Кэш память.

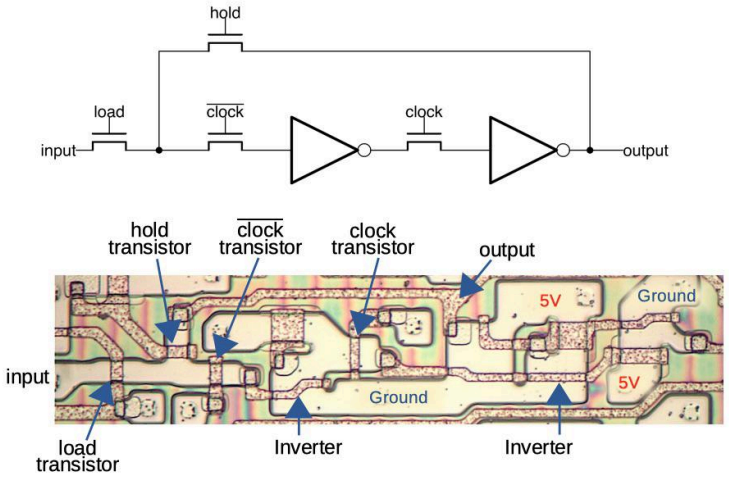
Ситуация, когда система имеет в наличии несколько разных видов памяти, различающихся по скорости работы и стоимости совершенно обычна. Вполне естественно и желание ускорить доступ к более медленной (но более объемной при этом) памяти за счет кэширования, как, например, мы это видели в главе 3.3. про виртуальную память.

Обычно есть дополнительные ограничения, например, память может быть энергозависимой (SRAM, DRAM теряют содержимое при обесточивании) или энергонезависимой (магнитные диски, твердотельные диски - flash память). Все виды памяти имеют специфику своего использования, например, flash память допускает ограниченное число перезаписей, поэтому при кэшировании с помощью неё магнитных дисков приходится применять меры, чтобы нагрузка на разные области была более-менее равномерной.

Но сейчас будем разбираться с энергозависимой памятью, которую обычно называют оперативной. Причем в её современном состоянии, которое можно считать относительно устоявшимся после десятилетий интенсивного роста и бурления идей.

Первой системой, использующей кэширование оперативной памяти, была по-видимому, малоизвестная военная М-100 (Китов А.И. 1958). В ней существовали раздельные интерфейсы для доступа к коду и данным (Гарвардская архитектура), Для доступа к данным (на магнитных сердечниках) использовался кэш, тоже на магнитных сердечниках, но сверхмалых и сверхбыстрых, объемом 5 тыс. разрядов, изготовленных по специальному заказу.,

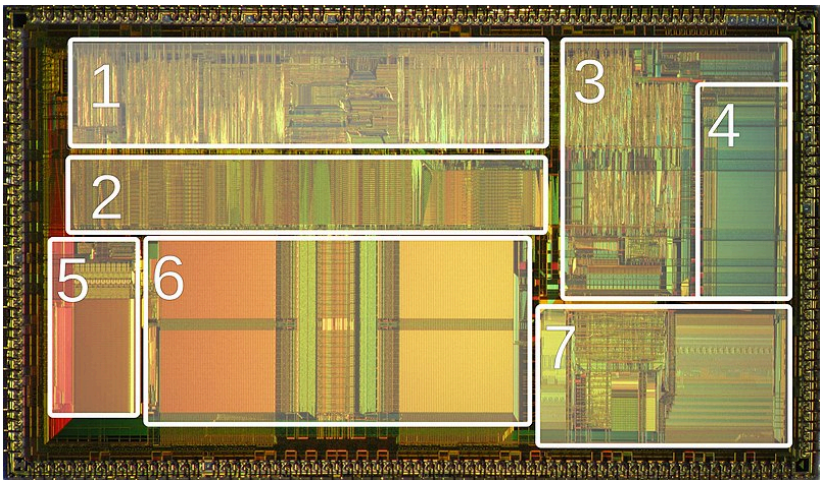
Первой серийной системой, использующей кэширование оперативной памяти, была БЭСМ-6 (1967). Кэш в ней был совсем крохотным (4 слова для данных, 4 для кода, 8 для очереди на запись, с доступом за 0.11 мкс), но тем не менее достаточно эффективным. Чуть позже появилась IBM System 360/85 (1968). Основная память в системе (до 4 Мб) была сделана на основе магнитных сердечников со временем доступа 0.96 мкс. Как сообщается, статическая кэш память (триггеры) объемом 16 Кб или 32 Кб (с циклом доступа в 0.08 мкс, за один такт) на некоторых задачах ускоряла доступ к основной памяти в 3-4 раза.



Фиг.3.4.1 Микрофотография триггера (без выходного буфера) в процессоре Intel 8086 (1978, отсюда)

Но это мэйнфрейм - высокопроизводительный компьютер коллективного пользования со значительными ресурсами. В производительных моделях мини компьютеров кэш также присутствовал (например, VAX 11/780, кэш 8К), но в первых однокристальных микропроцессорах его не было. Впервые внутри-кристальный кэш можно найти в Motorola 68020 (1982 г.), где было по 256 байт на код и данные. Кое-где предусмотрен внешний кэш, например, Intel 386DX(1985 г.), SPARC MB86900 (1986 г.), MIPS R3000 (1988 г.).

В Motorola 68040 (1990 г.) по 4К кэша на данные и код. Аналогично, в 80486 (1989 г.) 8К внутреннего кэша и он занимает примерно половину площади кристалла.



Фиг.3.4.2 Микрофотография 80486DX2, из общих ~1.2 млн транзисторов почти половина приходится на кэш (под номером 6). (отсюда)

С 1982 по 1990 гг. объем кэша увеличился в 8 раз. Эмпирическое правило: каждые два года количество транзисторов увеличивается в два раза, известное как “закон Мура”, продолжит действовать еще довольно долго. В современном Intel Core i7 (2023) 30 Мб так называемого Smart (L3) кэша, доступного всем ядрам.

Но прежде чем мы станем разбираться что такое ядра и почему кэш называется L3, посмотрим как собственно устроена кэш память.

Устройство кэш-памяти.

Прежде всего, кэш память - это память. Уже с самого начала (IBM System 360/85) в качестве физической реализации было выбрано СОЗУ(статическое ОЗУ, иногда говорили сверх-ОЗУ, SRAM)[1]. На каждый разряд такой памяти требуется 6..8 вентилей, следовательно 8К СОЗУ выливаются в минимум в 8192 * 8 * 6 = 393 216 вентилей, не так уж и мало.

