Санкт-Петербургский государственный университет

Математико-механический факультет

Кафедра системного программирования

Кэш-зависимые структуры данных

Курсовая работа студента 445 группы

Ерохина Георгия Алексеевича

Научные руководители

Чернышев Г.А.,

Ассистент кафедры информатики СПбГУ

Смирнов К.К.

Инженер ЗАО “Ланит-Терком”

Санкт-Петербург

2013

Оглавление

Введение

Важными характеристиками оперативной памяти являются объем и время обращения. И если объем памяти значительно возрос за последние годы, то время обращения, несмотря на небольшое уменьшение, является одним из узких мест многих программ. В то время, как производительность процессоров растет, накладные расходы на обращение к памяти все сильнее сказываются на времени выполнения программ [1]. Один из способов борьбы с разрывом в производительности процессора и памяти является использования кэша — невидимого для программиста буфера с быстрым доступом, содержащем информацию, которая может быть запрошена с большой вероятностью. Как показывает практика, в большинстве приложений имеет место пространственная и временная локальность обращений к данным. Пространственная локальность означает, что если произошло обращение к какому-то участку памяти, то в ближайшее время с высокой вероятностью произойдет обращение к нему же, либо к данным, которые располагаются поблизости от этого участка. Это можно наблюдать, например, при последовательном обходе массива по массиву, когда за небольшой промежуток времени считываются блоки данных, располагающиеся в памяти последовательно. В кэше это свойство обращений к памяти используется следующим образом: если происходит чтение некоторого участка памяти, в кэш копируется не только затребованный блок, но и некоторая часть данных, находящихся поблизости от него. Наименьшая область памяти, которой может оперировать кэш, соответствует кэш-линии, которая состоит из самих данных и тэга, содержащего адрес этих данных в основной памяти и некоторую дополнительную информацию, например, совпадают ли данные в кэше с соответствующими им данными в основной памяти. Таким образом, наименьший объем данных, который можно поместить в кэш, соответствует размеру кэш-линии. Временная локальность означает, что к данные, к которым было совершено обращение, с большой вероятностью будут затребованы в ближайшее время. В качестве примера можно привести любую переменную, используемую в небольшом фрагменте кода более одного раза. Временная локальность позволяет добиться прироста производительности в случае, когда повторное обращение производится к переменной, которая уже находится в кэше [7]. Эффективность использования временной локальности зависит от политики замещения кэша, то есть от того, когда и какие данные из него вытесняются.

Следует упомянуть, почему структуры, о которых пойдет речь в этой работе называются кэш-зависимыми (англ. cache-conscious). Дело в том, что в отличие от кэш-независимых алгоритмов (англ. cache-oblivious), при проектировании таких типов данных, могут учитываться конкретные характеристики кэша на целевой машине, такие как ассоциативность или размер кэш-блока. В свою очередь, для кэш-независимых алгоритмов и структур важно лишь само наличие кэша. Классический пример такого алгоритма — метод “разделяй и властвуй” когда исходная задача разделяются на несколько подзадач и те рекурсивно решаются таким же образом. На некотором этапе, все данные получившейся подзадачи полностью поместятся в кэш, что значительно уменьшит время их решения [5].

В последние годы огромный объем оперативной памяти стал доступен даже на персональных компьютерах и ноутбуках, не говоря уже о серверах и мейнфреймах. Это привело к тому, что даже среди баз данных, области программного обеспечения, неразрывно связанной с третичными носителями информации, появились экземпляры, предназначенные для работы полностью в оперативной памяти. Примером таких СУБД могут служить SAP HANA, MemSQL, eXtremeDB. Научное сообщество также не осталось в стороне: например, в рамках ежегодной конференции ACM SIGMOD, посвященной главным образом базам данных, традиционно проводится соревнование, на котором предлагается актуальная задача из области баз данных, и в последние годы тема соревнования основывается именно на системах, работающих целиком в оперативной памяти.

