Крис Касперски

ПОДСИСТЕМА КЭШ-ПАМЯТИ КАК ОНА ЕСТЬ

kk@sendmail.ru

Крис Касперски
kk@sendmail.ru

Подсистема кэш-памяти как она есть

"*Память определяет быстродействие*"
Фон-Нейман

"*Самый медленный верблюд определяет скорость каравана*"
Арабское народное

"Когда караван поворачивает назад, самый медленный верблюд оказывается впереди"
Арабское народное

Прозрачность кэш-подсистемы современных процессоров сочетается с их капризным и весьма эгоцентричным характерам. Кэш похож на девушку, которая "хочет, но молчит", заставляя окружающих догадываться: что же у нее на уме, и как же ей угодить. И хотя робкие намеки на демократичность уже начали прорезаться, в целом кэш-подсистема представляет собой сплошную скопление чудес, сюрпризов и загадок. Это дремучий лес и официальная документация – плохой путеводитель, постоянно ставящий вас в тупик неполнотой, а то и откровенной недостоверностью информации.

Я искренне надеюсь, что вы сочтете настоящее описание кэш-подсистемы лучшим из имеющихся, но даже оно не освещает и доли тайн кэш-памяти! Перед вами – лишь небольшая часть того, что мне удалось нарыть. Увы! Сжатые временные сроки и ограниченный объем журнальной статьи не позволили рассказать обо всем и пришлось ограничится только самой необходимой информацией…

Истоки

Оперативная память персональных компьютеров сегодня, как и десять лет тому назад, строится на базе относительно дешевой, но низко производительной динамической памяти – ***DRAM*** (***D****ynamic* ***R****andom Access* ***M****emory*). Причем, если персональные компьютеры конца восьмидесятых – начала девяностых оснащались ми-

кропроцессорами с тактовой частотой порядка 10 MHz и оперативной памятью со временем доступа в ~200 нс., типичная конфигурация современного ПК: 1.000 – 2.000 MHz CPU и 20 ns. DRAM. Нетрудно подсчитать, что соотношение производительности памяти и процессора *уменьшилось* более чем в *десять* раз! Несмотря на стремительный рост пропускной способности оперативной памяти, наблюдающийся в последние годы, разрыв "CPU vs Memory" сокращаться отнюдь не собирается. Напротив, с каждым днем он продолжает неотвратимо увеличивается!

Статическая оперативная память (***SRAM*** – ***S**tatic **R**andom **A**ccess **M**emory*) без проблем работает на частотах вплоть до нескольких ГГц, но ввиду значительно большей аппаратной сложности стоит значительно дороже DRAM, и использовать ее в качестве основной оперативной памяти ПК было бы слишком расточительно. Но, может, если не всю, то хотя бы часть памяти реализовать на SRAM? Знаете, а это мысль! Ведь что по сути представляет собой оперативная память? Правильно, – временное хранилище данных, загруженных с внешней, так называемой, дисковой памяти. Диски слишком медленны и интенсивная работа с ними крайне непроизводительна. Поэтому, разместив многократно используемые данные в оперативной памяти, мы резко сокращаем время доступа к ним, а, значит, – и время их обработки.

На первый взгляд, выигрыш в производительности достигается в тех, и только в тех случаях, когда загруженные данные используются многократно. А вот и нет! Допустим, потребовалось нам перекодировать содержимое некоторого файла. Поскольку, к каждому байту обращение происходит лишь однократно, – какой смысл загружать его в оперативную память? Почему бы не перекодировать файл непосредственно на диске? Увы! Дисковод в силу своих конструктивных особенностей просто "не хочет" считывать один-единственный байт, насильно заставляя нас оперировать минимум всем *сектором* целиком. А раз так, – прочитанный сектор приходится где-то хранить. К тому же, обмен данными можно значительно ускорить, есть обрабатывать не один, а сразу несколько секторов за раз. В этом случае дисководу не придется тратить время на позиционирование головки при обращении к каждому сектору. Наконец, хранение данных в оперативной памяти позволяет отложить их немедленную запись до тех времен, пока это не будет "удобно" дисководу.

Таким образом, ***вовсе не обязательно всю оперативную память реализовывать на дорогостоящих микросхемах SRAM***. Даже небольшое (в процентном отношении) количество статической памяти при грамотном с ней обращении значительно увеличивает производительность системы, как и сама оперативная память значительно увеличивает производительность обмена с дисками, значительно превосходящими последнюю по объему.

OK. С памятью мы разобрались. Теперь поговорим о способах ее адресации. Если бы кусочек этой "быстрой" памяти мы вздумали бы адресовать непосредственно, т. е. сделали его *непосредственно* доступным программисту, проектирование программ ощутимо усложнилось бы и, что еще хуже, привело к полной потере переносимости, поскольку такая тактика привязывает программиста к особенностям реализации конкретной аппаратной архитектуры. Поэтому, конструкторы решили сделать "быструю" память *невидимой* и *прозрачной* для программиста.

В результате этого и родился ***кэш***...

Цели и задачи кэш-памяти

...кэш (называемый так же ***сверхоперативной памятью***) представляет собой высокоскоростное запоминающее устройство небольшой емкости для временного хранения данных, значительно более быстродействующее, чем основная память, но, в отличии от оперативной памяти, не адресуемое непосредственно и не "видимое" для программиста.

В задачи кэша входит:

a) *обеспечение быстрого доступа к интенсивно используемым данным*;

b) *согласование интерфейсов процессора и контроллера памяти*;

c) *упреждающая загрузка данных*;

d) *отложенная запись данных.*

Обеспечение быстрого доступа к интенсивно используемым данным. Архитектурно кэш-память расположена между процессором основной оперативной памятью (см. рис. 1) и охватывает все (реже часть) адресного пространства. Перехватывая запросы к основной памяти, кэш-контроллер смотрит: есть ли ***действительная*** (*валидная* от английского *valid*) копия затребованных данных в кэше. Если такая копия там действительно есть, то данные наскоро извлекаются из сверхоперативной памяти и происходит так называемое ***кэш-попадание*** (*cache hit*). В противном случае говорят о ***промахе*** – (*cache miss*), и тогда запрос данных переадресуется к основной оперативной памяти.

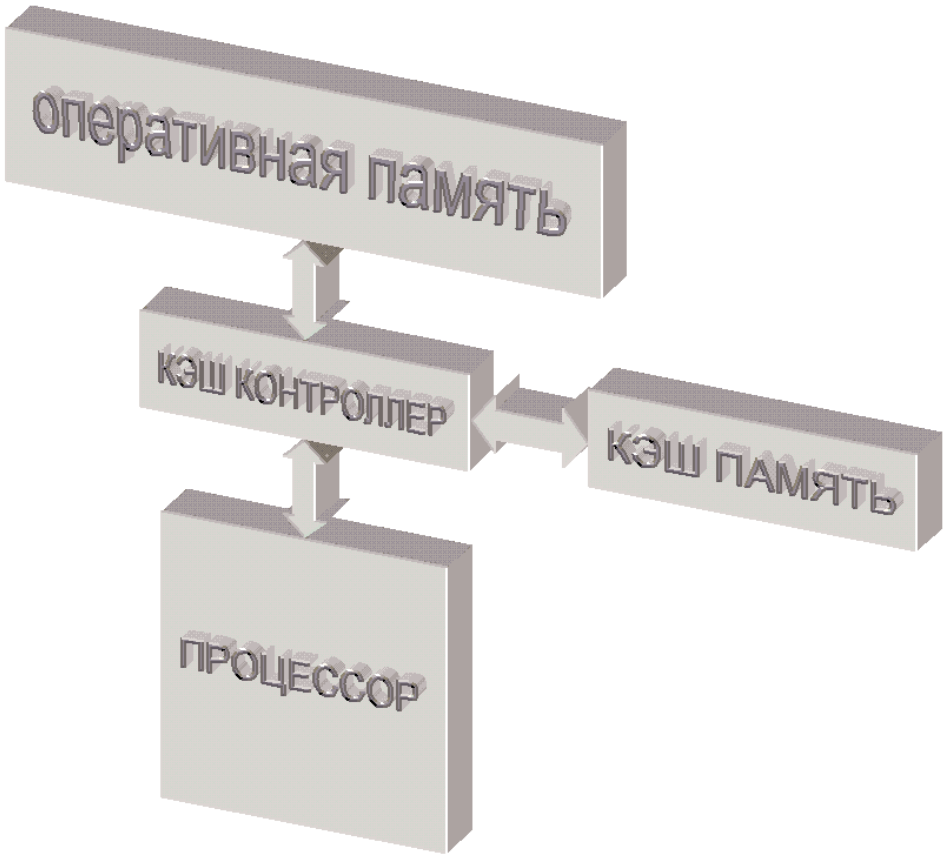

Рисунок 1. *Расположение кэша в иерархии оперативной памяти*

Для достижения наивысшей производительности кэш-промахи должны происходить как можно реже (а в идеале – не происходить вообще). Учитывая, что емкость сверхоперативной памяти намного меньше емкости основной оперативной памяти, добиться этого не так-то просто! И в служебные обязанности кэш-контроллера в первую очередь входит накопление в сверхоперативной памяти действительно нужных данных и своевременное удаление оттуда всякого "мусора", – данных, которые более не понадобятся. Поскольку, кэш-контроллер не имеет абсолютно никакого представления о назначении обрабатываемых данных, поставленная перед ним задача требует нехилого искусственного интеллекта. Но, увы! Кэш-контроллеры персональных процессоров интеллектом не обременены и слепо действуют по одному из нескольких шаблонов, называемых ***стратегиями кэширования***.

