Содержание

ГЛАВА 2

Оптимизация, влияющая на поведение компьютера

Ложь, умение рассказывать прекрасные истории, каких никогда не случалось, составляет истинную цель Искусства.[1]

— Оскар Уайльд (Oscar Wilde), "Упадок лжи", *Намерения* (1891)

Цель настоящей главы — обеспечить минимум справочной информации об оборудовании компьютера для мотивации оптимизаций, описанных в этой книге, чтобы читателю не пришлось сходить с ума, углубляясь в 600-страничный справочник по процессору. Здесь приводится поверхностный обзор архитектуры процессора, позволяющий выделить некоторые эвристики для оптимизации. Очень нетерпеливый читатель может пропустить эту главу и вернуться к ней позже, когда другие главы будут ссылаться на материал из данной главы. Тем не менее приводимая здесь информация является важной и полезной для понимания остального материала книги.

Микропроцессорные устройства в настоящее время невероятно разнообразны. Они варьируются от дешевых встроенных устройств, состоящих всего лишь из нескольких тысяч логических вентилей с тактовой частотой ниже 1 МГц, до настольных устройств с миллиардами вентилей и частотами, измеряемыми в гигагерцах. Мощные ЭВМ могут иметь размер большой комнаты, содержать тысячи независимых исполнительных устройств и потреблять электроэнергию, достаточную для освещения небольшого города. Заманчиво считать, что ничто не связывает это изобилие вычислительных устройств, но в действительности они имеют очень много общего. В конце концов, если бы между ними не было никакого сходства, было бы невозможно компилировать код на C++ для множества процессоров, для которых имеются соответствующие компиляторы.

Все широко используемые компьютеры выполняют команды, хранящиеся в памяти. Эти команды обрабатывают данные, которые также хранятся в памяти. Память делится на множество небольших *слов* по нескольку битов. Несколько драгоценных слов памяти представляют собой *регистры*, которые именуются в машинных

[1] Перевод А. Зверева.

командах непосредственно. Большинство же слов памяти именуются с использованием их числового *адреса*. Определенный регистр в каждом компьютере содержит адрес следующей выполняемой команды. Если память рассматривать как книгу, то *выполняемый адрес* выглядит как палец, указывающий на следующее читаемое слово. *Исполнительное устройство* (именуемое также процессором, ядром, ЦПУ и кучей других слов) считывает поток команд из памяти и действует в соответствии с ними. Команды говорят исполнительному устройству, какие данные читать (загружать, выбирать) из памяти, что с ними делать и где в памяти записывать (запоминать, сохранять) результаты. Компьютер состоит из устройств, которые подчиняются физическим законам. Чтение или запись каждого адреса памяти занимает некоторое ненулевое количество времени, как и выполнение некоторых действий над считанными данными.

За пределами этой базовой схемы, знакомой любому первокурснику, генеалогическое дерево компьютерных архитектур демонстрирует буйный рост и изобилие ветвей. Поскольку компьютерные архитектуры весьма изменчивы, трудно сформулировать какие-либо строгие числовые правила в отношении поведения аппаратного обеспечения. Современные процессоры выполняют так много различных взаимодействующих операций, чтобы ускорить выполнение команд, что какие-то конкретные сроки их выполнения в общем случае указать невозможно. С учетом того, что многие разработчики даже не знают точно, на каких процессорах будет работать их код, лучшее, что можно ожидать в смысле оптимизации, — это некоторые эвристики.

Ложь о компьютерах, в которую верит C++

Конечно, программа на C++ по крайней мере делает вид, что верит в изложенную в предыдущем разделе версию простой модели компьютера. Существует память, адресуемая в байтах размера `char`, которая является по существу бесконечной. Существует специальный адрес, именуемый `nullptr`, который отличается от любого допустимого адреса памяти. Целое число `0` преобразуется в `nullptr`, хотя `nullptr` не обязательно указывает на адрес `0`. Существует единый концептуальный адрес выполнения, указывающий на инструкцию исходного кода, выполняемую в настоящее время. Инструкции выполняются в том порядке, в котором они написаны, с учетом действия операторов управления потоком выполнения C++.

