кода имеет критически важное значение. Однако есть уверенность, что со временем EPIC будет играть столь же важную роль и на рынке высокопроизводительных встроенных систем. В этой области более жесткие требования к соотношению цена/производительность и при этом более низкие требования к совместимости на уровне объектных модулей, что заставляет использовать более «гибкие» архитектуры. Понимая данную ситуацию, HP Labs начала реализацию исследовательского проекта PICO (Program In, Chip Out), в рамках которого уже разработан прототип, где наряду с другими возможностями на основе встроенных приложений, написанных на стандартном языке C, можно автоматически проектировать архитектуру и микроархитектуру процессора EPIC, адаптированного к конкретной задаче. Таким образом, EPIC дает новые надежды на устойчивый рост производительности микропроцессоров общего назначения, выполняя конкретные приложения без кардинального переписывания кода.

Главной проблемой VLIW-процессоров по-видимому является противоречие между, по большей части, последовательной логикой приложений и природой подавляющего большинства вычислительных алгоритмов, с одной стороны, и параллельной природой исполнения VLIW-процессора — с другой. Ограничение числа функциональных устройств в классических процессорах вызвано скорее всего не только и не столько невозможностью организовать эффективное внеочередное исполнение, а просто-напросто ограниченным количеством способных исполняться независимо команд.

Хотя, если не ограничивать оптимизатор жесткими временными рамками (как в классических суперскалярных процессорах), то, наверное, на некоторых алгоритмах можно добиться достаточно плотной упаковки инструкций, но только на некоторых. Примером, в частности, могут выступать программы, управляемые по событию, с постоянно меняющимся набором данных, которые статической оптимизации поддаются с огромным трудом. Вполне можно предположить ситуацию, когда изменение набора данных приведет к фатальному падению производительности, например, при работе с базами данных. Способ предотвратить такое развитие событий видится такой — при компиляции программы «пройти» все ветки алгоритма во всех возможных сочетаниях данных, сгенерировав оптимальный код для каждой возможной ситуации. Но, очевидно, что такое решение повлечет за собой вполне объяснимый рост размеров скомпилированного приложения, а необходимость подгружать из ОЗУ на исполнение различные (по ситуации) куски оптимального кода понятным образом увеличит нагрузку и на подсистему памяти. В этой связи логичным представляется рост размеров кэш-памяти, например, у настоящего Itanium2 по сравнению с Itanium — и это несмотря на увеличенную в несколько раз пропускную способность шины данных.

Определенный интерес представляет собой теоретическая возможность упаковки в одну связку инструкций, принадлежащих, например, различным ветвям алгоритма с целью одновременного их исполнения с выбором в последствии актуального результата. Такой способ может исключить саму идею предсказания ветвлений, при этом оказавшись сравнительно упрощенным с точки зрения производительности, но размер кода и нагрузка на память вырастут несоизмеримо. С другой стороны, если ветвление не бинарное а больше, то в таком случае увеличится и нагрузка на устройства декодирования команд — при возросшей плотности подгрузки инструкций (обе ветки вместо одной) могут возникнуть, например, ситуации конкуренции за максимально возможную пропускную способность основной шины (FSB — Front Side Bus) и/или кэш-памяти (BSB — Back Side Bus).

Нельзя не отметить тот факт, что предварительное планирование последовательности операций в большем числе случаев, чем для классических процессоров, позволит максимально полно использовать реальную пропускную способность шины данных. Но столь высокая и прогнозируемая эффективность использования интерфейса требует, с другой стороны, принципиально нового подхода к построению системы.

Самым же «интересным» (и опасным для концепции) аспектом развития VLIW-архитектур можно считать проблему статической оптимизации кода в сочетании с возможными изменениями внутренней архитектуры процессоров. Вполне очевидно, что оптимизированная для одного устройства программа может оказаться совершенно непригодной (малоэффективной) для работы на следующем поколении процессоров. Тем не менее, существует метод, позволяющий изящно обойти данную проблему. Суть идеи состоит в том, что распространяемый «дистрибутив» приложения содержит не готовый к исполнению код, и не исходные тексты (по идее OSF), а результат некоторого промежуточного этапа компиляции: с описанными зависимостями, формализованным описанием процессов, развернутыми циклами — в общем содержит именно то, что традиционно считается самой сложной частью работы компилятора и, собственно, определяет качество его работы. Инсталлятор в процессе установки программы вызывает некий специфический для данного процессора компилятор, преобразующий этот промежуточный код в исполняемый — причем, оптимальным для данного процессора (и более того — конфигурации) образом. Развивая данную мысль, можно предложить вариант составления финального компилятора (опций его работы) на основании библиотек, описывающих используемые подсистемы. Итогом такого подхода может стать, во-первых, «выжимание» максимально возможной

сможет быть установлена на любой процессор, если только есть набор требуемых драйверов (описаний). Однако это слишком идеальная схема.

Альтернативой предложенным вариантам можно назвать Crusoe-подобную систему с динамической компиляцией приложений, поставляемых в виде промежуточного кода непосредственно во время исполнения. С одной стороны, очевидно, что производительность из-за дополнительной нагрузки в виде динамической компиляции будет ниже, а с другой — простота такого решения и его очевидно большая гибкость (например, динамический транслятор может учитывать поведение одновременно исполняющихся в системе приложений) делают такой способ достаточно привлекательным. Опять-таки, подобный динамический компилятор вполне может (и даже должен) руководствоваться во время работы информацией об имеющемся оборудовании.

Оглядываясь назад, можно предположить с довольно большой вероятностью, что Transmeta «погубило» нежелание использовать в полной мере существующие современные технологии памяти. Вполне очевидно, что при динамической компиляции приложений нагрузка на ОЗУ увеличится в разы. Иными словами, используя подсистему памяти с невысокой пиковой пропускной способностью, Transmeta обрекла себя на огромный проигрыш в приложениях, интенсивно обращающихся к системному ОЗУ, тогда как в производительности приложений, большую часть времени проводящих внутри процессора, практически паритет. Будущий путь Transmeta, рассуждая логически, должен быть направлен на расширение количества функциональных устройств процессора с возможностью параллельного исполнения двух и более веток программы (разных программ) в одном ядре, используя более масштабируемую и быстродействующую системную память.

В статье использовались материалы, опубликованные на сайте издательства Открытые системы
Огромная благодарность за неоценимую помощь в подготовке и обработке материала:
Виктору Картунову** aka **matik
Олегу Бессонову** aka **bess

15 ОКТЯБРЯ 2002 Г. | ИЛЬЯ ВАЙЦМАН МАКСИМ ЛЕНЬ

0 комментариев

Написать комментарий

ОТПРАВИТЬ

Добавить комментарий

Обсуждения в конференции:

[16:26] Отечественные микропроцессоры. Состояние и перспективы (часть 25) (4541 сообщений)

[16:05] Подержанные Intel Xeon и другие серверные процессоры (часть 2) (266 сообщений)

[14:26] Процессоры AMD RYZEN - возникающие проблемы и их решения. Техподд… (544 сообщений)

[14:13] AMD vs Intel: что лучше? Тестируем, сравниваем. Приветствуется те… (268 сообщений)

[13:13] Предел техпроцесса (1729 сообщений)

[12:57] 1366 в 2019? (182 сообщений)

[12:40] Клуб любителей AMD, Intel, Apple и др. фирм (бывшая CPU-флудилка)… (360 сообщений)