Программная настраиваемость аппаратной структуры

Виктор Корнеев

Среди существующих путей увеличения производительности, таких как наращивание тактовой частоты, повышение параллелизма и программная настраиваемость структуры для аппаратной реализации вычислений, первые два уже хорошо освоены, а третий только начинает применяться. Однако первые два пути уже практически исчерпали возможности дальнейшего повышения производительности без применения их в комбинации с программной настраиваемостью структуры. Поэтому для построения современных высокопроизводительных систем необходимо применение реконфигурируемых структур, настраиваемых на исполнение алгоритмов.

Программная настраиваемость структуры для аппаратной, или схемной, реализации вычислений сегодня применяется в основном в программируемых логических схемах, а не в процессорных и многопроцессорных СБИС и не на уровне вычислительных систем в целом. Нынешние терафлопные показатели производительности достигнуты за счет построения параллельных систем из серийно выпускаемых компонентов, таких как универсальные микропроцессоры и специально разработанные или коммерчески доступные коммуникационные среды. Для дальнейшего повышения производительности необходимы реконфигурируемые структуры, настраиваемые на исполнение алгоритмов.

В той или иной степени на всем протяжении развития средств вычислительной техники применялись все способы увеличения производительности, сформулированные еще в начале 60-х годов (повышение тактовой частоты и параллелизма, программная настраиваемость структуры для аппаратной реализации вычислений), однако в их акцентированном применении прослеживается этапность, обусловленная, с одной стороны, степенью развития элементной базы, а с другой — сохранением удобства и продуктивности программирования [1]. Освоив некоторый метод — а точнее, модель создания программ, большинство программистов склонны ее применять и совершенствовать с целью увеличения производительности труда и скорости выполнения создаваемых программ и не стремятся переходить на другую модель программирования. Это обстоятельство обусловливает консерватизм развития архитектур в ущерб производительности, достигаемой на используемой элементной базе. Однако физические свойства элементной базы определяют возможности создаваемых вычислительных средств и диктуют архитектурные решения, требующие смены модели программирования.

Первый этап развития средств вычислительной техники преимущественно определялся ростом тактовой частоты, что сохраняло модель создания программ на базе архитектуры компьютера с последовательным исполнением команд. Этот этап сопровождался расцветом алгоритмических языков программирования, так как повышение эффективности и удобства программирования представлялось в создании типизированных, объектно-ориентированных, функциональных и других высокоуровневых языков. Было предложено много языков высокого уровня, но цель не была достигнута, так как для повышения производительности требовалось привлечение низкоуровневых средств — команд процессора. Это связано с тем, что для преодоления постоянно увеличивающегося разрыва в быстродействии логических схем и элементов памяти пришлось вводить такие механизмы, как виртуальная память, таблицы отображения виртуальной памяти в физическую, многоуровневая кэш-память, таблицы предсказаний направлений переходов и др. Все эти механизмы, призванные сохранить удобство программирования, потребовали дополнительного оборудования к тому, которое непосредственно занято обработкой данных, что само по себе снизило показатель максимально достижимой производительности на единицу оборудования. Но, кроме того,

единообразно автоматически функционирующие механизмы замены страниц виртуальной памяти и строк кэш-памяти вызывают при исполнении ряда программ неприемлемо большие потери производительности [2, 3], поэтому в систему команд процессоров были добавлены команды управления содержимым кэш-памяти. Это в совокупности с возможностью управления замещением страниц виртуальной памяти позволяет программисту задать перемещение данных между уровнями памяти в соответствии с потребностями алгоритма обработки.

Повышение производительности на первом этапе развития средств вычислительной техники достигалось также за счет использования параллелизма на уровне функциональных устройств и реализации конвейерной обработки в векторных процессорах. Чтобы обеспечить эффективную загрузку функциональных устройств, потребовался большой объем оборудования для реализации суперскалярной архитектуры, а неудовлетворенность достигаемым на этом пути результатом породила архитектуру VLIW-процессоров с длинным командным словом. В течение нескольких лет разработчикам компиляторов с языков высокого уровня удалось справиться с проблемой эффективной загрузки векторных процессоров и функциональных устройств в суперскалярных и VLIW-процессорах. Поэтому преобладала тенденция неявного для программиста использования параллелизма, скрытого компиляторами языков высокого уровня. Однако следствием столь «сдержанного» применения параллелизма стало существенное замедление в течение десятилетия — с начала 80-х до 90-х годов — темпов роста производительности вычислителей, лучшими образцами которых были системы с небольшим числом векторных процессоров (см. www.top500.org).