C++ знает, что в действительности компьютеры сложнее, чем эта простая модель. Поэтому из сияющих внутренностей реализации C++ сквозь крышку пробиваются следующие сверкающие лучи.

- Программа на C++ должна лишь вести себя так, “как если бы” инструкции выполнялись в указанном порядке. Компилятор C++ и сам компьютер имеют право изменять порядок выполнения, чтобы добиться ускорения работы программы, если при этом не меняется смысл выполняемых вычислений.
- Что касается стандарта C++11, то C++ больше не считает, что существует только один адрес выполнения. Стандартная библиотека C++ теперь поставляется с возможностями запуска и остановки потоков выполнения и синхронизации

доступа к памяти между этими потоками. До C++11 программисты лгали компилятору C++ о потоках, что зачастую приводило к трудно отлаживаемым проблемам.

- Некоторые адреса памяти на самом деле вместо обычной памяти могут быть регистрами устройства. Значения в некоторых адресах могут изменяться между двумя последовательными чтениями одного и того же адреса в одном и том же потоке, отражая некоторые действия со стороны аппаратного обеспечения. Такие места описаны в C++ с помощью ключевого слова `volatile`. Объявление переменной как `volatile` требует от компилятора извлекать новую копию переменной всякий раз, когда она используется, вместо того, чтобы оптимизировать программу путем сохранения значения в регистре и его повторного использования. Могут быть также объявлены указатели на `volatile`-память.
- C++11 предлагает магическое заклинание `std::atomic<>`, которое заставляет память некоторое время вести себя так, как если бы это в действительности было простое линейное хранилище байтов, без учета всех сложностей современных микропроцессоров с их многочисленными потоками выполнения, многослойными кешами памяти и т.д. Некоторые разработчики считают, что для этого служит ключевое слово `volatile`, но они сильно ошибаются.

Операционная система также врет программам и их пользователям. На самом деле вся цель операционной системы — сообщить каждой программе набор очень убедительной лжи. Среди наиболее важной лжи — операционная система хочет убедить каждую программу, что она для компьютера единственная, что предоставляемая ей физическая память бесконечна и что есть бесконечное число процессоров, доступных для запуска потоков программы.

Операционная система умело использует аппаратное обеспечение компьютеров, чтобы скрывать свою ложь, так что программа на C++ не имеет иного выбора, кроме как поверить ей. Эта ложь обычно не слишком сильно влияет на работу программы, за исключением замедления ее работы. Однако она может осложнить выполнение измерений производительности программы.

Правда о компьютерах

Модели C++ соответствуют только простейшие микропроцессоры и некоторые древние ЭВМ. Что действительно важно для оптимизации — это то, что аппаратное обеспечение памяти реальных компьютеров очень медленное по сравнению со скоростью выполнения команд, что обращение к памяти в действительности не побайтное, что память не является простым линейным массивом одинаковых ячеек и что она имеет конечную емкость. Реальные компьютеры могут иметь более чем один адрес команды. Реальные компьютеры быстрые не потому, что они быстро выполняют каждую команду, а потому, что выполнение нескольких команд перекрывается, и сложные схемы следят за тем, чтобы такие перекрывающиеся команды вели себя так, как если бы они выполнялись одна за другой.

Медленная память

Основная память компьютера очень медленная по сравнению с его внутренними логическими вентилями и регистрами. Она медленная настолько, что процессор настольного компьютера может выполнить сотни команд за время, необходимое для извлечения из основной памяти единственного слова данных.

Следствием этого для оптимизации является то, что *доступ к памяти доминирует над другими действиями процессора*, включая выполнение команд.