А еще, кэш - это память ассоциативная, то есть хранилище пар (ключ, значение), где ключом является адрес а значением - данные по этому адресу. Но когда говорят об объёме кэша, имеют в виду только данные. Тем не менее на хранение ключа тоже нужны вентили. Если 32-разрядному адресу соответствуют 32-разрядные данные, количество необходимых вентилей удваивается.

Но и это еще не всё. Поскольку кэш должен отвечать быстро, к **каждой** паре (ключ, значение) необходимо добавить логику сравнения (компаратор), причем все эти компараторы будут работать параллельно, иначе как найти нужную пару за один такт. Общее число вентилей для 8К кэша легко может (могло бы) скакнуть за миллион.

Такой вариант устройства называется кэш с **полной ассоциативностью** (fully associative) и он вполне уместен там, где скорость работы важнее стоимости (и кэш не планируется сделать большим). Например, в TLB[2] или в предсказателе переходов[3] процессора.

Можно ли на чем-то сэкономить? Конечно.

- Хранить в кэше каждое слово (пусть как в примере выше - 32 разряда) отдельно - расточительно. С большой вероятностью программа далее обратится за следующим словом и опять его придётся искать.Поэтому одному ключу соответствует несколько подряд идущих слов - это называется **линия кэша** (блок кэша, cache-line). Типичное значение размера линии кэша для 32-разрядных систем - 16 или 32 байта, для 64-разрядных - 64. Т.е. 4..8 слов. Сколько линий умещается в имеющиеся 8К? Для линии в 32 байта: 8 192 / 32 = 256.
  Для кодирования 256 линий достаточно 8 разрядов.
- Упрощается компаратор. Если для поиска предъявлен 32 разрядный адрес, не обязательно сравнивать все 32 разряда. При размере линии кэша в 32 байта, младшие 5 разрядов ($2^5 = 32$) можно игнорировать, они указывают внутрь

[1] Статическое Оперативное Запоминающее Устройство, иногда используют термин Сверх вместо Статическое. Слово Статическое означает, что памяти не нужно динамически перезаписывать содержимое для корректной работы. SRAM - Static Random Access Memory
[2] Translation Lookaside Buffer - кэш перекодировки виртуальных адресов в физические
[3] предсказатель переходов - устройство процессора, которое запоминает в какую сторону сворачивала программа ранее при ветвлении (ЕСЛИ…ТО…ИНАЧЕ), весьма эффективно в случае циклов.

линии.
Теперь необходимо предъявить функцию, которая 27 разрядов (32-5) превратит в 8. Обычно не мудрствуют лукаво, а просто берут для сравнения 8 разрядов из середины адреса. Пусть это будут разряды 8, 10, 12, 13, 14, 15, 17, 19. Какие конкретно разряды использовать, определяется с помощью статистики, собранной при прогоне тестовых программ на эмуляторах процессора в процессе его разработки.

Для такой функции и сравнивать ничего не обязательно, достаточно взять соответствующие разряды и получить индекс линии кэша.

Однако, оставшиеся 27 - 8 = 19 разрядов адреса придётся хранить вместе с линией кэша, иначе, как мы проверим что это именно наш блок данных? Эти 19 (в данном случае) разрядов называются **тег** (tag) линии.

- кроме тега вместе с каждой линией кэша придётся хранить и служебную информацию, как минимум 1 разряд - занято/свободно. Это называется атрибут/флаг линии.
- Вариант, когда выбранные разряды адреса определяют индекс конкретной линии данных, называется кэшем с **прямой индексацией** (или с **ассоциативностью** 1). Обобщим устройство - пусть разряды адреса указывают не на конкретную линию кэша, а на “корзину” с линиями. Если в корзину помещается одна линия - имеем кэш с прямой индексацией, если 2 - двух-ассоциативный кэш, если 4 - четырёх-ассоциативный.

Опыт показывает, что увеличение ассоциативности (в разумных пределах) при том же объёме кэша, уменьшает число промахов. Оптимальная ассоциативность кэша определяется, опять же, с помощью статистики, полученной на эмуляторе процессора.

Итак, допустим, в нашем примере с 8К кэшем выбрана ассоциативность 4.
То есть 256 имеющихся линий распределяются по 64 корзинам. Индекс корзины становится 6-разрядным, то есть из исходного адреса только 6 разрядов (например, 10, 12, 13, 14, 15, 17) определяют интекс корзины.

Возможны разные варианты, как линия кэша расположена внутри корзины. Самый простой вариант - в соответствии с содержимым тех двух разрядов, что мы отрезали от индексной части адреса. Либо более сложный вариант - внутри корзины линия кэша может находиться в любом положении, поиск нужной происходит с использованием тэгов линий. Это называется **псевдо-ассоциативный** кэш.

Поиск.

На том же примере.

1) на вход поступает 32-разрядный адрес
2) выбираем разряды 10, 12, 13, 14, 15, 17, склеиваем их в 6-разрядное число, получаем номер корзины, пусть 11.
3) берём корзину с индексом 11, сравниваем теги всех 4 линий с оставшимися разрядами исходного адреса (кроме 5 младших, они внутри линии кэша). Обратим внимание, т.к. работа ведётся с единственной корзиной, нужно всего 4

параллельно работающих компаратора в отличие от 256 в случае полно-ассоциативного кэша.
Часть линий может содержать флаг “не занято”, с ними сравнивать не нужно.

4) если совпадение есть, возвращаем найденную линию кэша, обновляем статистику использования корзины.
5) если нет совпадения, фиксируем промах кэша

Запись.

На том же примере.

1) аналогично поиску
2) …
3) …
4) если совпадение есть, записываем свежие данные в найденную линию кэша, обновляем статистику использования корзины.
5) если нет совпадения, выбираем линию для замещения, записываем в нее свои данные и тэг, обновляем статистику. Это может быть линия с флагом “не занято” или, если таковых нет, линия с худшей статистикой.

Статистика.

При выборе кандидата на вытеснение при записи в корзину, можно придерживаться разных стратегий. Опустим тривиальный случай, когда есть незанятые позиции. Например, выбирать случайную позицию, почему нет?
Еще один вариант - держать циклический указатель на позицию для вытеснения - сейчас вытесняем линию с номером 2, в следующий раз - 3, потом - 0, …
Наиболее распространённым и всё ещё относительно легко реализуемым методом является LRU[4] - вытесняется тот, кого дольше всего не использовали, такой вариант демонстрирует наибольшую эффективность кэша.