Второй этап развития ассоциируется с построением массово-параллельных систем из коммерчески доступных микропроцессоров с явным для программиста использованием параллелизма. Создание параллельных программ с явным программированием межпроцессорных обменов данными, в том числе с использованием библиотеки MPI [4], позволило достигнуть значений производительности, прямо пропорциональных произведению тактовой частоты на число используемых процессоров с коэффициентом пропорциональности, равным количеству результатов, получаемых процессором за один такт. Однако сам подход к программированию с явным указанием межпроцессорных обменов признается неудобным и малопродуктивным, что выразилось в эмоциональной оценке таких систем — «преждевременный терафлопс» [5].

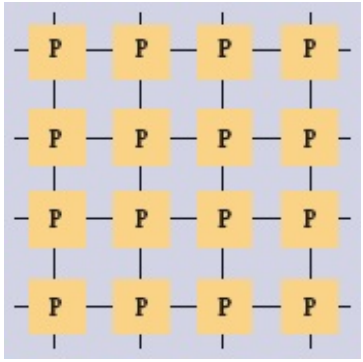
Дальнейшие попытки создать специальные языки для параллельного программирования не привели к ожидаемому повышению продуктивности труда программистов. Проблема оказалась не в средствах представления, а в модели организации параллельных вычислений, которая формируется интеллектом программиста. Поэтому в последние годы популярной становится идея «инкрементального распараллеливания», при котором вместо создания новой параллельной программы в текст последовательной программы просто добавляются специальные директивы, облегчающие компилятору принятие решения о последовательном или параллельном исполнении секций программного кода. Предполагается, что программа с этими директивами без какой-либо модификации при компиляции должна исполняться как на массово параллельных системах, так и на однопроцессорных. Эта идея использована в HPF [6], OpenMP (www.openmp.org), системе разработки параллельных программ DVM (www.keldysh.ru/dvm), созданной в Институте прикладной математики им. М. В. Келдыша РАН, и в Т-системе [7], разработанной Институтом программных систем РАН. Но в целом, несмотря на неудовлетворенность средствами библиотеки MPI, модели параллельного программирования до сих пор не создано, возможно, потому, что обработка данных и их доставка к месту обработки должны рассматриваться неразрывно.

Сегодня основными препятствиями на пути дальнейшего увеличения производительности служат ограничение скорости распространения сигналов на кристалле, а также проблемы энергопотребления и тепловыделения. Если принять, что такт процессора равен восьми задержкам вентилей с четырьмя нагрузками, то расстояние, проходимое сигналом за один такт процессора в кристалле со стороной 20 мм, составляет около 10 мм для технологии 180

нм и меньше 1 мм для 35 нм, что затрудняет синхронизацию работы блоков, размещенных в разных частях кристалла.

