

Лекция 2

Оптимизация работы с кеш-памятью процессора

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com

WWW: www.mkurnosov.net

Курс «Параллельные вычислительные технологии»

Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)

Организация подсистемы памяти (история)

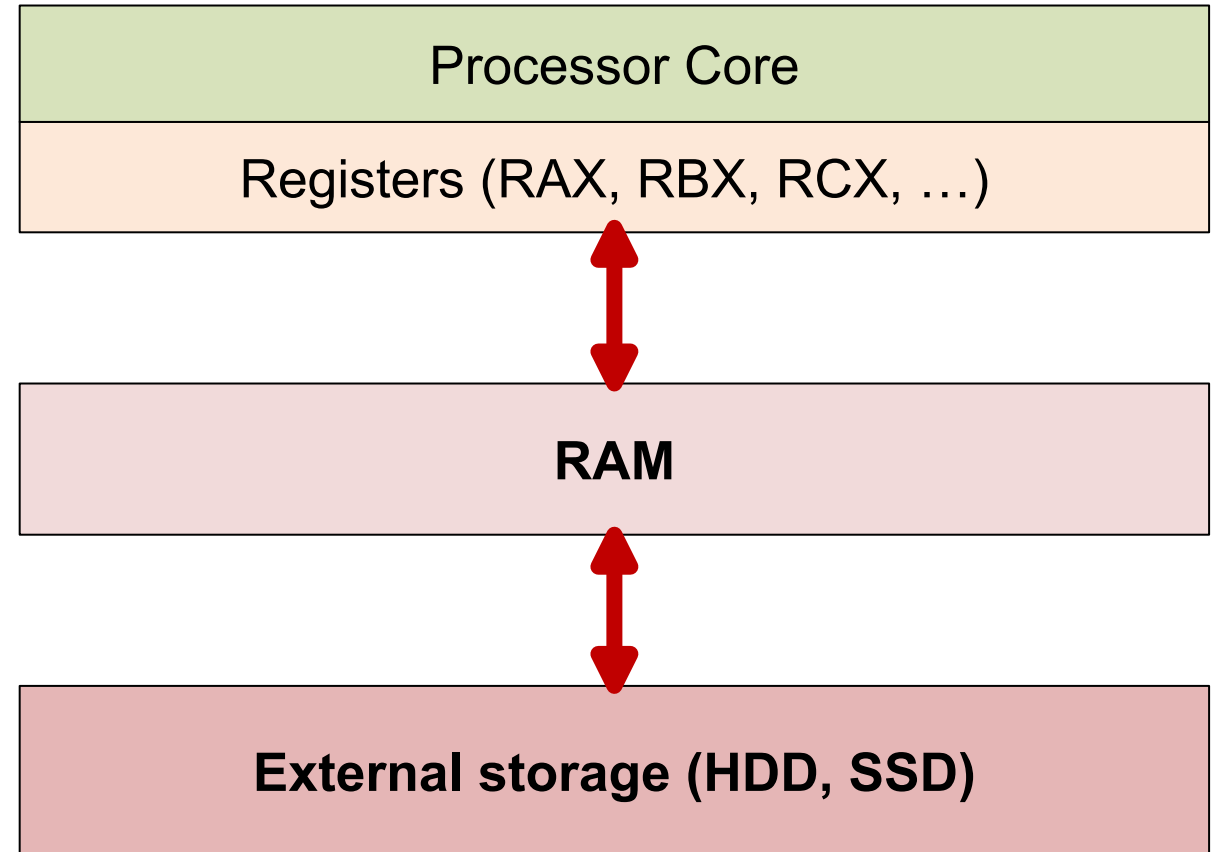
$v[i] = v[i] + 10$

```
addl $0xa, v(,%rax,4)
```

1. Load $v[i]$ from memory
2. Add 2 operands
3. Write result to memory

Время доступа к памяти (load/store)
для многих программ является
критически важным

*(memory bound application, memory
intensive application)*



Доступ к памяти

```
int a[100];

int sum = 0;
for (int i = 0; i < 100; i++) {
    sum += a[i];
}
```

```
$ gcc -o prog ./prog.c --save-temps
```

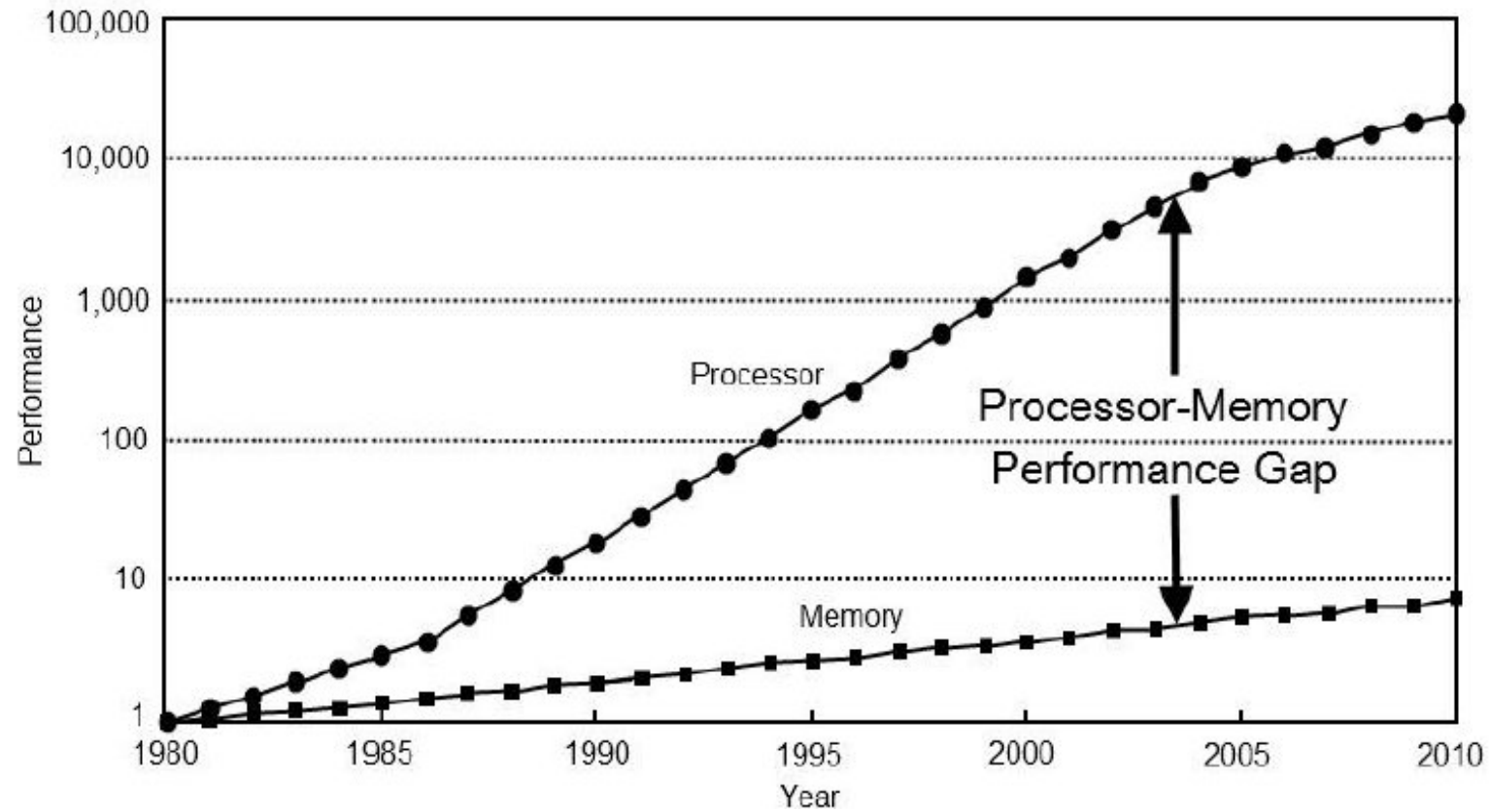


```
        movl    $0, -4(%rbp)           // sum = 0
        movl    $0, -8(%rbp)           // i = 0
        jmp     .L2

.L3:
        movl    -8(%rbp), %eax
        cltq
        movl    -416(%rbp,%rax,4), %eax // Convert Long To Quad
        addl    %eax, -4(%rbp)          // a[i] -> %eax
        addl    %eax, -4(%rbp)          // sum = sum + %eax
        addl    $1, -8(%rbp)           // i++

.L2:
        cmpl   $99, -8(%rbp)
        jle    .L3
```