Политика (write policy).

Имеется в виду порядок обновления данных в основной памяти после внесения изменений в сам кэш. Возможны два крайних варианта:

1) Сквозная запись (Write Through). Запись в основную память делается сразу после изменения данных в кэш-памяти.
2) Отложенная запись (Write Back). Запись происходит позднее, при вытеснении измененной линии из кэша.

Оба варианта имеют свои плюсы и минусы. Сквозная запись сильнее нагружает интерфейс памяти т.к. может произойти несколько изменений линии кэша и каждый раз данные будут записаны в основную память. Обратная запись меньше нагружает память, но её работа делает работу системы менее предсказуемой. Так, в случае записи при вытеснении, фактически потребуется два доступа к памяти: запись этой линии и чтение вытесняющей.

Возможны и промежуточные варианты, например, данные при сквозной записи не

[4] LRU - Least Recently Used

записываются в память сразу, а попадают в буфер - отстойник, где многократные изменения одних и тех же линий объединяются перед фактической записью. Какую политику выбрать в каждом конкретном случае определяют с помощью цифровых экспериментов.

Кэш и виртуальная память.

До сих пор мы не уточняли, с какими именно адресами имеем дело - виртуальными или физическими. Виртуальные адреса и их преобразование в физические мы разбираем в другой главе. Сейчас же необходимо понять, как это может сказаться на работе кэша.

На первый взгляд, вариантов два: кэширование идёт либо в физических адресах, либо в виртуальных. Казалось бы, а какая разница?

Во первых, физический и виртуальный адреса могут иметь разную ширину. Например, в AMD64 виртуальный адрес имеет ширину 48 разрядов, а физический - 52. Наоборот, в микроархитектуре Ice Lake (Intel) дозволено (в одном из режимов) использовать 57 разрядов виртуального адреса (и по-прежнему 52 физического). Чем уже адрес тем меньше вентилей.

Во-вторых. При кэшировании в виртуальных адресах надо как-то решать проблему так называемых “синонимов”. При использовании разделяемой между процессами памяти (а это весьма распространённая практика), возникают разные виртуальные адреса, которые указывают на одну физическую память. В результате в кэше может возникнуть дублирование данных.

Есть и обратная проблема т.н. “омонимов”, когда один виртуальный адрес смотрит на разные физические, ведь у каждого процесса виртуальное пространство своё и они пересекаются. Так что виртуальный адрес для кэширования необходимо расширять на идентификатор процесса (больше адрес - больше вентилей). Либо удалять из кэша все данные процесса при потере им контекста (операционная система даёт поработать другому процессу).

В третьих. При кэшировании в физических адресах, надо еще дождаться пока завершится его преобразование из физического, это задержка. При кэшировании в виртуальных адресах преобразование необходимо только в случае промаха.

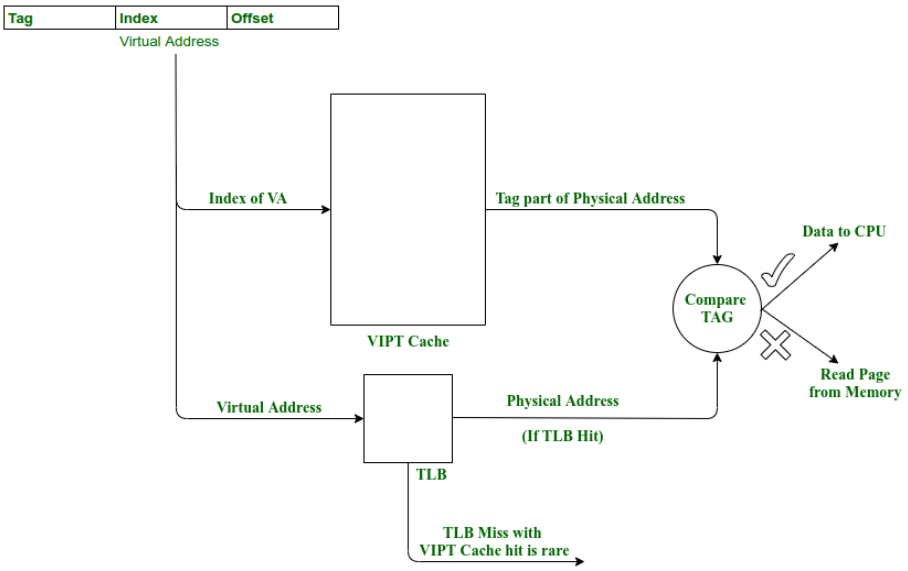
В четвертых. Если кэш имеет несколько уровней (зачем - отдельный вопрос, об этом дальше), после промаха в кэше верхнего уровня у нас вполне может быть на руках уже преобразованный физический адрес, странно было бы им не воспользоваться.

В пятых. Вообще, кэш TLB работает весьма эффективно, он находит нужный адрес за 1-2 такта в 98..99% случаев. Это из-за того, что работающая программа обычно пишет в не очень большой набор рабочих страниц. Но в случае промаха, задержка может быть большой - до нескольких десятков тактов (поиск в таблице страниц, если все страницы в памяти и их не придётся подкачивать).

В шестых. Память разделена на страницы (обычно, 4К), для адресации внутри такой страницы необходимо 12 разрядов и эти разряды в виртуальных и физических адресах совпадают. Младшие разряды (4..6) из этих 12 указывают внутрь линии кэша, оставшиеся (6..8) вполне можно взять прямо из виртуального адреса до его преобразования в физический. Этого вполне достаточно для индексной части кэша небольшого объема, с учетом ассоциативности.

И тут мы вернулись к тому факту, что для кэширования используются две части адреса - индексная, которая указывает на корзину и тег для проверки попадания. Теоретически, возможны четыре варианта организации кэша:

1) индексная часть из виртуального адреса, тег из виртуального (VIVT[5]). Здесь для проверки не нужно дожидаться преобразования адреса, но необходимо бороться с упомянутыми выше синонимами и омонимами. На практике не используется, разве что в экспериментальных системах.
2) индексная часть из виртуального адреса, тег из физического (VIPT[6]).



