
MOVING JAVA FORWARD

ORACLE

# Драконы в домашнем хозяйстве:
скалируемся на многоядерных машинах

**Алексей Шипилёв**
Java SE Performance
aleksey.shipilev@oracle.com
@shipilev

**Сергей Куксенко**
Java SE Performance
sergey.kuksenko@oracle.com


## India

3–4 May 2012

## San Francisco

September 30–October 4, 2012

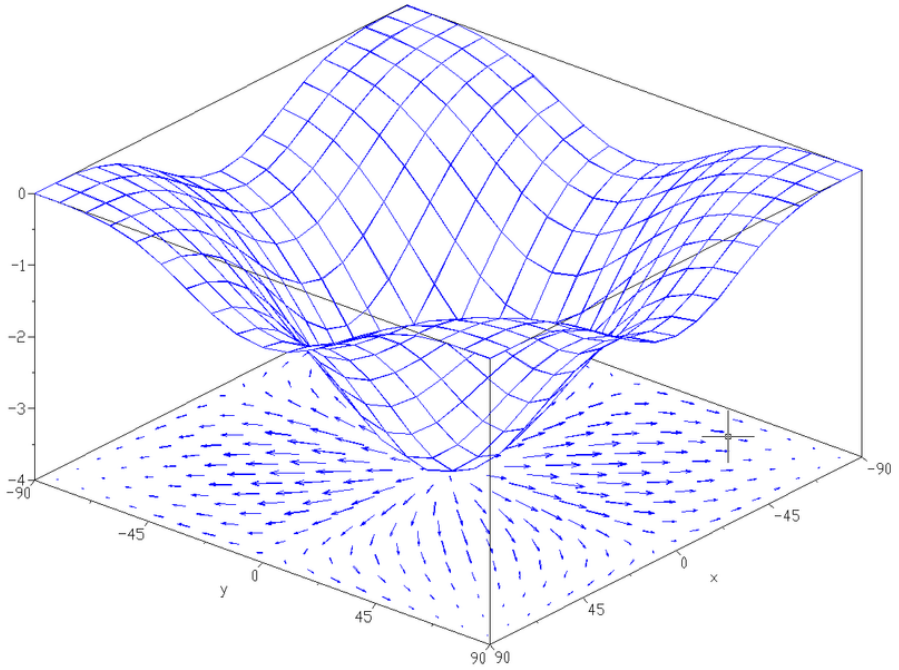
 

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

# Введение (немножко матана)

- Пространство ресурсов:
  - В общем случае N-мерное
  - $K^n$
- Производительность:
  - Скалярное поле
  - $P : K^n \rightarrow \mathbb{R}$
- Масштабируемость:
  - Градиент производительности
  - $S = \nabla P$
- Масштабируемость по ресурсу:
  - $S_i = \frac{\partial P}{\partial R_i}$



# Кульминация

- Пространство велико: полный обход невозможен
- Случайные блуждания в направлении градиента?
  - Нужна оценка градиента в текущей точке (N направлений)
- Локальная оценка!
  - В каждой точке исследованием состояния можно оценить, будет ли расти P, если добавить конкретного ресурса
  - Таким образом, оценить, в какую сторону растёт градиент P
  - Для этого нужно уметь диагностировать bottleneck'и

# Главные леммы доклада

**Л1. Если производительность больше не растёт, значит, где-то что-то кончилось**

Следствие: надо найти, что кончилось, и добавить.

**Л2. Если производительность падает, значит кончился более дорогой ресурс**

Следствие: надо найти, что кончилось, и сэкономить.

**Л3. На многопроцессорных машинах процессоры кончаются в последнюю очередь.**

Следствие: искать прежде всего в другом месте.

# Главные леммы доклада

**Л1. Если производительность больше не растёт, значит, где-то что-то кончилось**

Следствие: надо найти, что кончилось, и добавить.

**Л2. Если производительность падает, значит кончился более дорогой ресурс**

Следствие: надо найти, что кончилось, и сэкономить.

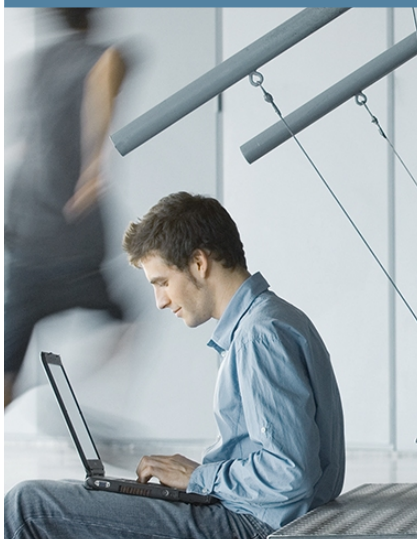
**Л3. На многопроцессорных машинах процессоры кончаются в последнюю очередь.**

Следствие: искать прежде всего в другом месте.

Доклад окончен.
Всем спасибо. Все свободны.

# Программа



- Теория
- **Практика**
  - **I/O**
    - **Network**
    - Disk
  - Memory
    - Caches
    - Coherency
    - TLB
    - Memory & Buses
  - CPU
    - Multicore
    - Sharing
    - ILP
- Q/A

# Network I/O: техпределы (latency)

- Скорость света!
  - Внутри Москвы по воздуху (20 км, n = 1): 0.06 мс
  - MSK → SPB по оптоволокну (800 км, n = 1.5): 0.4 мс
  - MSK → NYC по оптоволокну (7500 км, n = 1.5): 4 мс
- Пакетные среды передачи данных
  - ICMP ping внутри Москвы <= 1мс
  - ICMP ping MSK → SPB <= 10 мс
  - ICMP ping MSK → NYC <= 200 мс

# Network I/O: техпределы (bandwidth)

- Скрадывает latency
  - несколько пакетов в полёте
- Congestion control
  - Устойчивая передача требует подтверждения доставки и контроля целостности данных
  - Вариант 1: подтверждения на транспортном уровне (TCP)
  - Вариант 2: подтверждения на уровне приложения
- Bandwidth*Delay Product (BDP)
  - Определяет количество данных, могущих быть “в полёте”
  - В итоге, размеры буферов должны соотноситься с BDP

# Network I/O: техпределы (bandwidth, #2)

- IP – пакетный протокол, pps важнее kbps
  - Оборудованию нужно обработать каждый пакет
  - По поводу каждого пакета принять решение
  - А сетевому драйверу в ядре нужно ещё и отдать в user-space
- kbps/pps ограничен сверху MTU
  - MTU = Maximum Transmission Unit
  - Для справки: в Ethernet default MTU = 1500 bytes
    - 1 Gbps ~ 666 Kpps
  - Невозбранно повышать MTU нельзя
    - Вероятность ошибок при передаче тоже растёт

# Network I/O: оценка пределов

- Core i5, Linux 3.0, netperf:

| | Local Loopback | Gigabit Ethernet | Wireless (g) |
|---|---|---|---|
| MTU | 16384 | 1500 | 1500 |
| Bandwidth, Mbps | 14000 ± 1500 | 630 ± 18 | 25 ± 5 |
| RTT, us | 40 ± 1 | 240 ± 3 | 1200 ± 20 |

- Ограничители:
  - Local Loopback: стоимость memcpy() в ядре
  - Gigabit Ethernet: реалтековская сетевуха через D-Link :)
  - Wireless: шум в канале

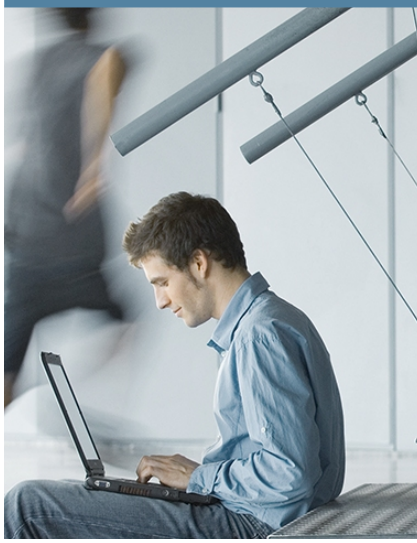
# Network I/O: что делать?

- Симптомы:
  - Большие задержки
  - Высокий system time
- Диагностика:
  - **netperf**, **iperf**: пиковая пропускная способность интерфейсов
  - **iptraf**, **nicstat**, **bwm-ng**: актуальная пропускная способность
  - **netstat**, **lsof**: мониторинг подключений
- Решения:
  - Консолидировать все приложения на одном хосте (hint: виртуализация)
  - Крутить сетевые настройки
    - TCP offload, IRQ balancing
    - буфера под BDP, MTU: http://www.psc.edu/networking/projects/tcptune/
  - Уходить на более быстрые интерфейсы (1G/10G Ethernet, Infiniband, Myrinet)
    - + bonding

# Программа

- Теория
- **Практика**
  - **I/O**
    - Network
    - **Disk**
  - Memory
    - Caches
    - Coherency
    - TLB
    - Memory & Buses
  - CPU
    - Multicore
    - Sharing
    - ILP
- Q/A



# Disk I/O: техпределы (интерфейсы)

- На быстрых дисках узкое место – интерфейс
  - SAS, SAS2: 1500, 4800 Mbps
  - SATA I, II, III: 1500, 3000, 6000 Mbps
  - USB 2.0, 3.0: 480, 5000 Mbps
  - iSCSI (10G): 10000 Mbps
- Не только само соединение, но и контроллеры
  - Кто-нибудь видел USB 2.0 девайс, реально выжимающий 480 Mbps?
  - А ещё есть RAID-контроллеры
- Трудно отличить от проблем самого диска
  - Пока вы не знаете, сколько диск может выжать сам по себе

# Disk I/O: техпределы и феномены (НЖМД)

- Скорость вращения (5400 – 15000 RPM)
  - Добавка к времени доступа (11 мс – 0.4 мс)
  - Напряжения в диске растут как квадрат радиуса и квадрат частоты
  - Но реальный ограничитель – трение блинов о воздух:
    - TDP ~ #блинов * RPM^2.8 * R^4.8
- Скорость позиционирования головок (1-2 мс)
  - Добавка к времени доступа
  - Огромные токи на сервоприводах
- Огромная плотность записи (1-100 Тбит / см^2)
  - Частично компенсирует линейную скорость чтения
  - Кроме плотности записи, скорость чтения пока не ограничена

# Disk I/O: техпределы и феномены (SSD)

- Делают много быстрых операций одновременно
  - Очень легко насыщают интерфейсы, но...
- Media Wearout: ресурс ячеек ограничен
  - Wear leveling: распределяем нагрузку между физическими страницами
  - Write amplification: ввиду внутренних копирований количество записей актуально выше
- Garbage Collection: стирание требует перемещения данных
  - Пишем страницами (~ 4 Kb), но стирать можем только блоками (~512 Kb)
  - Контроллер вынужден агреггировать грязные страницы в блоки
  - Нет свободных страниц для записи? Write stall!
- Read Disturb: чтение ячейки может изменить состояние соседних
  - Проявляется через ~100K+ чтений
  - Контроллеру нужно следить и регулярно перезаписывать данные.
  - Маленький read stall!