Стена памяти (memory wall)

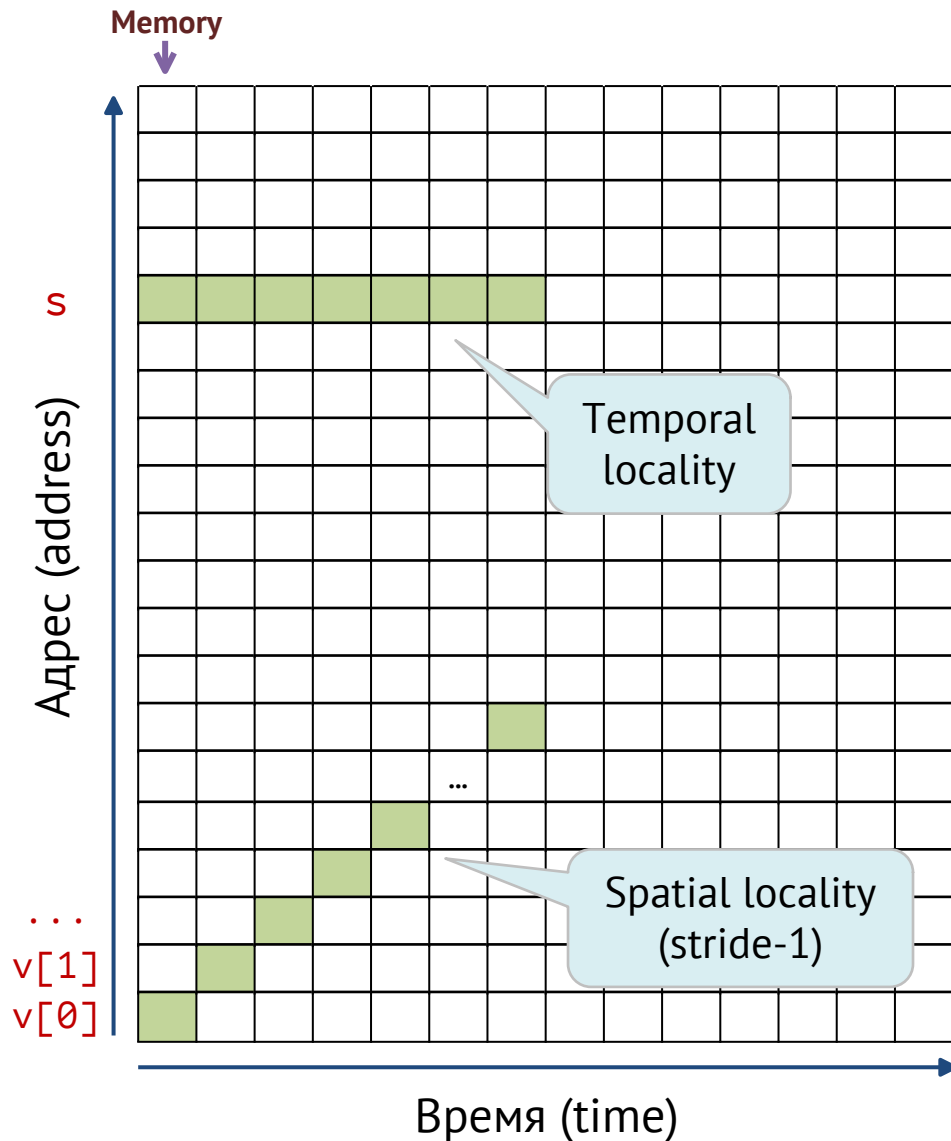


- С 1986 по 2000 гг. производительность процессоров увеличивалась ежегодно на **55%**, а производительность подсистемы памяти возрастала на **10%** в год

Пространственная и временная локальность ссылок

- **Локальность ссылок** (locality of reference)
- **Пространственная локальность** (spatial locality) – свойство программ обращаться по адресам, находящимся рядом с текущим адресом (к инструкциям и данным)
- **Временная локальность** (temporal locality) – свойство программ повторно обращаться к одному набору адресов через короткий промежуток времени
 - *temporal data* – область памяти (данные), которая будет востребована в ближайшем будущем
 - *non-temporal data* – данные после модификации не будут востребованы длительное время

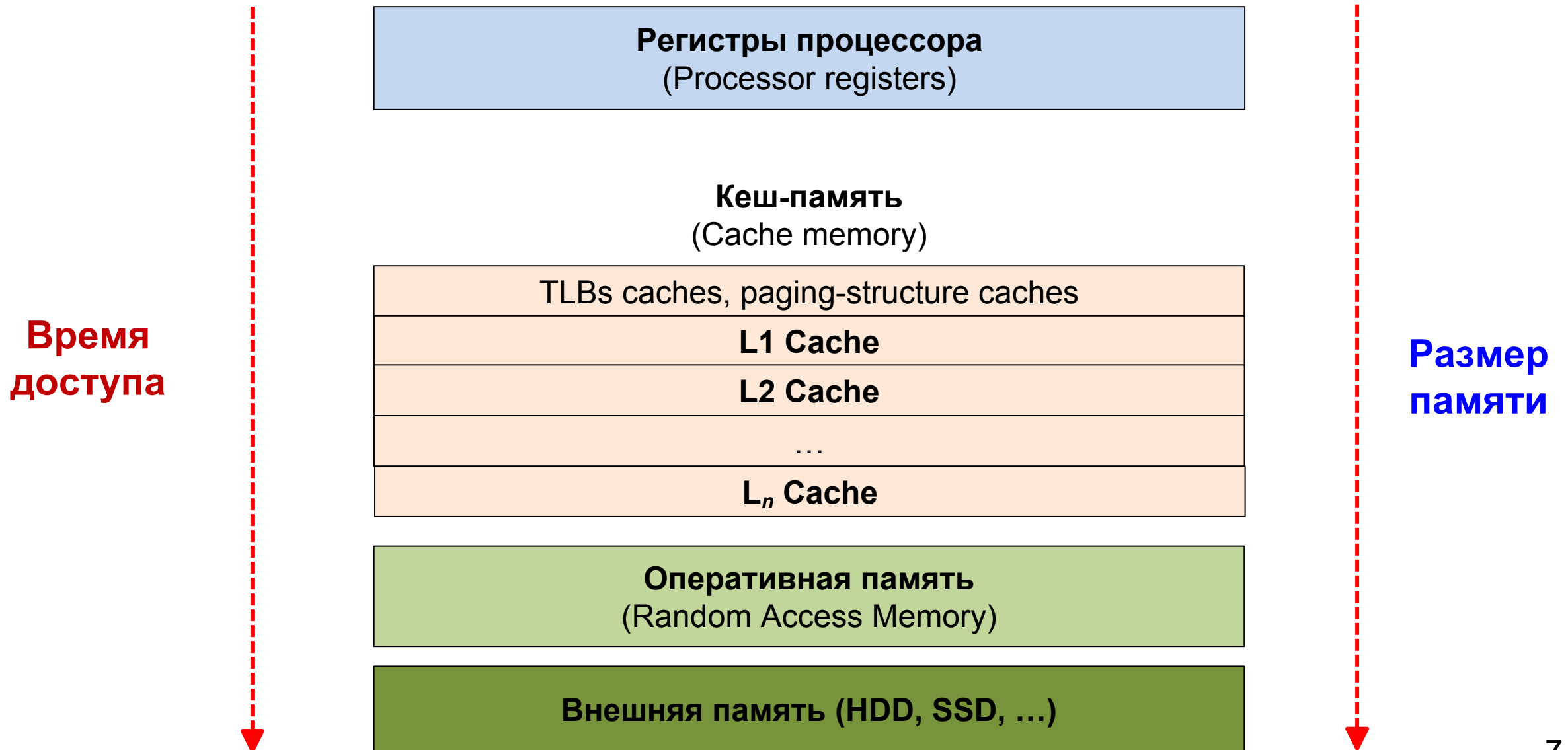
Пространственная и временная локальность ссылок