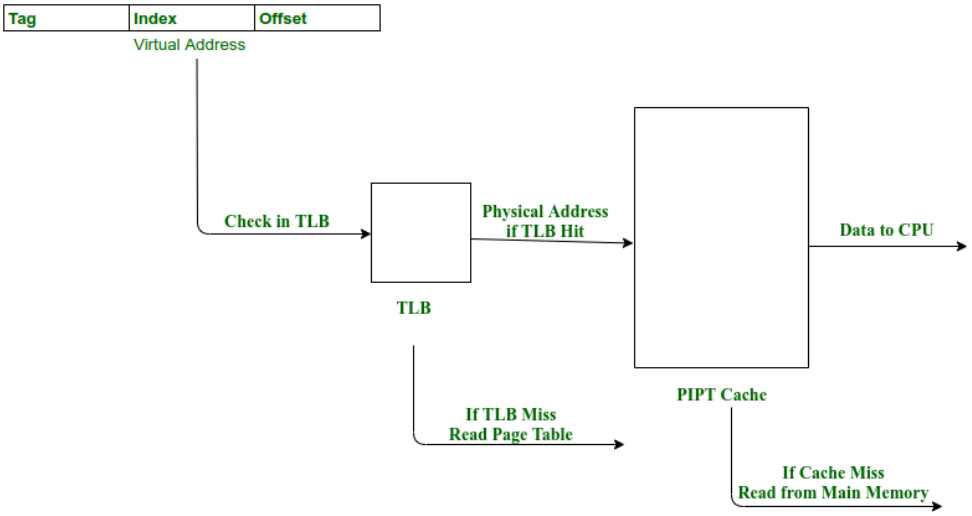
Фиг.3.4.3 Работа VIPT кэша (отсюда)

Этот вариант хорош тем, что проверку можно начинать не дожидаясь окончания преобразования адреса, хотя для проверки тега физический адрес нужен. Это позволяет сэкономить 1-2 такта. Заплатить придётся увеличением ширины тега, ведь индексную часть мы брали из виртуального адреса, в физическом она другая, ее тоже придётся включить в тег. Однако, если индексная часть использует общие в виртуальном и физическом адресе разряды, этого можно избежать.

3) индексная часть из физического адреса, тег из виртуального (PIVT). На практике не используется, хотя, с этим экспериментировали.

[5] virtually indexed, virtually tagged
[6] virtually indexed, physically tagged

4) индексная часть из физического адреса, тег тоже из физического (PIPT).



Фиг.3.4.4 Работа PIPT кэша (отсюда)

Распространённый вариант. Используется в кэшах большого размера, обычно так устроены нижние уровни многоуровневого кэша. К тому моменту как запрос доберётся до нижнего уровня, физический адрес уже обычно известен либо можно подождать, дополнительные пара тактов не сильно испортят общее время доступа.

Многопроцессорные системы

Системы с несколькими (двумя) процессорами появились, как только это позволили успехи миниатюризации и общая надёжность вычислительных систем.

В Соединённых Штатах первой такой машиной была, по видимому, Burroughs B5000 (1961). В СССР - 5Э92б, (1961) военное развитие архитектуры М-50, а заодно и первая полностью полупроводниковая советская ЭВМ. В обеих системах процессоры были неравноценны, один процессор исполнял пользовательские программы, второй использовался операционной системой в том числе для работы с периферийными устройствами.

В дальнейшем процессоры стали универсальными (к середине 1960-х), так оказалось проще и разрабатывать компьютеры и поддерживать их работу. Но самое главное, для одинаковых процессоров (SMP[7]) проще организовать “справедливое” распределение ресурсов между программами.

Однако, две параллельно работающих задачи могут столкнуться, обращаясь к одному ресурсу, например, если станут одновременно посылать что-нибудь на принтер или просто пользоваться одной областью памяти. Возникает проблема с синхронизацией.

[7] Symmetric Multi-Processing

Требуется механизм, который гарантирует что в некоторый момент ни одна другая программа не сможет воспользоваться конкретным ресурсом.

Так же к середине 1960-х в операционные системы стали многопоточными в нынешнем понимании. В первых серийных компьютерах не было операционных систем, программы вручную загружались оператором с перфоленты/перфокарт. Далее появились операционные системы, которые запускали программы сначала по очереди, в дальнейшем в пакетном режиме, подразумевающем управление ресурсами. Возникла так называемая **кооперативная многозадачность**, когда в случае системного вызова, (например, обращения за ресурсом или операции ввода-вывода) операционная система могла приостановить работу задачи и отдать управление другой.

И, наконец, к середине 1960-х, появилась **вытесняющая многозадачность** (preemptive multitasking), когда задача для работы получала время квантами, по истечении которого операционная система отдавала управление другой задаче. Для этого использовался аппаратный таймер, который вызывал аппаратное прерывание, обрабатываемое ядром операционной системы.

При этом, квант времени мог истечь в любой момент, после любой исполнившейся инструкции. Это вызывает ту же проблему с синхронностью, даже если в компьютере всего один процессор. Например, пока одна программа печатала на принтере, у нее истёк квант времени и другая программа занялась тем же самым.

Что с этим можно сделать?
Во первых, можно решить данную проблему программным путём. Для синхронизации двух задач существует алгоритм Деккера (улучшенный Эдсгером Дейкстрой, 1965). На случай синхронизации N-задач есть, например, алгоритм Лампорта (алгоритм кондитера, 1974). Но, обратим внимание, алгоритм Деккера опубликован в 1965 г., когда уже существовали и многопроцессорные системы и вытесняющая многозадачность.

Как же обходились до этого момента?

Объекты синхронизации.

С помощью простого, но очень мощного аппаратного трюка - инструкции с условным названием test-and-set (проверь и замени). Она устанавливает новое значение переменной и возвращает старое. Инструкция эта атомарна, задача не прервётся, пока инструкция не завершится. Гарантируется, что при наличии нескольких процессоров, во время работы этой инструкции другой процессор не сможет обратиться к этой ячейке памяти (как вариант, с помощью блокировки доступа к памяти, поначалу так и было, но при большом числе процессоров так, конечно, делать нельзя из-за сильного удара по производительности). Впрочем, в те времена не существовало компьютеров с числом процессоров, достаточным для того, чтобы это стало узким местом.

Тем не менее, операция test-and-set позволяет работать с атомарными счетчиками, которые лежат в основе разнообразных примитивов/объектов синхронизации. Одним из первых таких механизмов был семафор (наряду с барьерами и спинлоками). Семафор предоставляется ядром операционной системы, которая блокирует процесс/поток до тех пор, пока счетчик не обнулится. Барьер используется для того, чтобы гарантировать, что все зарегистрированные потоки достигли нужной точки кода. Спинлок - тот же семафор, но без обращения к операционной системе, обычно это цикл ожидания до тех пор, пока счетчик не обнулится, на коротких блокировках это эффективнее, чем приостановка работы на время.