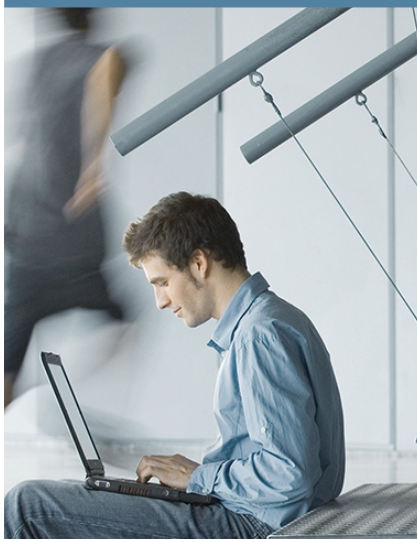
# Disk I/O: что делать?

- Симптомы:
  - Большие iowait%, r_await, w_await, %util
- Диагностика:
  - **iostat**: статистика в разрезе дисковых устройств
  - **iotop:** статистика в разрезе отдельных процессов
  - **strace**: инструментирование I/O syscall'ов
  - **smartctl, mhdd:** чтение внутреннего здоровья и тесты
- Решения:
  - Отложенная запись: лучше писать в память, чем сразу на диск
  - Больше кешей чтения: лучше читать из памяти, чем каждый раз с диска
  - SSD для операций с низкой латентностью и/или высокими IOPS
  - RAID для операций с большим throughput

# Программа

- Теория
- **Практика**
  - I/O
    - Network
    - Disk
  - **Memory**
    - **Caches**
    - Coherency
    - TLB
    - Memory & Buses
  - CPU
    - Multicore
    - Sharing
    - ILP
- Q/A



# Memory (Caches): теория

- Технически считаются частью CPU
- Средняя задержка:

```
<average latency> =
    (1 - <cache miss rate>)*<hit latency> +
          <cache miss rate>*<miss latency>
```

- Как можно улучшить:
  - Hit latency: скорость выбоки в доли наносекунды
  - Hit latency: несколько портов для расшаренных кешей
  - Miss rate: размеры во многие мегабайты
  - Miss rate: высокая ассоциативность
  - Miss latency: многоуровневые кеши

# Memory (Caches): техпределы

- Увеличить скорость выборки?
  - Только уменьшая физический размер
    - А дальше квантовые эффекты
- Увеличить объём?
  - Только увеличивая физический размер
  - Нужно больше логики выборки (растёт как O(log n))
  - Нужна большая ассоциативность
- Увеличить ассоциативность?
  - Ещё больше вырастет логика выборки (растёт как O(a))
- Увеличить количество портов?
  - Ещё больше вырастет логика выборки

# Memory (Caches): техпределы (2)

- Физический предел:
  - Выборка за один такт: 5 Ггц ~ 0.2 нс
  - За это время свет проходит 6 см в вакууме
  - А ещё надо попереключать транзисторы
- Перевод:
  - Нельзя сделать больше 5*10^9 операций с памятью в одной нитке в секунду (и это если вам повезло поместиться в L1)
  - С учётом нескольких портов можно выжать пару Тбит/сек
  - Реальный Nehalem:
    - L1 bandwidth = 1 / (3 ГГц) * 128 бит/порт * 1 порт = 426*10^(-10) бит/сек = 42.6 Гбит/сек

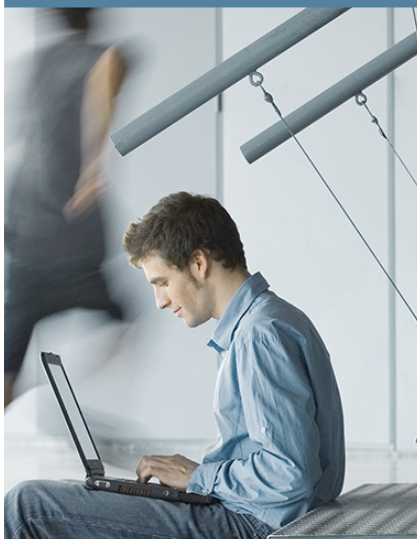
 

# Memory (Caches): что делать?

- Симптомы:
  - Равномерное замедление по всему коду, работающему с памятью
- Диагностика:
  - А можно и не диагностировать.
  - **vtune, solstudio analyzer, perf, oprofile**: процессорные счётчики и маппинг обратно на адреса в программе
  - **cachegrind**: симуляция кешей
- Решения:
  - spatial locality: упаковываем структуры данных ближе друг к другу
  - temporal locality: группируем операции над данными
  - non-temporal operations, которые не гадят в кеш
  - ЗАБИВАЙ @ ПАРАЛЛЕЛЬ

# Программа



- Теория
- **Практика**
  - I/O
    - Network
    - Disk
  - **Memory**
    - Caches
    - **Coherency**
    - TLB
    - Memory & Buses
  - CPU
    - Multicore
    - Sharing
    - ILP
- Q/A

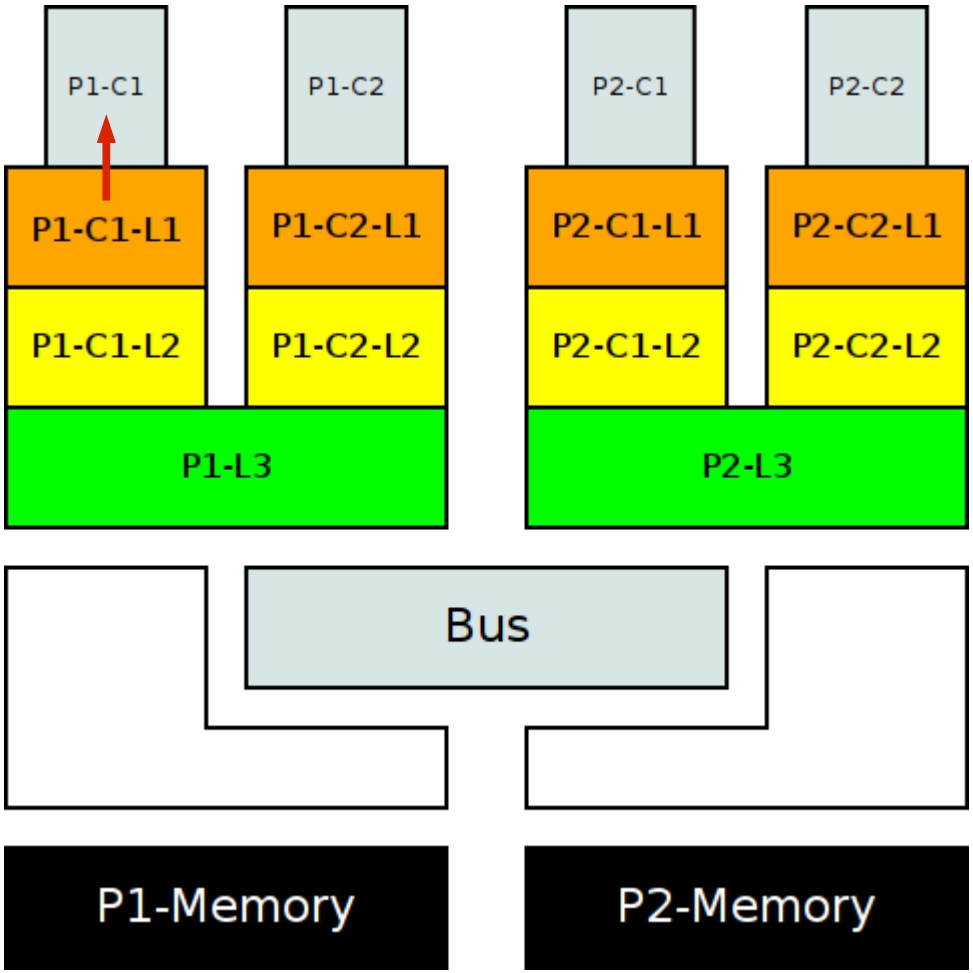
# Memory (Coherency): теория

- Куча процессоров?
  - Каждый пишет и читает
- Нужно поддержать абстракцию общей памяти
  - Кеши существенно усложняют схему
  - Требуется обмениваться сообщениями между разными кешами
  - Конфликты за общую память тоже надо разруливать
  - google://”cache coherency”
- Пертубации в зависимости от расположения данных
  - На самом деле, мы сделали очень много, чтобы быть быстрее в некоторых случаях
  - ...а гундим из-за того, что плохо во всех остальных

# Memory (Coherency): оценка пределов

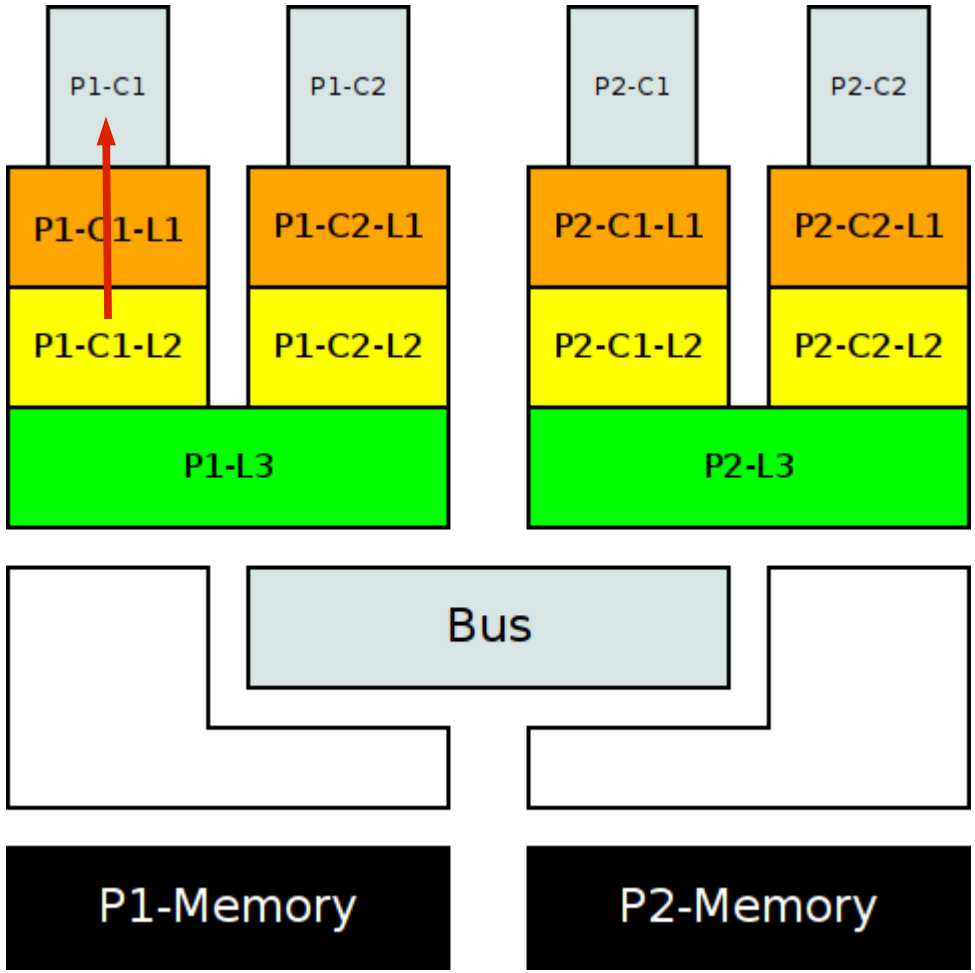
- Читает ядро P1-C1
- Данные из P1-C1-L1
- Очень хорошо:
  - 3 clks
  - 45 Gbps