Стратегия помещения данных в кэш-память представляет собой алгоритм, определяющий: стоит ли помещать копию запрошенных данных в сверхоперативную память или нет? Процессоры класса Intel Pentium и процессоры AMD от K5 и выше не мудрствуя лукаво, помещают в кэш *все* данные, к которым хотя бы однократно происходит обращение.

Поскольку, мы не можем сохранить в кэше содержимое всей оперативной памяти и рано или поздно кэш заполняется по самую макушку (а с такой стратегией он заполняется скорее рано, чем поздно) настанет время, когда для помещения новой порции данных, нам придется в спешном порядке выкинуть из кэша что-нибудь ненужное, чтобы освобождать для них место. (Помните, как говорил кот Матрискин: "...ч*тобы продать что-нибудь ненужное, надо сначала купить что-нибудь ненужное..*")

Поиск вот таких наименее нужных данных и называется *стратегия замещения*. Можно принимать решение, основываясь на количестве обращений к каждой порции данных (частотный анализ), можно – на времени последнего обращения, выбрав ту, к которой дольше всего не обращались (алгоритм **LRU** – ***L**east **R**ecently **U**sed*), можно – на времени загрузки из основной памяти, вытеснив ту, которая была загружена раньше всех (алгоритм **FIFO** – ***F**irst **I**nput **F**irst **O**utput*), а можно просто подкинуть монетку (*randomize*-алгоритм) – на кого судьба ляжет, – ту и вытеснять (кстати, именно такая стратегия замещения использовалась в процессорах AMD K5).

В современных процессорах семейства x86 встречаются исключительно стратегии FIFO и LRU, частотный же анализ ввиду сложности его реализации в них не используется.

Согласование интерфейсов процессора и контроллера памяти. "Ячейка памяти" в понятии современных процессоров представляет собой как правило байт или двойное слово. С другой стороны, минимальной порцией обмена с физической оперативной памятью является *пакет*, состоящий по меньшей мере из четырех 64-разрядных ячеек.

Здесь можно провести аналогию с оптовой торговлей, – производитель не отпускает товар по штукам и если нам, положим, требуется один карандаш, мы все равно вынуждены приобретать целую упаковку. Естественно, до той поры, пока остальные карандаши не будут реально востребованы (конечно, если они вообще

будут востребованы), их необходимо где-то хранить. Решение: извлечь один-единственный карандаш из упаковки и выбросить остатки, – слишком нерационально, поэтому здесь не рассматривается. Тем более, что подходящее хранилище для пакетов данных у нас есть – это кэш. Получив пакет данных со склада, пардон, загрузив их из основной оперативной памяти, кэш позволяет процессору в последствие обрабатывать эти данные с любой разрядностью. Именно этим, кстати, объясняется выбранная стратегия загрузки данных (см. "*Стратегия помещения данных"*). Кэш-контроллер вынужден помещать в сверхоперативную памяти все ячейки, к которым происходит обращение, уже хотя бы потому, что выкидывать их, как карандаши в приведенном выше примере, было бы крайне нерационально.

Упреждающая загрузка данных. Существует несколько стратегий загрузки данных из основной оперативной памяти в кэш-память. Простейший алгоритм загрузки, называемый *загрузкой по требованию* (*on demand*), предписывает обращаться к основной памяти только после того, как затребованных процессором данные не окажется в кэше (то есть, попросту говоря, после возникновения кэш-промаха). Использование такой стратегии приводитк тому, что в кэш попадают действительно нужные нам данные (и это плюс!), однако, при первом обращении к ячейке, процессору придется очень долго ждать – приблизительно 20 тактов системной шины, что есть несомненный минус!

Стратегия *спекулятивной (speculative)* загрузки, напротив, предписывает помещать данные в кэш задолго то того, как к ним произойдет реальное обращение. Откуда же кэш-контроллеру знать, какие именно ячейки памяти потребуется процессору в ближайшем будущем? Ну... наверняка знать этого он этого, конечно, не может, но почему бы ему не попробовать просто *угадать*?

Алгоритмы угадывания делятся на *интеллектуальные* и *неинтеллектуальные*. Типичный пример неинтеллектуального алгоритма – *опережающая загрузка*. Исходя из предположения, что данные из оперативной памяти обрабатываются последовательно в порядке возрастания адресов, кэш-контроллер, перехватив запрос на чтение первой ячейки, в порядке собственной инициативы загружает некоторое количество ячеек, последующих за ней. Если данные действительно обрабатываются последовательно, то остальные запросы процессора будут выполнены практически мгновенно, ведь запрошенные ячейки уже присутствуют в кэше! Следует заметить, что стратегия опережающей загрузки возникает уже в силу необходимости согласования разрядности оперативной памяти и процессора (см. *"Согласование интерфейсов процессора и контроллера памяти"*).

Серьезный минус опережающей (и вообще неинтеллектуальной) загрузки состоит в том, что выбранный программистом алгоритм обработки данных далеко не всегда совпадает с алгоритмом их загрузки и зачастую ячейки памяти востребуются совсем не в том порядке, в котором кэш-контроллер запрашивает их из основной памяти. Как следствие, – мы имеем значительный падеж производительности, поскольку данные в этом случае загружаются в холостую.

Интеллектуальный кэш-контроллер предсказывает адрес следующей запрашиваемой ячейки не по слепому шаблону, а на основе анализа предыдущих обращений. Исследуя последовательность кэш-промахов, контроллер пытается установить какой именно зависимостью связны ее элементы и, если это ему удается,

предвычисляет ее последующие члены. Если обращение к памяти происходит по регулярному шаблону, интеллектуальная стратегия спекулятивной загрузки при благоприятном стечении обстоятельств может полностью ликвидировать задержки, возникающие при ожидании загрузки данных из основной памяти.

До недавнего прошлого интеллектуальные кэш-контроллеры использовались разве что в суперкомпьютерах и высокопроизводительных рабочих станциях, но теперь они реализованы в процессорах P-4 и AMD Athlon XP.

Стратегии поиска данных. В соответствии с выбранной стратегией загрузка данных из памяти может начинаться либо *после* фиксации кэш-промаха (*стратегия Look Through*), либо осуществляться *параллельно* с проверкой наличия соответствующей копии данных в сверхоперативной памяти и прерываться в случае кэш-попадания (*стратегия Look aside*). Последнее сокращает накладные расходы на кэш-промахи, уменьшая тем самым латентность загрузки данных, но зато увеличивает энергопотребление, что в ряде случаев оказывается неприемлемо большой платой за в общем-то довольно незначительную прибавку производительности.

Отложенная запись данных. Наличие временного хранилища данных позволяет накапливать записываемые данные и затем, дождавшись освобождения системой шины, выгружать их в оперативную память "одним махом". Это ликвидирует никому не нужные задержки и значительно увеличивает производительность подсистемы памяти (подробнее об этом см. "*Политики записи и поддержка когерентности*").

В x86 процессорах механизм отложенной записи реализован начиная с Pentium и AMD K5. Более ранние модели были вынужденные непосредственно записывать в основную память каждую модифицируемую ячейку, что серьезно ограничивало их быстродействие. К счастью, сегодня такие процессоры практически не встречаются и об этой проблеме уже можно забыть.

Организация кэша

Для упрощения взаимодействия с оперативной памятью (и еще по ряду других причин), кэш-контроллер оперирует не байтами, а блоками данных, соответствующих размеру пакетного цикла чтения/записи. Программно, кэш-память представляет собой совокупность блоков данных фиксированного размера, называемых ***кэш-линейками*** *(cache-line)* или ***кэш-строками***.

Каждая кэш-строка полностью заполняется (выгружается) за один пакетный цикл чтения и всегда заполняется (выгружается) целиком. Даже если процессор обращается к одному байту памяти, кэш-контроллер инициирует полный цикл обращения к основной памяти и запрашивает весь блок целиком. Причем, адрес первого байта кэш-линейки всегда кратен размеру пакетного цикла обмена. Другими словам: начало кэш-линейки всегда совпадает с началом пакетного цикла.

Поскольку, объем кэша много меньше объема основной оперативной памяти, каждой кэш-линейке соответствует множество ячеек кэшируемой памяти, а отсю-

да с неизбежностью следует, что приходится сохранять не только содержимое кэшируемой ячейки, но и ее адрес. Для этой цели каждая кэш-линейка имеет специальное поле, называемое *тегом*. В теге хранится линейный и/или физический адрес первого байта кэш-линейки. Т. е. кэш-память фактически является *ассоциативной памятью* (*associative memory*) по своей природе.