Мы говорили про инструкцию test-and-set, которая изначально была сделана через блокировку доступа к памяти на время ее работы. Представим, что с помощью этой инструкции в цикле опрашивается переменная (это вполне обычный вариант использования). Так можно парализовать работу всей системы.

Для поддержки объектов синхронизации придумывались специальные аппаратные механизмы. Например, в советских Эльбрусах таких объектов было два вида - семафоры и спинлоки (инструкция с мнемоникой “ЖУЖ”, да, от слова жужжать),.

Чтобы обеспечить это, для межпроцессорного взаимодействия введены инструкции:

- “ПРЕРВАТЬЦП” с аргументом - маской заинтересованных процессоров. В результате во всех указанных процессорах вызывается прерывание и запоминается что объект синхронизации захвачен/отпущен.
- “ЖДАТЬЦП” - выполнение команды заключается в ожидании исполнения команды “ОТВЕТЦП” от всех процессоров, указанных в команде (параметре команды)
- “ОТВЕТЦП” - подтверждение после выполнения прерывания

Но это не решает проблемы с локальными кэшами. В Эльбрусах межпроцессорному обмену подлежало лишь небольшое количество т.н. глобальных данных. Но и они синхронизировались только когда изменялись внутри критической секции кода (от захвата семафора или спинлока до освобождения). После выхода из критической секции, глобальные данные записывались в память и сбрасывались во всех остальных локальных кэшах. В прочих случаях кэширование шло локально.

Обратим внимание, опрос памяти спинлоком в цикле нагружает только один процессор и никак не мешает остальным пользоваться памятью.

В современных архитектурах нет разделения данных на локальные и глобальные, все (за небольшими исключениями, есть некэшируемые диапазоны адресов, используемые аппаратурой, в некоторых архитектурах ввод-вывод происходит с помощью записи по предопределённым адресам) данные кэшируются и доступны из всех процессоров.

Поддержка когерентности кэша.

Проблема с синхронизацией возникает при наличии кэша в системе с несколькими процессорами весьма серьёзна. Например, программа работала на одном процессоре, у нее истек квант времени. Через какое-то время ОС снова запустила эту программу, но уже на другом процессоре. Но в этом процессоре свой кэш с другим содержимым. Вполне вероятна потеря данных.

Две совершенно разные проблемы - реализация механизмов синхронизации и корректная работа кэша в многопроцессорной среде. В современных компьютерах для их решения используют один механизм, но так было не всегда.

Под когерентностью подразумевается невозможность в разных процессорах одновременно прочитать разные данные из одной ячейки памяти.

Вариант с общим кэшем не годится т.к. он должен быть равноудалён от всех процессоров и длинные линии связи неизбежно снизят время доступа к нему по сравнению с обычным внутрикристальным кэшем. К тому же, процессоры начнут мешать друг другу при работе с ним.

Директория(справочник).

Первое, что приходит в голову в качестве решения этой проблемы - устройство, знающее в чьем локалоном кэша находятся нужные данные (т.н. директория).
В чистом виде директория предполагает наличие описателя к каждой потенциальной строке кэша. Т.е. если размер строки 64 байта, то на каждые 64 байта основной памяти должен быть описатель, говорящий, в каких процессорах есть копии этой строки.

Понятно, что это требует большого количества дополнительной памяти, которую в силу этого нельзя сделать быстрой. Следовательно такая схема сильно замедлит работу с памятью.

С другой стороны, directory сильно разрежена т.к. объём кэша намного меньше основной памяти. Поэтому делались попытки (например, здесь) хранить информацию в сжатом виде и всё же втиснуть её в быструю память. Здесь проблема довольно простая - объем памяти на директорию фиксированный, а степень сжатия - величина непостоянная. Гарантировать что удастся вместиться в фиксированный объем не так то и просто.

Отсюда вывод - схема с директорией имеет естественные пределы роста числа процессоров.

Протоколы слежения (Snooping).

Идея довольно простая - раз нет возможности собрать всю информацию о строках кэша в одном месте, давайте хранить её распределённо. Идеальный вариант - прямо с рядом со строкой кэша. Правда придётся делать широковещательную рассылку при

каждом изменении состояния каждой строки. Эта рассылка делается с помощью т.н. следящих или snoop-пакетов.

Впервые идея представлена в 1983 г. Равишанкаром и Гудманом (Chinya Ravishankar, James Goodman) и лежит в основе многих последующих протоколов

Рассмотрим (для простоты) MESI (Intel: Pentium, Core; PowerPc 604), который хоть и считается устаревшим, является основой для других протоколов и не так специфичен, как, например, MESIF (Intel Nehalem) или MOESI (AMD Opteron).

Итак, MESI.
Запросы со стороны процессора к кэшу бывают:

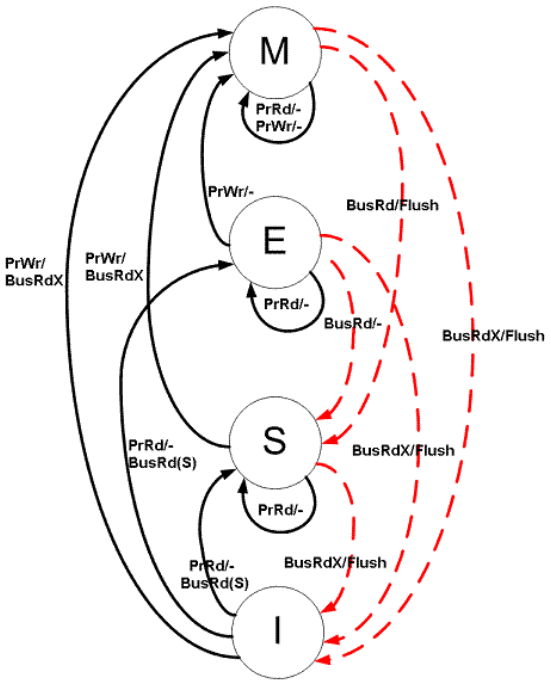
1. **PrRd**: процессор просит прочитать строку кэша.
2. **PrWr**: процессор просит записать строку кэша.