Узкое место фон Неймана

Интерфейс к основной памяти является узким местом, которое ограничивает скорость работы. Это узкое место даже имеет имя. Оно называется узким местом (или бутылочным горлышком) фон Неймана, в честь знаменитого пионера компьютерной архитектуры и математика Джона фон Неймана (John von Neumann) (1903–1957).

Например, компьютер, оснащенный памятью DDR2 с частотой 1000 МГц (типичной для компьютеров несколько лет назад), имеет теоретическую пропускную способность 2 млрд слов в секунду, или 500 пс на слово. Но это не значит, что компьютер может считывать или записывать случайное слово данных каждые 500 пс.

Начнем с того, что за один такт (половина такта часов с частотой 1000 МГц) может выполняться только последовательный доступ. Доступ к не последовательным местам в памяти выполняется примерно за 6–10 тактов.

Далее, за доступ к шине памяти конкурируют несколько действий. Процессор постоянно проводит выборку очередных выполняемых команд из памяти. Контроллер кеш-памяти выполняет выборку блоков данных памяти для кеша и сбрасывает записанные строки кеша. Контроллер DRAM также требует циклы для обновления заряда в динамических ячейках устройства памяти. Числа ядер многоядерного процессора достаточно, чтобы гарантировать перенасыщенность шины памяти. Фактическая скорость, с которой данные могут быть прочитаны из основной памяти в определенное ядро, составляет более 20–80 нс на одно слово.

Закон Мура позволяет каждый год получать все больше и больше ядер в процессорах. Но чтобы обеспечить более быстрый интерфейс основной памяти, этого мало. Таким образом, удвоение числа ядер в будущем будет оказывать на производительность отрицательное влияние. Ядра будут бороться за доступ к памяти. Надвигающееся ограничение производительности называется *стеной памяти*.

Недоступность байтов

Хотя C++ считает, что каждый байт доступен по отдельности, компьютеры часто компенсируют медленность физической памяти путем выборки данных большими блоками. Самые мелкие процессоры могут выбирать из основной памяти отдельные байты, но процессоры настольных компьютеров могут выбирать за один раз по 64 байта. Некоторые суперкомпьютеры и графические процессоры выполняют выборку еще большего размера.

Когда C++ выбирает многобайтные данные, такие как тип `int`, `double` или указатель, может оказаться, что байты, составляющие данные, входят в два слова физической памяти. Это называется *невыровненным доступом к памяти*. Важным для оптимизации является то, что *невыровненный доступ требует в два раза больше времени*, чем если бы все байты были в одном и том же слове, поскольку при этом требуется чтение двух слов. Компилятор C++ выравнивает структуры таким образом, чтобы каждое поле начиналось с адреса байта, кратного размеру поля. Но это создает собственную проблему: "дыры" в структурах, содержащие неиспользуемые данные. Обращая внимание на размер полей данных и их порядок в структуре, можно сделать структуры максимально компактными при сохранении выровненности.

Одни обращения к памяти медленнее других

Для компенсации "медленности" основной памяти многие компьютеры содержат *кеш-память*, разновидность быстрой временной памяти, располагающейся близко к процессору, чтобы ускорить доступ к наиболее часто используемым словам памяти. Одни компьютеры обходятся без кеша; другие имеют один или несколько уровней кеша, каждый из которых меньше, быстрее и дороже предыдущего. Когда процессору нужны байты из слова кешированной памяти, они могут быть быстро извлечены без повторного обращения к основной памяти. Насколько кеш-память быстрее? Эмпирическое правило гласит, что каждый уровень кеш-памяти примерно в 10 раз быстрее, чем предыдущий уровень в иерархии памяти. На процессорах настольных компьютеров время доступа к памяти может меняться на пять порядков в зависимости от того, к чему осуществляется доступ: к кеш-памяти первого, второго или третьего уровня, к основной памяти или к странице виртуальной памяти на диске. Это одна из причин, по которым увлеченность тактами выполнения команд процессором и другими подобными тайнами так часто оказывается глупой и бесполезной — состояние кеша делает время выполнения команды весьма неопределенным.