В некоторых процессорах (например, в младших моделях процессоров Pentium) используется только один набор тегов, хранящих физические адреса. Это удешевляет процессор, но для преобразования физического адреса в линейный требуется по меньшей мере один дополнительный такт, что снижает производительность.

Другие же процессоры (например, AMD K5) имеют два набора тегов, для хранения физических и линейных адресов соответственно. К физическим тегам процессор обращается лишь в двух ситуациях: при возникновении кэш-промахов (в силу используемой в x86 процессорах схемы адресации одна и та же ячейка может иметь множество линейных адресов и потому несовпадение линейных адресов еще не свидетельство промаха) и при поступлении запроса от внешних устройств (в т. ч. и других процессоров в многопроцессорных системах): имеется ли такая ячейка в кэш памяти или нет (см. "*Протокол MESI*"). Во всех остальных случаях используются исключительно линейные теги, что предотвращает необходимость постоянного преобразования адресов.

Доступ к ассоциативной памяти, в отличии от привычной нам адресной памяти, осуществляется не по номеру ячейки, а по ее содержанию, поэтому такой тип памяти еще называют *content addressed memory*. Кэш-строки, в отличии от ячеек оперативной памяти, не имеют адресов и могут нумероваться в произвольном порядке, да и то чисто условно. Выражение "кэш-контроллер обратился к кэш-линейке №69" лишено смысла и правильнее сказать: "кэш-контроллер, обратился к кэш-линейке 999", где 999 – *содержимое* связанного с ней тега.

Таким образом, полезная емкость кэш-памяти всегда меньше ее полной (физической) емкости, т.к. некоторая часть ячеек расходуется на создание тегов, совокупность которых так и называется "память тегов" (остальные ячейки образуют "память кэш-строк"). Следует заметить, что производители всегда указывают именно *полезную*, а не полную емкость кэш-памяти, поэтому, за память, "отъедаемую" тегами, потребителям можно не волноваться.

Блокируемая и не блокируемая кэш память. Существует две основных разновидности сверхоперативной памяти: ***блокируемая кэш-память*** и ***не блокируемая***. Странно, но расшифровка этого термина во многих популярных изданиях отсутствует (я, например, впервые обнаружил ее в технической документации по процессору AMD K5), поэтому имеет смыл рассмотреть этот вопрос поподробнее.

Блокируемая кэш память, как и следует из ее названия, блокирует доступ к кэшу после всякого кэш-промаха. Независимо от того, присутствуют ли запрашиваемые данные в сверхоперативной памяти или нет, до тех пор, пока кэш-строка, вызвавшая промах не будет целиком загружена (выгружена), кэш не сможет обрабатывать никаких других запросов ("*...a read hit or miss after a read miss waits until the prior miss fills the cache*", – как говорят англичане). В настоящее время блокируемая кэш память практически не используется, поскольку при частных кэш-промахах, она работает крайне непроизводительно.

Не блокируемая кэш память, напротив, позволяет работать с кэшем параллельно с загрузкой (выгрузкой) кэш-строк. То есть, кэш-промахи не препятствует кэш-попаданиям. И это – хорошо! Несмотря на то, что не блокируемая кэш память имеет значительно большую аппаратную сложность (а, значит, – и стоимость), в силу своей привлекательности, она широко используется в старших процессорах семейства x86, как, впрочем, и во многих других современных процессорах.

Понятие ассоциативности кэша. Проследим по шагам как работает кэш. Вот процессор обращается к ячейке памяти с адресом xyz. Кэш-контроллер, перехватив это обращение, первым делом пытается выяснить: присутствует ли запрошенные данные в кэш-памяти или нет? Вот тут-то и начинается самое интересное! Легко показать, что проверка наличия ячейки в кэш-памяти фактически сводится к *поиску* соответствующего диапазона адресов в памяти тегов.

В зависимости от архитектуры кэш-контроллера просмотр всех тегов осуществляется либо параллельно, либо последовательно. Параллельный поиск, конечно, чрезвычайно быстр, но и сложен в реализации (а потому – дорог). Последовательный же перебор тегов при большом их количестве крайне непроизводителен. Кстати, а сколько у нас тегов? Правильно – ровно столько, сколько и кэш-строк. Так, в частности, в 32 килобайтном кэше насчитывается немногим более *тысячи* тегов.

Стоп! Сколько времени потребует просмотр тысячи тегов?! Даже если несколько тегов будут просматриваться за один такт, поиск нужной нам линейки растянется на сотни тактов, что "съест" весь выигрыш в производительности. Нет уж, какая динамическая память ни тормозная, а к ней обратится побыстрее будет, чем сканировать кэш...

Но ведь кэш все таки работает! Спрашивается: как? Оказывается (и это следовало ожидать), что последовательный перебор – не самый продвинутый алгоритм поиска. Существуют и более элегантные решения. Рассмотрим два наиболее популярные из них.

В кэше ***прямого отображения*** проблема поиска решается так: пусть каждая ячейка памяти соответствует не любой, а одной, строго определенной, строке кэша. В свою очередь, каждой строке кэша будет соответствовать не одна, а множество ячеек кэшируемой памяти, но опять-таки, не любых, а строго определенных.

Пусть наш кэш состоит из четырех строк, тогда (см. рис. 2) первый пакет кэшируемой памяти связан с первой строкой кэша, второй – со второй, третий – с третьей, четвертый – с четвертой, а пятый – вновь с первой! Достаточно очевидно, что адреса ячеек кэшируемой памяти связаны с номерами кэш-строк следующим отношением: $N = \frac{ADDR}{CACHE.LINE.SIZE} \bmod \frac{CACHE.SIZE}{CACHE.LINE.SIZE}$, где N – условный номер кэш-линейки, ADDR – адрес ячейки кэшируемой памяти; CACHE.LINE.SIZE – длина кэш-линейки в байтах; CACHE.SIZE – размер кэш-памяти в байтах; "—" – операция целочисленного деления.

Таким образом, чтобы узнать: присутствует ли искомая ячейка в кэш-памяти или нет, достаточно просмотреть всего один-единственный тег соответствующей кэш-линейки, для вычисления номера которого требуется совершить всего три

арифметические операции (поскольку, длина кэш-линеек и размер кэш памяти всегда представляют собой степень двойки, операции деления и взятия остатка допускают эффективную и простую аппаратную реализацию). Необходимость просматривать все теги в этой схеме естественным образом отпадает.

Да, отпадает, но возникает другая проблема. Задумайтесь, что произойдет, если процессор попытается последовательно обратиться ко второй, шестой и десятой ячейкам кэшируемой памяти? Правильно – несмотря на то, что в кэше будет полно свободных строк, каждая очередная ячейка будет вытеснять предыдущую, т.к. все они жестко закреплены именно за второй строкой кэша. В результате кэш будет работать максимально неэффективно, полностью ***вхолостую*** (*trashing*).

Программист, заботящийся об оптимизации своих программ, должен организовать структуры данных так, чтобы исключить частое чтение ячеек памяти с адресами, кратными размеру кэша. Понятное дело – такое требование не всегда выполнимо и кэш прямого отображения обеспечивает далеко не лучшую производительность. Поэтому, в настоящее время он практически нигде не встречаются и полностью вытеснен наборно-ассоциативной сверхоперативной памятью.

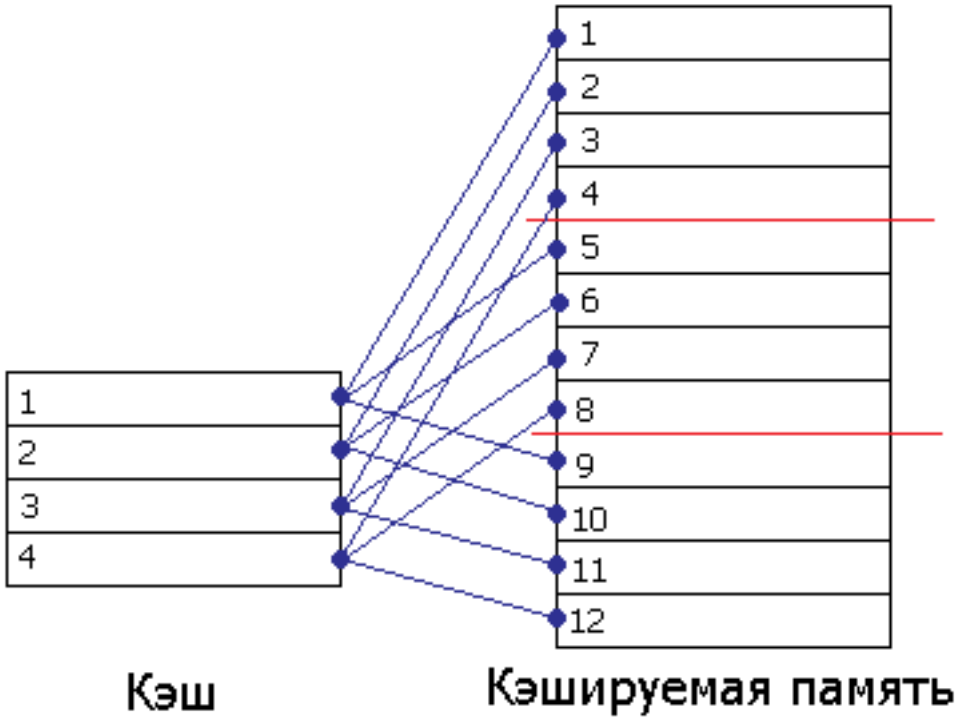

Рисунок 2. *Устройство кэша прямого отображения*

Наборно-ассоциативным кэш состоит из нескольких независимых *банков,* каждый из которых представляет собой самостоятельный кэш прямого отображения. Взгляните на рисунок 3. Видите, каждая ячейка кэшируемой памяти может быть сохранена в *любой* из двух строк кэш-памяти. Допустим, процессор читает шестую и десятую ячейку кэшируемой памяти. Шестая ячейка идет во вторую строку первого банка, в десятая – во вторую строку следующего банка, т.к. первый уже занят.