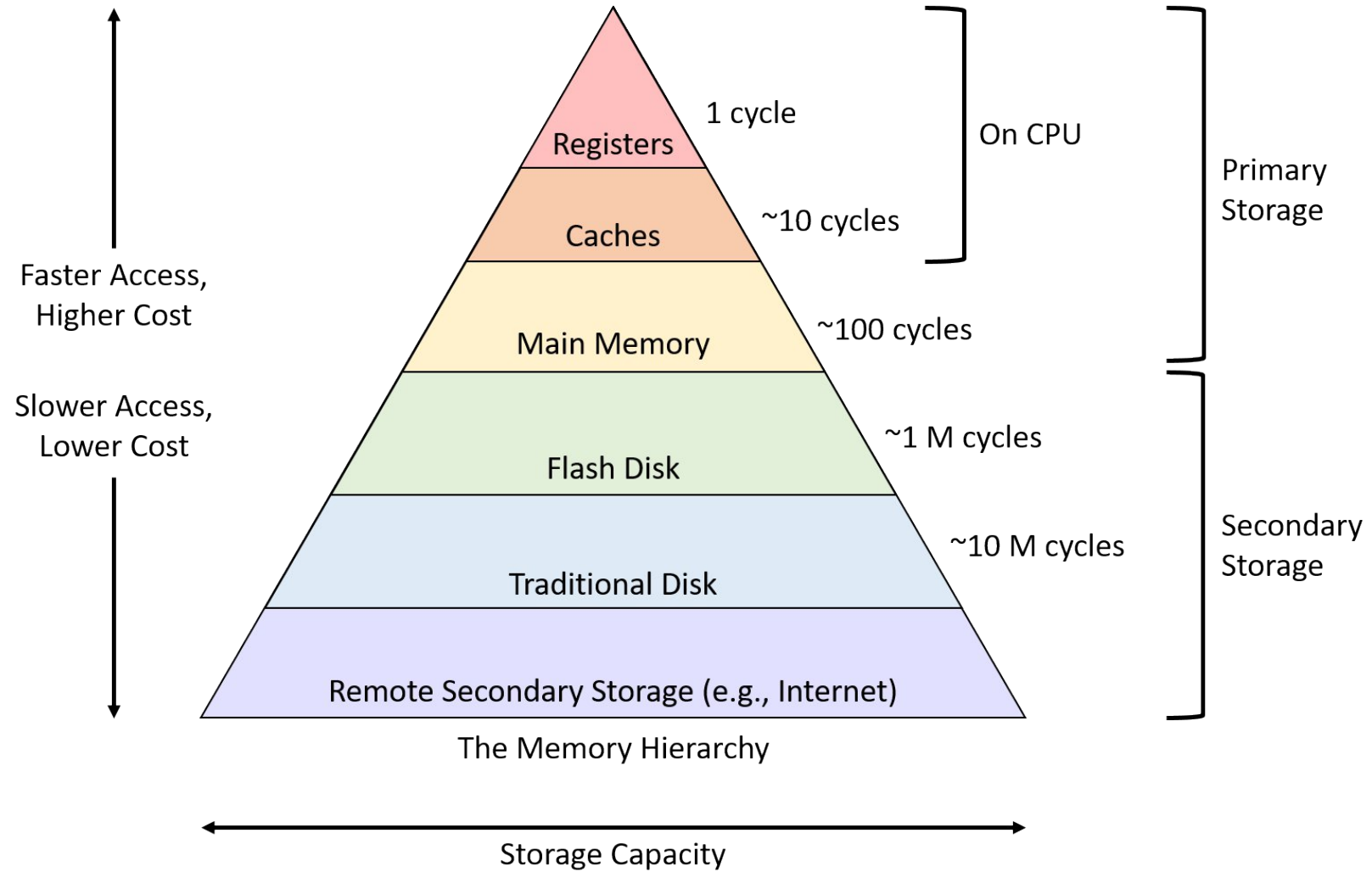
```
flag = 1;  
for (int i = 0; i < n; i++) {  
    s = s + v[i];  
}
```

```
// s - temporal locality (temporal data)  
// v[] - spatial locality (stride-1)  
// flag - non-temporal data
```

Иерархическая организация памяти



Иерархия памяти

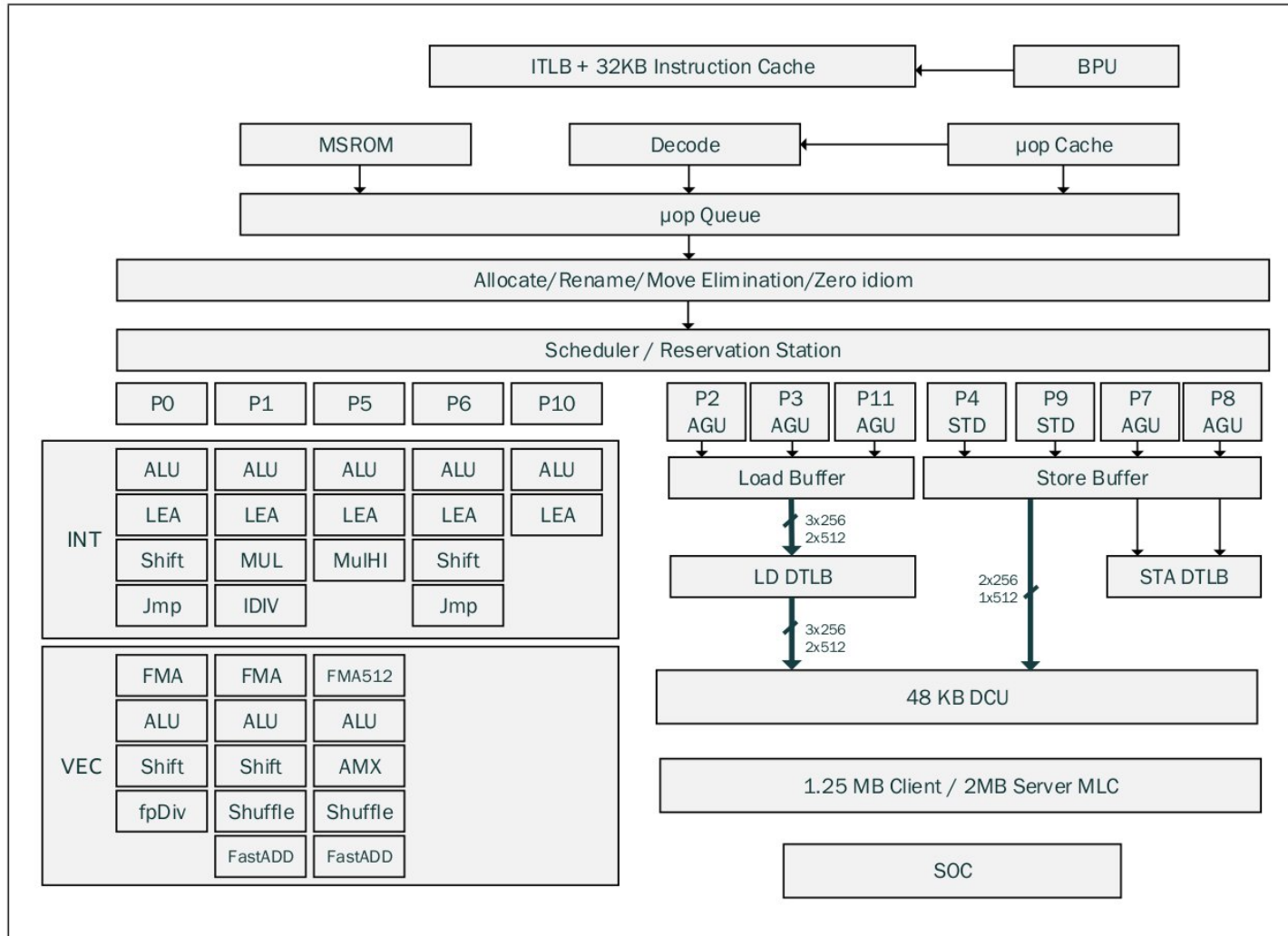


Стена памяти (memory wall)

Как минимизировать латентность (задержку) доступа к памяти?