Когда процессор нуждается в выборке данных, находящихся не в кеше, другие данные, расположенные в кеше, могут быть удалены из него для того, чтобы освободить место. Данные, выбираемые для удаления, как правило, являются наиболее давно использовавшимися. Это важно для оптимизации, поскольку означает, что *доступ к интенсивно используемым ячейкам памяти можно получить быстрее, чем к используемым реже*.

Чтение даже одного байта данных, который не находится в кеше, приводит к кешированию множества близлежащих байтов (как следствие это означает, что множество байтов, находящихся в настоящее время в кеше, удаляются из него).

Эти близлежащие байты будут готовы для быстрого доступа. Это важно для оптимизации, поскольку означает, что *обращение к соседним ячейкам памяти (в среднем) быстрее, чем к памяти в отдаленных местах*.

В терминах C++ это означает, что блок кода, содержащий цикл, может выполняться быстрее, поскольку инструкции, составляющие цикл, активно используются, располагаются близко одна к другой и, таким образом, вероятно, будут оставаться в кеше. Блок кода, содержащий вызовы функций или инструкции `if`, которые приводят к дальним переходам, могут выполняться более медленно, потому что используются части кода, далеко отстоящие одна от другой. Такой код использует больше пространства кеша, чем цикл. Если программа большая, а кеш конечен, часть кода будет удаляться из кеша, чтобы освободить место для других вещей, что приведет к замедлению доступа в следующий раз, когда потребуется этот код. Аналогично доступ к структуре данных, состоящей из последовательных местоположений в памяти, такой как массив или вектор, может быть быстрее, чем к структуре данных, состоящих из узлов, связанных указателями, поскольку данные в последовательных местоположениях с большей вероятностью будут оставаться в кеш-памяти. Доступ к структуре данных, состоящих из записей, связанных с помощью указателей (например, к списку или дереву), может быть медленнее из-за необходимости считывания данных каждого узла из основной памяти в новые строки кеша.

Остроконечные и тупоконечные слова

Из памяти может быть выбран один байт данных, но зачастую одновременно извлекается несколько последовательных байтов, образующих число. Например, в Microsoft Visual C++ значение типа `int` образуют четыре байта, считываемые из памяти вместе. Так как к памяти можно обращаться двумя способами, люди, которые проектируют компьютеры, должны ответить на важный вопрос: что содержится в первом байте (адрес которого наименьший) — старшие или младшие биты значения `int`?

На первый взгляд кажется, что это не должно иметь значения. Конечно, важно, чтобы все части компьютера одинаково считали, с какого конца `int` адрес меньше, иначе воцарится хаос. Вот в чем заключается разница между этими способами хранения. Если значение `int`, равное `0x01234567`, хранится по адресам 1000–1003 и первым хранятся старшие биты, то по адресу 1000 содержится байт `0x01`, а по адресу 1003 — байт `0x67`, в то время как если сначала хранится младший байт, то по адресу 1000 содержится `0x67`, а по адресу 1003 — `0x01`. Компьютеры, которые хранят старшие биты в байте с младшим адресом, называются компьютерами с *обратным порядком байтов* (“тупоконечниками”, big-endian). Компьютеры с *прямым порядком байтов* (“остроконечники”, little-endian) сначала читают младшие биты. Итак, имеется два способа хранения целого числа (или указателя), и нет никаких причин предпочесть один другому, так что разные команды, работающие на разных процессорах для разных компаний, могут делать разный выбор.

Проблемы начинаются, когда данные, записанные на диск или отправленные по сети одним компьютером, должны быть прочитаны другим компьютером. Диски и сети пересылают информацию побайтно, а не весь `int` одновременно. Поэтому

оказывается важно, какой конец числа сохраняется (или отправляется) первым. Если отправляющий и принимающий компьютеры не согласованы, то значение, отправляемое как `0x01234567`, может быть получено как `0x67452301`.