Постановка задачи

В этом году автор в составе команды RotaFortunae принимал участие в уже упомянутом ранее соревновании ACM SIGMOD Programming Contest. На момент написания этого текста известно, что команды вышла в финал, то есть заняла одно из первых пяти мест в турнирной таблице[1], занимая второе место в публичном рейтинге[2]. В задании требовалось написать систему, обрабатывающую поток документов и запросов. Документ представляет собой набор слов, запрос — небольшой набор слов (до пяти включительно), одну из трех заданных метрик на множестве слов и некоторое число (от нуля до трех включительно), которое означает, что слово из запроса встречается в документе, если там есть слово, расстояние до которого в заданной метрике не превосходит этого числа. Программа должна работать только с данными, которые находятся в оперативной памяти, на вход ей поступает поток документов и команд на добавление и удаление запросов из системы. Если запрос был добавлен и ещё не был удален, он называется активным. При поступлении документа, система должна выдать список активных запросов, которым этот документ удовлетворяет. В реальности такая схема встречается, например, в интернет-сервисе Twitter. Пользователь может подписаться на определенный хэштег, и при появлении сообщения с этим хэштегом, необходимо найти всех пользователей, которые в нем заинтересованы. При этом хэштеги соответствуют запросам, а сообщения (твиты) — документам. Другой пример использования такой схемы — подписка на некоторый новостной ресурс, когда необходимо оповещать пользователя при появлении статей, соответствующих его кругу интересов. Чтобы приблизить задание к реальным условиям, когда у множества пользователей имеются схожие предпочтения, в задании требовалось уделить внимание случаю, при котором в запросах некоторые слова часто повторяются.

Чтобы не тратить время на обработку повторяющихся слов, в системе интенсивно используются структуры, в которых одинаковые элементы не различимы. Например, один из самых популярных видов представления текстовых документов в сфере информационного поиска — модель множества слов (bag-of-words), в которой, как не трудно догадаться, документ представляется множеством слов, которые в него входят, при этом не учитывается порядок слов и связь между ними. В одном из своих проявлений, эта

[1] http://sigmod.kaust.edu.sa/finalists.php [Дата просмотра 26.05.2013]
[2] http://sigmod.kaust.edu.sa/leaderboard.php [Дата просмотра 26.05.2013]

модель используется и в нашей системе, и это не единственный пример использования в ней множеств. Стоит отметить, что все используемые в нашей программе множества обладают одной характерной чертой — на протяжении всего времени их использования, элементы в них остаются неизменными и не удаляются: поступивший в систему запрос или документ не может быть изменен, а при удалении запроса из системы, слова, которые в нем встречались, сохраняются, чтобы при добавлении в систему новых запросов с такими же словами не пришлось заново вычислять посчитанные ранее расстояния. Это не учитывается в структурах из стандартных библиотек, где универсальность структур более приоритетна, чем их быстродействие. Как показало профилирование системы инструментом Callgrind[3], функции работы с множествами являются одним из узких мест программы. Учитывая все вышесказанное, эффективная реализация множества, адаптированного под условия использования нашей системой, даст ощутимый прирост в производительности. Таким образом, перед нами встали следующие задачи.

- Изучить подходы к эффективному использованию кэша процессора.
- Оптимизировать систему для работы в оперативной памяти, в частности эффективно реализовать структуру данных «множество» в оперативной памяти.

[3] http://valgrind.org/info/tools.html#callgrind [Дата просмотра 26.05.2013]

Решение

Способы эксплуатации кэша

Один из способов повысить быстродействие структур, располагающихся в оперативной памяти — учитывать кэш процессора при их проектировании, такие структуры называются кэш-зависимыми. Существует множество способов сделать это, и среди них можно выделить три основных подхода [3]:

- Проектировать структуру типов данных с учетом кэша. Например, при работе с объектами, размер которых превышает размер кэш-блока, желательно группировать поля, доступ к которым будет производиться одновременно, таким образом, чтобы эти поля попадали в один кэш-блок (hot-cold structure splitting). Наверное, главный недостаток этой стратегии — необходимость знать алгоритмы и структуры, использующиеся в программе, что делает невозможным инкапсуляцию логики самой программы от логики, обеспечивающей оптимизации, связанные с кэшем. Однако, это является недостатком только при написании универсальных инструментов, вроде уже упомянутых выше ccmalloc или ccmorph. Основные примеры применения такой стратегии:
    - Раскрашивание (англ. coloring): способ сделать так, чтобы участки памяти, используемые в программе одновременно, отображались в разные области кэша, что уменьшит количество коллизий. Тут следует упомянуть такое свойство кэша, как ассоциативность. Существует три типа кэша:
        - прямого отображения (direct-mapped cache), в этом случае каждая область памяти отображается в единственную область кэша.
        - наборно-ассоциативный (set-associative или N-way set-associative), задается константа N, и гарантируется, что любая область памяти отображается ровно в N областей кэша.
        - Полностью ассоциативный (fully-associative), при этом каждая область памяти может быть помещена в какую угодно область кэша.