# Memory (Coherency): оценка пределов

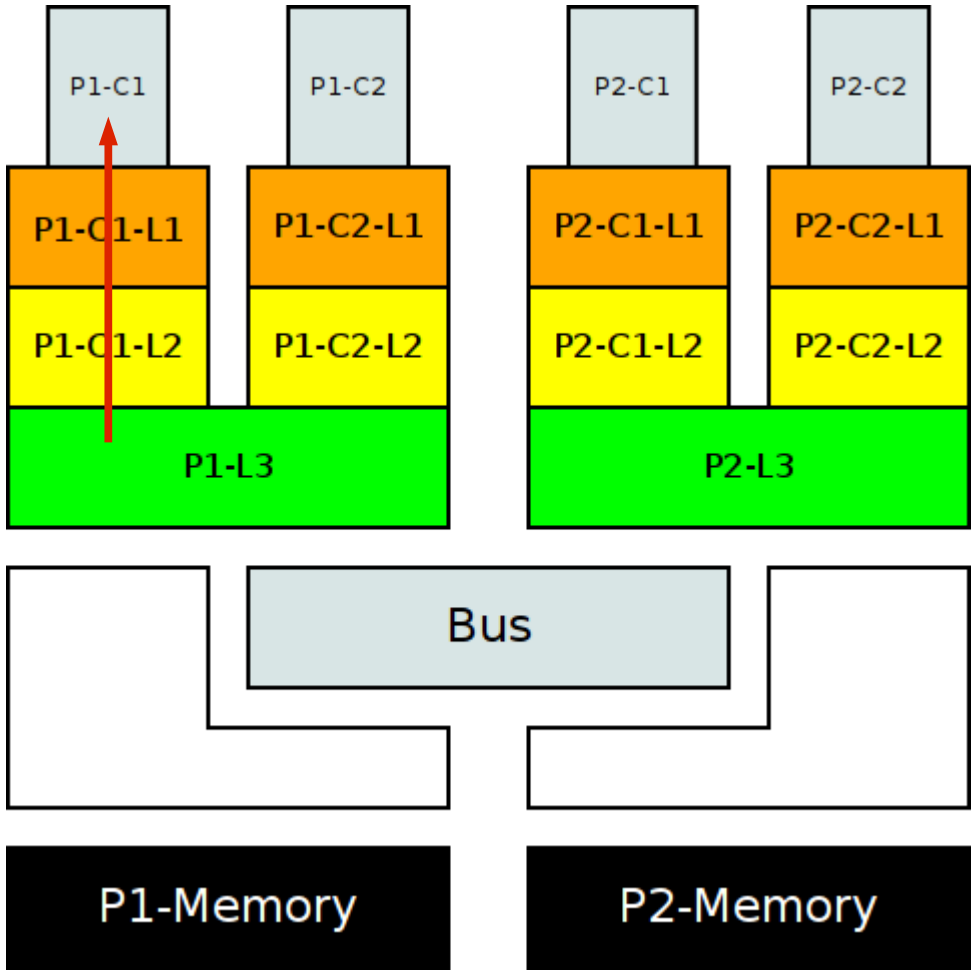
- Читает ядро P1-C1
- Данные из P1-C1-L2
- L2 больше, но медленее
- Нормально:
  - 12 clks
  - 31 Gbps



# Memory (Coherency): оценка пределов

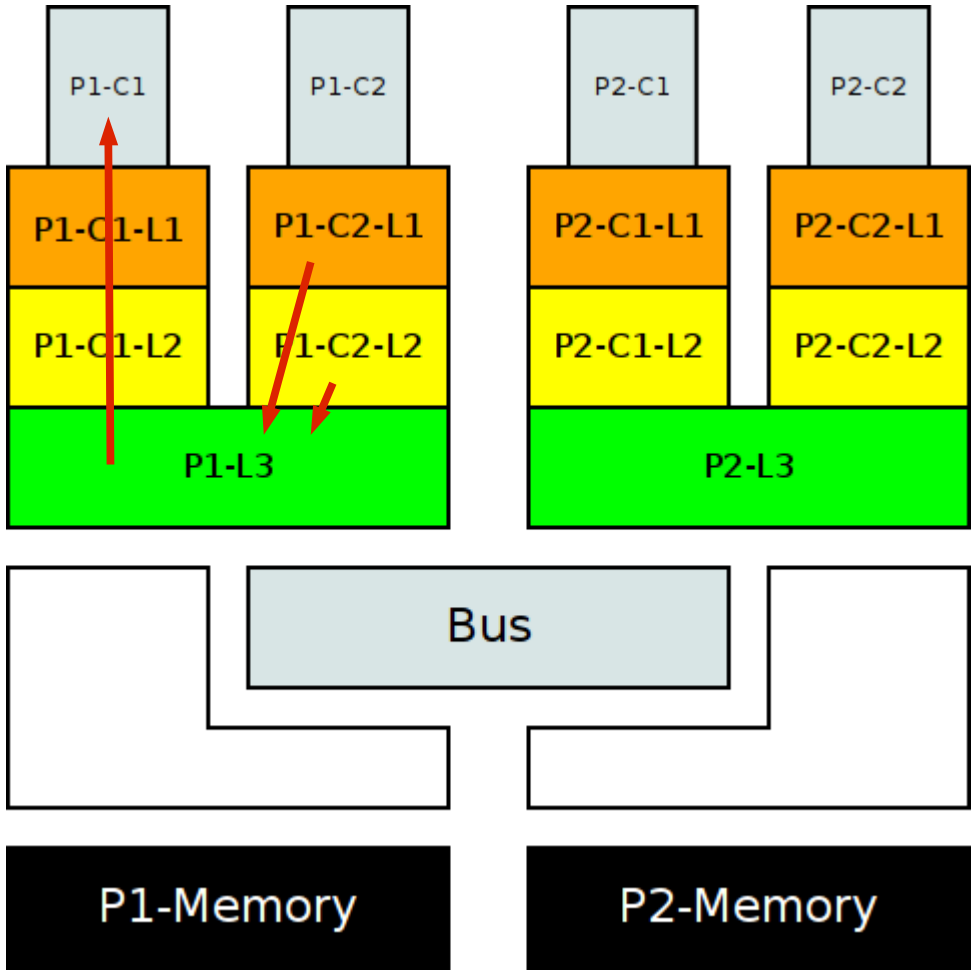
- Читает ядро P1-C1
- Данные в P1-C2-L1/2
- Кешлиния в “shared” состоянии
- Приемлемо:
  - 40 clks
  - 26 Gbps



# Memory (Coherency): оценка пределов

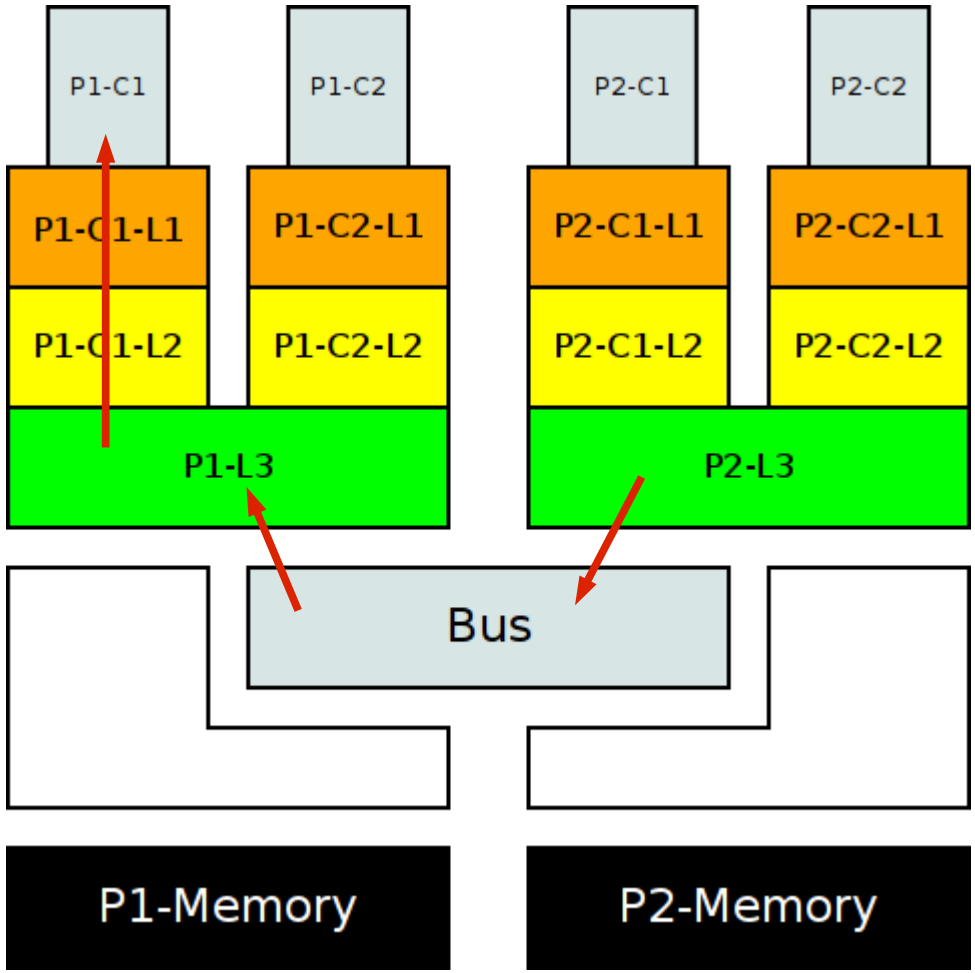
- Читает ядро P1-C1
- Данные в P1-C2-L1/2
- Кешлиния в “exclusive” состоянии
  - P1-C2 мог её перевести в “modified” без нотификации
  - L3 нужно опросить P1-C2-L1/L2
- Существенно:
  - 75-83 clks
  - 10-13 Gbps