- **Внеочередное выполнение** (out-of-order execution) – динамическая выдача инструкций на выполнение по готовности их данных
- **Вычислительный конвейер** (pipeline) – совмещение (overlap) во времени выполнения инструкций
- **Суперскалярное выполнение** (superscalar) – выдача и выполнение нескольких инструкций за такт ($CPI < 1$)
- **Одновременная многопоточность**
(simultaneous multithreading, hyper-threading)

Intel 64 (Golden Cove uarch, 2021)



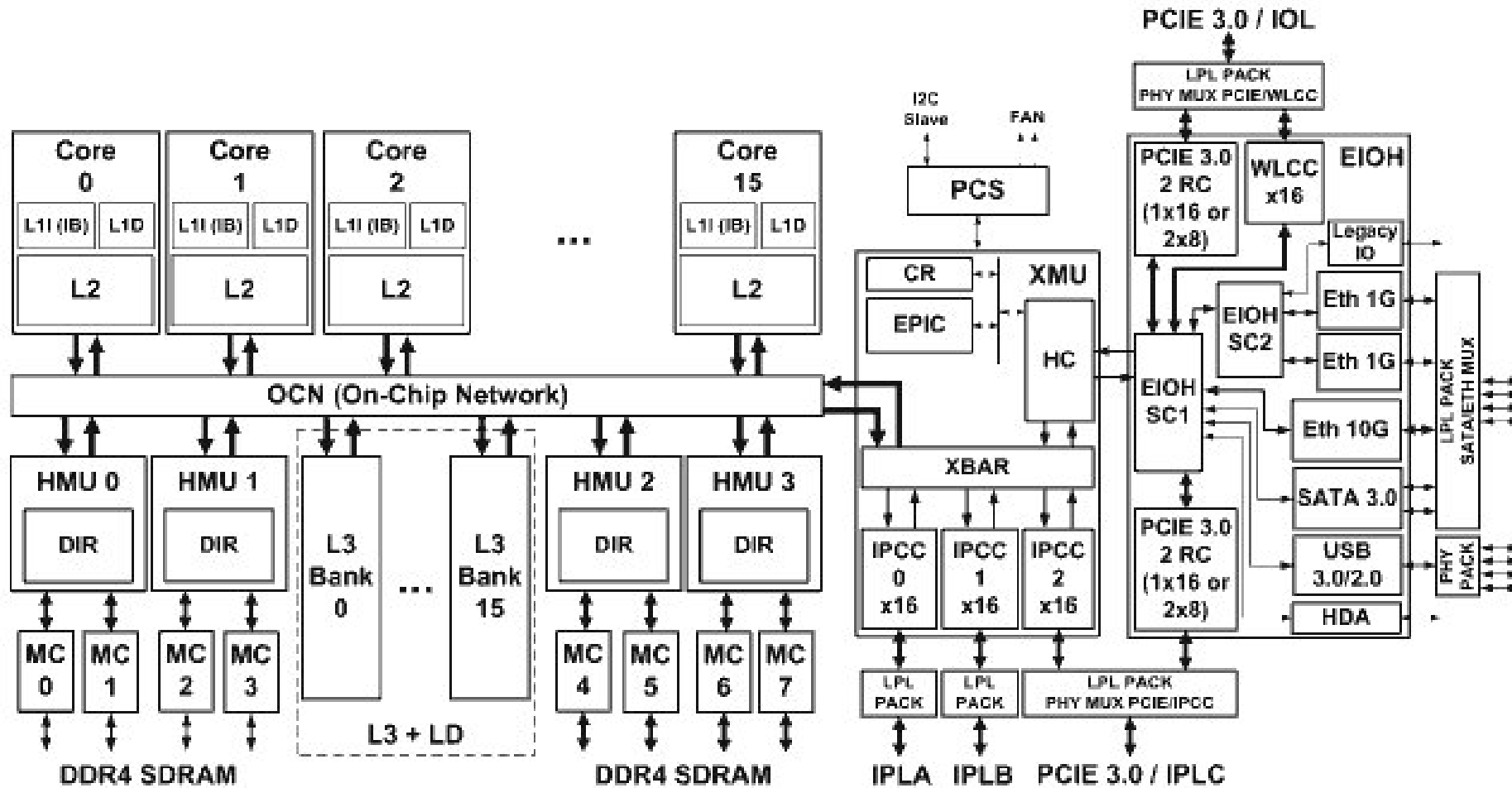
- Caches
- TLB (iTLB, dTLB), STLB (second-level TLB)
- Store buffers
- Write Combining (WC) buffer

L1 cache (DCU – Data Cache Unit)

L2 cache (mid-level cache, MLC)

L3 (last level cache, LLC; uncore)

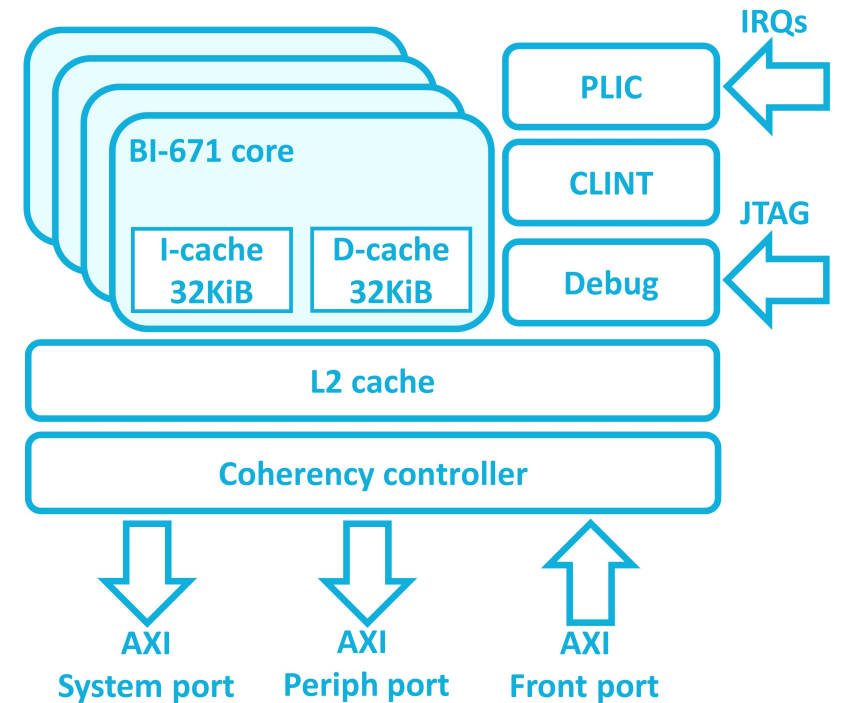
Эльбрус-16С



- L1: 64 Кбайт данные + 128 Кбайт команды в каждом ядре
- L2: 1 Мбайт в каждом ядре, 16 Мбайт суммарно
- L3: 32 Мбайт в процессоре

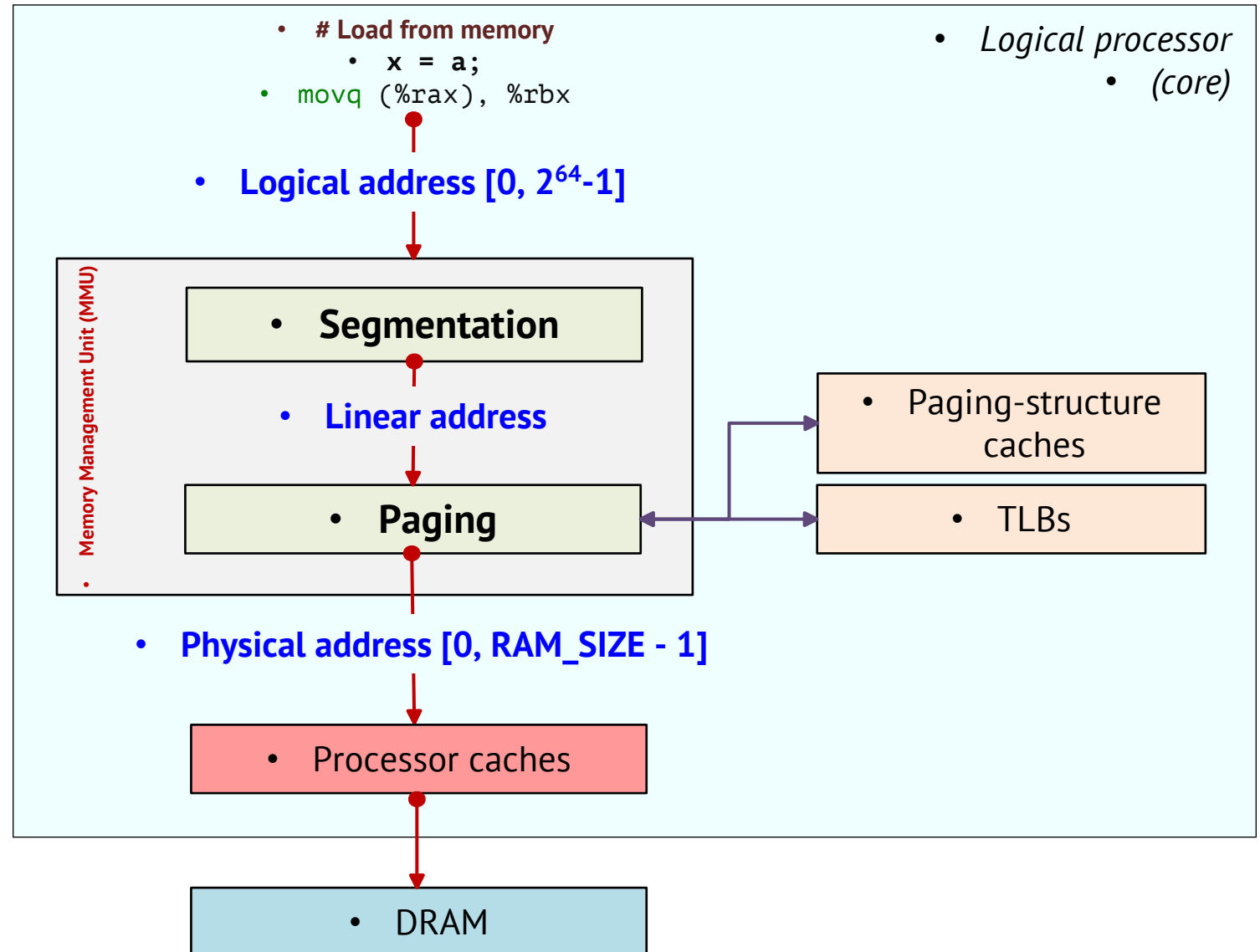
CloudBear BI-671 (64 разрядное ядро RISC-V)