Количество банков кэша и называют его *ассоциативностью (way)*. Легко видеть, что с увеличением степени ассоциативности, эффективность кэша существенно возрастает.

В идеале, при наивысшей степени дробления, в каждом банке будет только одна линейка, и тогда любая ячейка кэшируемой памяти сможет быть сохранена в любой строке кэша. Такой кэш называют ***полностью ассоциативным кэшем*** или просто ассоциативным кэшем.

Ассоциативность кэш-памяти, используемой в современных персональных компьютеров колеблется от двух (2-way cache) до восьми (8-way cache), а чаще всего равна четырем (4-way cache).

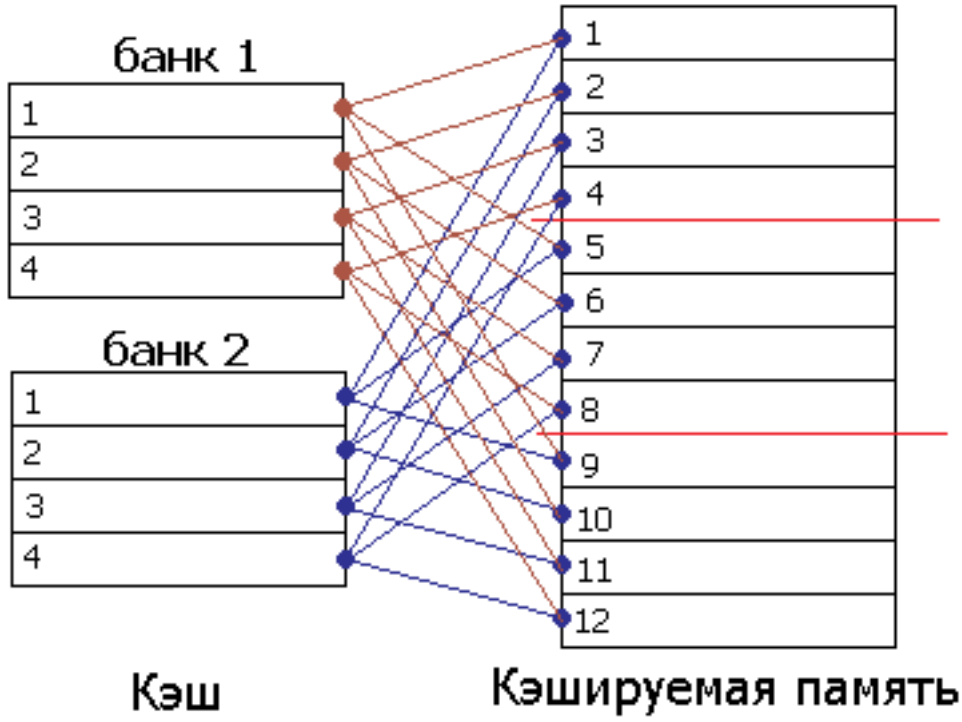

Рисунок 3. *Устройство наборно-ассоциативного кэша*

Политики записи и продержка когерентности. Если бы ячейки памяти были доступны только на чтение, то их скэшированная копия всегда совпадала бы с оригиналам. Возможность записи (ну какая же программа обходится без операций записи?) рождает следующие проблемы: во-первых, кэш-контроллер должен отслеживать модификацию ячеек кэш-памяти, выгружая в основную память модифицированные ячейки при их замещении, а, во-вторых, необходимо как-то отслеживать обращения всех периферийных устройств (включая остальные микропроцессоры в многопроцессорных системах) к основной памяти. В противном случае, мы рискуем считать совсем не то, что записывали!

Кэш-контроллер обязан обеспечивать *когерентность* (*coherency*) – согласованность кэш-памяти с основной памятью. Допустим, к некоторой ячейке памяти, уже модифицированной в кэше, но еще не выгруженной в основную память, обра-

щается периферийное устройство (или другой процессор) – кэш-контроллер должен немедленно обновить основную память, иначе оттуда почитаются "старые" данные. Аналогично, если периферийное устройство (другой процессор) модифицирует основную память, например посредством DMA, кэш-контроллер должен выяснить – загружены ли в модифицированные ячейки в его кэш-память, и если да – обновить их.

Поддержка когерентности – задача серьезная. Самое простое (но не самое лучшее) решение, мгновенно приходящее на ум, – кэшировать ячейки основной памяти только для чтения, а запись осуществлять напрямую, минуя кэш, сразу в основную память. Это, так называемая, ***сквозная*** (***W**rite **T**rue write policy*) политика. Сквозная политика легка в аппаратной реализации, но крайне неэффективна.

Частично компенсировать задержки обращения к памяти помогает *буферизация*. Записываемые данные на первом этапе попадают не в основную память, а в специальный *буфер записи* (*store/write buffer*), размером порядка 32-байт. Там они накапливаются до тех пор, пока буфер целиком не заполниться или не освободиться шина, а затем все содержимое буфера записывается в память "одним скопом". Такой режим ***сквозной записи с буферизацией*** (***W**rite **C**ombining write policy*) значительно увеличивает производительность системы, но решает далеко не все проблемы. В частности, значительная часть процессорного времени по-прежнему расходуется именно на выгрузку буфера в основную память. Тем более обидно, что в подавляющем большинстве компьютеров установлен всего один процессор и именно он, а не периферия, интенсивнее всех работает с памятью – не слишком ли дорого обходится поддержка когерентности?

Более сложный (но и совершенный!) алгоритм реализует ***обратная политика записи*** (***W**rite **B**ack write policy*), до минимума сокращающая количество обращений к памяти. Для отслеживания операций модификации с каждой ячейкой кэш-памяти связывается специальный флаг, называемый *флагом состояния*. Если кэшируемая ячейка была модифицирована, то кэш-контроллер устанавливает соответствующий ей флаг в *грязное* (*dirty*) состояние. Когда периферийное устройство обращается к памяти, кэш-контроллер проверяет – находится ли соответствующий адрес в кэш-памяти и если да, тогда он, глядя на флаг, определяет: грязная она или нет? Грязные ячейки выгружаются в основную память, а их флаг устанавливается в состояние "*чисто*" (*clear*). Аналогично – при замещении старых кэш-строк новыми, кэш-контроллер в первую очередь стремится избавиться от чистых кэш-строк, т. к. они могут быть мгновенно удалены из кэша без записи в основную память. И только если все строки грязные – выбирается одна, наименее ценная (с точки зрения политики замещения данных) и "сбрасывается" в основную память, освобождая место для новой, "чистой" строки.

Протокол MESI. Под загадочной аббревиатурой MESI, частенько встречающийся в отечественной и зарубежной литературе, скрывается ни что иное, как первые буквы четырех статусов кэш-линейки ***M**odified **E**xclusive **S**hared **I**nvalid (Модифицированная Девственная Скоммунизденая Инвалидная)*.

Но что каждый из этих статусов обозначает? Вот это мы сейчас и рассмотрим! Итак...

Статус "***M**odified"* автоматически присваивается кэш-строкам при их модификации. Строка с таким атрибутом не может быть просто выброшена из кэша и при ее вытеснении обязательно должна выгружаться в кэш память более высокой иерархии или же основную память;

Статус "***E**xclusive"* автоматически присваивается кэш-строкам при их загрузке из кэша более высокой иерархии или основной оперативной памяти. Модификация строки с атрибутом **E**xclusive влечет его автоматическую смену на атрибут **M**odified.

Строка с атрибутом **E**xclusive при ее вытеснении из кэша в зависимости от архитектурных особенностей системы либо просто уничтожается (inclusive-кэш), либо обменивается своим содержимым с одной из строк кэш-памяти более высокой иерархии (exclusive–кэш). см. так же. "*Двухуровневая организация кэша*"

Статус "***S**hared"* присваивается кэш-строкам, потенциально присутствующим в кэш-памяти других процессоров (если это многопроцессорная система). Помимо этого, атрибут **S**hared указывает еще и на то, что строка когерентна содержимому соответствующих ей ячеек основной памяти. Поскольку, многопроцессорные системы далеко выходят за рамки нашего разговора, отложим этот вопрос до "лучших времен".

(***Примечание**: в AMD Athlon добавился новый статус "Owner" – "Владелец", и сам протокол стал записываться так: MOESI*).

Статус "***I**nvalid"* строка отсутствует в кэше и должна быть загружена из кэш памяти более высокой иерархии или же основной памяти.