# Memory (Coherency): оценка пределов

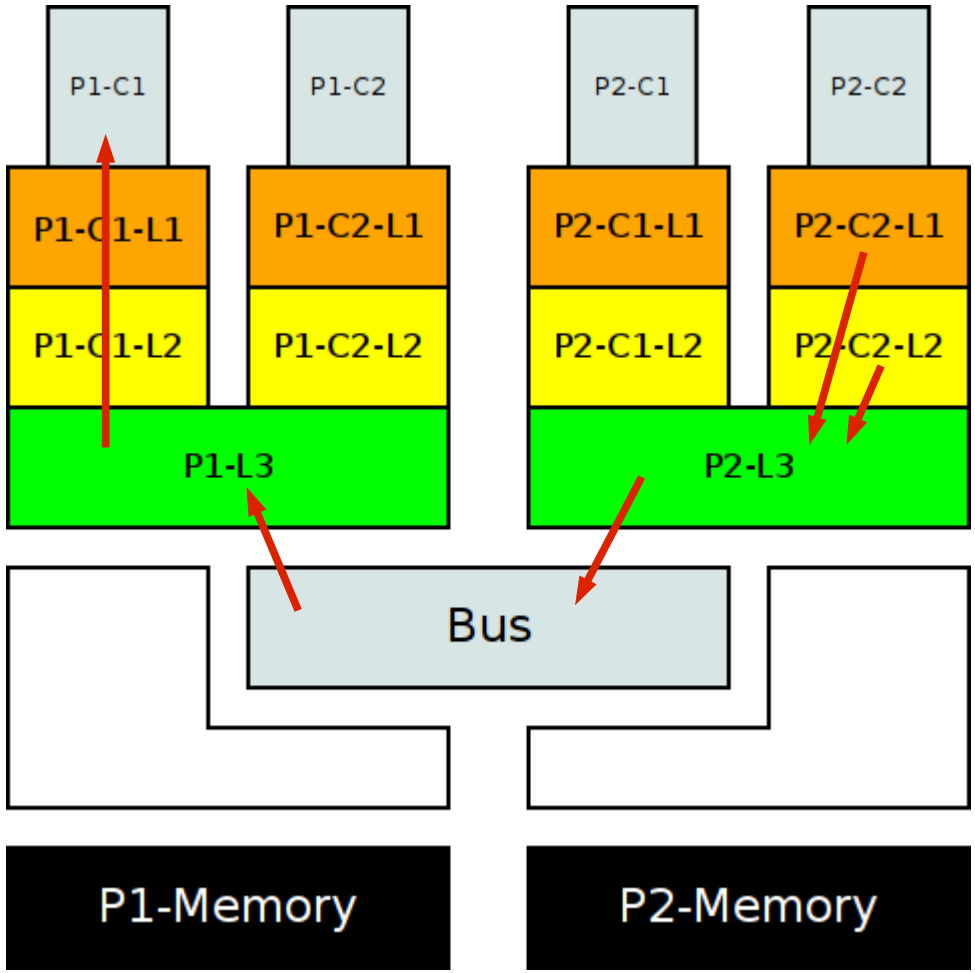
- Читает ядро P1-C1
- Данные в P2-L3
- Кешлиния в “shared” состоянии
  - L3 может обслужить
- Сравнительно дорого:
  - 170 clks
  - 10 Gbps



# Memory (Coherency): оценка пределов

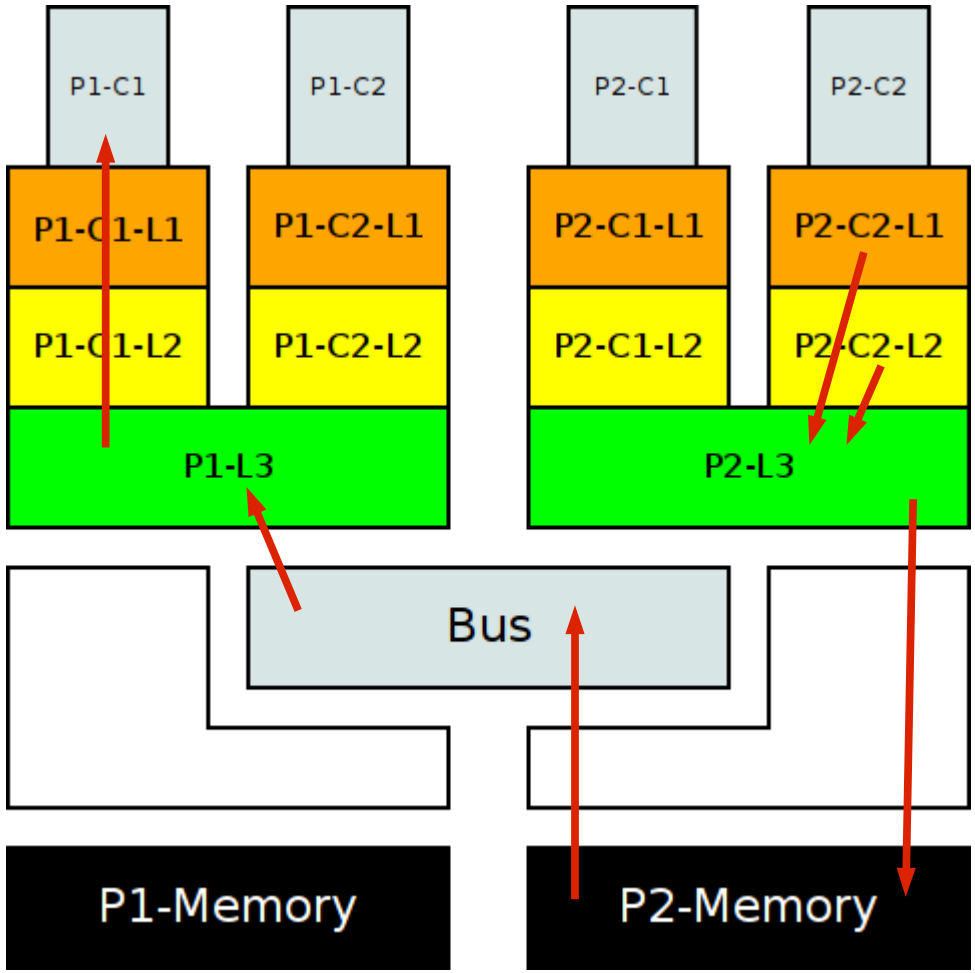
- Читает ядро P1-C1
- Данные в P2-L3
- Кешлиния в “exclusive” состоянии
  - Нужно проверить P2-C*-L*
- Очень дорого:
  - 190 clks
  - 9 Gbps



# Memory (Coherency): оценка пределов

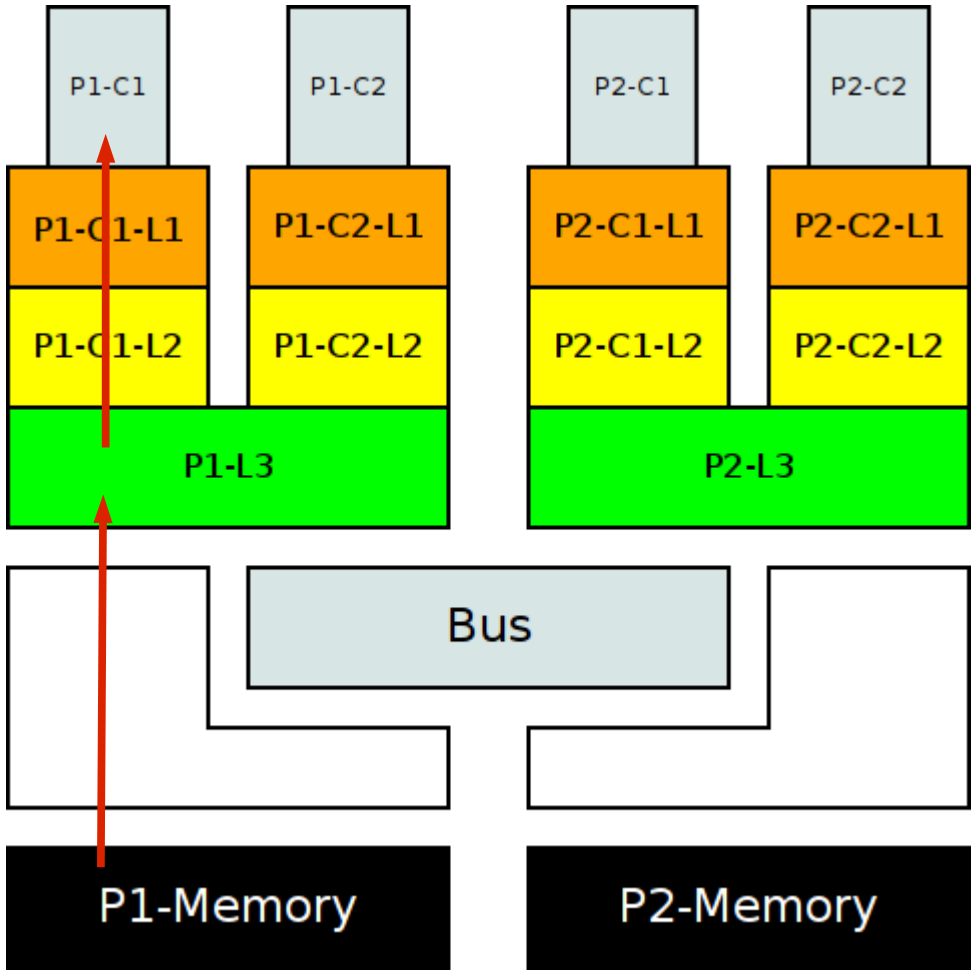
- Читает ядро P1-C1
- Данные в P2-L3
- Кешлиния в “modified” состоянии
  - Нужен writeback!
  - …и он точно произойдёт в “чужую” память
- Очень дорого:
  - 310 clks
  - 5.5 Gbps



# Memory (Coherency): оценка пределов

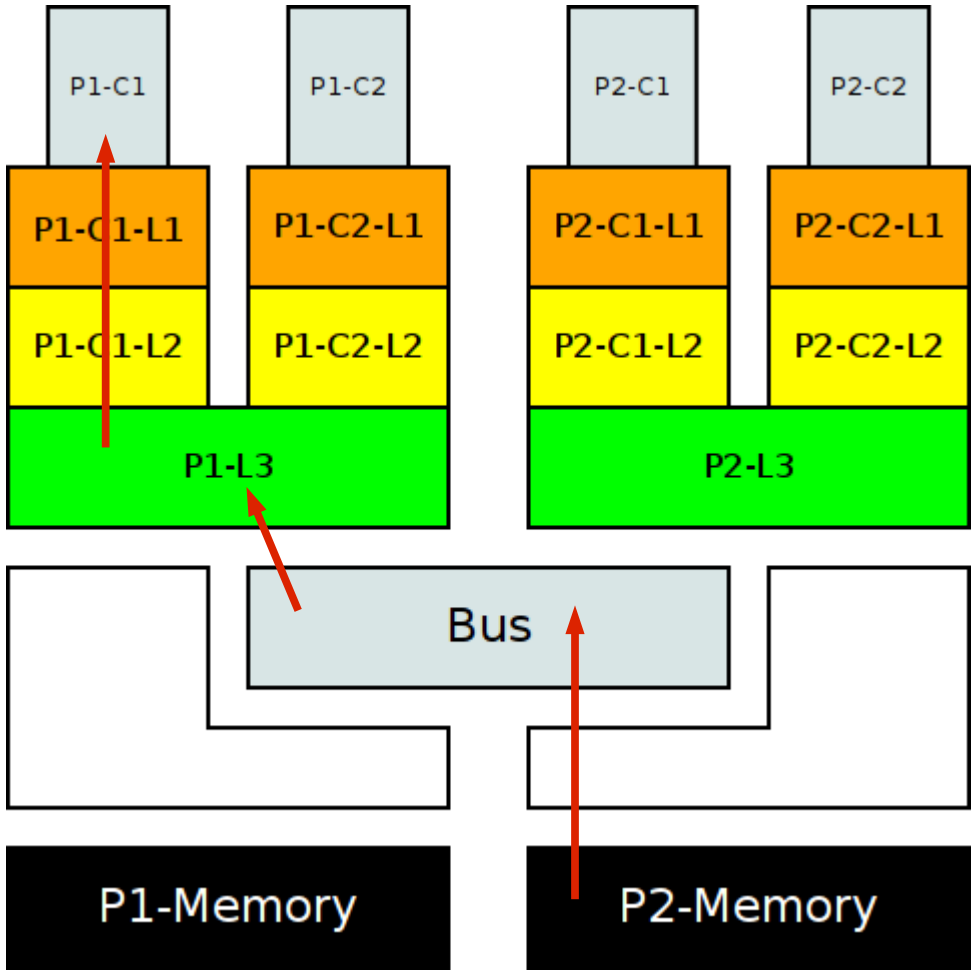
- Читает ядро P1-C1
- Данных нет в P*-L3
- Данные в своей ноде
- Очень дорого:
  - 190 clks
  - 9 Gbps