- 9 стадийный конвейер с внеочередным выполнением команд
- 4-32КБ, 2-8-канальный L1 кэш инструкций
- 4-32КБ, 2-8-канальный L1 кэш данных
- интегрированный 128КБ-2МБ L2 кэш
- предварительная загрузка данных в L2 кэш с определением шага запросов (stride prefetcher)
- поддержка Linux



Страничная организация памяти

- Линейный адрес $[0, 2^{64}-1]$
- Физический адрес $[0, \text{RAM_SIZE} - 1]$
- Как линейные адреса отобразить в физические?
 $2^{64} > \text{RAM } 16 \text{ GiB}$
- Страница памяти 4 KiB

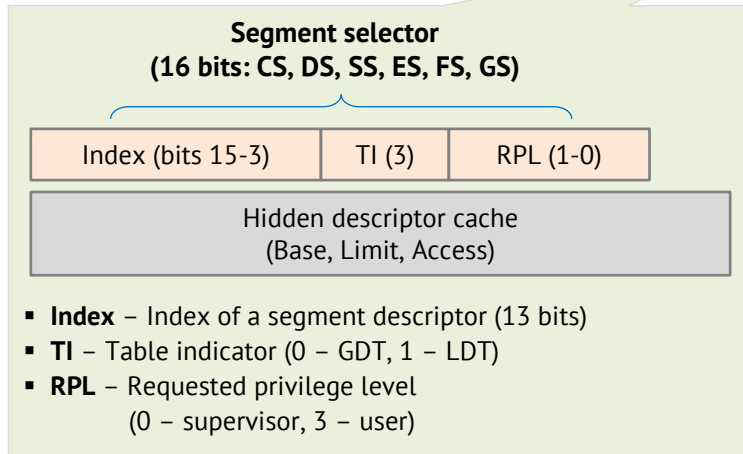


Translation: Logical address (SegSel : Offset) --> Linear address (48 bits)

(Vol. 3A Intel 64 and IA-32 Architectures Software Developer's Manual, Chapter 3 Protected Mode Memory Management)

x = a;
Load from memory
movq (%rax), %rbx

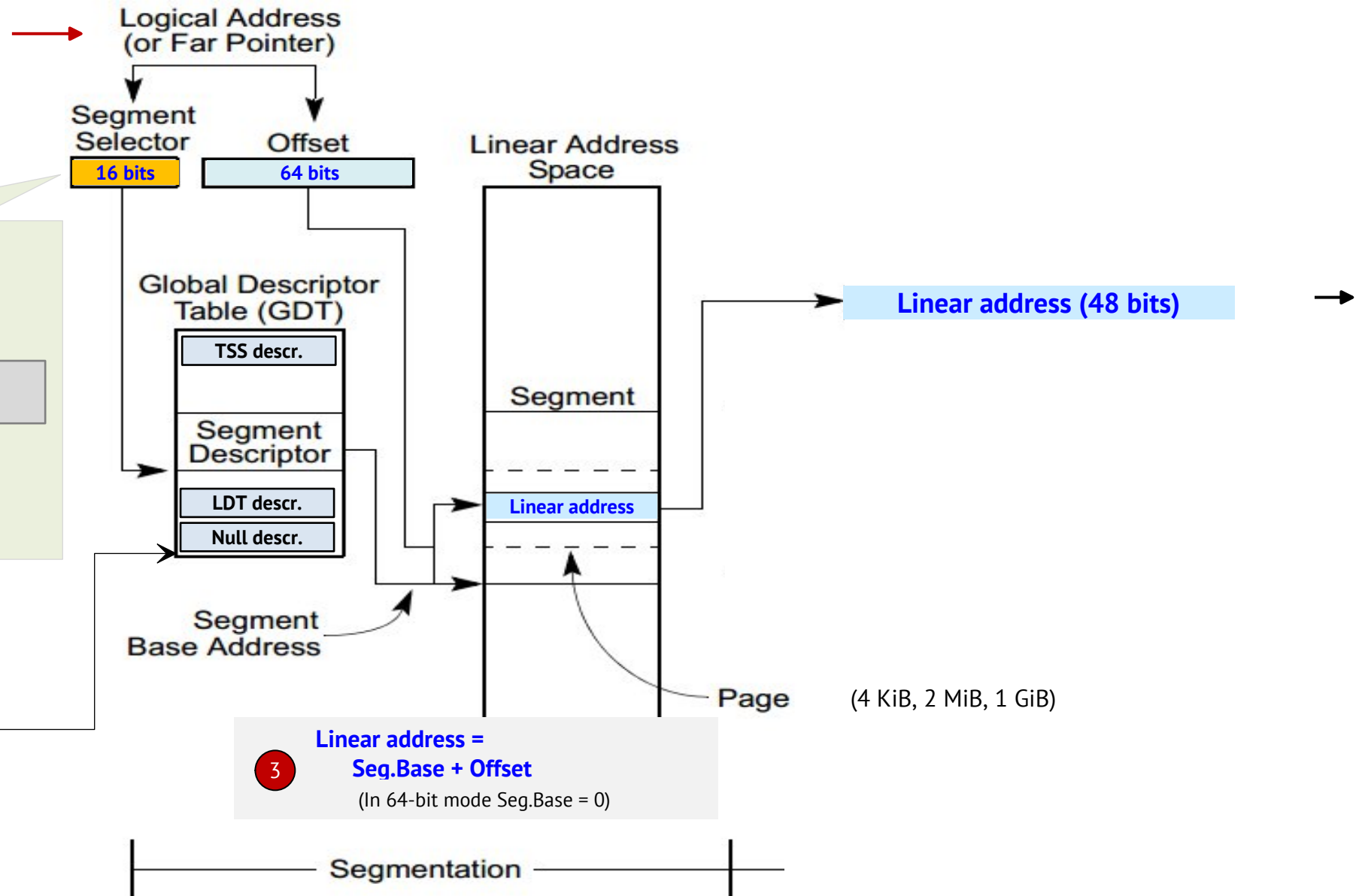
1 Select descriptor table
(TI = 0 ? GTD : LDT)



2 Read segment descriptor Check rights access and limits

GDTR
80 bits: Base (64 bits), Limit (16 bits)

LDTR
Seg. Sel. (16), Base (64), Limit (16), Descriptor attributes

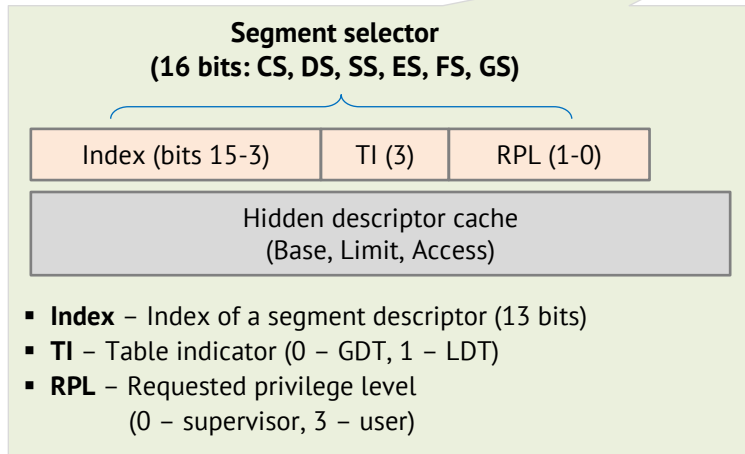


Translation: Logical address (SegSel : Offset) --> Linear address (48 bits)