Запросы со стороны шины бывают:

1. **BusRd**: Снуп возникающий, когда есть запрос на чтение этой строки кэша со стороны другого процессора
2. **BusRdX**: Снуп говорящий, что есть запрос на чтение этой строки кэша со стороны другого процессора, который собирается её изменить, но в данный момент она у него отсутствует
3. **BusUpgr**: Некоторый процессор просит (более свежую) строку кэша для изменения при том, что старая её версия у него есть
4. **Flush**: Сигнал о том, что строка кэша была кем-то выгружена в память.
5. **FlushOpt**: Передача строки из кэша в кэш. Необязательный запрос с целью оптимизации.

Состояния строки кэша:

1. **Modified(M)** Строка присутствует только в кэше данного процессора, но её содержимое отличается от того, что лежит в памяти. Однажды эту строку придётся записать обратно в память, при этом состояние перейдет в **S**.
2. **Exclusive(E)** Неизмененная строка присутствует только в кэше данного процессора.
3. **Shared(S)** Данная строка может быть и в кэшах других процессоров, при этом её содержимое совпадает с оным в памяти и в любой момент может быть безопасно сброшено.
4. **Invalid(I)** в кэше процессора нет данной строки.



Фиг.3.4.5 Диаграмма состояний MESI (отсюда)

Посмотрим на примере, пусть имеются две переменные a и b, изначально равные 0. Два потока (thread) одновременно работают с ними.

{
a = 1;
b = 1;
}
	{
	while (0 == b);
	…
	}

Один устанавливает переменные в единицу, второй ожидает, пока b не станет единицей. Может ли так случиться, что **b=1** обгонит **a=1** и при выходе из цикла прочитаем **0 == a** ? Будем исходить из того, что компилятор по каким-то своим соображениям не переставит инструкции записи в память местами и они будут исполнены именно в таком порядке.

- изначально переменных **a** и **b** нет в кэше и их состояния равны I(invalid)
- задача 1: **PrWr a**
- одновременно задача 2: **PrRd b**, запросы к памяти идут последовательно, пусть порядок таков
- задача 1: **PrWr b**
- задача 1: получает **a=1** в свой кэш и состояние **a** становится **M**
  если бы переменная была в кэше других процессоров, она бы там инвалидировалась
- задача 2: получает результат **b=0** в свой кэш и поскольку переменной нет в кэшах других процессоров, состояние **b** становится **E**
- задача 2 делает еще одну итерация цикла: **PrRd b**
- задача 1: получает результат в свой кэш и состояние **b** становится **M**
  при этом состояние переменной в кэшах других процессоров инвалидируется (**I**)
- задача 2: получает результат **b=1** в свой кэш, выходит из цикла, состояние **b** в обоих задачах становится **S.** Задача 1 слышит, что кто-то читает переменную и отвечает на **BusRd**, задача 2 получает ответ на свой **PrRd** от задачи 1.

- в кэше задачи 2 нет сведений о переменной **a (I),** а даже если оно и было то инвалидировалось в процессе, поэтому при чтении её значение будет получено из кэша задачи 1 и окажется равным 1.

Выглядит не очень сложно, но как всё это устроено? Кто занимается синхронизацией, что произойдёт, если будет два одновременных обращения к одному адресу памяти, скажем, на запись и на чтение?

Введем такую сущность, как арбитр памяти или контроллер когерентности (CC).
CC наряду со всеми ядрами слушает снуп-пакеты. Если ядро сделало запрос на чтение и за отведённое время никто не ответил, значит запрашиваемых данных нет в кэше и их необходимо прочитать из памяти.

CC поддерживает реестр запросов, которые в данный момент выполняются и блокирует запросы к памяти по адресам, которые находятся в процессе чтения/записи вплоть до завершения этих запросов.

CC может заниматься мелкой оптимизацией, например, держать небольшой буфер запросов, вдруг будет несколько подряд записей по одному адресу, их можно скомбинировать. Или при наличии запроса на чтение сразу после запроса на запись, можно не читать данные, а организовать их пересылку из кэша в кэш (FlushOpt)...

После того, как Intel поддержала MESI в Pentium, выяснилось, что в многопроцессорном режиме снуп-пакеты сильно забивают системную шину, это сказывается на производительности. В результате, в процессоре Pentium Pro шин стало две - внешняя (FSB - Front Side Bus) и внутренняя (BSB - Back Side Bus). Внутренняя шина предназначалась для обмена снуп-пакетами и данными между кэшами разных процессоров. На архитектуру с двумя шинами перешли и разработчики процессора PowerPC (IBM, Apple, Motorola).

Кэш стал двухуровневым - небольшой L1 доступен внутри ядра, L2 разделяется всеми ядрами

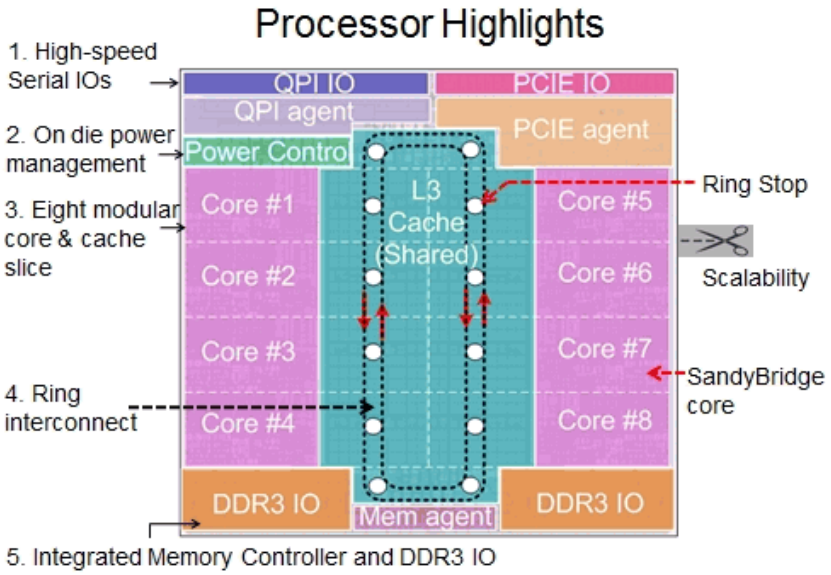
Фиг.3.4.6 процессорный модуль Pentium II - справа видны два блока кэша L2

Очень быстро выяснилось что с ростом числа процессоров даже выделенная под поддержку когерентности кэша шина становится узким местом. Пропускная способность у неё фиксированная и все процессоры её делят.