Порядок байтов является лишь одной из причин, по которым C++ не определяет, как биты располагаются в `int` или как значение одного поля в объединении влияет на другие поля. Это одна из причин, по которым программу можно написать так, что она будет успешно работать на компьютере одного вида, но приводить к аварийному завершению на другом.

Количество памяти ограничено

Память компьютера не бесконечна. Чтобы сохранить иллюзию бесконечной памяти, операционная система может использовать физическую память наподобие кеш-памяти и хранить данные, которые не помещаются в физическую память, в виде файла на диске. Эта схема называется *виртуальной памятью*. Виртуальная память создает иллюзию большего количества физической памяти. Однако получение блока памяти с диска занимает десятки миллисекунд — вечность для современного компьютера.

Быстрая кеш-память весьма дорогостоящая. В настольном компьютере или смартфоне может быть гигабайт памяти, но кеш размером лишь в несколько мегабайтов. Программы и их данные обычно в кеш не помещаются.

Результатом кеширования и применения виртуальной памяти может быть то, что *из-за кеширования определенная функция, работающая в контексте всей программы, может выполняться медленнее, чем та же функция в тестовой программе*, в 10 тысяч раз. В контексте всей программы функции и данные не могут все время оставаться в кеше, в то время как в контексте теста это именно так и происходит. Этот эффект усиливает преимущества оптимизаций, которые снижают использование памяти или диска, в то время как преимущества оптимизаций, которые уменьшают размер кода, остаются небольшими.

Вторым результатом кеширования является то, что если большая программа выполняет рассеянный доступ ко многим участкам памяти, то кеш-памяти может оказаться недостаточно для хранения данных, непосредственно используемых программой. Это приводит к снижению производительности, которое называют *пробуксовкой страницы*. Когда пробуксовка страницы происходит во внутреннем кеше микропроцессора, результатом является снижение производительности. Когда это происходит в файле виртуальной памяти операционной системы, производительность падает тысячекратно. Эта проблема возникала чаще, когда физическая память была дороже и меньше по размеру, но она встречается и сейчас.

Медленное выполнение команд

Простые микропроцессоры наподобие встроенных в кофеварки и микроволновые печи предназначены для выполнения команд с той же скоростью, с которой они могут извлекаться из памяти. Микропроцессоры настольного компьютера имеют дополнительные ресурсы для параллельной обработки нескольких команд, поэтому они способны выполнять команды во много раз быстрее, чем те могут быть извлечены из

основной памяти, так что бо́льшую часть времени для хранения команд и их передачи процессору используется быстрая кеш-память. Важность этого для оптимизации заключается в том, что *время доступа к памяти превышает время вычислений*.

Современные настольные компьютеры выполняют команды с удивительной скоростью, *если* им ничто не мешает. Они могут завершать команды каждые несколько сотен пикосекунд (пикосекунда представляет собой 10^{-12} с, до смешного короткое время). Но это не значит, что каждая команда выполняется так быстро. Процессор содержит "конвейер" одновременно выполняемых команд. Команды проходят через конвейер, дешифруются, получают свои аргументы, выполняют вычисления и сохраняют результаты. Чем более мощный процессор, тем более сложен его конвейер, разбивающий выполнение команды на десяток этапов так, чтобы как можно больше команд могли быть обработаны одновременно.

Если команда А вычисляет значение, которое требуется команде Б, то команда Б не может выполнить свое вычисление до тех пор, пока команда А не даст необходимый результат. Это приводит к *остановке конвейера*, короткой паузе в выполнении команд, которая возникает, когда выполнение двух команд не может полностью перекрываться. Остановка конвейера в особенности долгая, если команда А извлекает значение из памяти, а затем выполняет вычисление, которое дает значение, необходимое команде Б. Остановке конвейера подвержены все современные микропроцессоры, что делает их время от времени почти такими же медленными, как процессор в вашем тостере.