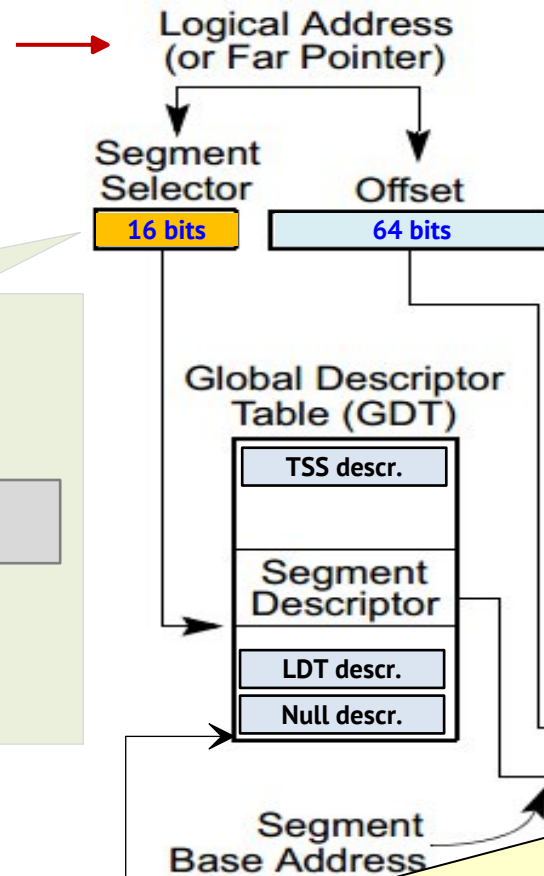
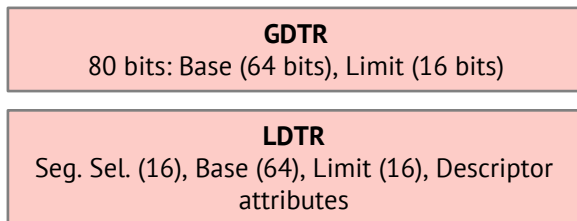
(Vol. 3A Intel 64 and IA-32 Architectures Software Developer's Manual, Chapter 3 Protected Mode Memory Management)

x = a;
Load from memory
movq (%rax), %rbx

1 Select descriptor table
(TI = 0 ? GTD : LDT)



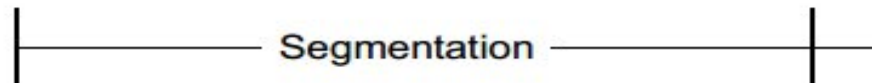
2 Read segment descriptor Check rights access and limits



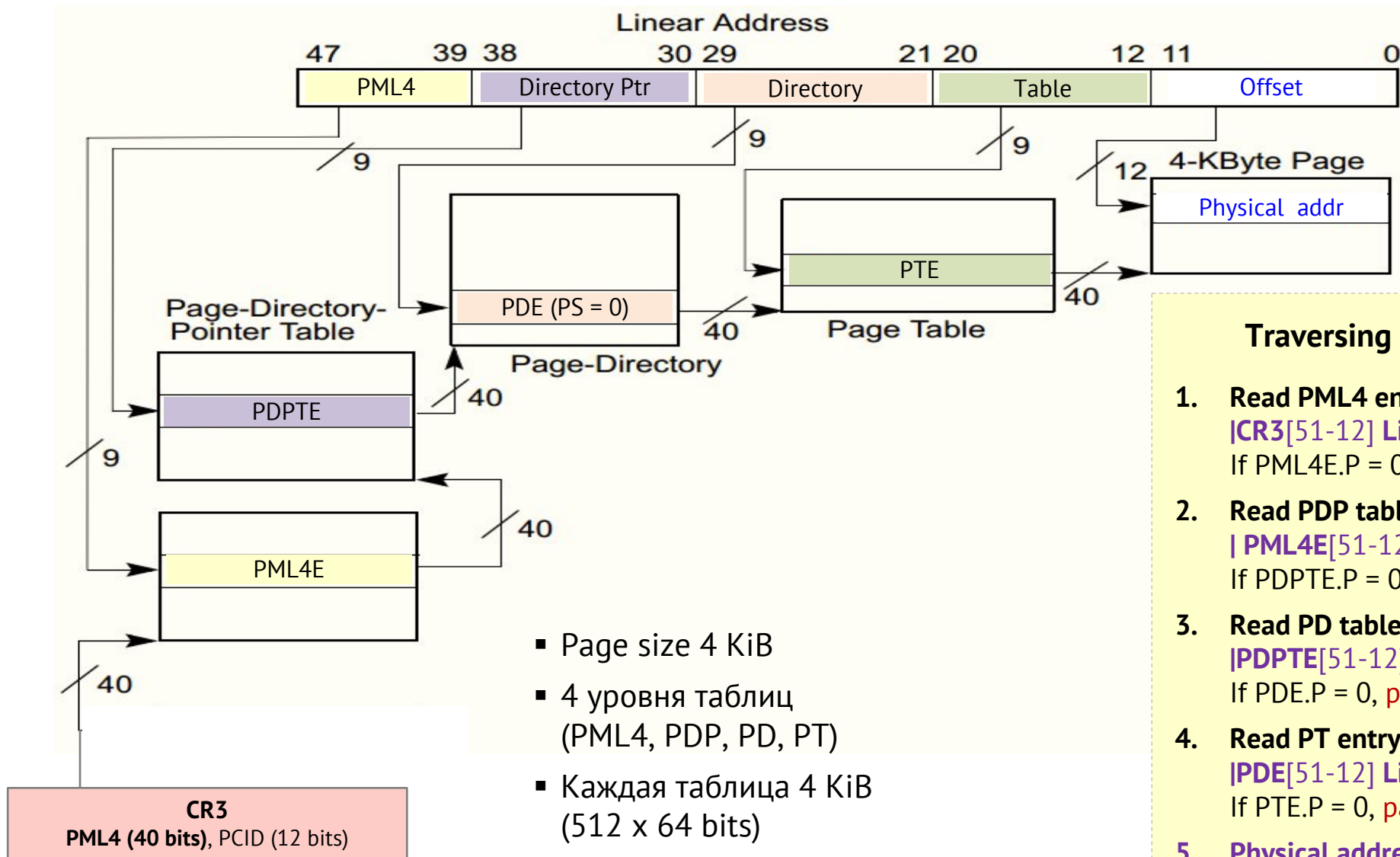
- **Average/best case**
Поля дескриптора сегмента читаются из скрытого регистра (Seg.Base, ...)
- **Worst case**
Дескриптор сегмента загружается из оперативной памяти в скрытый регистр (требуется трансляция адреса)
- **В 64-битном режиме сегментация не используется (Seg.Base = 0)**

- Таблицы дескрипторов сегментов (GDT и LDT) хранятся в оперативной памяти
- Трансляция адреса GDTR/LDTR осуществляется при установке сегментного регистра и загрузке соответствующего ему дескриптора в скрытый регистр

3 Linear address =
Seg.Base + Offset
(In 64-bit mode Seg.Base = 0)



Translation: Linear address (48 bits) --> Physical address (52 bits)

