Василенко Анатолий, 321 группа

**Предложить схему организации кеш-памяти, при наличии малой ассоциативной памяти. Описать её преимущества по сравнению с полностью ассоциативной кеш-памятью, кеш-памятью с прямым отображением и частично ассоциативной кеш-памятью.**

Значительное влияние на работу кэш-памяти оказывает алгоритм отображения адресов оперативной памяти на адреса в кэше. Чем быстрее мы можем проверить наличие определенного слова в кэш-памяти, тем быстрее будет работать кэш. Кроме того, схема отображения оперативной памяти на кэш-память влияет на стоимость и размеры центрального процессора. Различные схемы организации кэш-памяти позволяют балансировать между скоростью поиска и числом мест, на которых может находиться определенный блок памяти. Например, если некоторый блок основной памяти может располагаться на

ограниченном множестве мест в кэш-памяти, то кэш называется множественно-ассоциативным. Обычно множество представляет собой группу из двух или большего числа блоков в кэше. Если множество состоит из нескольких блоков, то такое размещение называется многоканальным множественно-ассоциативным. Для размещения блока прежде всего необходимо определить множество. Адрес множества кэш-памяти равен адресу блока оперативной памяти, нацело деленному на число множеств в кэш-памяти. Блок может размещаться на любом месте данного множества.

Пример схемы реализации:

……..

 K k1 k2

 …..

…

 Кеш

 ОЗУ

Кеш состоит из некоторого набора строк

Ключ K ассоциируется с первыми N битами в адресации ОЗУ (мы разбили ОЗУ на блоки размером с количество строк в кеше)

Строки кеша в свою очередь подразделяются на клетки в которых можно записать ключ, ассоциирующийся с последними (т.е. младшими) битами в адресе по которому располагается слово в ОЗУ, а рядом с ключём записывается само слово из ОЗУ. (Можно менять размер того, что записывается, например не слово, а 8 байт, это не принципиально для нашей схемы, однако, если мы будем записывать не слово, а больше, то нужно учесть это при разбиении ОЗУ на блоки) (Заметим, что в одной строке хранятся данные, как в ассоциативном кеше)

При запросе чтения информации по некоторому адресу, мы сначала проводим небольшой ассоциативный поиск по строкам кеша, если нужной строки не найдено, то либо у нас есть свободные строки, и тогда мы прописываем в кеше необходимый ключ у свободной строки и прописываем ключ и содержимое одной из клеток этой строки, если же у нас нету свободных строк, то происходит вытеснение, описанное в (\*). Найдя нужную строку (если нужных строк оказалось несколько, то необходимо будет провести следующую операцию с каждой из них), мы проверяем, есть ли нужная нам информация в какой-либо клетке. Если нет, то идём к алгоритму вытеснения (\*\*), если да, то у нас попадание, и всё хорошо.

(\*) если нет свободного места под строку, то мы выбираем ту строку, которую будем вытеснять по алгоритму LRU среди всех строк.

(\*\*) Мы находим LRU строку, после чего, если эта строка – не наша, то вытесняем её и инициализируем, аналогично описанному выше новую строку и клетку. Если LRU строка – это наша строка, то мы находим LRU клетку в нашей строке и вытесняем её, записывая вместо ней нашу новую информацию.

Когда мы подсчитываем LRU среди строк, то мы берём по каждой строке некоторое среднее LRU в её клетках, после чего находим минимальное среднее по всем строкам.

Приведённую выше схему можно так же рассматривать, как иерархию из 2-х уровней таблиц (это лишь вопрос представления).

Эта схема лучше, чем полностью ассоциативная кеш память, потому что при ассоциативном поиске сам поиск происходит не по всему кешу, а чаще всего только сначала по строкам, а потом по клеткам этой строки. (Т.к. программы и данные обычно располагаются локально, то строки с одинаковым ключём будут появляться крайне редко)

Эта схема лучше, чем схема с прямым отображением, потому что, в схеме с прямым отображением есть проблема, что когда выбираются данные, попадающие в одинаковые строки, то происходит вытеснение ранее занесенных строк, несмотря на возможное наличие свободных строк в кеше. В описанной выше схеме данной проблемы не появляется.

Эта схема лучше, чем частично-ассоциативная схема, потому что в частично-ассоциативной схеме может происходить кеш-трешинг, в предложенной выше схеме, трешинг может происходить только в случае, если забит абсолютно весь кеш, кроме, быть может, одной строки.

Однако как и в любой другой схеме, в предложенной выше есть свои недостатки, в частности трудность реализации, а также некоторое падение в скорости работы (но только по сравнению с полностью ассоциативной кеш-памятью).