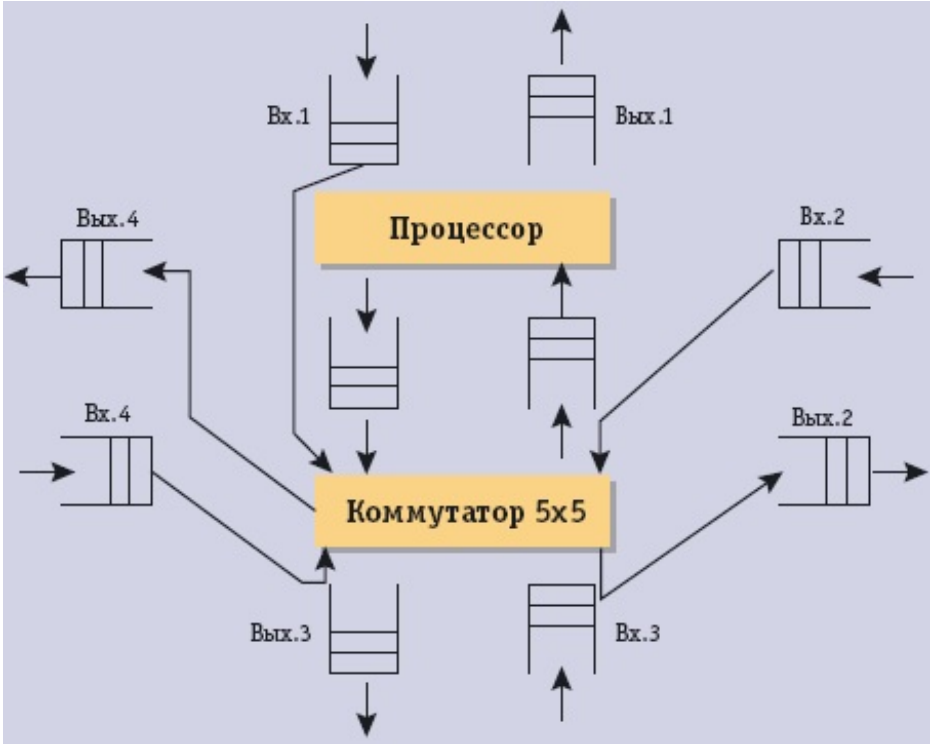
Проблема роста энергопотребления и тепловыделения связана с тем, что большая часть энергии начинает потребляться проводниками. Уже при технологических нормах, равных 0,13 мкм, 64-разрядный блок операций с плавающей запятой занимает площадь менее 1 мм^2 и расходует около 50 пДж энергии на одну операцию на тактовой частоте 500 МГц, в то время как передача данных на расстояние, равное одной из сторон кристалла 14x14 мм^2, требует около 1 пДж, то есть в 20 раз больше [8]. Кроме того, рассеиваемая мощность на единицу площади современных кристаллов достигает значений, делающих невозможным дальнейший рост частоты для традиционных процессоров с единой сеткой тактовых сигналов, разведенной по всему кристаллу.

Нельзя сказать, что эта реальность развития микроэлектроники осознана только недавно — основные положения были изложены еще в [1], а в [9] была рассмотрена архитектура и организация функционирования вычислительной системы с программируемой структурой на базе многопроцессорных кристаллов. При этом коммуникации между процессорами как внутри кристаллов, так и между процессорами разных кристаллов основаны на принципе близкодействия, что позволяет преодолеть ограничения на рост тактовой частоты за счет использования только коротких проводников.



При структуре кристалла, изображенной на рис. 1, тактовый сигнал должен синхронно распространяться только внутри области кристалла, занимаемой одним процессорным ядром (Р). Все процессорные ядра функционируют на одной тактовой частоте, но между ними возможен произвольный сдвиг фаз тактовых сигналов. Поэтому линии связи между процессорными ядрами должны обеспечивать асинхронную передачу данных. Собственно ядро процессора (рис. 2) состоит из обрабатывающего блока (процессор) и коммуникационного блока, который в свою очередь включает в себя входные и выходные очереди портов процессорного ядра и коммутатор. Коммуникационный блок служит, во-первых, для передачи в процессор программ и данных и приема из процессора результатов, а во-вторых, для транзитных передач программ и данных в соседние процессорные ядра.

Как при использовании одиночных многоядерных кристаллов, так и при построении вычислительных систем на их базе возникает проблема выбора модели программирования. Для выполнения задания в [9] предлагается формировать подсистемы процессорных ядер. Процессорные ядра подсистемы могут программно набираться в требуемом количестве из совокупности кристаллов, при этом выбираются все процессорные ядра каждого кристалла или только часть их. Каналы, установленные как между процессорными ядрами внутри кристаллов, так и между процессорными ядрами разных кристаллов, обеспечивают потоковую аппаратную реализацию алгоритма исполняемого задания, параллельные ветви которого загружены в каждое из процессорных ядер подсистемы. Такой способ выполнения вычислений является вариантом программной настраиваемости структуры.