Трудное принятие решений

Еще одна вещь, которая может вызвать остановку конвейера, — это принятие решения компьютером. После большинства команд выполнение продолжается с команды, находящейся в памяти по следующему адресу. Бо́льшую часть времени эта следующая команда уже находится в кеше. Последовательные команды можно загружать в конвейер, как только на первом этапе конвейера для этого появляется место.

Но не таковы команды передачи управления. Команда перехода или вызова подпрограммы заменяет адрес выполнения произвольным новым значением. "Следующая" команда не может быть считанной из памяти и попасть в конвейер до тех пор, пока в некоторый момент в процессе обработки команды перехода не будет обновлен адрес выполнения. У слова памяти по новому адресу выполнения меньше шансов находиться в кеше. Конвейер останавливается на время обновления адреса выполнения и загрузки в конвейер новой "следующей" команды.

После команды условного ветвления выполнение продолжается в одном из двух разных мест: выполняется либо следующая команда, либо команда по адресу, который является целевым для команды ветвления, в зависимости от результатов некоторых предыдущих вычислений. Конвейер останавливается, пока не будут завершены все команды, участвующие в предыдущих вычислениях, и остается в этом состоянии до тех пор, пока не будет определен следующий выполняемый адрес и не будет прочитана соответствующая команда по этому адресу.

Важность этого явления для оптимизации заключается в том, что *вычисление быстрее принятия решения*.

Множественные потоки выполнения

Любая программа, работающая в современной операционной системе, разделяет компьютер с другими программами, выполняемыми в то же время, с периодическими обслуживающими процессами типа проверки диска или поиска обновлений Java или Flash, и с различными частями операционной системы, управляющими сетевым интерфейсом, дисками, звуковыми устройствами и другими периферийными устройствами. *Каждая программа конкурирует с другими программами за ресурсы компьютера*.

Программа обычно не слишком об этом осведомлена. Она просто работает немного медленнее. Исключением является ситуация, когда одновременно запускается много программ, и все они конкурируют за память и диск. Для настройки производительности, *если программа должна запускаться при запуске системы или в периоды пиковой нагрузки, ее производительность должна измеряться под нагрузкой*.

По состоянию на начало 2016 года настольные компьютеры имеют до 16 ядер процессора, а микропроцессоры, используемые в телефонах и планшетах, — до восьми. Беглый взгляд на диспетчер задач Windows, вывод состояния процесса Linux или список задач Android обычно показывает гораздо больше процессов, чем ядер, и большинство процессов имеют несколько потоков выполнения. Операционная система выполняет каждый поток в течение короткого времени, а затем переключает контекст на другой поток или процесс. С точки зрения программы это выглядит, как если бы одна команда выполнялась наносекунду, а следующая — 60 мс.

Что означает переключение контекстов? Если операционная система переключается от одного потока к другому в одной программе, это означает сохранение регистров процессора для приостановки потока и загрузки сохраненных регистров для возобновления другого потока. Регистры современного процессора содержат сотни байтов данных. Когда новый поток возобновляет выполнение, данные могут не быть в кеше, так что имеется начальный период медленного выполнения, пока новый контекст загружается в кеш. Таким образом, при переключении контекстов потоков имеются значительные затраты.

Процедура переключения операционной системой контекста от одной программы к другой еще более дорогостоящая. Все “грязные” страницы кеша (с записанными данными, которые еще не внесены в основную память) должны быть сброшены в физическую память, а все регистры процессора сохранены. Затем сохраняются регистры страниц отображения физической памяти на виртуальную в диспетчере памяти. Далее для нового процесса загружаются соответствующие регистры памяти и регистры процессора. И наконец выполнение программы может возобновиться. Но кеш в этот момент пуст, так что начальный период характеризуется низкой производительностью и конфликтами памяти.