# Memory (Coherency): оценка пределов

- Читает ядро P1-C1
- Данных нет в P*-L3
- Данные в чужой памяти
- Очень дорого:
  - 310 clks
  - 5.5 Gbps



# Memory (Coherency): следствие

- Обмениваемся данными между потоками?
- Внутри сокета на 3 ГГц:
  - **`3*10^9 [ticks/sec] / 70 [ticks/op] = 42*10^6 ops/sec`**
  - **`42*10^6 [ops/sec] * 256 [bits/op] = 10*10^9 bits/sec`**
- Между сокетами на 3 ГГц:
  - **`3*10^9 [ticks/sec] / 310 [ticks/op] = 10*10^6 ops/sec`**
  - **`10*10^6 [ops/sec] * 256 [bits/op] = 2.5*10^9 bits/sec`**
- ...и это хардварный предел
  - В реальности ещё и протокол когерентности
  - ...ещё и конкуренция с другими ядрами

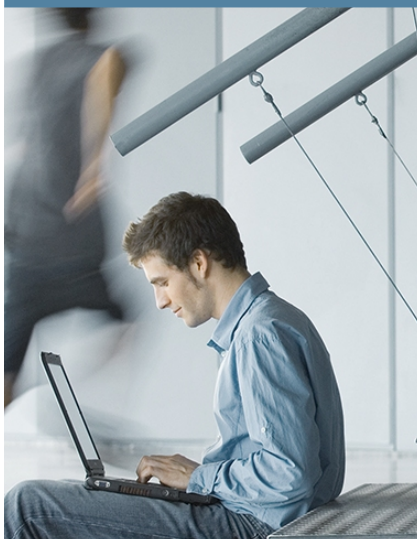
# Memory (Coherency): что делать?

- Симптомы:
  - Высокий CPI
  - Много coherence miss'ов
- Диагностика:
  - **vtune, solstudio analyzer, perf, oprofile**: процессорные счётчики
  - **busstat, numastats**: мониторинг трафика на шине
- Решения:
  - море симптоматических решений
  - лучше много immutable объектов, чем один постоянно реиспользуемый
  - связанные потоки на одном процессорном сокете / NUMA-node
  - пользуемся плюшками -XX:+UseNUMA

# Программа

- Теория
- **Практика**
  - I/O
    - Network
    - Disk
  - **Memory**
    - Caches
    - Coherency
    - **TLB**
    - Memory & Buses
  - CPU
    - Multicore
    - Sharing
    - ILP
- Q/A



# Memory (TLB): теория

- Virtual Memory – ещё одна абстракция
  - Процессы адресуют память логическими адресами
  - Контроллеры памяти адресуют физическими адресами
- Кто преобразует виртуальные адреса в физические?
  - Memory Management Unit (MMU)
  - Уже давно часть CPU
- Как договариваются с ОС?
  - Page Tables: многоуровневые таблицы соответствий адресов
  - Одно преобразование: Page Walk, стоит тысячи циклов
  - Не найдена страница? Page fault.
- Кэш трансляции – TLB (translation lookaside buffer)

# Memory (TLB): теория

- Размер страниц можно менять
  - Чем больше страница, тем меньше запросов в TLB
    - (для памяти того же размера)
  - По умолчанию, 4 Кб; есть 2 Мб, 4 Мб, на совсем новых процах 1 Гб
- Размер TLB фиксирован
  - L1 DTLB: 64 entries (4K page), 32 entries (2M/4M), 4 entries (1G page)
  - L2 DTLB: 512 entries (4K page)
- Протоколы когерентности: физические адреса
  - Aliasing: процессы имеют общую физическую страницу?
  - L1 может быть индексирован *логическими* адресами
  - L2/L3 обычно индексированы физическими адресами
  - Каждое обращение к памяти – обращение к TLB

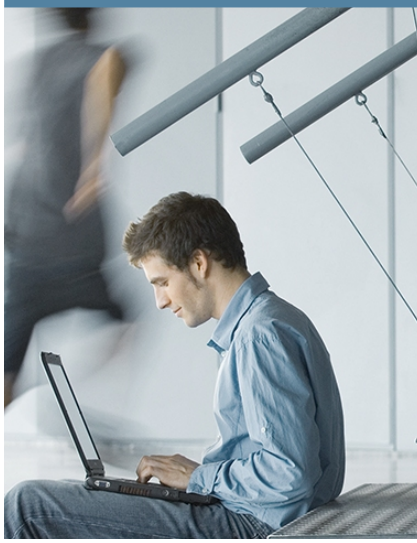
# Memory (TLB): что делать?

- Симптомы:
  - Равномерное замедление на операциях с памятью
  - Много TLB miss'ов
- Диагностика:
  - **vtune, solstudio analyzer, perf, oprofile**: процессорные счётчики
- Решения:
  - Группируем данные плотнее в адресном пространстве
  - Используем -XX:+UseLargePages
    - Иногда требует привилегий суперпользователя, RTFM

# Программа

- Теория
- **Практика**
  - I/O
    - Network
    - Disk
  - **Memory**
    - Caches
    - Coherency
    - TLB
    - **Memory & Buses**
  - CPU
    - Multicore
    - Sharing
    - ILP
- Q/A



# Memory (Buses & RAM): теория

- Контроллеры памяти адресуют очень много медленной памяти
  - Чем ближе к ядрам, тем меньше latency
  - Если интегрирован в кристалл, то в системе несколько MC, лучше throughput
  - IMC естественным образом сегментируют память!
- Максимальный bandwidth можно определить:
  - Каждое ядро имеет fill buffer, в котором хранятся запросы в память
  - Тогда трафик можно прикинуть, зная latency:
    - bandwidth = 64 [bytes/line] * 10 [slots] / 60 ns = ~10.6 GB/sec
  - Для справки: средняя скорость одного DDR3-10600: 10.6 GB/sec
    - Т.е. одного канала хватит на кормёжку одного ядра
  - Несколько каналов могут улучшить throughput, но не улучшат latency

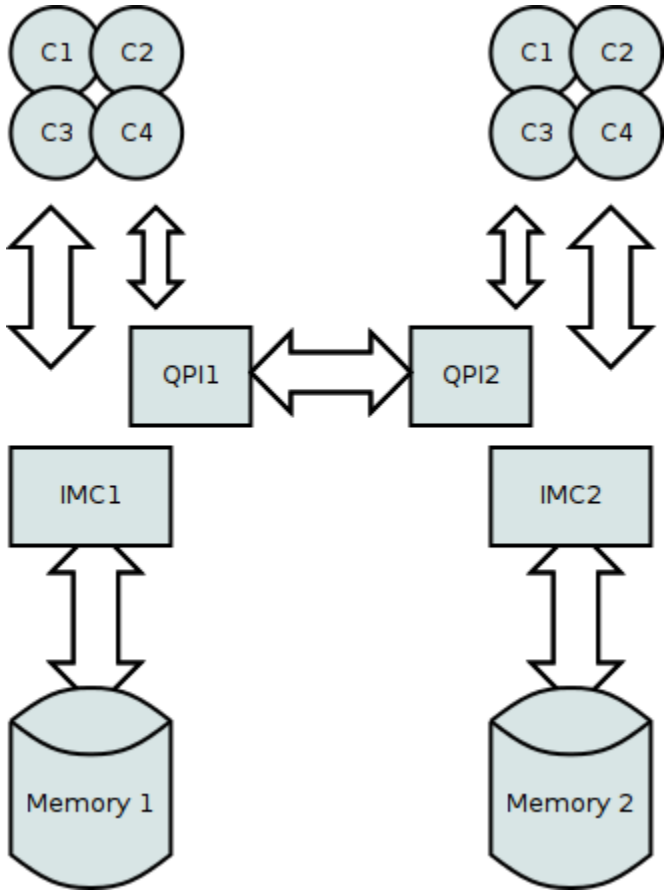
# Memory (Buses & RAM): техпределы

- DRAM ~ конденсатор с ключом
  - Нетривиальная логика выборки, ADC, и т.п.
- DDR1 → DDR2 → DDR3 → DDR4
  - Увеличивают throughput
    - Увеличивают ширину чтения: 2 → 4 → 8 → 8
  - Упёрлись в барьер для latency
    - DRAM Latency Barrier: 10-12 ns
  - Увеличивают размер чипов
    - ...пытаясь при этом не сильно просадить latency

# Memory (Buses & RAM): проблемы

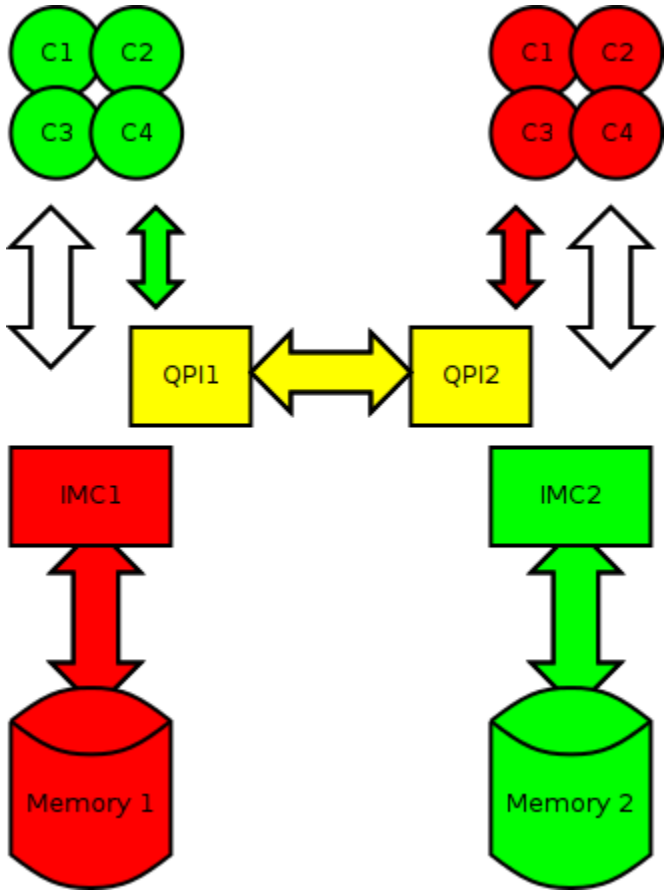
- Нет однозначно правильного расположения данных в памяти
  - Для коммуникации лучше сгруппировать
  - Для сырого throughput'а лучше расклеить
- (Ответ на вопрос, почему это не делается автоматически)