Программная настраиваемость структуры для аппаратной (схемной) реализации вычислений имеет широкий спектр воплощений, определяемый выбором элементарных блоков, между которыми возможно установить реконфигурируемые соединения с целью формирования требуемой электронной схемы. Один край спектра образуют уже упомянутые *многоядерные кристаллы*, элементарные блоки которых устанавливаются в компьютеры, имеющие процессор, собственную память и интерфейсы ввода/вывода.

На другом краю спектра находятся *программируемые логические интегральные схемы (ПЛИС)*, элементарные блоки которых способны реализовать любую логическую функцию от небольшого числа переменных, а для соединения этих блоков служат трассы, проложенные между строками и столбцами элементарных блоков. Функции, реализуемые каждым блоком ПЛИС, и подсоединение блоков к трассам задаются настроечной битовой последовательностью, загружаемой в специальную память конфигурации ПЛИС. Различные настроечные битовые последовательности и определяют множество схем, реализуемых ПЛИС.

С применением ПЛИС можно реализовать любое функциональное устройство, но объем оборудования (используемая площадь кристалла) и энергопотребление при этом будут на порядок больше, чем необходимо для реализации этого устройства как фиксированной схемы. Поэтому при построении систем с аппаратной реализацией вычислений важно выбрать компромисс между функциями, осуществляемыми в программируемой логике и фиксированной, что и является предметом современных интенсивных исследований в области архитектуры высокопроизводительных компьютеров.

Сегодня активно ведутся разработки как существенно многоядерных кристаллов, так и архитектур систем с программируемой структурой. В качестве примера развития первого направления можно указать на проект Intel Terascale, а второе направление представлено проектом системы Merrimac [8].

В рамках проекта Terascale изготовлен кристалл Teraflops Research Chip размером приблизительно 22x14 мм^2, состоящий из 80 процессорных ядер, расположенных на кристалле в виде прямоугольного массива 8x10. Каждое процессорное ядро содержит вычислительный блок (два АЛУ для выполнения операций с плавающей запятой) и пятипортовый коммутатор, обеспечивающий передачу данных и команд в другие процессорные ядра. Оценки Intel применительно к проекту Terascale показывают, что

процессорное ядро традиционной архитектуры занимает площадь 21 мм^2 и обеспечивает удельную производительность 1,5 GFLOPS/мм^2, в то время как каждое ядро кристалла с 8x10 процессорными ядрами занимает только 6 мм2 при удельной производительности 6,4 GFLOPS/мм^2.

Для снижения энергопотребления вычислительный блок каждого ядра запитывается независимо от коммутатора, что позволяет отключать питание АЛУ ядра, если оно не участвует в решении текущей задачи. В этом случае ядро будет выполнять только функцию канала передачи данных в соседние ядра.

В Teraflops Research Chip воплощена еще одна интересная идея — сделать доступным каждому ядру 256 Мбит статической памяти, расположив ее на другом кристалле, поверх процессорного, и соединив каждый процессорный элемент с соответствующим ему блоком памяти посредством высокоскоростного канала.

В проекте Merrimac процессорные ядра через файлы потоковых регистров (Stream Register File, SRF), выполняющих роль очередей, связаны коммуникационной сетью друг с другом и блоками памяти. При этом обеспечена возможность передачи потока данных с выхода одного процессорного ядра на вход другого ядра или блока памяти и, соответственно, получения потока данных от другого ядра или из памяти. Одна команда задает передачу потока между SRF и памятью, которая совмещается с обработкой в функциональных устройствах.

Язык Brook, предложенный для программирования Merrimac, представляет собой расширение Си двумя ключевыми словами — stream («поток») и kernel («ядро»), а также библиотекой времени исполнения, включающей в себя функции для порождения потоков и манипулирования ими [10]. Поток — это последовательность записей одного типа. Ядра обрабатывают элементы записей потока и формируют выходные потоки. Для поддержки многомерных массивов в язык введен атрибут потока shape («форма»), используемый для задания шаблона выборки данных вокруг элемента потока. Программа на языке Brook состоит из операторов перемещения данных и обрабатывающих ядер.