Когда программа должна ожидать некоторое событие, это ожидание может продолжаться даже после того, как это событие произойдет, пока операционная система не освободит процессор для продолжения программы. При выполнении программы в контексте работы других программ, конкурирующих за ресурсы компьютера, это может увеличить время выполнения программы и сделать его более неопределенным.

Исполнительные устройства многоядерных процессоров и связанная с ними кеш-память для достижения лучшей производительности работают более или менее независимо друг от друга. Однако все исполнительные устройства имеют одну и ту же основную память. Они вынуждены конкурировать за доступ к оборудованию, связывая его с основной памятью, что делает узкое место фон Неймана в компьютере с несколькими исполнительными устройствами еще более ограничивающим.

Когда исполнительное устройство записывает значение, оно сначала попадает в кеш-память. В конечном итоге оно должно быть записано из кеша в основную память, так что это значение станет видимым для других исполнительных устройств. Однако из-за конфликтов доступа к основной памяти среди исполнительных устройств основная память может не обновляться в течение сотен команд после того, как значение было изменено.

Если на компьютере имеется несколько исполнительных устройств, то одно из них может, таким образом, на протяжении длительного периода времени не увидеть данные, записанные другим устройством в основной памяти, а изменения в основной памяти могут произойти не в том же порядке, что и порядок выполнения команд. В зависимости от непредсказуемых временны*х* факторов исполнительное устройство может увидеть как старое значение общей памяти, так и обновленное значение. Для того чтобы различные исполнительные устройства видели согласованные представления памяти, должны использоваться специальные команды синхронизации. Значение этого эффекта для оптимизации состоит в том, что *обращение к совместно используемым потоками выполнения данным гораздо медленнее, чем к не разделяемым данным.*

Вызовы операционной системы являются дорогостоящими

Все процессоры, кроме самых мелких, имеют аппаратное обеспечение для обеспечения изоляции между программами, так что программа А не может читать, писать или выполнять команды в физической памяти, принадлежащей программе Б. То же самое оборудование защищает ядро операционной системы от перезаписи программами. С другой стороны, ядру операционной системы требуется доступ к памяти, принадлежащей каждой программе, чтобы эти программы могли делать системные вызовы операционной системы. Некоторые операционные системы также позволяют программам делать запросы для совместного использования памяти. Способы организации системных вызовов и совместно используемой памяти разнообразны и полны тайной магии. С точки зрения оптимизации важным является то, что *системные вызовы дороги*, в сотни раз дороже, чем вызовы функций внутри одного потока одной программы.

C++ тоже лжет

Самая большая ложь C++ в том, что он рассказывает своим пользователям, что компьютер, который выполняет программу, представляет собой простую последовательную структуру. В обмен на то, что разработчики делают вид, что верят в эту ложь, C++ позволяет разработчикам программировать без знаний интимных подробностей каждого микропроцессорного устройства, которые совершенно необходимы при использовании языка ассемблера.

Не все инструкции одинаково дорогие

В мирные, давно минувшие дни программирования на языке C Кернигана и Ритчи каждая инструкция была примерно такой же дорогой, как и любая другая. Вызов функции может содержать вычисления произвольной сложности. Однако оператор присваивания в общем случае копирует нечто, помещающееся в машинном регистре, во что-то другое, что может хранить содержимое регистра компьютера. Таким образом, инструкция

```
int i,j;
...
i = j;
```

копирует 2 или 4 байта из `j` в `i`. Объявление может быть `int`, `float` или `struct big struct*`, но инструкция присваивания все равно выполняет одно и то же количество работы.