# Memory (Buses & RAM): проблемы

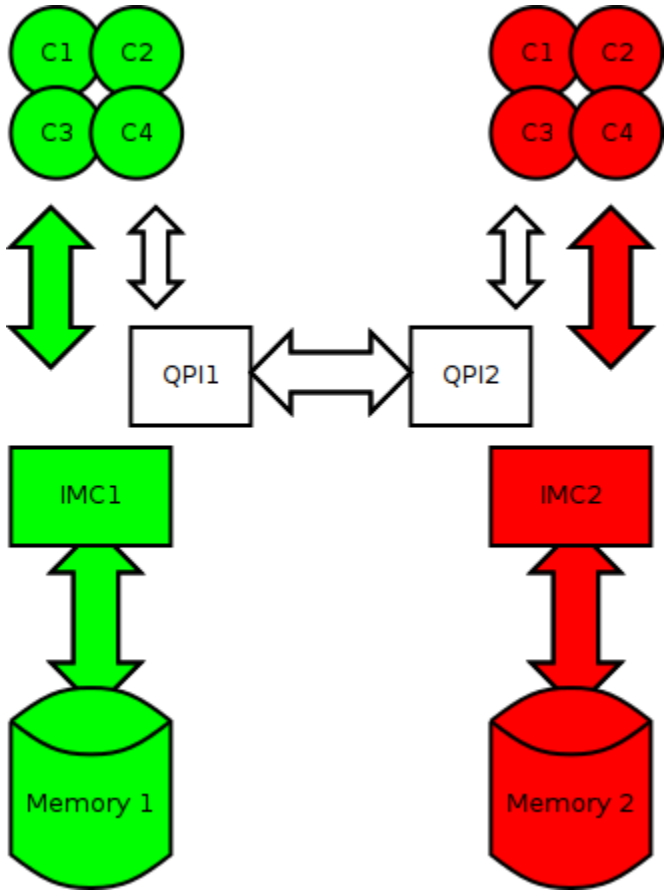
- Явная проблема:
  - Выбираем весь remote трафик
  - Доступна куча локального трафика



# Memory (Buses & RAM): проблемы

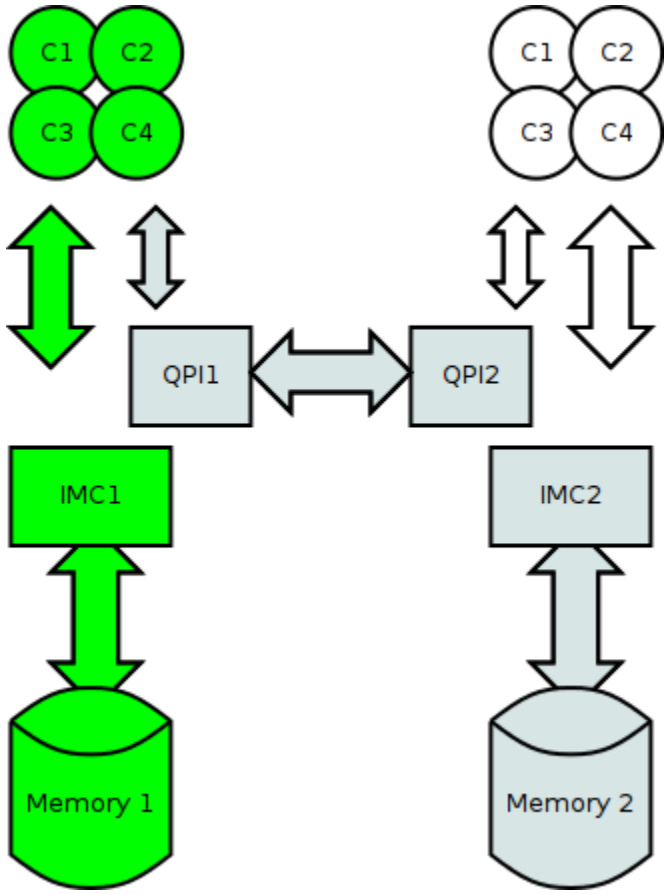
- Отличное балансирование:
  - Только для хорошо симметричных задач



# Memory (Buses & RAM): проблемы

- Компенсация:
  - Если IMC1 уже захлебнулся
  - Имеет смысл сходить в IMC2
  - И это может оказаться быстрее!



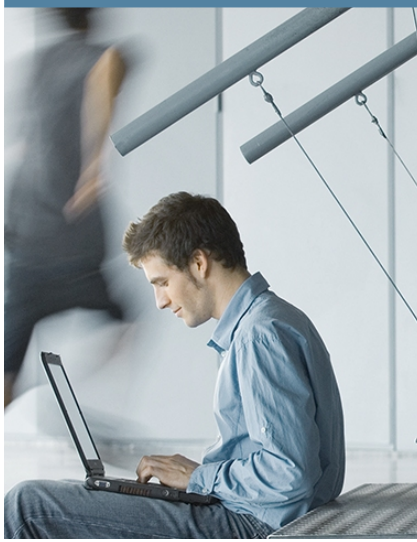
# Memory (Buses & RAM): что делать?

- Симптомы:
  - Высокая утилизация шин
  - Мало локального трафика
- Диагностика:
  - **vtune, solstudio analyzer, perf, oprofile**: процессорные счётчики
  - **busstat**: утилизация шин
- Решения:
  - Больше памяти, быстрее контроллеры
  - Интеллектуальная раскладка данных по памятям
  - -XX:+UseNUMA

# Программа

- Теория
- **Практика**
  - I/O
    - Network
    - Disk
  - Memory
    - Caches
    - Coherency
    - TLB
    - Memory & Buses
  - **CPU**
    - **Multicore**
    - Sharing
    - ILP
- Q/A



# CPU (multicore): теория

- "Free lunch is over"
  - частоту дальше наращивать нельзя
  - разгонять последовательное исполнение за счет ILP уже не имеет смысла
- Закон Мура: транзисторов всё больше и больше
  - Куда их девать?
  - Нарастить кэш? (И так уже 60-70% площади кристалла)
  - Выход: наращивать ядра

# CPU (multicore): пример

- Переходим с 1-го на 100 ядер
- Задача очень хорошо распараллелена – последовательная часть всего 1%
- Ускорение?
  - В 100 раз
  - В 99 раз
  - В 50 раз
  - В 2 раза
  - Не ускорится

# CPU (multicore): пример

- Переходим с 1-го на 100 ядер
- Задача очень хорошо распараллелена – последовательная часть всего 1%
- Ускорение?
  - ~~В 100 раз~~
  - ~~В 99 раз~~
  - **В 50 раз**
  - ~~В 2 раза~~
  - ~~Не ускорится~~

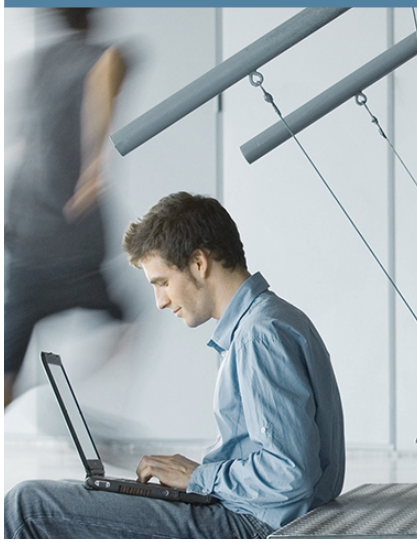
 

# CPU (multicore): что делать?

- Симптомы
  - Слабая утилизация ядер
  - Низкое количество тредов
- Диагностика
  - **vmstat**, **mpstat**, **jvisualvm**: оцениваем фактический параллелизм
- Решение
  - Лучше параллелим то что есть
  - Не получается – изобретаем новые алгоритмы.
  - Achtung!
    - Не перестараться
    - Не забываем про балансировку задач.

# Программа

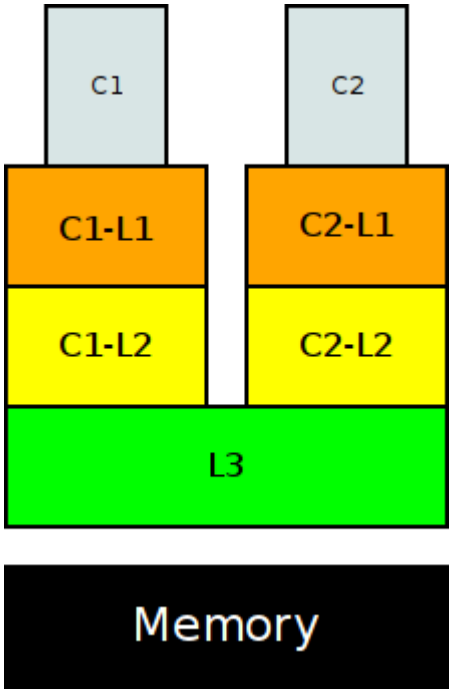


- Теория
- **Практика**
  - I/O
    - Network
    - Disk
  - Memory
    - Caches
    - Coherency
    - TLB
    - Memory & Buses
  - **CPU**
    - Multicore
    - **Sharing**
    - ILP
- Q/A

# CPU (sharing): хитрый пример

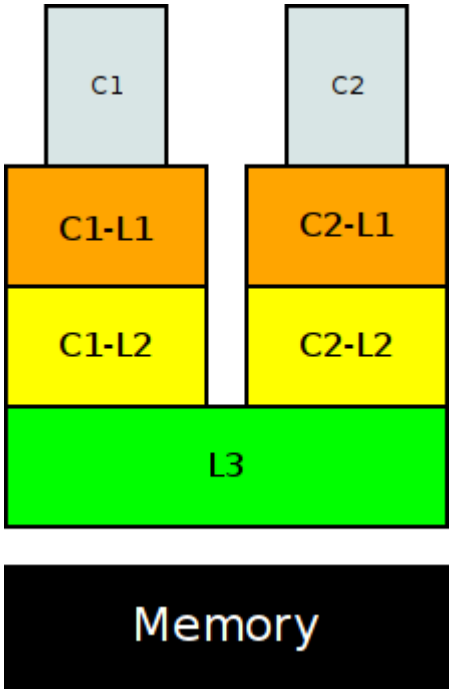
- Инклюзивные L1/L2/L3
- На C1 работает CPU-bound задача
  - Сидит себе в C1-L1
  - C1-L1D почти не используется
  - C1-L1I используется для кода
- На C2 работает mem-bound задача
  - Агрессивно гадит в C2-L1D/L2/L3



# CPU (sharing): хитрый пример

- Инклюзивные L1/L2/L3
- На C1 работает CPU-bound задача
  - Сидит себе в C1-L1
  - C1-L1D почти не используется
  - C1-L1I используется для кода
- На C2 работает mem-bound задача
  - Агрессивно гадит в C2-L1D/L2/L3
- C1 работает в разы хуже
  - C2 постоянно вытесняет код из C1-L1I



# CPU (sharing): теория

- Память – основной разделяемый ресурс
  - HW memory model
    - Правила видимости значений в памяти
  - Memory Barriers
    - Усиление порядка операций над памятью
    - Сериализуют чтения/записи
  - True sharing
    - Настоящий конфликт за данные, например, запись в одну ячейку памяти
  - False sharing
    - Наведённый конфликт за данные
    - Пишем в разные ячейки, но кеш-линия одна

# CPU (sharing): базовые примитивы в Java

- Java Memory Model
  - happens-before
- Atomics
  - Compare-and-Set (CAS)
  - Load-Linked / Store-Conditional (LL/SC)
- Locks
  - synchronized
  - java.util.concurrent.Lock