Кэш данных первого уровня и кэш второго уровня Pentium- и AMD K6\Athlon процессоров поддерживает все четыре статуса, а кэш кода – только два из них ***S**hared и **I**nvalid*. Остальные не поддерживаются по той простой причине, что кодовый кэш не допускает модификации своих линеек. А как же в этом случае работает самомодифицирующий код? – удивится иной читатель. А кто вам сказал, что он вообще работает? – возражу я. Независимо от того, присутствует ли модифицируемая ячейка в кодовом кэше или нет, инструкция записи не может непосредственно изменить ее содержимое, и она помещается в кэш первого (второго) уровня или основную оперативную память. Несмотря на то, что процессор все-таки отслеживает эти ситуации и обновляет соответствующие строки кодового кэша, самомодифицирующегося кода по возможности следует избегать, поскольку: во-первых, при обновлении строк гибнет вся преддекодированная информация, а, во-вторых, процессору приходится очищать конвейер и вновь начинать его заполнять сначала.

Причем, под самомодифицирующимся кодом в современных системах понимается не только истинно самомодифицирующийся код в его каноническом понимании, но и вообще всякая запись в область памяти, находящуюся в кодовом кэше. То есть, смесь кода и данных, которая так часто встречается в "ручных" ассемблерных программах, будет исполняться не скорее асфальтового катка, хотя формально она и не изменяет машинный код (но процессор-то об этом не знает!).

Статус	**Modified**	**Exclusive**	**Shared**	**Invalid**
эта кэш линия действительна?	да	да	да	нет
копия в памяти действительна?	устарела	действительна	действительна	этой строке вообще не соответствует никакая память
содержится ли копия этой строки в других процессорах?	нет	нет	может быть	может быть
запись в эту линию осуществляется...	только в эту строку, без обращения к шине	только в эту строку, без обращения к шине	сквозной записью в память с аннулированием строки в кэшах остальных процессоров	непосредственно через шину

Двухуровневая организация кэша. Предельно достижимая емкость кэш-памяти ограничена не только ее ценой, но и электромагнитной интерференцией, налагающей жесткие ограничения на максимально возможное количество адресных линий, а значит – на непосредственно адресуемый объем памяти. В принципе, мы можем прибегнуть к мультиплексированию выводов или последовательной передаче адресов (как, например, поступили разработчики Rambus RDRAM), но это неизбежно снизит производительность и доступ к ячейке кэш-памяти потребует более одного такта, что не есть хорошо.

С другой стороны, двух портовая статическая память действительно очень дорога, а одно-портовая не в состоянии обеспечить параллельную обработку нескольких ячеек, что приводит к досадным задержкам.

Естественный выход состоит в создании многоуровневой кэш-иерархии (см. рис. 4). Большинство современных систем имеют как минимум два уровня кэш памяти. Первый, наиболее "близкий" к процессору (условно обозначаемый ***Level 1*** или сокращенно ***L1***), обычно реализуется на быстрой двух портовой синхронной статической памяти, работающей на полной частоте ядра процессора. Объем L1-кэша весьма не велик и редко превышает 32 Кб, поэтому он должен хранить только самые-самые необходимые данные. Зато на обработку двух полно разрядных ячеек уходит всего один такт. (***Внимание***: *процессоры x86 оснащаются не истинно двух портовой памятью! Двух портовый у нее лишь интерфейс, а ядро памяти состоит из нескольких независимых банков, – обычно восьми, реализованных на одно-портовый матрицах, и параллельный доступ возможен лишь к ячейкам разных банков*).

Между кэшем первого уровня и оперативной памятью расположен *кэш второго уровня* (условно обозначаемый ***Level 2*** или сокращенно ***L2***). Он реализуется на одно-портовой конвейерной статической памяти (BSRAM) и зачастую работает на пониженной тактовой частоте. Поскольку одно-портовая память значительно

дешевле, объем L2 кэша составляет сотни килобайт, а зачастую достигает и нескольких мегабайт! Между тем скорость доступа к нему относительно невелика (хотя, естественно, многократно превосходит скорость доступа к основной памяти).

Во-первых, минимальной порцией обмена между L1 и L2 кэшем является отнюдь не байт, а целая кэш-линейка, на чтение которой уходит в среднем 5 тактов частоты кэша второго уровня (формула BSRAM памяти выглядит так: 2-1-1-1). Если L2-кэш работает на половинной частоте процессора, то обращение к одной ячейке займет целых 10 тактов. Разумеется, эту величину можно сократить. В серверах и высокопроизводительных рабочих станциях кэш второго уровня чаще всего работает на полной частоте ядра процессора и зачастую имеет учетверенную разрядность шины данных, благодаря чему пакетный цикл обмена завершается всего за один такт. Однако и стоимость таких систем соответствующая.

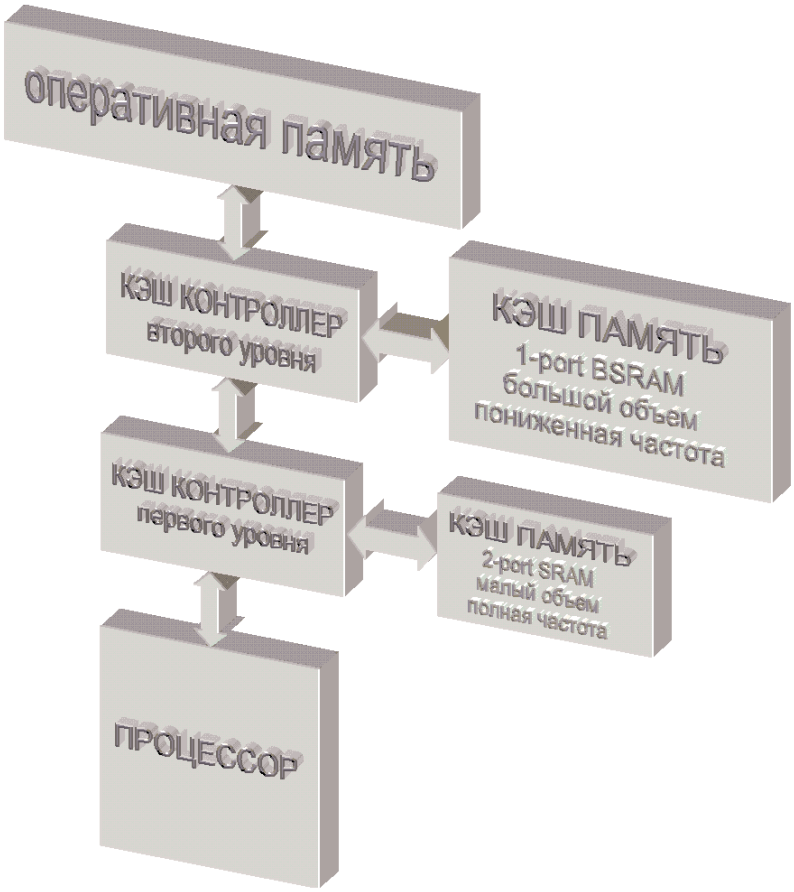

Рисунок 4. *Двух уровневая кэш-иерархия*

Включающая (inclusive) архитектура. Кэш второго уровня, построенный по inclusive-архитектуре, всегда дублирует содержимое кэша первого уровня, а потому эффективная емкость сверхоперативной памяти обоих иерархий равна: L2.CACHE_SIZE – L1.CACHE.SIZE + L1.CACHE.SIZE == L2.CACHE.SIZE.

Давайте рассмотрим, что происходит в системе, когда при полностью заполненном кэше второго уровня, процессор пытается загрузить еще одну ячейку. Обнаружив, что все кэш-линейки заняты, кэш второго уровня избавляется от наиме-

нее ценной из них, стремясь при этом найти линейку, которая еще не была модифицирована, поскольку в противном случае еще придется выгружать в основную оперативную память, а это – время.

Затем кэш второго уровня передает полученные из памяти данные кэшу первого уровня. Если кэш первого уровня так же заполнен под завязку ему приходится избавляться от одной из строк по сценарию, описанному выше.

Таким образом, загруженная порция данных присутствует и кэш-памяти первого уровня, и в кэш-памяти второго, что не есть хорошо. Между тем, практически все современные процессоры (AMD K6, P-II, P-III) построенные именно по включающей архитектуре.