      Из-за того, что кэш по объему значительно меньше оперативной памяти, очевидно, что в любом из этих трех случаев возможно конфликты, то есть одному участку кэш-памяти будут соответствовать разные участки основной (оперативной) памяти.

При раскрашивании считается известным, какие участки памяти отображаются в одну область кэша. По этому признаку, области памяти можно сгруппировать: в одной группе будут находиться участки, которые отображаются в одну область кэша. Отсюда происходит название этого метода — участки памяти “раскрашиваются” таким образом, чтобы фрагменты разного цвета в принципе не могли в отображались в одну область кэша. Также, востребованные и невостребованные данные можно хранить в областях памяти разного цвета, это приводит к тому, что часто запрашиваемые данные не вытесняются из кэша остальными, редко запрашиваемыми.

В современных операционных системах, этот метод реализуется на этапе трансляции виртуальной памяти в физическую.

- Сжатие (англ. compression): если структуры занимают меньше места, есть возможность поместить больше данных в кэш. Помимо сложных способов сжатия для конкретных структур (пример — сжатие многомерных параллелепипедов в CR-tree [6]), существуют более общие методы:
  - Избавление от указателей (pointer elimination): в некоторых случаях, возможно не хранить указатели, а вычислять смещение. Например, в дереве с фиксированным числом детей у каждого узла, можно не хранить указатели на дочерние узлы, а располагать все узлы дерева в массиве в оговоренном порядке, например, в начале располагается корень (0 уровень), затем его дети (первый уровень), затем второй уровень и так далее. Другой пример — развернутый список.
  - Разделение данных на холодные и горячие (англ. hot/cold structure splitting): для структур, размером более одного кэш-блока, эффективным способом экономии кэш-памяти может быть следующий способ. Если известно, что некоторые поля структуры используются гораздо чаще других, можно хранить отдельно данные, к которым часто происходят обращения (“горячие данные”) и отдельно те, к которым редко обращаются (“холодные данные”). Таким образом, при обращении к полям из горячей части структуры, только они попадут в кэш, а вместо “холодных” полей, в кэше окажутся потенциально более востребованные данные.

    - Кластеризация (англ. clustering): распределить данные по кластерам, где в одном кластере будут храниться элементы, доступ к которым предположительно будет производиться с небольшой разницей во времени. Например, при работе с деревом, в одном кластере могут оказаться узел и его дети, из-за этого, во время спуска по дереву, при работе с некоторым внутренним узлом, все его дети уже будут находиться в кэше.
- Выделять память с учетом кэша, то есть на этапе аллокации делать так, чтобы переменные, обращения к которым происходят одновременно, либо в течение короткого промежутка времени, помещались в одну область кэша. Пример — использование функции ccmalloc, описанной в [4], или же программа может профилироваться на этапе компиляции, и с учетом информации об использовании переменных, компилятор улучшит размещение данных в памяти [2].
- Реорганизовывать расположение данных в памяти во время исполнения программы. Пример — функция ccmorph, из статьи [4].

Реализация кэш-зависимого множества для решения задач информационного поиска

Вернемся к исходной задаче: реализовать эффективное множество для системы информационного поиска, работающей в оперативной памяти.

Как было сказано выше, в из множеств не требуется удалять вставленные элементы. Однако, при появлении в системе слова, которое не встречались в более ранних запросах или документах, требуется полностью проходиться по множествам слов, длины которых несильно отличаются от длины этого слова. Это значит, что обход элементов множества необходимо оптимизировать в первую очередь. Так как порядок обхода слов не важен, то есть в множестве не требуется поддерживать определенный порядок элементов, в качестве структуры для реализации множества была выбрана хэш-таблица. Из всевозможных структур данных, ввиду последовательного расположения данных в памяти и отсутствия переходов по указателям, быстрее всего происходит обход обычного массива. Поэтому было решено хранить полезную нагрузку нашей структуры в массиве (хранилище). В хэш-таблице используются не просто объекты, а связные списки, поэтому в хранилище необходимо хранить элементы списков, то есть пары вида <объект, ссылка на следующий элемент списка>. Получается структура наподобие списка на массиве, за исключением того, что на одном массиве построено множество списков. Конечно, последовательно располагающиеся элементы списка могут находиться в разных частях массива, что отрицательно влияет на локальность данных, но в связи с особенностями использования списка в нашей системе, это не играет особой роли по нескольким причинам. Во-первых, вставка и поиск (только при выполнении этих операций может потребоваться обход элементов в порядке их расположения в списке) происходят несколько реже обхода множества. Во-вторых, при достаточном размере хэш-таблицы, коллизии встречаются очень редко, поэтому список редко состоит более чем из одного элемента.