И не менее важное, эта шина - внешняя по отношению к процессору, её тактовая частота ограничена этим фактом (вспоминаем про скорость света), плюс на высоких частотах надо заботиться об электромагнитной совместимости т.к. проводники имеют неприятное свойство влиять друг на друга.

Так что эволюция пошла по пути: несколько процессоров на одном кристалле и теперь они называются **ядрами** (core).

Рассмотрим относительно современное семейство Intel Sandy Bridge (2011), с тех пор принципиально ничего не изменилось.



Фиг.3.4.7 Кольцевая шина Sandy Bridge (отсюда)

Впервые двунаправленная кольцевая шина в качестве транспорта snoop-пакетов появилась в девятиядерных (PowerPC + 8 графических ускорителей) процессорах Sony/IBM Cell BE (PlayStation, 2007 г.)

В Sandy Bridge:

- В процессоре уже три уровня кэша - L1, L2 и L3
- L1 и L2 доступны только внутри ядра, L3 нужен для обмена между ядрами
- L1 - по 32 Кб для данных и кода, ассоциативность 8, доступ за 4..7 тактов, тип - включающий
- L2 - 256 Кб, ассоциативность 8, доступ за 11..12 тактов, тип - включающий
- L3 - 1..3 Мб на ядро, ассоциативность 8, 12, 16, 20, тип - включающий, доступ за 26..31 такт
- Блоки L3 при каждом ядре связаны четырьмя двусторонними (двунаправленными) кольцевыми шинами (фактически шин 8) - одна 32-байтовая для данных, остальные для снуп-запросов и ответов. При необходимости послать пакет, контроллер L3 отсылает его сразу в оба конца, где они ретранслируются дальше, пока не вернутся назад, где будут

проигнорированы. Так, за N тактов, равное максимум половине числа ядер, пакет дойдёт до самого дальнего адресата, еще за N тактов будет получен ответ.

Но что делать, если ядер одного процессора недостаточно и действительно требуется собрать в одной системе несколько процессоров? Эта техника называется SMP (Symmetric Multi-Processing). Именно архитектура с равноценными процессорами оказалась наиболее “справедливо” распределяющей ресурсы системы, будь то память, пропускная способность памяти или вычислительная мощность.

На этот случай предусмотрены дополнительная маска в каждой линии кэша. Эта маска содержит 10 разрядов, которые указывают, в каком из ядер процессора еще есть данная строка плюс 7 разрядов для указания на внешние процессоры. Таким образом максимальное число процессоров в компьютере - 8 (плюс текущий). Этой маски из 17 разрядов достаточно для работы MESI протокола в системе из 8 десяти-ядерных процессоров. Фактически, эта маска есть распределенная по L3 “директория” покрывающая весь L3. По какой топологии связаны между собой процессоры, не так уж и важно (например, аналогичной двунаправленной кольцевой шиной), главное, чтобы snoop пакеты доходили до адресата.

А как быть, когда недостаточно и 8 многоядерных процессоров? При желании, например, по топологии гиперкуба можно было бы связать в единую сеть и 16 и 32 … процессоров с максимальной стоимостью snoop-пакета в две межпроцессорные пересылки. Здесь в полный рост встаёт другая проблема - производительность памяти. Проблема старая - пропускная способность подсистемы памяти фиксированная и разделяется между всеми ядрами и процессорами. В какой-то момент наращивать число процессоров становится бессмысленно - они не смогут работать в полную мощь.

Приходится прибегать к технологии неоднородной памяти - NUMA (Non Uniform Memory Access).

Неоднородная память (NUMA).

Поскольку централизованное владение ресурсами (в частности, памятью) имеет естественные ограничения на размер, для построения бОльших систем требуется фрагментация. Система теперь состоит из т.н. нод (узлов), каждая из которых обычно устроена в духе SMP. Для связи между нодами существуют специальные средства, принципиальным отличием является тот факт, что доступ к памяти теперь не однороден - память на своей ноде как минимум на десятки процентов быстрее.

Разделяемый кэш теперь стал трёхуровневым - на своем процессоре - на своей ноде - на чужой ноде. Соответственно и стоимостей стало больше. Снуп-протоколы в масштабах всей системы больше невозможны, в том или ином виде требуется наличие директории.

С точки зрения директории эта ситуация принципиально отличается от SMP тем, что сама директория стала двухуровневой. Т.е. теперь строка может разделяться не

только с другим процессором, но и с другой нодой. Это деление необходимо в силу того, что процессоров может быть очень много - сотни или тысячи, невозможно под каждый из них зарезервировать разряд в дескрипторе каждой строки кэша. Кроме того, стоимость доступа в этих ситуациях тоже разная.

Поэтому придётся явно отделять чужие ноды от своих процессоров. Например, в директории четырёхядерного Эльбрус-4С+ заведено три разряда для указания с какими чужими процессорами мы делим строку, т.е.максимальное число ядер в системе - 8X4. А если мы зарезервируем 6 разрядов - 3 для процессоров своей ноды и 3 для чужих нод, получим 4X8X8 ядер. Теперь при необходимости получить данные с удаленной ноды нет необходимости делать широковещательную рассылку, можно обратиться к конкретной ноде, а там дальше она сама разберется на каком процессоре и в кэше какого ядра лежит нужная нам строка.

С точки зрения протокола когерентности, в основном используются варианты MESI с учетом того факта что разделение строки может происходить на трёх уровнях - (ядро-процессор-нода). Стоит сказать, что игроков на рынке NUMA-систем не так уж много и они ведут позиционную войну на патентном минном поле.

В случае NUMA нет универсальных рецептов написания программ. Операционная система старается выдавать (в каждом новом кванте времени) программе если не один процессор, то хотя бы ту же ноду. Но если так случилось, программа возобновила работу на неродной ноде, стоит попытаться минимизировать потери. Например, можно, конечно пытаться устанавливать affiniti (маску желательных для использования процессоров), но это скорее подсказка планировщику ОС. Есть также возможность принудительно “перевезти” память с одних нод на другие, в любом случае, программирование для NUMA это всё еще скорее искусство, чем ремесло.

Транзакционная память.

Транзакция - это кусок кода, который выполняется атомарно (неделимо). Т.е. если транзакция удаляет элемент из одной очереди и помещает в другую, никаким образом снаружи невозможно увидеть некорректное состояние, в котором элемента нет ни в одной из очередей или он есть в обеих.