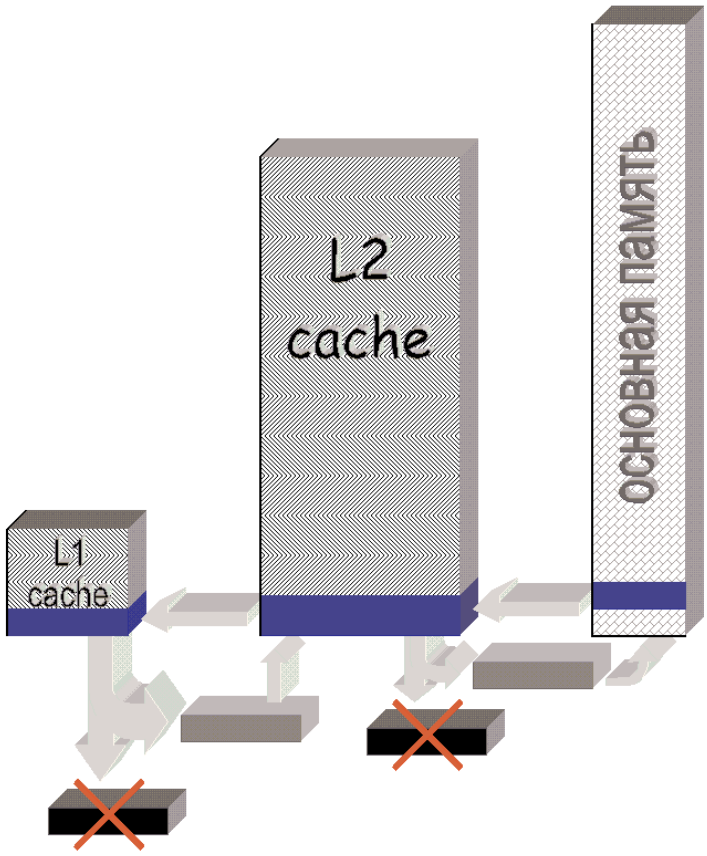

Рисунок 5. *inclusive- (слева) и exclusive- (справа) архитектуры*

Исключающая (exclusive) архитектура. Кэш-подсистема, построенная по exclusive-архитектуре, никогда не хранит избыточных копий данных и потому эффективная емкость сверхоперативной памяти определяется *суммой* размеров сверхоперативной памяти всех иерархий.

Кэш-первого уровня никогда не уничтожает кэш-линейки при нехватке места. Даже если они не были модифицированы, – данные в обязательном порядке вытесняются в кэш-второго уровня, помещаясь на то место, где находилась только что переданная кэшу первого уровня линейка. Т. е. кэш первого и кэш второго уровней как бы обмениваются друг с другом своими линейками, а потому сверхоперативная память используется весьма эффективно.

На сегодняшний день эксклюзивная кэш-подсистема реализована в одном лишь процессоре AMD Athlon, да и то не с первых моделей.

Буфера записи. Для предотвращения задержки, возникающей при промахах записи, современные процессоры активно используют различные приемы буферизации. Вместо того, чтобы немедленно отправлять записываемые данные по месту назначения, процессор временно помещает их в специальный буфер, откуда, по мере освобождения шины и/или кэш-контроллера они выгружаются в кэш первого (второго) уровня или в основную оперативную память.

Такая схема особенно полезна при значительном превышении количества операций чтения над числом операций записи. Если частота возникновения промахов записи не превышает скорости выгрузки буферов, штрафных задержек не возникает вовсе и эффективная скорость выполнения инструкции записи составляет 1 такт.

Часто приходится слышать: чем больше в процессоре буферов, тем выше предельная частота промахов записи, которую без ущерба для производительности он способен выдержать. На самом деле, это неверно. *Максимальная частота промахов определяется не количеством буферов, а скоростью (и политикой) их выгрузки.*

В частности, процессоры AMD K6 и Athlon всегда выгружают содержимое буферов записи в кэш первого уровня, благодаря чему скорость их опорожнения весьма велика (при благоприятном стечении обстоятельств каждый 32/64- байтовый буфер выгружается за один такт), а выгруженные данные вплоть до вытеснения их из L1-кэша доступны на чтение практически мгновенно!

Процессоры P6 и P-4, напротив, направляют содержимое буферов записи в кэш второго, а не первого уровня. (Конечно, если записываемые данные уже содержаться в L1-кэше они помещаются именно туда). Эффективность такой стратегии не бесспорна. С одной стороны: это приводит к тому, что процессор переносит значительно меньшую интенсивность промахов записи, а с другой: выгрузка записываемых данных в кэш второго уровня не загрязняет кэш первого уровня, в конечном итоге увеличивая его эффективную емкость.

Кстати, нетривиальным следствием такой политики выгрузки буферов, становится *искажение политики кэш-записи.* Несмотря на то, что кэш первого уровня формально построен по Write Back архитектуре, фактически он работает по схеме Write True, поскольку промах записи приводит к непосредственному обновлению кэш-памяти более высокой иерархии без загрузки модифицируемых данных в кэш. Забавно, но Intel впервые "проговорилась" об этом факте лишь в документации по процессору Pentium-4, где в графе "политика записи кэша первого уровня" честно указано "Write True", а не "Write Back", как это утверждалось в документации по процессорам предыдущего поколения.

Важно понять, что увеличение количества буферов записи не компенсируют медлительность их выгрузки в память. Практически единственная польза емкого буфера в том, что он безболезненно переносит *локальные* перегрузки, когда несколько промахов возникают одновременно (или идут с минимальным отрывом друг от друга), а затем вновь наступает "тишина".

Чтение после записи. Вопреки своему названию, каждый из буферов доступен не только на запись, но и на *чтение*. Причем, чтение данных из буфера записи осуществляется по крайней мере на один такт быстрее, нежели из кэш-памяти первого уровня.

Тем не менее, чтение содержимого буферов (особенно на процессорах семейства Pentium) следует осуществлять с большой осторожностью, т.к. только что записанные данные в любой момент могут перекочевать в кэш второго уровня, что резко увеличит время доступа к ним.

Упорядочивание записи. Помимо всего прочего, на буфера еще возложена и функция *упорядочивания записи*. Старшие представители семейства x86 разбивают машинные инструкции на микрооперации, выполняя их в наиболее предпочтительном с точки зрения RISC-ядра процессора, порядке. Но ведь нарушение очередности операций чтения/записи в память (кэш) может запросто нарушить нормальную работу программы!

Задумайтесь, что произойдет если в следующем блоке кода `"a = *p; *p = b;"` запись ячейки `*p` завершится раньше ее чтения? Спрашивайте, а почему такое вообще может случиться? Да мало ли почему! Допустим, блок чтения данных занят обработкой предыдущей инструкции и команда `a = *p` вынуждена терпеливо дожидаться свой очереди, в то время как команда `*p = b` захватывается бездельничающим в этот момент блоком записи.

Конечно, блок записи можно до завершения чтения и притормозить, но производительности такая мера не добавит – это уж точно. Выход?! Выход: временно сохранить записываемые данные не в кэше (основной памяти), а в некотором промежуточном буфере. Чтобы не захламлять архитектуру микроядра и не вводить еще один буфер, конструкторы решили для этих целей использовать все те же буфера записи. Политика выгрузки данных из буфера гарантирует, что данные, помещенные в буфер, покинут его "застенки" не раньше, чем завершаться все, предшествующие им инструкции. Таким образом, с буфера данные сходят уже упорядоченными и потому никаких конфликтов не возникает.

Причем, буферизация записи в определенной степени сокращает количество обращений к памяти, поскольку если одна и та же ячейка записывалась несколько раз в память (кэш) попадает лишь последний результат.

Реализация и характеристики буферов записи. Младшие модели Pentium имели всего два буфера записи (*write buffers*) – по одному на каждую U- и V-трубу, но уже в Pentium MMX количество буферов возросло до четырех, а в Pentium II их и вовсе насчитывается *двенадцать*! Причем, начиная с Pentium II буфера записи переименованы в *Складские Буфера* (*store buffers*), однако, во избежании путаницы, здесь и далее мы будем пользоваться исключительно прежним термином. Все равно "store" и "write" в русском переводе – синонимы (во всяком случае, в рамках данного контекста)

Учитывая, что размер каждого из буферов составляет 32 байт, можно безболезненно сохранять до 384 байт (96 двойных слов) за раз, не беспокоясь за кэш-промахи. Но помните, что попытки записи в некэшируемую память при полностью заполненный буферах приводят к неустранимым кэш-промахам и вытекающих отсюда штрафным задержкам.

Поэтому, целесообразно чередовать операции записи с вычислениями – давая буферам время на выгрузку своего содержимого.

Кэш-подсистема современных процессоров

Кэш подсистема процессоров P-II, P-III, P-4 и AMD Athlon представляет собой многоуровневую иерархию, состоящую из следующих компонентов: ***кэша данных первого уровня***, ***кэша команд первого уровня***, ***общего кэша второго уровня***, ***TLB-кэша страниц данных***, ***TLB-кэша страниц кода***, ***буфера упорядочивая записи*** и ***буферов записи*** (см. рис. 6).

MOB: Данные, сходящие с вычислительного конвейера, первым делом попадают на **MOB** (***M**emory **O**rder **B**uffer* – *буфер упорядочивая записи к памяти*) где они, постепенно накаливаясь, ожидают своей очереди выгрузки в паять. Грубо говоря, буфер упорядоченной записи играет тут же самую роль, что и зал ожидания в аэропорту. Пассажиры прибывают туда в более или менее случайном порядке, но улетают в строгом соответствии со временем, указанном в билете, да и то при условии, что к этому моменту выдастся летная погода и самолету предоставят "коридор" (кто летал – тот поймет).