Но тут появляется одна проблема. С одной стороны — необходимо динамическое множество, так как количество слов, проходящих через систему заранее неизвестно, а в купе с желаемой локальностью, это значит, что хранилище необходимо время от времени полностью переаллоцировать, используя больший объем памяти и копируя туда старые данные. С другой стороны, для обеспечения локальности, в множестве должны храниться сами слова, а не ссылки или указатели на них. Так как при повторном выделении памяти адрес элемента меняется, для идентификации элемента множества извне нельзя пользоваться ссылками или указателями. В стандартной библиотеке C++ эта проблема

решается при помощи итераторов — аналогов обычных указателей, которые остаются актуальными при подобных преобразованиях. В нашем случае возможно было также определить тип данных, по функционалу аналогичный указателям, но простота внутренней структуры позволила поступить проще и в то же время улучшить структуру с учетом кэша процессора путем избавления от указателей, а именно замены указателя (итератора) на более компактную структуру. Достаточно на этапе переаллокации хранилища сохранить порядок элементов, и по индексу становится возможным однозначно найти нужное слово. Целевой платформой для нашей системы является сервер с 64-разрядным процессором, а это значит, что обычный указатель занимает 64 бита, в то время как для хранения индекса в хранилище достаточно использовать 32-битный целый тип. Таким образом, размер хэш-таблицы уменьшается в 2 раза. При использовании своего множества, быстродействие всей системы повысилось на 14% (43.5 с против 50.3)

Элементы, находящиеся в хранилище, имеют вид <объект, ссылка на следующий элемент списка>, при этом, второе поле этой структуры используется только при вставке или поиске элементов в множестве, а в момент итерации требуется значение только первого. Иными словами, первое поле является часто используемым (во введенной ранее терминологии — горячим), а второе редко используемым (холодным). Применяя разделение данных на горячие и холодные, можно разбить хранилище на 2 части: массив объектов и массив индексов, тогда во время обхода множества, в кэш попадет больше данных, которые будут использованы программой в ближайшее время. Подобная оптимизация одного из множеств привела к уменьшению времени прохождения основного теста на 4% (12.2 с против 12.7 с), при этом разница статистически значима.

Измерение производительности системы производились на стороне организаторов соревнования, на сервере, выделенном специально для проверки и сравнения решений участников.

Конфигурация тестового сервера организаторов выглядит следующим образом[4]:

Процессор	Intel Xeon X5650
Тактовая частота	2.67 GHz
Конфигурация	2 процессора (всего 12 ядер)

[4] http://sigmod.kaust.edu.sa/task-details.php [Дата просмотра 26.05.2013]

Кэш первого уровня	64 KB/core
Кэш второго уровня	256 KB/core
Кэш третьего уровня	12MB
Объем оперативной памяти	96 GB
Операционная система	Ubuntu 12.10 (kernel: 3.5.0-17-generic)
Компилятор	GCC 4.7.2

Заключение

В работе были представлены основные способы использования кэша процессора в современных программах. Некоторые из них были использованы при реализации системы фильтрации документов, которая была представлена на соревновании ACM SIGMOD 2013 Programming Contest. В частности, описан способ реализации кэш-зависимого неупорядоченного множества. В условиях системы, представленной на упомянутом выше соревновании, реализованное таким образом множество показало ощутимый прирост в производительности по сравнению со структурами из стандартной библиотеки языка C++.

Список литературы

1 S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski, and J. Rattner. Platform 2015: Intel Processor and Platform Evolution for the Next Decade. Intel white papers. 2005.

2 Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. 1998. Cache-conscious data placement. SIGPLAN Not. 33, 11 (October 1998), 139-149.

3 Trishul M. Chilimbi , Bob Davidson , James R. Larus, Cache-conscious structure definition, Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation, p.13-24, May 01-04, 1999, Atlanta, Georgia, United States.

4 Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. 1999. Cache-conscious structure layout. SIGPLAN Not.34, 5 (May 1999), 1-12.

5 Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnels, and Fred Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. In Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures (SPAA '07). 2007. ACM, New York, NY, USA, 93-104.

6 Kihong Kim, Sang K. Cha, and Keunjoo Kwon. Optimizing multidimensional index trees for main memory access. SIGMOD Rec. 30, 2 (May 2001), 139-150.

7 Marc Snir and Jing Yu. On the Theory of Spatial and Temporal Locality. Technical Report No. UIUCDCS-R-2005-2611, Department of Computer Science, UIUC, July 2005.