# CPU (sharing): базовые примитивы

- Низкоуровневые примитивы (HW):
  - Memory barriers
    - Относительно дорого
    - На многих платформах – глобальные барьеры
  - HW CAS, LL/SC
    - Локальные – дёшево
    - Глобальные – дорого
- Низкоуровневые примитивы (OS):
  - OS mutex, conditional wait
    - Относительно дорого

# CPU (sharing): spinloop

```
for() {
    while(owner != __nobody) { // ожидание
        pause;
    }
    if (CAS(&owner, __nobody, __me)) { // захват
        break;
    }
}
```

- Плюсы:
  - не требует остановки/старта треда
- Минусы:
  - съедает процессорные ресурсы

# CPU (sharing): synchronized vs. RL

- synchronized {}
  - **init**
  - **biased** – очень дешево в локальном случае, дорогой переход в другие состояния
  - **thin** – CAS в неконфликтном случае
  - **fat** – в случае конфликта
- java.util.concurrent.Lock
  - CAS для захвата
  - park/unpark в случае конфликта
  - fair/unfair mode

# CPU (sharing): synchronized vs. RL

- synchronized {}
  - Лучше чем java.util.concurrent.Lock
    - В случае biased locking
    - Реализован adaptive spinning
  - Такой же как java.util.concurrent.Lock
    - Если находится в состоянии thin
  - Хуже чем java.util.concurrent.Lock
    - Нет политики честности, большая зависимость от ОС
    - Fat состояние дорого, нет inflation

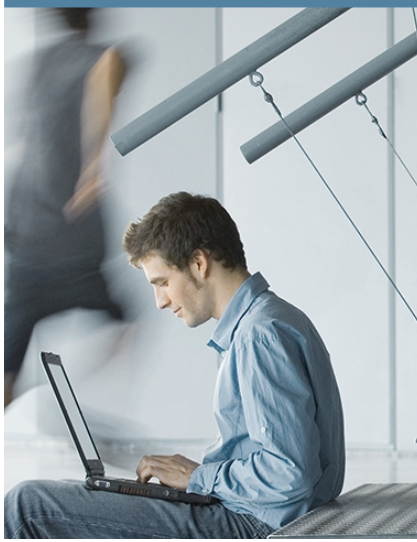
 

# CPU (sharing): что делать?

- Симптомы:
  - Для явных блокировок: низкая утилизация и видно локи в профиле
  - Для неявных конфликтов: высокая утилизация и видно горячие места в профиле
- Диагностика:
  - **jvisualvm, solstudio, jrmc**: профилируем приложение
- Решения:
  - Минимизируем коммуникацию, насколько возможно
  - Миниатюрные блокировки → spinloops
  - Тяжёлые spinloops → блокировки

# Программа



- Теория
- **Практика**
  - I/O
    - Network
    - Disk
  - Memory
    - Caches
    - Coherency
    - TLB
    - Memory & Buses
  - **CPU**
    - Multicore
    - Sharing
    - **ILP**
- Q/A

# CPU (ILP): мотивация

- Проблема:
  - Невозможно получить достаточную параллелизацию (TLP)
- Как увеличить производительность последовательной программы?

# CPU (ILP): мотивация

- Проблема:
  - Невозможно получить достаточную параллелизацию (TLP)
- Как увеличить производительность последовательной программы?
  - Необходимо увеличить что-нибудь другое:
    - частота (CPU bound workloads)
    - размеры кэшей (memory bound workloads)
    - мелкогранулярная параллелизация (ILP)
  - ILP = Instruction-Level Parallelism

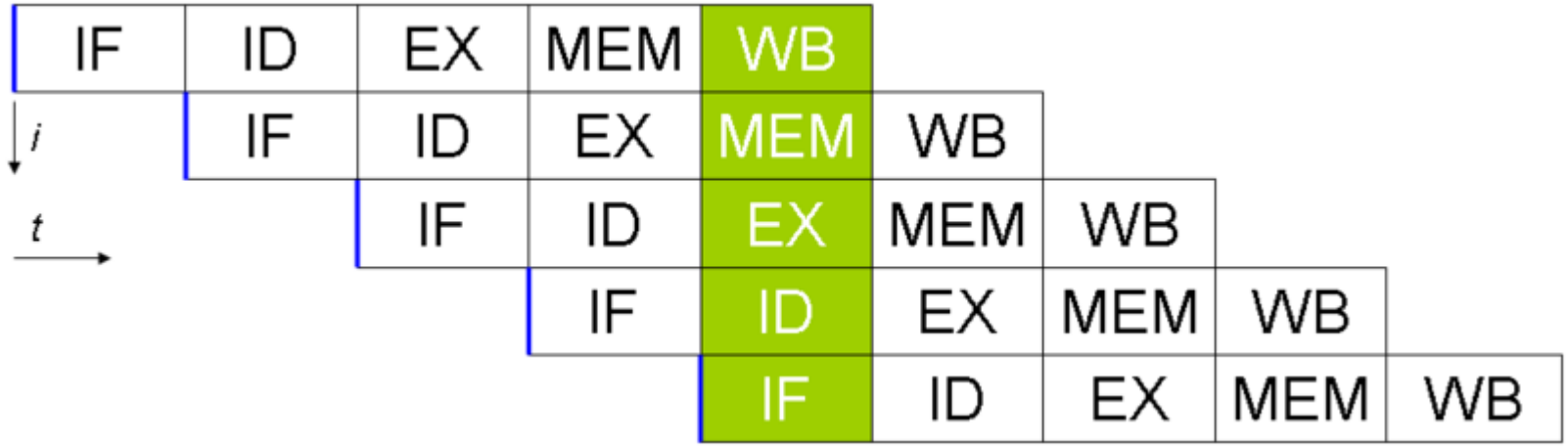
# CPU (ILP): типы

- Статический ILP
  - Создается компилятором (векторизация)
  - Создается разработчиком (non-portable)
  - Явно выражен в ассемблерном коде:
    - SIMD (SSE, AVX, VIS)
    - VLIW
- Динамический ILP
  - Скрыт за HW:
    - Конвееризация (pipeline)
    - Superscalar
    - Out of Order (OoO)

# CPU (ILP): конвейер

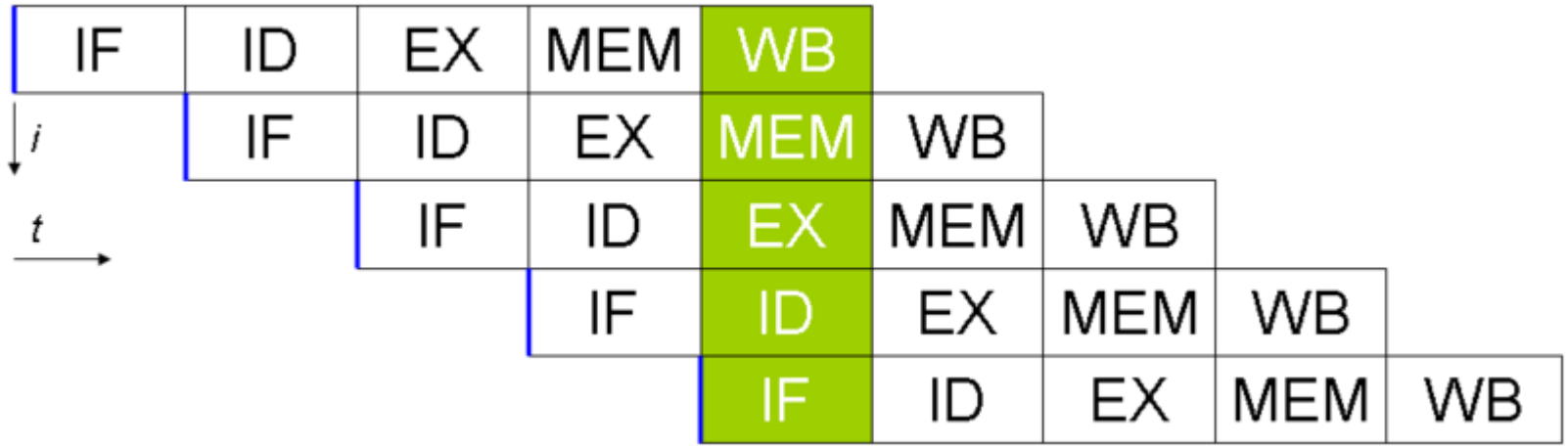
- Конвейер (как нас все учат):



# CPU (ILP): конвейер

- Конвейер (как нас все учат):



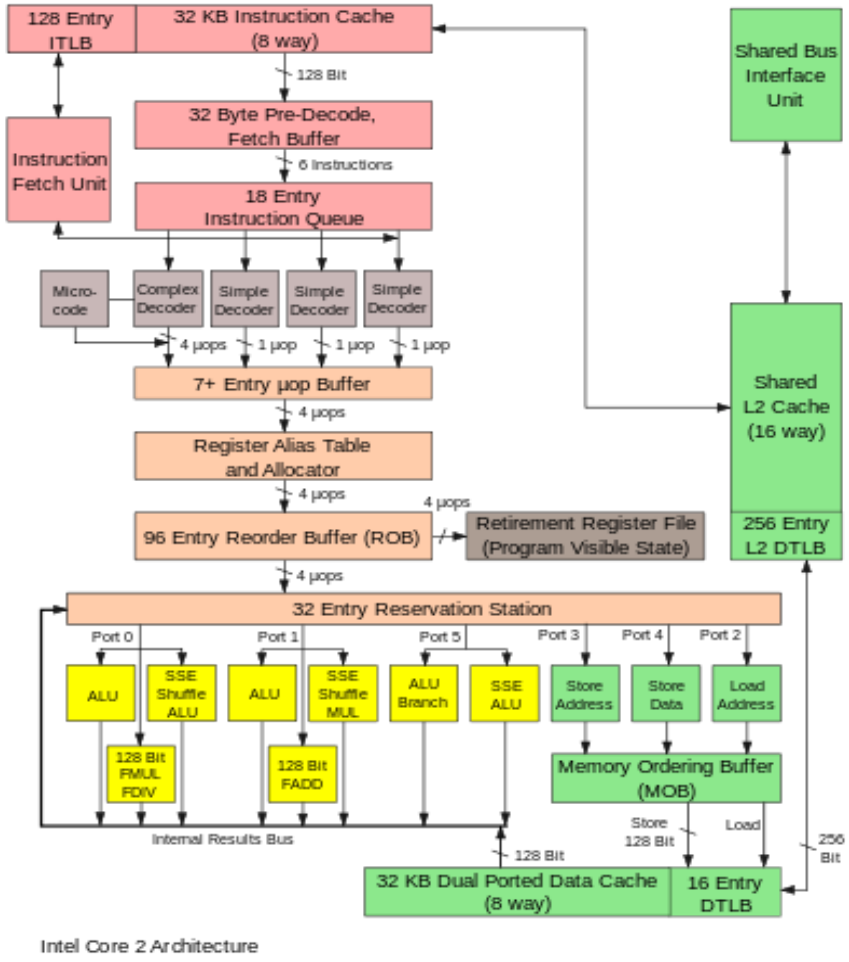

~~ВЛАСТИ СКРЫВАЮТ!~~

НА САМОМ ДЕЛЕ ВСЕ НЕ ТАК!