**Виртуальные и физические теги. Кеш память в многоядерных процессорах.**

Вся физическая память, которая может быть помещена в кэш, разделена на виртуальные «линии», размер которых невелик, но, тем не менее, больше размера порции данных, необходимых для выполнения одной команды. Обычно на x86- совместимых процессорах, размер линии равен 64 байтам. Каждая линия в оперативной памяти имеет номер (это адрес памяти с отброшенными младшими битами). В кэше каждой линии ставится в соответствие тег, который является адресом (в оперативной памяти) продублированых в этой линии данных. Теперь вспомним о том, что большинство современных операционных систем используют механизмы виртуальной памяти. Было бы естественно, чтобы кэш архитектурно учитывал эту особенность работы операционных систем. Решение довольно естественно: будем хранить в теге линии кэш-памяти не физический адрес, а виртуальный. Таким образом мы избежим проблемы необходимости трасляции виртуального адреса в физический, если запрашиваемый блок виртуальной памяти уже хранится в кэше процессора. Однако, не все так просто: один и тот же адрес виртуальной памяти для нескольких различных процессов может указывать на разные места в физической памяти. Возможны также ситуации (например, при использовании механизмов разделяемой памяти для синхронизации между процессами), когда несколько различных виртуальных адресов ссылаются на один и тот же блок данных в

физической памяти. Механизмы решения этих проблем различны в различных процессорах. Так, например, процессоры ARMv5 при переключении на другой процесс полностью выгружают кэш процессора в основную память устройства и загружают данные из физической памяти, принадлежащей процессу, на который происходит переключение. Хотя такая схема и кажется неоптимальной, в случае небольшого числа процессов и редкого переключения между ними, она оказывается достаточно выгодной. Для многоуровневых кэшей тоже оказывается достаточно выгодно использовать механизм виртуальных тегов. Вспомним, что при всякой загрузке данных в кэш уровня N, данные сначала загружаются в кэш уровня (N + 1) (кэшем максимального уровня можно считать непосредственно оперативную память). Если хранить в кэше уровня N адрес блока данных в кэше уровня (N + 1), то мы сможем избежать дополнительных расходов, связанных с поиском блока данных в кэше по его физическому адресу. Это дает значительный прирост производительности кэш-памяти, хотя и усложняет ее архитектуру.

Системы с общей оперативной памятью подразделяются на UMA (Uniform Memory Access) (система с однородным доступом в память) и NUMA (NonUniform Memory Access) (система с неоднородным доступом в память). В качестве конкретной реализацией UMA системы можно привести SMP – симметричная мультипроцессорная система (Symmetric Multiprocessor).

Когда у систем есть общая оперативная память, то стаёт вопрос о том, как её кешировать!

Кэш называется когерентным, если выполняются следующие условия:

1. Если процессор Р записывает значение в переменную Х, то при следующем считывании Х он должен получить ранее записанное значение, если между записью и чтением Х другой процессор не производил запись в Х. Это условие связано с сохранением [порядка выполнения программы](http://ru.wikipedia.org/wiki/%D0%92%D0%BD%D0%B5%D0%BE%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D0%BD%D0%BE%D0%B5_%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5), это должно выполняться и для однопоточной архитектуры.
2. Операция чтения Х процессором P1, следующая после того, как другой процессор P2осуществил запись в Х, должна вернуть записанное значение, если другие процессоры не изменяли Х между двумя операциями. Это условие определяет понятие когерентной видимости памяти.
3. Записи в одну и ту же ячейку памяти должны быть последовательными. Другими словами, если два процессора записывают в переменную Х два значения: А, затем В — не должно случиться так, чтобы при считывании процессор сначала получал значение В, а затем А.

В этих условиях предполагается, что операции чтения и записи происходят мгновенно. Однако этого не происходит на практике из-за [задержек памяти](http://ru.wikipedia.org/wiki/CAS-%D0%BB%D0%B0%D1%82%D0%B5%D0%BD%D1%82%D0%BD%D0%BE%D1%81%D1%82%D1%8C) и других особенностей архитектуры. Изменения, сделанные процессором P1, могут быть не видны процессору P2, если чтение произошло через очень маленький промежуток времени после записи. Модель консистентности памяти определяет, когда записанное значение будет видно при чтении из другого потока.

В системе с общей оперативной памятью имеется ОЗУ, и любой процессорный элемент имеет доступ к любой точке общего ОЗУ, то есть любой адрес может быть исполнительным для любого процессора.

SMP(UMA) система:

Характеристики доступа любого процессорного элемента в любую точку ОЗУ не зависят от конкретного элемента и адреса (Все процессоры равноценны относительно доступа к памяти).