Данные, находящиеся в MOB, всегда доступы процессору, даже если они еще не выгружены в память, однако емкость буфера упорядоченной записи довольно невелика (40 входов на P6) и при его переполнении вычислительный конвейер блокируется. Поэтому, содержимое MOB должно при всякой возможности незамедлительно выгружаться оттуда. Это происходит по крайней мере тремя путями:

а) если модифицируемая ячейка уже присутствует в кэш-памяти первого уровня, то она прямиком направляется в соответствующую ей кэш-строку, на что уходит всего один такт, в течении которого в кэш может быть записана одна или даже две любых несмежные ячейки (максимальное количество одновременно записываемых ячеек определяется архитектурой кэш-подсистемы конкретного процессора;

б) если модифицируемая ячейка отсутствует в кэш-памяти первого уровня, она, при наличии хотя бы одного свободного буфера записи, попадает туда. Это так же занимает всего один такт, причем, максимальное количество параллельно записываемых ячеек определяется количеством портов, имеющихся в "распоряжении" у буферов записи (например, процессоры AMD K5 и Athlon содержат только один такой порт);

с) если модифицируемая ячейка отсутствует в кэш-памяти первого уровня и ни одного свободного буфера записи нет, – процессор самостоятельно загружает соответствующую копию данных в кэш-первого уровня, после чего переходит к пункту а). В зависимости от ряда обстоятельств, загрузка данных занимает от десятков до сотен (а то и десятков тысяч!) тактов процессора, поэтому, таких ситуаций по возможности следует избегать.

L1 CAHE. Кэш первого уровня размещается непосредственно на кристалле и реализуется на базе двух портовой статической памяти. Он состоит из двух независимых банков сверхоперативной памяти, каждый из которых управляется "своим" кэш-контроллером. Один кэширует машинные инструкции, другой – обрабатываемые ими данные. В кратной технической спецификации процессора

обычно указывается суммарный объем кэш-памяти первого уровня, что приводит к некоторой неопределенности, т. к. емкости кэша инструкций и кэша данных не обязательно должны быть равны (а на последних процессорах они и не равны).

Каждый банк кэш-первого уровня помимо собственно данных и инструкций содержит и буфера ассоциативной трансляции (TLB) страниц данных и страниц кода соответственно. Под буфера ассоциативной трансляции отводятся фиксированные линейки кэша и занимаемое ими пространство "официально" исключено из емкости кэш-памяти. Т. е. если в спецификации сказано, что на процессоре установлен 8 Кб кэш данных, – все эти 8 Кб непосредственно доступны для кэширования данных, а реальная емкость кэш-памяти в действительности же превосходит 8 Кб.

Буфера Записи. Если честно, то у автора нет полной ясности где конкретно в кэш-иерархии расположены буфера записи. На блок-диаграммах процессоров Intel Pentium и AMD Athlon, приведенных в документации, они вообще отсутствуют, а в § 9.1 "INTERNAL CACHES, TLBS, AND BUFFERS" главы "MEMORY CACHE CONTROL" руководства по системному программирования от Intel, буфера записи изображены чисто условно и явно не в том месте, где им положено быть (сам Intel пишет, что "буфера записи связны с исполнительным блоком процессора", а на рисунке подсоединяет их к блоку интерфейсов с шиной – с каких это пор последний стал "исполнительным блоком процессора"?!).

Проанализировав всю документированную информацию, так или иначе касающуюся буферов и основываясь на результат собственных экспериментов, автор склоняется к мысли, что буфера записи напрямую связаны как минимум с Буфером Упорядоченной Записи (ROB Wb), Блоком Интерфейса с Памятью (MIU) и Блоком Интерфейсов с Шиной (BIU). А на K5 (K6/Athlon) Буфера Записи связаны еще с кэш-памятью первого уровня.

Но, так или иначе, Буфера Записи позволяют на некоторое время откладывать фактическую запись в кэш и/или основную память, осуществляя эту операцию по мере освобождения кэш-контроллера, внутренней или системной шины, что ликвидирует целый ряд задержек и тем самым увеличивает производительность процессора.

Блок Интерфейсов с Памятью (MIU). Блок Интерфейсов с Памятью представляет собой одно из исполнительных устройств процессора и функционально состоит из двух компонентов*: устройства чтения памяти* и *устройства записи памяти*. Устройство чтения соединено с буферами записи и кэшем первого уровня. Если требуемая ячейка памяти присутствует хотя бы в одном из этих устройств, на ее чтение расходуется всего один такт. Устройство записи памяти соединено с Блоком Упорядоченной Записи (ROB Wb), уже рассмотренным выше.

Блок Интерфейсов с Шиной Блок Интерфейсов с Шиной (BIU) является единственным звеном, связующим процессор с внешним миром, эдакое своеобразное "окно в Европу". Сюда стекается все информация, вытесняемая из Буферов Записи и кэш-памяти первого уровня, сюда же поступают запросы за загрузку данных и машинных команд от кэша данных и кэша команд соответственно. Со стороны

"Европы" к Блоку Интерфейсов с Шиной примыкает кэш-память второго уровня и основная оперативная память. Понятно, что от поворотливости BIU зависит быстродействие всей системы в целом.

Кэш второго уровня. В зависимости от конструктивных особенностей процессора кэш второго уровня может размещаться либо непосредственно на самом кристалле, либо монтироваться на отдельной плате вне его.

Однокристальная (*On Die*) реализация обладает практически неограниченным быстродействием, – поскольку длины проводников, соединяющих кэш второго уровня с Блоком Интерфейсов с Шиной относительно невелики, кэш свободно работает на полной процессорной частоте, а разрядность его шины в процессорах P-III и P-4 достигает 256-бит. С другой стороны, такое решение значительно увеличивает площадь кристалла, а значит и его себестоимость (процент брака с увеличением площади кристалла растет экспоненциально). Тем не менее, благодаря совершенству производственных технологий (и не в последнюю очередь – жесточайшей конкурентной борьбе), – интегрированных кэшем второго уровня обладают все современные процессы.

Двойная независимая шина (DIB – Dual Independent Bus). Для увеличения производительности системы, кэш второго уровня "общается" с BIU через свою собственную *локальную* шину, что значительно сокращает нагрузку, выпадающую на долю FSB.

В силу геометрической близости кэша второго уровня к процессорному ядру, длина локальной шины относительно невелика, а потому она может работать на значительно более высоких тактовых частотах, чем системная шина. Разрядность локальной шины долгое время оставалась равной разрядности системной шины и составляла 64 бита. Впервые эта традиция нарушилась лишь с выходом Pentium-III Copper mine, оснащенным 256 битной локальной шиной, позволяющей загружать целую 32 байтную кэш-линейку всего за один такт! Это фактически уравняло кэш первого и кэш второго уровня в правах! К сожалению, процессоры AMD Athlon не могут похвастаться шириной своей шины...

Архитектура двойной независимой шины значительно снижает нагрузку на FSB, т.к. большая часть запросов к памяти обрабатывается локально. По статистике, коэффициент загрузки системной шины в однопроцессорных рабочих станциях, составляет порядка 10% от ее максимальной пропускной способности, а остальные 90% запросов ложатся на локальную шину. Даже в четырех процессорном сервере нагрузка на системную шину не превышает 60%, создавая тем самым обманчивую видимость, что производительность системной шины перестает быть самым узким местом системы, ограничивающим ее производительность.

Несмотря на то, что статистика не лжет, интерпретация казалось бы самоочевидных фактов, мягко говоря не совсем соответствует действительности. Низкая загрузка системной шины объясняется высокой латентностью основной оперативной памяти, приводящей к тому, что по меньшей мере половину времени шина тратит не на передачу, а на ожидание выполнения запроса. Помните как в анекдоте, – почему у вас нет черной икры? Да потому что спроса нет! К счастью, в старших моделях процессоров появились команды предвыборки, позволяющие предотвратить латентность и разогнать шину на всю мощь.