# CPU (ILP): конвейер

- “Конвейер” на самом деле:

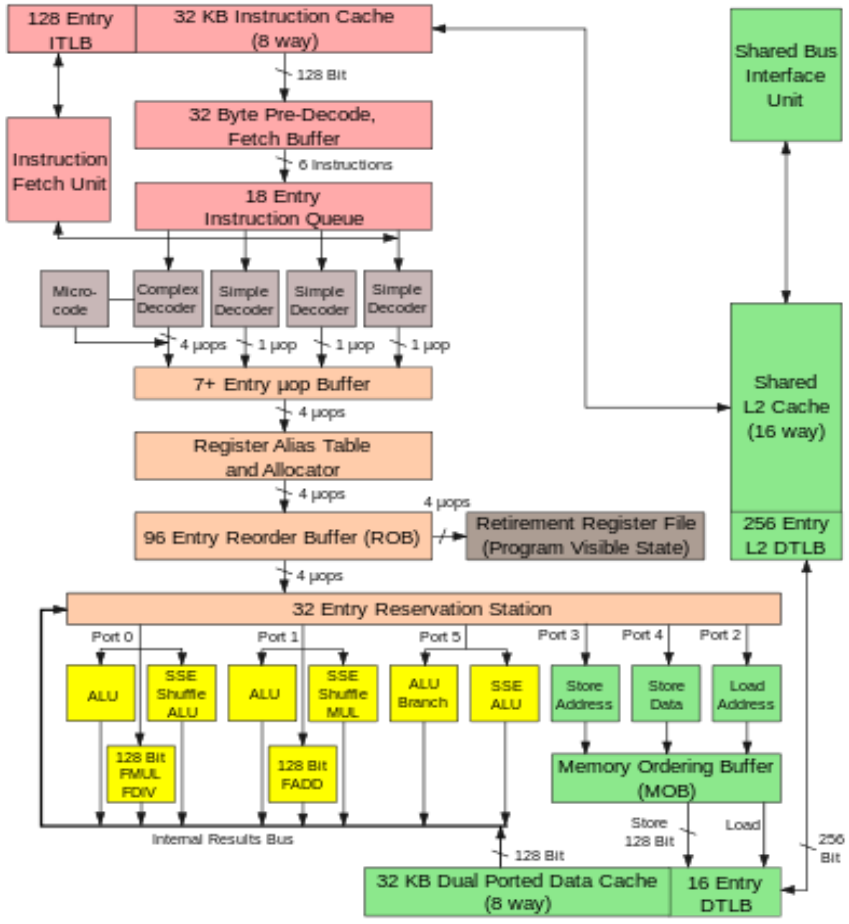


Intel Core 2 Architecture

# CPU (ILP): конвейер

- “Конвейер” на самом деле:
- Ключевые слова для любопытных:
  - Tomasulo
  - Reservation station
  - RAT
  - ROB
  - etc...

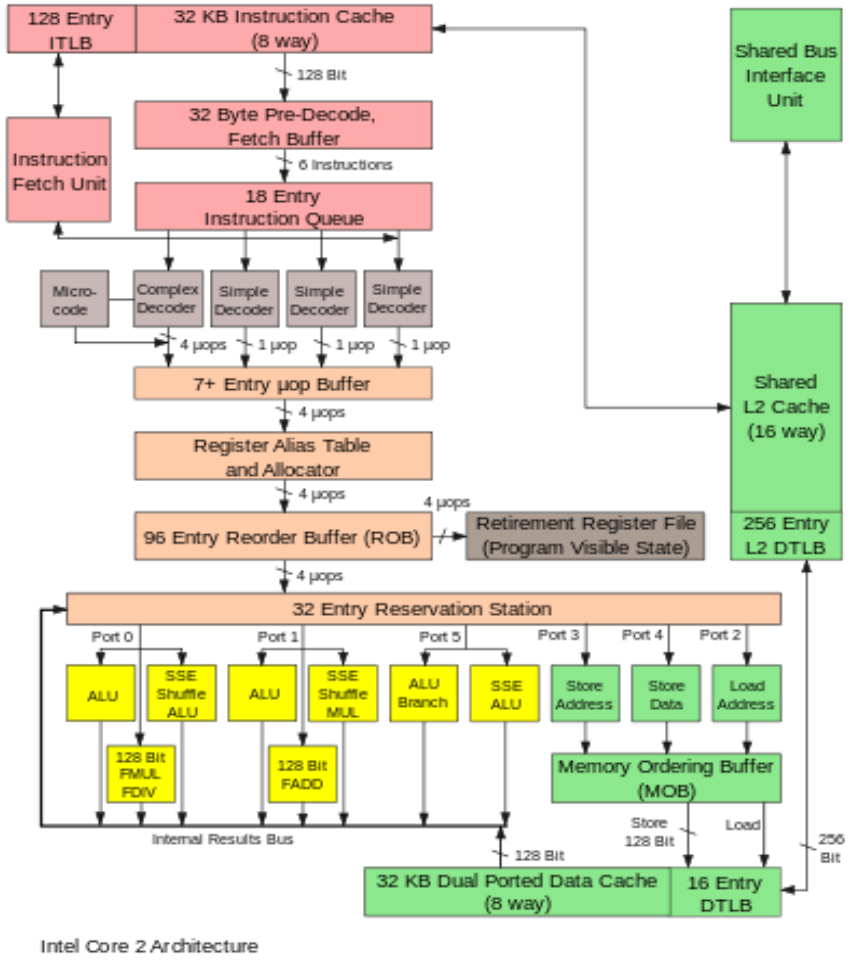



Intel Core 2 Architecture

# CPU (ILP): конфликты

- Hazards (конфликты)
  - Структурные
  - Управляющие
  - Зависимости по данным



Intel Core 2 Architecture

# CPU (ILP): примеры конфликтов

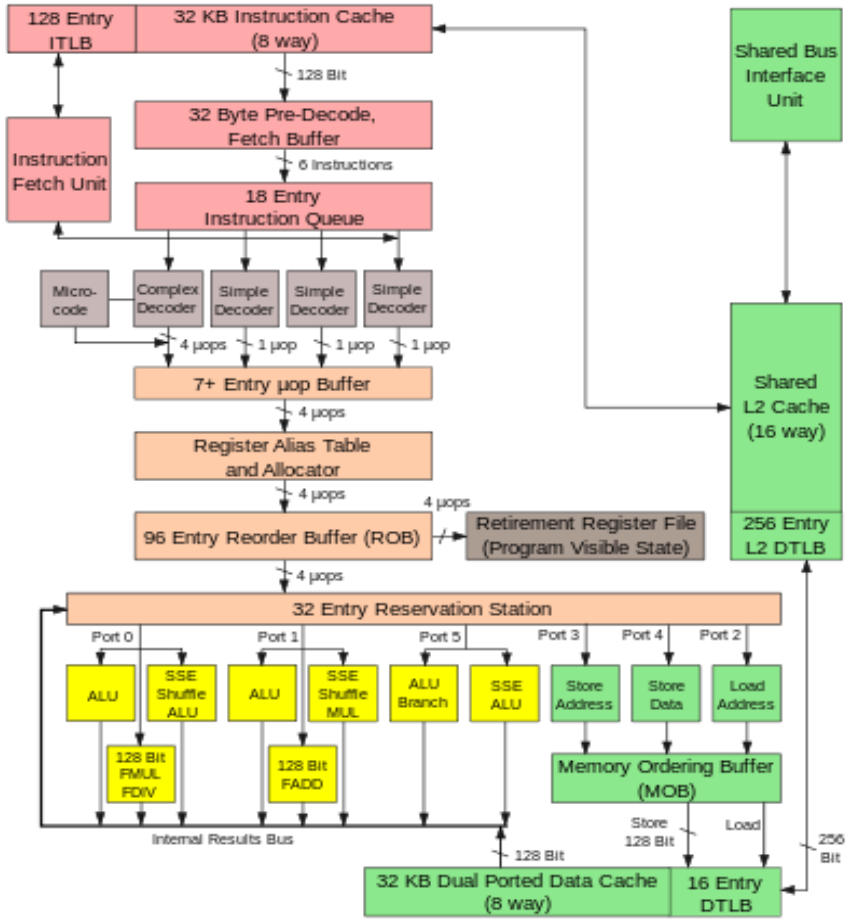
- Hazards (конфликты)
  - Структурные
  - Управляющие
  - Зависимости по данным
  - CPI =

    IdealCPI (0.2)

    + Struct Stalls (>1)

    + Control Stalls (~10-20)

    + Data Stalls (~1-30)




Intel Core 2 Architecture

# CPU (ILP): конфликты

- Control Hazard
  - Проблемы при передаче управления
  - Ещё не знаем, куда послали, чем заняться ближайшие 20 тактов?
- Ответ: спекулятивное исполнение
  - Предсказание переходов
  - Есть пределы: переход от 4K-entry branch predictor к бесконечно большому улучшает точность предсказания на 1-2%
- Ответ №2: качественная кодогенерация

# CPU (ILP): конфликты

- Data Hazard: зависимости по данным
  - RAW (Read After Write)
    - “true dependence” – не разрешимно
  - WAW (Write After Write)
    - эффективно обрабатываются переименовыванием регистров и т.п.
  - WAR (Write After Read)
    - эффективно обрабатываются переименовыванием регистров и т.п.

# CPU (ILP): сухой остаток

- Пределы ILP
  - Даже на идеальных моделях CPU и реальных приложениях выигрыш от ILP близок к фактическому
  - Перевод: смысла инвестировать в ILP больше нет
- Нельзя предсказывать “ленинским прищуром”
  - Например, считать ассемблерные инструкции

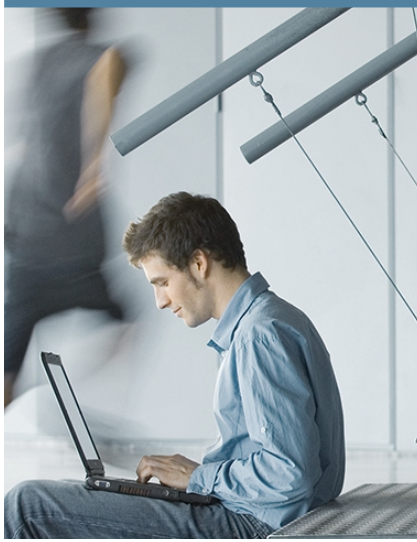
 

# CPU (ILP): что делать?

- Симптомы:
  - Высокий CPI без причины
  - Больше ниток – меньше скорость: кончились execution unit'ы
- Диагностика:
  - **vtune, solstudio analyzer, perf, oprofile**: процессорные счётчики
- Решения:
  - Берём компилятор по-круче
  - Берём JVM по-свежее
  - Перемешиваем задачи в разных потоках

# Программа



- Теория
- Практика
  - I/O
    - Network
    - Disk
  - Memory
    - Caches
    - Coherency
    - TLB
    - Memory & Buses
  - CPU
    - Multicore
    - Sharing
    - ILP
- **Q/A**


НЕСМОТРЯ НА ДЕТАЛЬНЫЙ АНАЛИЗ ТЕКУЩЕЙ СИТУАЦИИ, Я ТАК И НЕ СМОГ СОСТАВИТЬ ЧЁТКОЕ ПРЕДСТАВЛЕНИЕ ОБ ОБСУЖДАЕМОЙ ПРОБЛЕМЕ В СИЛУ ВОЗНИКШЕГО КОНГИТИВНОГО ДИССОНАНСА.

# Q/A


JavaOne™