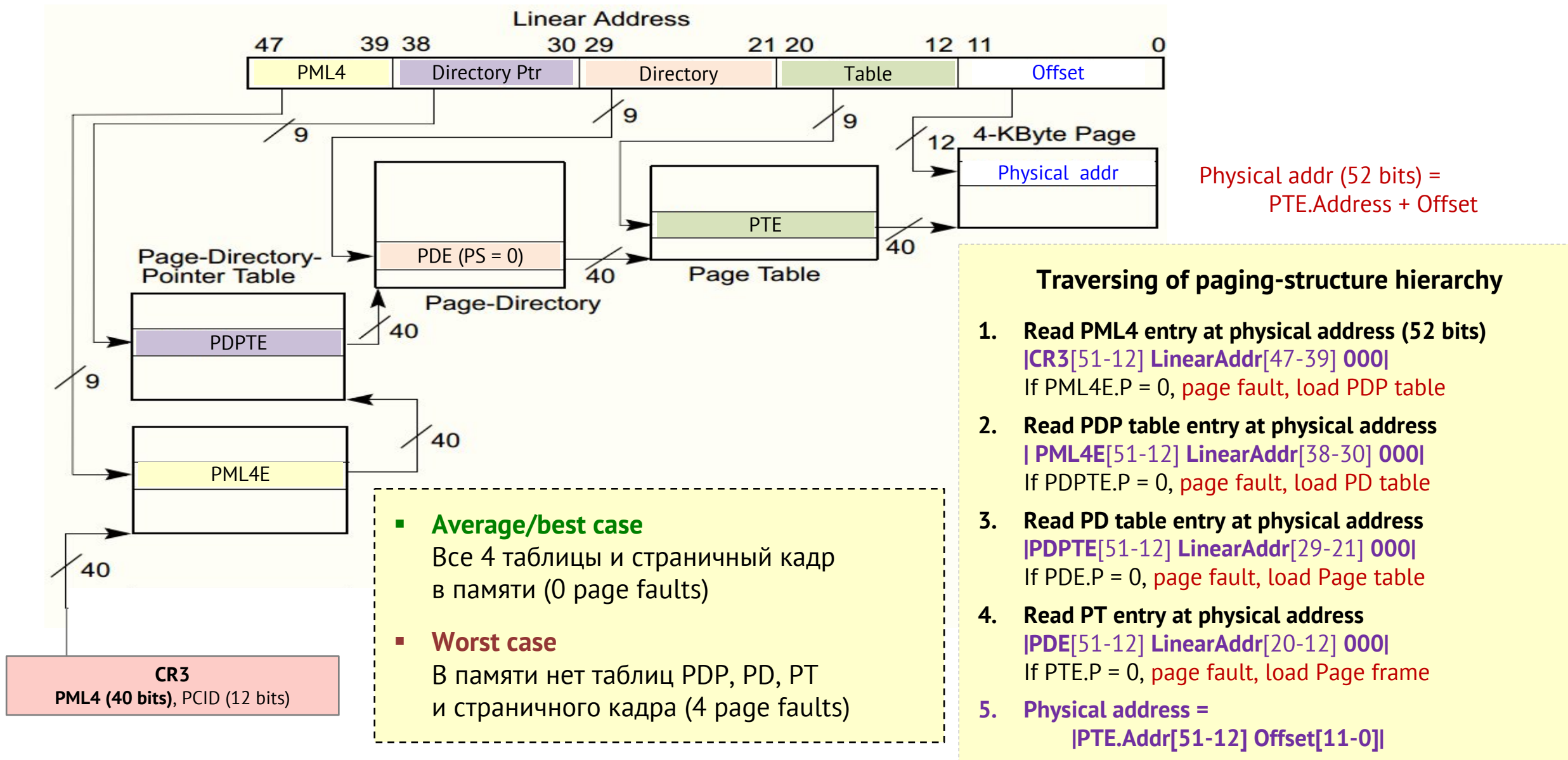


- Page size 4 KiB
- 4 уровня таблиц (PML4, PDP, PD, PT)
- Каждая таблица 4 KiB (512 x 64 bits)

Traversing of paging-structure hierarchy

1. Read PML4 entry at physical address (52 bits)
 $[CR3[51-12] \text{ LinearAddr}[47-39] \text{ 000}]$
 If PML4E.P = 0, page fault, load PDP table
2. Read PDP table entry at physical address
 $[PML4E[51-12] \text{ LinearAddr}[38-30] \text{ 000}]$
 If PDPTE.P = 0, page fault, load PD table
3. Read PD table entry at physical address
 $[PDPTE[51-12] \text{ LinearAddr}[29-21] \text{ 000}]$
 If PDE.P = 0, page fault, load Page table
4. Read PT entry at physical address
 $[PDE[51-12] \text{ LinearAddr}[20-12] \text{ 000}]$
 If PTE.P = 0, page fault, load Page frame
5. Physical address =
 $[PTE.Addr[51-12] \text{ Offset}[11-0]]$

Translation: Linear address (48 bits) --> Physical address (52 bits)



GNU/Linux

- **Minor page fault** – физическая страница присутствует в оперативной памяти, но не отображена в адресное пространство процесса (обращение к разделяемой библиотеке, инициализация памяти)
- **Major page fault** – физическая страница отсутствует в оперативной памяти, выполняется загрузка из файла подкачки (swap) на внешнем хранилище

```
$ /bin/time ls /
bin boot cdrom dev etc home lib lib32 lib64 libx32 lost+found media mnt opt proc      root  run  sbin
snap srv  swapfile sys  tmp usr  var

0.00user 0.00system 0:00.00elapsed 100%CPU (0avgtext+0avgdata 2512maxresident)k
0inputs+0outputs (0major+114minor)pagefaults 0swaps
```

```
$ ps -eo min_flt,maj_flt,cmd
MINFL MAJFL CMD
39261  165 /sbin/init splash
      0    0 [kthreadd]
      0    0 [rcu_gp]
133930  18  /opt/yandex/browser-beta/yandex_browser --type=renderer ...
```

```
$ perf stat -e minor-faults,major-faults ls /
bin boot cdrom dev etc home lib lib32 lib64 libx32 lost+found media mnt opt proc      root  run  sbin
snap srv  swapfile sys  tmp usr  var

Performance counter stats for 'ls /':
      94      minor-faults
       0      major-faults
```

Структурная организация кеш-памяти

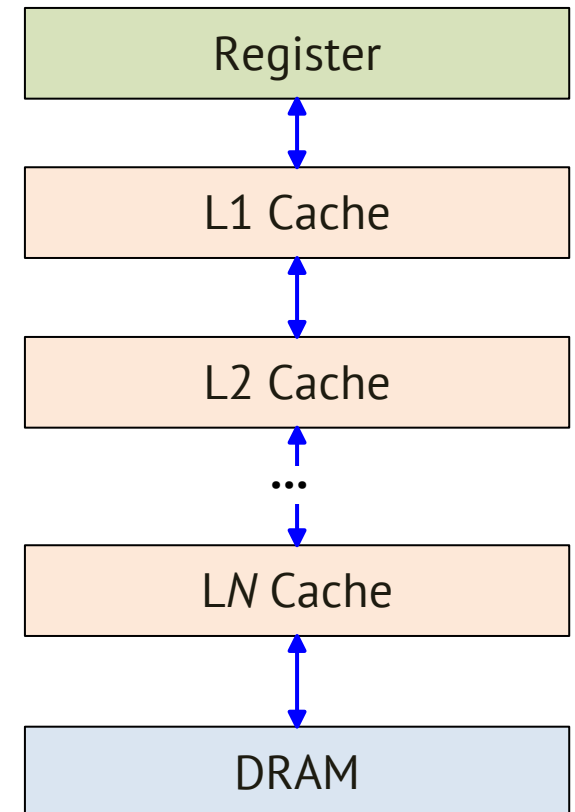
- **Размер кеш-памяти** (количество строк/блоков с данными)
- **Функция отображения** (mapping) — алгоритм сопоставления физическому адресу записи в кеш-памяти
- **Алгоритм замещения** строк (replacement policy) в случае нехватки места в кеш-памяти
- **Политика записи данных** к кеш-память — кеширование данных или немедленная запись в системную память
- **Обеспечения согласованного состояния** данных в кеш-памяти ядер процессоров и основной памяти (протоколы обеспечения когерентности кеш-памяти MESI, MESIF)

Поиск записи в кеш-памяти процессора

Physical address



- **Поиск записи в L1:** нашли запись (L1 cache hit) – возвращаем данные
- **[L1 cache miss]** L1 кеш обращается в L2 и замещает полученной строкой одну из своих записей (replacement), данные передаются ниже
- **Поиск записи в L2:** нашли запись (L2 cache hit) – возвращаем запись в L1
- **[L2 cache miss]** L2 кеш обращается в L3 и замещает полученной строкой одну из своих записей (replacement), запись передается в L1
- ...
- **Поиск записи в LN:** нашли запись (LN cache hit) – возвращаем запись в L{N-1}
- **[L{N-1} cache miss]** LN кеш обращается в DRAM и замещает одну из своих строк (replacement), запись передается в L{N-1}

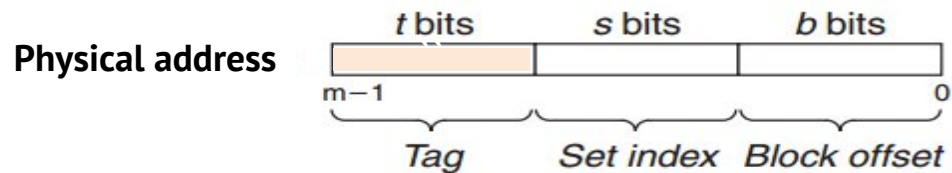


Чтение данных

```
if <Блок с адресом ADDR в кеш-памяти> then  
  /* Cache hit */  
  <Вернуть значение из кеш-памяти>  
else  
  /* Cache miss */  
  <Загрузить блок данных из кеш-памяти следующего уровня (либо DRAM)>  
  <Разместить загруженный блок в одной из строк кеш-памяти (вытеснить строку)>  
  <Вернуть значение из кеш-памяти (загруженное)>  
end if
```

- Из основной памяти в кеш загружается строка (64B), даже если в инструкции обращение к 4 байтам
- Что делать если кеш-память заполнена, нет свободных строк во множестве, которое соответствует адресу?