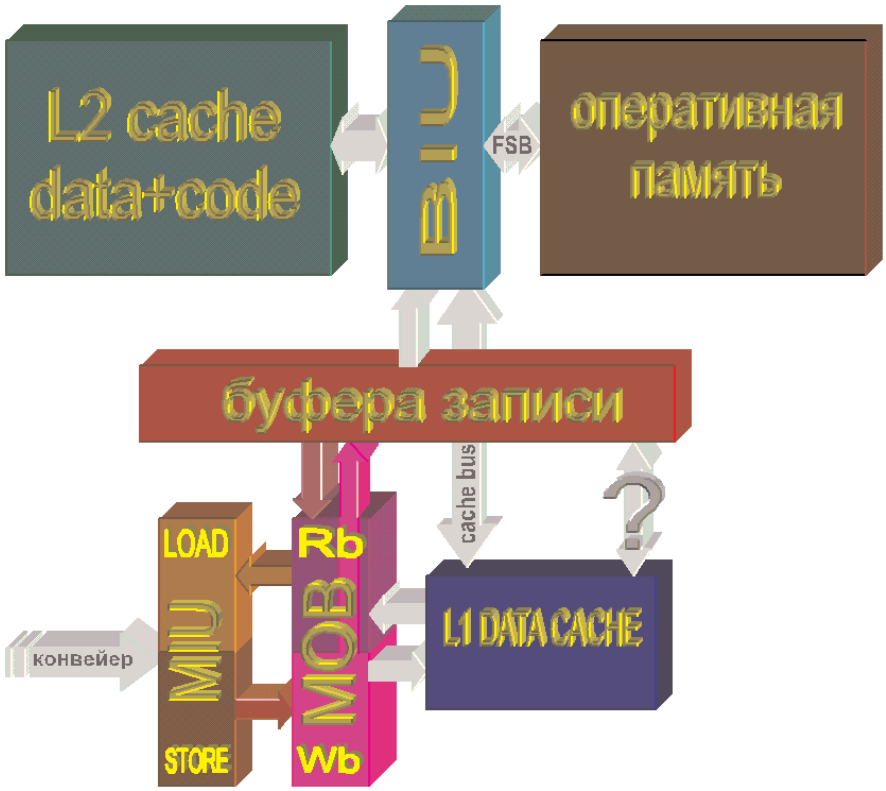

Рисунок 6. *Кэш-подсистема современных процессоров (кэш кода не показан)*

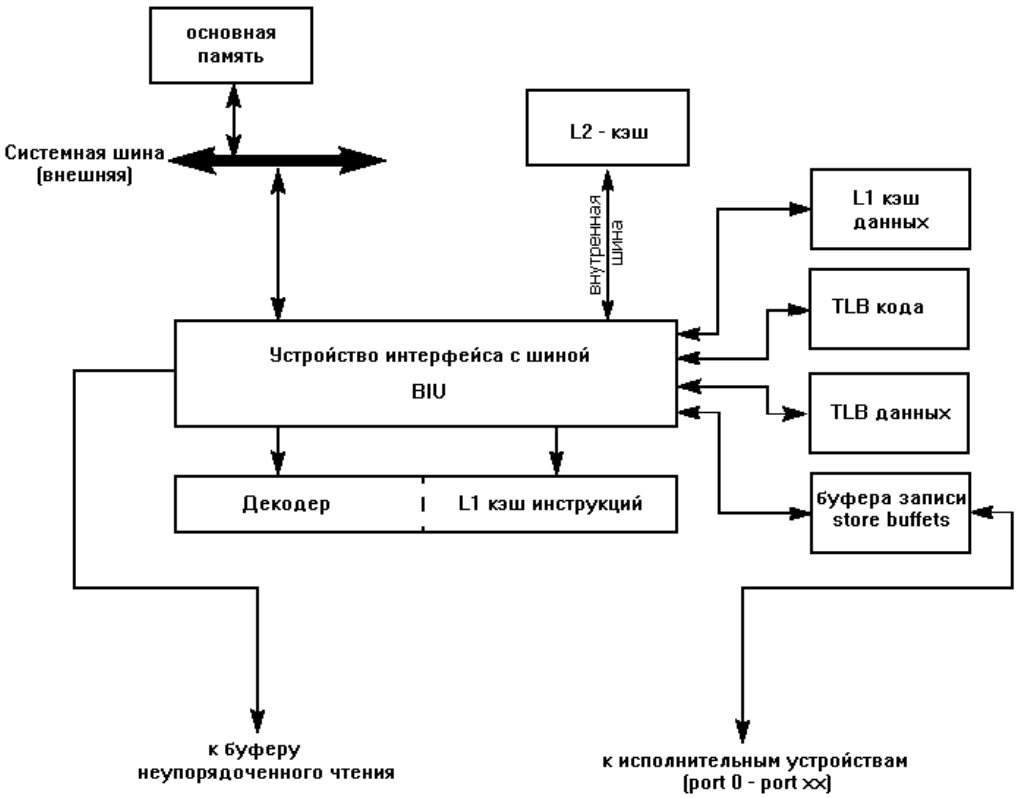

Рисунок 7. *Блок-схема подсистемы кэш-памяти процессоров семейства Intel P6*

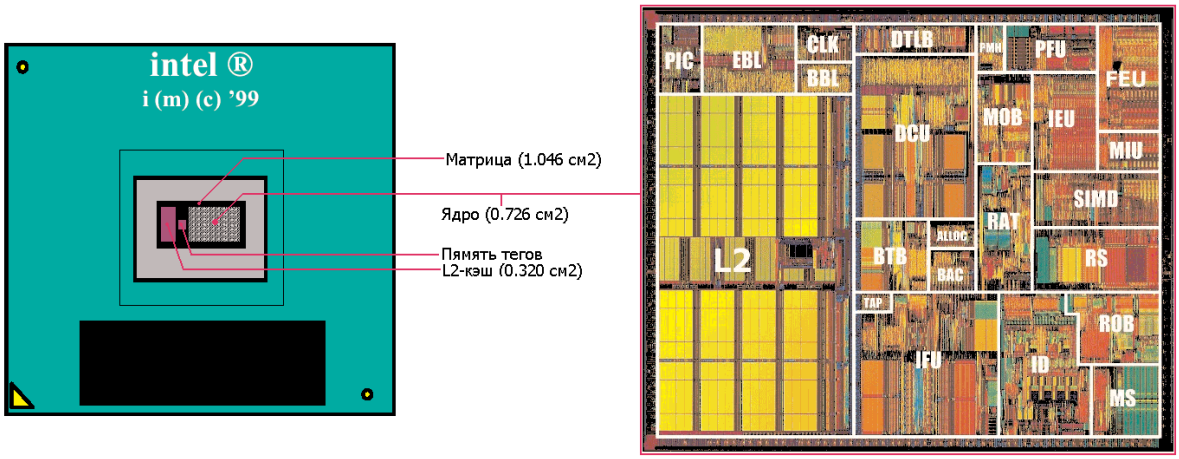

Рисунок 8. *Физическое воплощение подсистемы кэш-памяти на примере процессора Intel Pentium-III Coppermain*

Архитектура и характеристики кэшей современных микропроцессоров. Сводная Таблица

характеристика \ процессор			Pentium II CELERON	Pentium III CELERON	Pentium 4	Athlon
L1	КОД	размер (полный)	32 Кб	32 Кб	н/д	128 Кб
		тип	раздельный	раздельный	раздельный	раздельный
		размер	16 Кб	16 Кб	12K ops	64 Кб
		протокол	SI	SI	?	SI
		ассоциативность	4-way	4-way	4?	2
		размер линеек	32 байта	32 байта	6 mOPs	64 байт
		банков в линии *1	1	1	н/д	?
		размер банка *2	4 Кб	4 Кб	н/д	32 Кб
		кол-во портов	1	1	1?	1?
		алгоритм замещения	LRU	LRU	?	LRU
		политика записи	-	-	?	-
		блокировка	не блок.?	не блок.?	не блок.?	не блок.?
		частота	1.0 x ядра	1.0 x ядра	1.0 x ядра	1.0 x ядра
		время доступа: нормальное	1 такт	1 такт	1 такт	1 такт
		время доступа: line-splint	6 - 12 тактов	6 - 12 тактов	н/д	н/д
	ДАННЫЕ	размер	16 Кб	16 Кб	8 Кб	64 Кб
		протокол	MESI	MESI	MESI	MOESI
		ассоциативность	4-way	4-way	4-way	2-way
		размер линеек	32 байта	32 байта	64 байта	64 байта
		банков в линии	8	8	8?	8?
		размер банка	4 Кб	4 Кб	2 Кб	32 Кб
		кол-во портов	2	2	2	2
		алгоритм замещения	LRU	LRU	LRU	LRU
		политика записи	WA	WA	WT	WA
		блокировка	не блок.	не блок.	не блок. +4	не блок.
		частота	1.0 x ядра	1.0 x ядра	1.0 x ядра	1.0 x ядра
		время доступа				
L2		размещение	unified / on-die	unified / on-die	on-die / ?	
		размер, Кб	128, 256, 512, >	128, 256, 512, >	128, 256, 512, >	512,1024,2048
		тип	inclusive	inclusive	exclusive	inclusive / 1.0x
		протокол	MESI	MESI	MESI	MESI
		ассоциативность	4-way	4 / 8	4 / 8	2 / 1-1-1-1
		размер кэш-линий	32 байта	32	64x2	64?
		размер банка, Кб	32, 64, 128	32, 64, 128	32, 64, 128	32 / ? / 128
		кол-во портов	1?	1?	1?	1?
		алгоритм замещения	LRU *3	LRU	LRU	LRU
		политика записи	WB	WB	WB	WB
		блокировка	не блок.	не блок.	не блок.	не блок.?
		частота	0.5x / 1.0x	0.5x / 1.0x	?	1.0x
		время доступа	10 тактов?	4 такта?	2 такта?	8?
		формула	2-1-1-1	1-1-1-1	1-1-1-1	1-1-1-1
ROB, входов			40	40	?	?
RS, входов			20	20	?	?
Read Buffer			4x32 байт?	4x32 байт?	?	?
Write Buffer			32 байт?	32 байт?	6 x 64 байт	?
частота системной шины, MHz			66 / 100	66 / 100 / 133	100x4 / 133x4	100x2
разрядность шины	L2 Яа ↔ L1		64	256	256	64
	L2 Яа ↔ DRAM		64	64	64 / 128?	64

Таблица 2. *Основные характеристики L1 и L2 кэшей современных процессоров*