Преобразование программ на традиционных языках Фортран и Си в Brook предусматривает два шага. На первом должны быть определены данные, которые преобразуются в потоки. На втором — выделены вычислительные ядра, которые должны обрабатывать и формировать потоки, созданные на первом шаге. Разделение программы на последовательности доступа к данным по задаваемым шаблонам и собственно вычислительные процедуры делает код программы понятным и легко исполняемым.

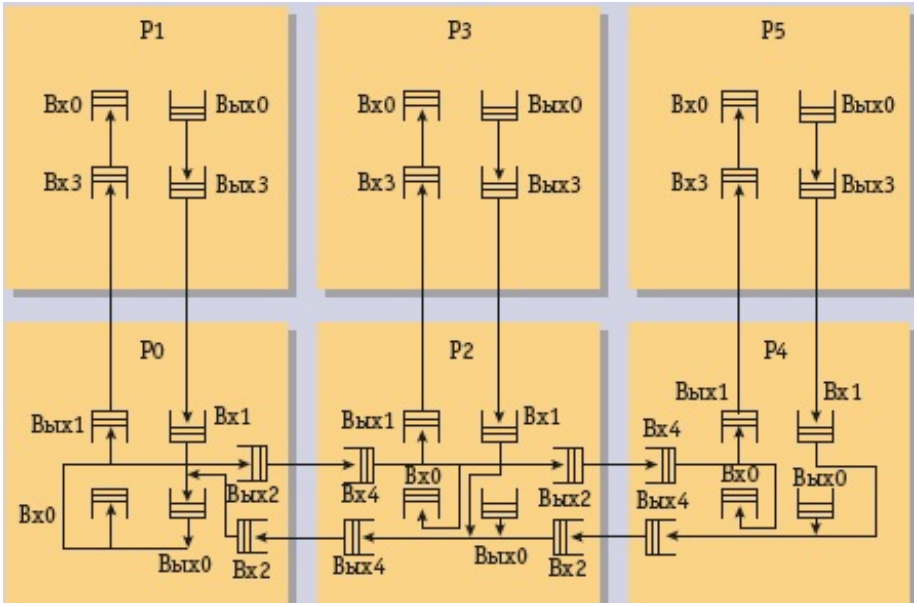
В работе [9] предложено создавать параллельные программы, в которых межъядерные потоки программируются исходя из параметрического описания графов связей подсистем, на которых эти программы способны выполняться. Для выполнения таких параллельных программ операционная система должна сформировать связную подсистему процессорных ядер с требуемым программой типом графа межъядерных связей и алгоритмом нумерации ядер, приписывающим каждому ядру уникальный номер из диапазона 0, .., N-1, где N — количество ядер в подсистеме. Алгоритм нумерации должен позволять по номеру узла, на котором он исполняется, вычислять для любого i, принадлежащего множеству {0, .., N-1}, выходное направление, ведущее к ядру с номером i. При этом, если ядро имеет номер i и требуется определить выходное направление к ядру i, то результат равен 0, что означает нахождение в требуемом ядре. В случае использования сосредоточенного коммутатора, как это имеет место в Merrimac, для всех номеров узлов, кроме собственного номера, должно выдаваться одно и то же направление, ведущее к коммутатору. Для распределенных коммутаторов эти направления могут принимать значения 1, 2, 3, 4, как показано на рис. 2.

Введены четыре типа графов, для которых могут быть созданы требуемые алгоритмы нумерации: *линейка, кольцо, дерево и решетка*.

Подсистемы с типами графов *линейка и дерево* могут быть сформированы на любом связном подмножестве ядер. Ядра графов типов *линейка и дерево* нумеруются в прямом порядке нумерации вершин, а типа *решетка* — по строкам.

В каждом ядре подсистемы имеется таблица направлений, в которой для каждого направления хранится номер соседнего по этому направлению ядра и количество ядер поддерева, корнем которого служит соседнее ядро. Тогда, имея в каждом ядре его окружение {номер j, таблицу направлений T_j, N — количество ядер в подсистеме}, можно определить направление передачи к каждому ядру подсистемы.