Кеш в данном случае у каждого элемента свой, и для того, чтобы у разных процессоров лежали в кеше правильные одинаковые значения, существует протокол согласования кешей (caсhe coherence protocol)

Нужно также понимать, что при синхронизации кешей, приходится загружать шину соответствующей служебной информацией.

Механизмы когерентности кешей:

1. Когерентность с использованием справочника (directory). Информация о состоянии блока физической памяти содержится только в одном месте, называемом справочником (физически справочник может быть распределен по узлам системы).
2. Когерентность с использованием отслеживания ([snooping](http://ru.wikipedia.org/w/index.php?title=Snoop&action=edit&redlink=1)). Каждый кэш, который содержит копию данных некоторого блока физической памяти, имеет также соответствующую копию служебной информации о его состоянии. Централизованная система записей отсутствует. Обычно кэши расположены на общей (разделяемой) шине и контроллеры всех кэшей наблюдают за шиной (просматривают ее) для определения того, не содержат ли они копию соответствующего блока.
3. Перехват (snarfing). Когда из какого-либо одного кэша данные переписываются в оперативную память, контроллеры остальных получают сигнал об этом изменении ("перехватывают" информацию об изменении данных) и, если необходимо, изменяют соответствующие данные в своих кэшах.

Это были перечислены механизмы синхронизации, а моделей и протоколов, т.е. собственно реализаций существует достаточно большое количество:

* [Протокол MSI](http://ru.wikipedia.org/w/index.php?title=%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB_MSI&action=edit&redlink=1)
* Протокол MESI [en:Illinois protocol](http://en.wikipedia.org/wiki/Illinois_protocol) (intel pentium, Core)
* Протокол [MOSI](http://ru.wikipedia.org/w/index.php?title=MOSI&action=edit&redlink=1)
* Протокол [MOESI](http://ru.wikipedia.org/wiki/MOESI) (amd opteron[[1]](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B3%D0%B5%D1%80%D0%B5%D0%BD%D1%82%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%BA%D1%8D%D1%88%D0%B0#cite_note-w21-1))
* [MOWESI](http://ru.wikipedia.org/w/index.php?title=MOWESI&action=edit&redlink=1)[[2]](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B3%D0%B5%D1%80%D0%B5%D0%BD%D1%82%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%BA%D1%8D%D1%88%D0%B0#cite_note-mowesi-2)
* Протокол [MERSI](http://ru.wikipedia.org/w/index.php?title=MERSI&action=edit&redlink=1)
* Протокол [MESIF](http://ru.wikipedia.org/wiki/MESIF) (intel nehalem[[1]](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B3%D0%B5%D1%80%D0%B5%D0%BD%D1%82%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%BA%D1%8D%D1%88%D0%B0#cite_note-w21-1))
* Протокол [Write-once](http://en.wikipedia.org/wiki/Write-once_%28cache_coherency%29)
* [en:Synapse protocol](http://en.wikipedia.org/wiki/Synapse_protocol)
* [en:Berkeley protocol](http://en.wikipedia.org/wiki/Berkeley_protocol)
* [en:Firefly protocol](http://en.wikipedia.org/wiki/Firefly_protocol) (DEC)
* [en:Dragon protocol](http://en.wikipedia.org/wiki/Dragon_protocol) (Xerox)

Пример синхронизации кеша на основе механизма перехвата в SMP системе:

|  |  |  |
| --- | --- | --- |
| Операции | Локальный кэш | Кэш других процессоров |
| Промах при чтении | Запись из памяти в кэш | Ничего |
| Попадание при чтении | Использование кэша | Ничего (операция «не видна») |
| Промах при записи | Запись в память | Соответствующая запись в кэше удаляется |
| Попадание при записи | Запись в память и кэш | Соответствующая запись в кэше удаляется |

Рассмотрим проблему когерентности кешей на примере NUMA-системы (NUMA: Процессорные элементы работают на общем адресном пространстве, но характеристики доступа процессора к ОЗУ зависят от того, куда он обращается).

Для решения проблемы когерентности кешей можно использовать следующие решения:

* использовать процессоры без кэша (использовать только Cache L2)
* использовать ccNUMA (NUMA-системы с когерентными кэшами)

**Управление физической памятью в приложениях**

В наше время для большинства разработчиков управление физической памятью сводится к распределению страниц и отдельных блоков памяти и их освобождении. За распределение страниц памяти отвечает менеджер основной физической памяти (primary physical memory manager) ядра Linux. В его функции входит

* Выделение/освобождение физических страниц.
* Распределение областей физически непрерывных страниц по запросу

Работа с физической оперативной памятью целиком ложится на плечи операционной системы. Работа с кэш-памятью в большинстве современных процессоров реализована аппаратно и не подразумевает управление со стороны исполняемого кода.