В настоящее время это не так. В C++ присвоение одного `int` другому представляет собой точно такой же объем работы, как и в случае соответствующей инструкции C. Но инструкция наподобие `BigInstance i = OtherObject;` может копировать целые структуры. Более того, этот вид присваивания вызывает конструктор `BigInstance`, который может скрывать произвольно сложный механизм. Конструктор вызывается также для каждого выражения, переданного функции в качестве формального аргумента, и вновь, когда функция возвращает значение. Арифметические операторы и операторы сравнения также могут быть перегружены, так что выражение `A = B*C;` может умножать *n*-мерные матрицы, а `if (x<y) ...` может сравнивать два пути через ориентированный граф произвольной сложности. Значение этого для оптимизации заключается в том, что *некоторые инструкции скрывают большие количества вычислений. Вид инструкции ничего не говорит об ее стоимости.*

Разработчики, которые начинали изучение языков программирования с C++, могут не увидеть в этом ничего удивительного; но для тех, кто начинал с изучения C, их инстинкты могут привести к катастрофическим заблуждениям.

Инструкции выполняются не по порядку

Программы на C++ ведут себя так, как если бы они выполнялись в порядке, указанном инструкциями управления потоком выполнения C++. Хитрая оговорка “как если бы” в предыдущем предложении является главной, на которой построено множество оптимизаций и трюков современного компьютерного оборудования.

За кулисами компилятор может — и зачастую так и делает — переупорядочивать инструкции для повышения производительности. Но компилятор знает, что переменная должна содержать последний результат вычисления, присвоенный до проверки или присваивания его другой переменной. Современные микропроцессоры также могут выбрать команды для выполнения не по порядку их следования, но они содержат логические схемы, которые гарантируют выполнение записи в память до последующего чтения того же места. Логика управления памятью микропроцессора может даже выбрать задержку записи в память для оптимального использования шины памяти. Однако контроллер памяти знает, что именно в настоящее время находится в

процессе записи от исполняющего устройства через кеш-память в основную память, и гарантирует, что если будет считываться тот же адрес, то будет получено корректное значение, даже если его запись выполнена еще не до конца.

Параллелизм усложняет эту картину. Программы на C++ компилируются без знания о других потоках, которые могут выполняться одновременно с данным. Компилятор C++ не знает, какие переменные, если таковые имеются, совместно используются потоками. Совокупный эффект изменения порядка инструкций компилятором и компьютером и задержка записи в основную память разрушают иллюзию выполнения инструкций в указываемом программой порядке, если программа содержит параллельные потоки, которые совместно используют данные. Разработчик должен добавить явную синхронизацию кода многопоточных программ для гарантии получения согласованного предсказуемого поведения. *Синхронизация кода снижает степень параллелизма, получаемого при совместном использовании данных параллельными потоками.*

Резюме

- *Доступ к памяти доминирует над другими действиями процессора.*
- *Невыровненный доступ требует в два раза больше времени, чем когда все байты находятся в одном и том же слове.*
- *Доступ к интенсивно используемым ячейкам памяти можно получить быстрее, чем к используемым реже.*
- *Обращение к соседним ячейкам памяти выполняется быстрее, чем к памяти в отдаленных местах.*
- *Из-за кеширования определенная функция, работающая в контексте всей программы, может выполняться медленнее, чем та же функция в тестовой программе.*
- *Обращение к совместно используемым потоками выполнения данным выполняется гораздо медленнее, чем к не разделяемым данным.*
- *Вычисления осуществляются быстрее принятий решений.*
- *Каждая программа конкурирует с другими программами за ресурсы компьютера.*
- *Если программа должна запускаться при запуске системы или в периоды пиковой нагрузки, ее производительность должна измеряться под нагрузкой.*
- *Каждое присваивание, инициализация аргумента функции и возврат из функции вызывает конструктор — функцию, которая может скрывать произвольно большое количество кода.*
- *Некоторые инструкции скрывают большие количества вычислений. Вид инструкции ничего не говорит об ее стоимости.*
- *Синхронизация кода снижает степень параллелизма, получаемого при совместном использовании данных параллельными потоками.*