При создании параметрически настраиваемой программы пользователь должен выбрать тип графа подсистемы, на которой будет исполняться его программа, и затем, пользуясь атрибутами окружения, запрограммировать путевую процедуру, управляющую передачами данных в подсистеме. Подход к программированию аналогичен применяемому при использовании библиотеки MPI [4] для работы с группой процессов, но принципиальное отличие состоит в том, в библиотеке MPI топология группы процессов определяет виртуальные связи между процессами группы, в то время как в рассматриваемом подходе атрибуты окружения в каждом ядре получают реальные значения в сформированной подсистеме.



Пример подсистемы, состоящей из процессорных ядер Pi, где i принадлежит множеству {0, .., 5}, с числом ядер N=6 и с типом графа дерево приведен на рис. 3. Параллельная программа состоит из двух частей: путевой и вычислительной процедур. Взаимодействие между ними выполняется через FIFO-очереди ВХ0 и ВЫХ0. Вычислительная процедура берет данные из ВХ0, обрабатывает их и помещает результат в ВЫХ0, а путевая процедура берет данные из ВЫХ0 и помещает в ВХ0. Путевая процедура в каждом ядре в соответствии с реализуемым алгоритмом задачи осуществляет выборки данных из входных очередей направлений и выходной очереди собственного процессора и передачу этих данных в выходные очереди направлений или входную очередь процессора. Тем самым, используя значения атрибутов окружения и элементы данных из очередей, в каждом ядре программно формируются потоки данных, специфичные для реализуемого алгоритма, и организуется параллельное функционирование процессорных ядер и передач данных в подсистеме.

В [9] приведены примеры создания путевых и вычислительных процедур для ряда задач линейной алгебры и математической физики с ускорением параллельных вычислений, прямо пропорциональным количеству используемых процессорных ядер при соответствующем размере задачи. При использовании 10^5 кристаллов, подобных Intel Teraflops Research Chip с производительностью на уровне 10^{12} FLOPS, производительность системы будет равна 10^{17} FLOPS.

Литература

1. Э. В. Евреинов, Ю. Г. Косарев. Однородные универсальные вычислительные системы высокой производительности. Новосибирск: Наука, 1966.
2. S. Williams, J. Shalf, L. Oliker, Sh. Kamil, P. Husbands, K. Yelick. The Potential of the Cell Processor for Scientific Computing. CF'06, May 3-5, 2006, Ischia, Italy.
3. V. Korneev, A. Kiselev. Modern Microprocessors. Third edition. Charles River Media. USA 2004.
4. MPI: A Message-Passing Interface Standard. Message Passing Interface Forum. www-unix.mcs.anl.gov/mpi.
5. G. Bell. Ultracomputers: A Teraflop Before Its Time. Communications of the ACM. 1992. Vol.35 No. 8.
6. High Performance Fortran Language Specification. dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/hpf-v20
7. С. М. Абрамов, А. А. Кузнецов, В. А. Органов. Кросс-платформенная версия Т-системы с открытой архитектурой. Вычислительные методы и программирование. 2007. Том 8.
8. W. Dally et al. Merrimac: Supercomputing with Streams SC'03, November 15-21, 2003, Phoenix, Arizona, USA.
9. В. В. Корнеев. Архитектура вычислительных систем с программируемой структурой. Новосибирск: Наука, 1985. andrei.klimov.net/reading/1985.Korneev.-.Arkhitektura.vychislitel'nykh.sistem.s.programmiruemoi.strukturoi.zip.
10. M. Fatica, A. Jameson, J. Alonso. StreamFLO: an Euler solver for streaming architectures. 42nd AIAA Aerospace Sciences Meeting and Exhibit. January 5-8, 2004. Reno, NV.

Виктор Корнеев (korv@rdi-kvant.ru) — сотрудник ФГУП НИИ «Квант» (Москва).

18.01.2008г.

Постоянный URL статьи: http://www.osp.ru/os/2007/10/4705462/