В сущности, транзакционной была работа с разделяемой памятью в советских Эльбрусах (1979..1984). Там транзакция начиналась с захватом семафора и успешно завершалась с его освобождением. Варианта отката транзакции не существовало - в случае ошибки программа останавливалась.

В современном программировании, чтобы добиться такого эффекта, пользуются блокировками, которые часто работают через объект синхронизации (объект ядра операционной системы) и обращение к нему недёшево (порядка тысячи тактов). А если произошло блокирование работы, можно стоимость умножать на десять (переключение контекста исполнения в современных системах).

В некоторых ситуациях можно обойтись “малой кровью” - с помощью атомарных операций с памятью - вариантов test-and-set. Которые в современных системах работают через механизм поддержки когерентности кэша.

В начале 1980-х Херлихи, Раджвар и Шавит (Maurice Herlihy, Ravi Rajwar, Nir Shavit) позаимствовали из области баз данных идею транзакционных вычислений и начали ее развивать. Идея эта нашла своё отражение в некоторых языках программирования, например, Clojure, Haskell, Scala, даже в C/C++ (GCC 4.7+, 2014 г.).

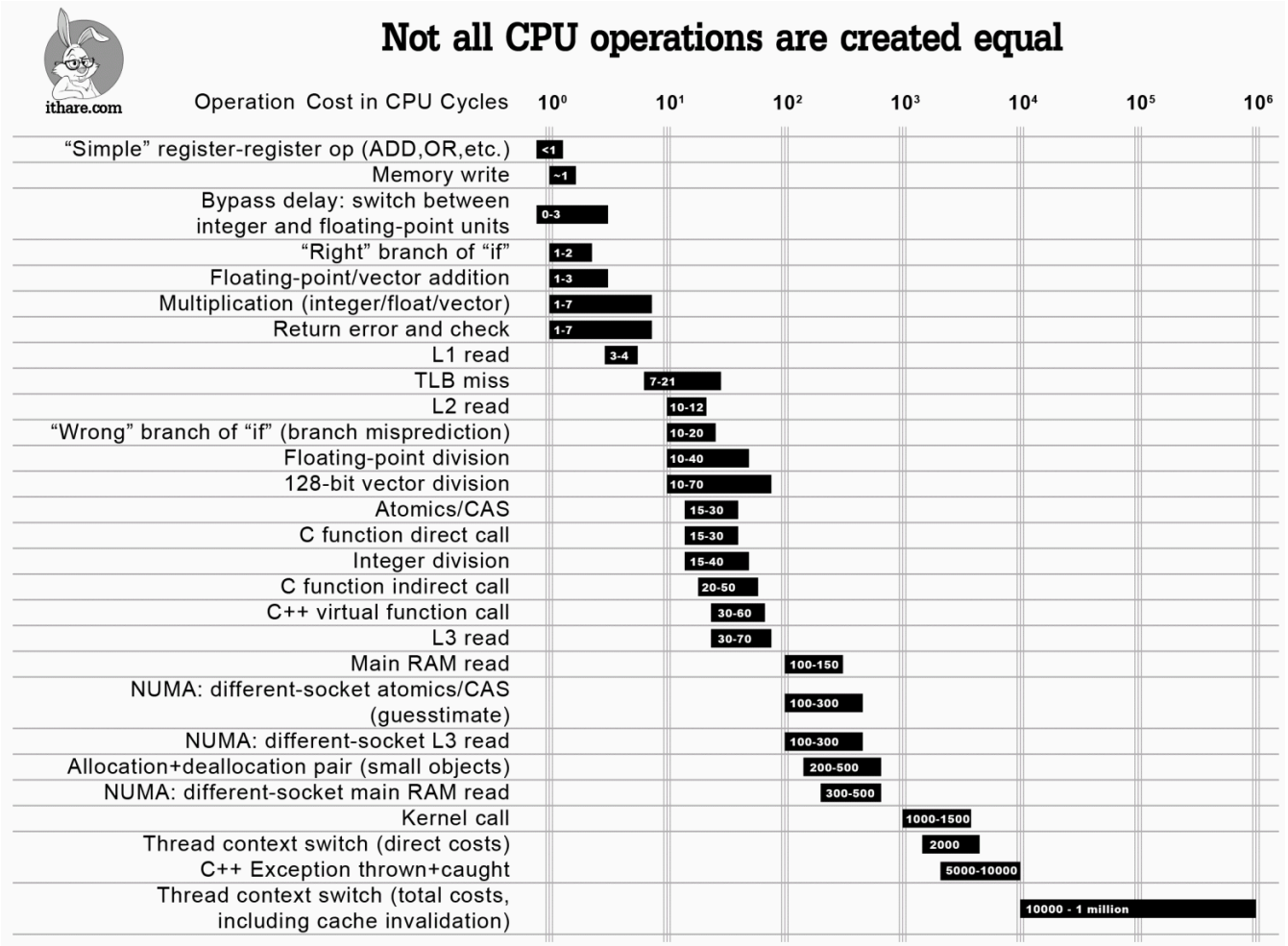
С аппаратной поддержкой всё было сложнее.
Первым был микропроцессор Rock (2007..2008 г.) от компании Sun Microsystems. 64-разрядный процессор архитектуры SPARC v9 имел 16 ядер. Проект был не слишком удачным (низкая скорость, высокое тепловыделение) и в 2008 г. после покупки Sun Microsystems компанией Oracle, его остановили.

Возможно, все дело в неудачной реализации, ведь здесь можно задействовать существующий механизм поддержки когерентности кэша. Этот механизм уже умеет отслеживать изменения в памяти, различать версии строк. Признак в линии кэша, что она была изменена транзакцией, пара новых видов снуп-пакетов и память с поддержкой когерентности кэша превращается в транзакционную.

Примерно так и сделала компания Intel, когда выпустила процессор Haswell (2013) у которого в наборе инструкций есть поддержка TSX (Transactional Synchronization Extensions). Также с транзакционной памятью работают процессоры IBM Power8 (2013 г.), IBM zEnterprise EC12 (2012 г.), IBM BlueGene/Q (PowerPC A2, 2011 г.).

В целом, технология развивается, но она требует кроме самих технологий еще и переосмысления подходов к программированию, а устоявшийся образ мышления - штука весьма инертная.

И, в качестве обобщения, картинка с сравнением стоимости в тактах разного рода программных событий:



Фиг. 3.4.8 примерная стоимость программных событий в тактах (отсюда)