Все операционные системы, использующие ядро GNU/Linux используют стандарты X/Open, в котором управление физической памятью полностью скрыто от приложения.

В операционных системах семейства Windows тоже нельзя управлять физической памятью напрямую — программисту доступна только виртуальная память. Однако стоит отметить, что выше обозначенные ограничения действительны только для обычных приложений пользовательского уровня. У ядра операционной системы, естественно, есть доступ к физической памяти компьютера. Впрочем, такие привилегии есть не только у ядра.

Доступом к физической памяти системы обладают также и драйвера — программные объекты, расположенные «между» пользовательскими приложениями и ядром операционной системы — поскольку драйверам необходим доступ к физическим устройствам, которыми они управляют.

Таким образом, написан специализированный драйвер для требуемой операционной системы, установив его (это потребует решения проблем, связанных с обеспечением безопасности) и настроив синхронизацию между драйвером и приложением пользовательского уровня, программист может получить доступ к физической памяти компьютера.

Кроме уже вышеперечисленных случаев, следует отметить операционные системы. Работающие на разнообразных системах на кристалле (system-on-chip). Поскольку их операционные системы и процессоры, как правило, устроены проще, чем операционные системы и процессоры general-purpose компьютеров, то при программировании на system-on-chip можно получить доступ к физической памяти значительно более простыми средствами.

Так же необходимо упомянуть ПЛИС (Программируемые логические интегральные схемы), они являются особенным классом устройств, однако в них пользователю (программисту) доступен очень низкоуровневый интерфейс, который позволяет управлять устройством на очень низком уровне, в частности управлять физической памятью, установленной на этом устройстве. (Необходимо иметь ввиду, что устройства ПЛИС бывают самыми разносторонними, и далеко не на всех вообще может иметься память, но там, где она есть, к ней имеется низкоуровневый доступ)

Ещё можно упомянуть, с какими ещё задачами могут столкнуться программисты, которые занимаются распределением памяти, хотя и не физической, а виртуальной, ведь это не менее важно.

Очень важно выделять память локально, даже, если её будет много, всё равно лучше, что бы взаимодействующие элементы находились как можно ближе.

Не менее важно создавать качественные аллокаторы, например, аллокатор, написанный Doug Lea, или многие другие аллокаторы, которых существет бесчисленное множество, из-за того, что каждый из них заточен под свою цель (скорость, минимизация служебной информации, и пр.). Если же перед программистом встают какие-то необычные условия, то ему придётся написать свой собственный аллокатор. (Однако это некоторое отступление от темы, потому что в данном случае программисы работают не с физической памятью, а с вирутальной)

**Реализация условных операторов в системе EPIC**

Важной особенностью EPIC-архитектуры является возможность параллельного ветвления в двух случаях: при выполнении команд условного перехода и при выполнении конструкций if-then-else в составе арифметических операторов. Условный переход в обычном понимании может привести к перезагрузке конвейера. В EPIC-архитектуре предусмотрен запуск дополнительного конвейера по команде подготовки перехода за несколько тактов до ветвления, т.е. таким образом у нас на куждую ветвь будет работать свой конвейер. Интенсивное ветвление в выполняемой программе способно привести к лавинообразному запуску дополнительных конвейеров и необходимости статистической оценки достаточного их количества при обосновании средств аппаратной поддержки.

Поэтому второй случай ветвления, осуществляемого внутри линейного участка программы, предпочтителен, хотя также приводит к избыточности оборудования, используемого при вычислениях.

Предикация - способ обработки условных ветвлений. Суть этого способа - компилятор указывает, что обе ветви выполняются на процессоре параллельно.

Опишем предикацию более подробно. Если в исходной программе встречается условное ветвление (по статистике - через каждые 6 команд), то команды из разных ветвей помечаются разными предикатными регистрами (команды имеют для этого предикатные поля), далее они выполняются совместно, но их результаты не записываются, пока значения предикатных регистров неопределены. Когда, наконец, вычисляется условие ветвления, предикатный регистр, соответствующий "правильной" ветви, устанавливается в 1, а другой - в 0. Перед записью результатов процессор будет проверять предикатное поле и записывать результаты только тех команд, предикатное поле которых содержит предикатный регистр, установленный в 1.

Поддержка предикатно-выполняемых команд, так же позволяет: 1) избежать излишних инструкций ветвления, если количество команд в ветвях условного оператора невелико; 2) уменьшить нагрузку на устройство предсказания переходов.

Введена поддержка инструкций циклического выполнения команд без потерь времени на инструкции циклического выполнения.