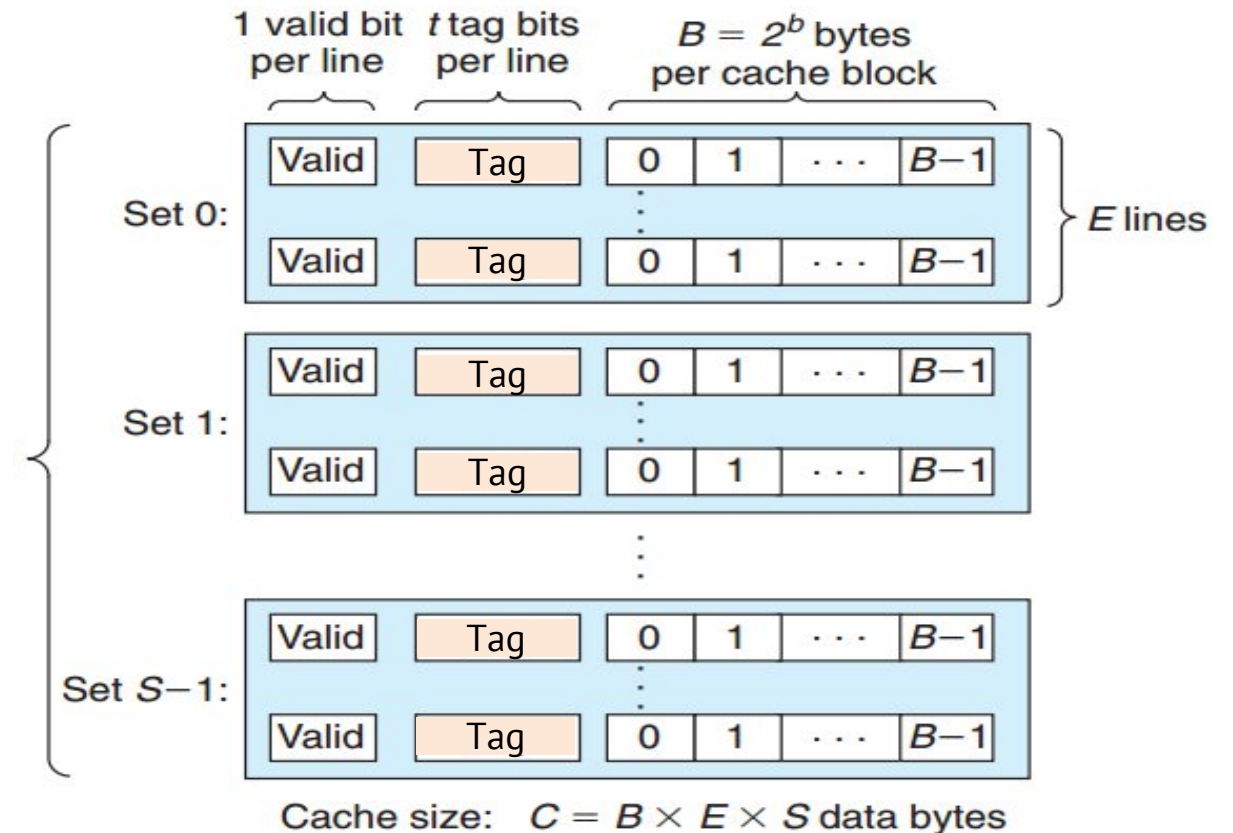
Структурная организация кеш-памяти



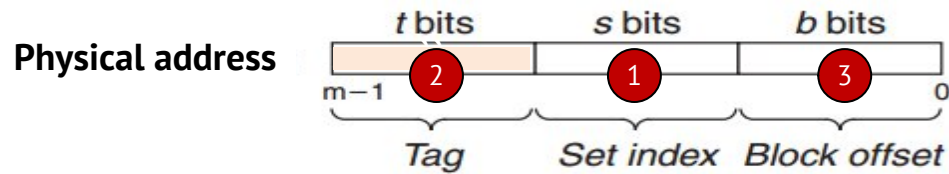
- Кеш содержит $S = 2^s$ множеств (**sets**)
- Каждое множество содержит E строк (**cache lines**)
- Каждая строка содержит блок данных ($B = 2^b$ байт) и метаданные – поля *valid bit*, *tag* (t бит)
- Размер кеш-памяти (данных)

$$C = S * E * B$$

Множественно-ассоциативная кеш-память

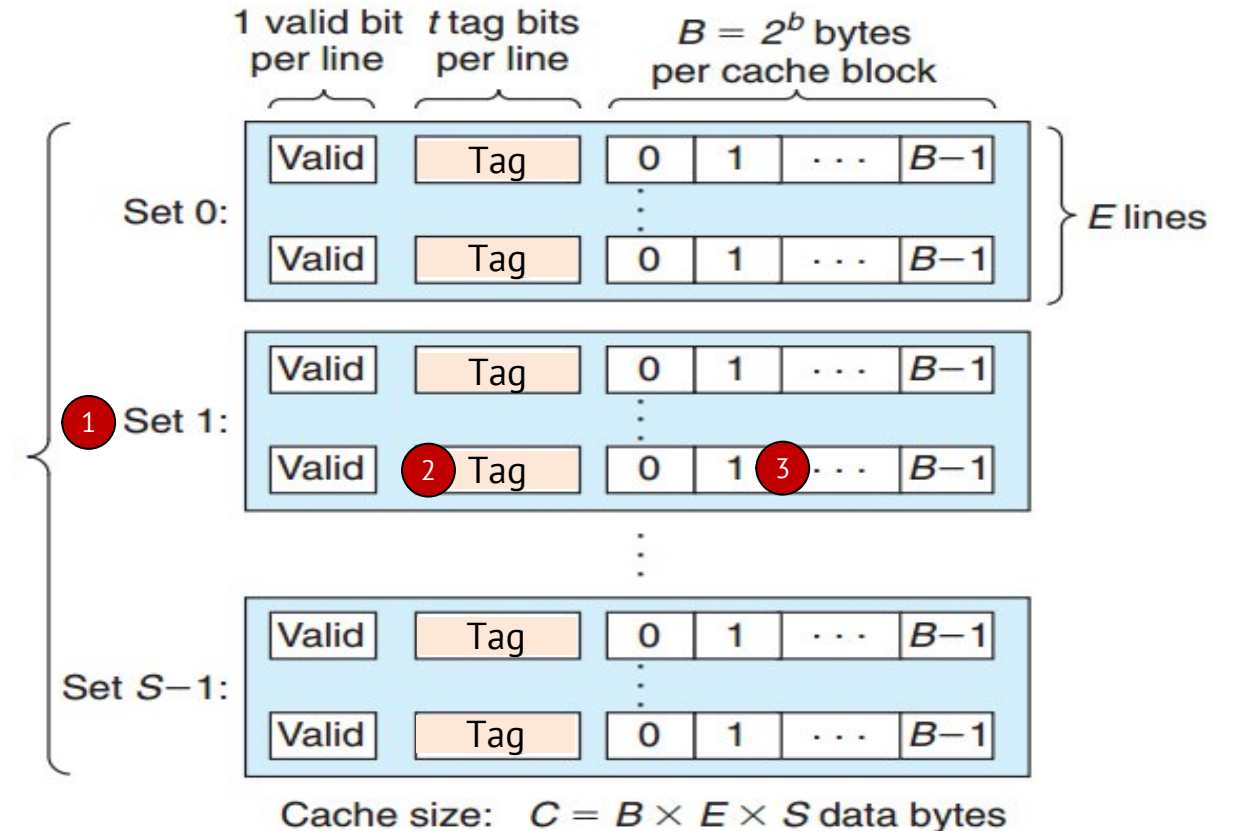


Поиск записи в кеш-памяти с множественно-ассоциативной функцией отображения



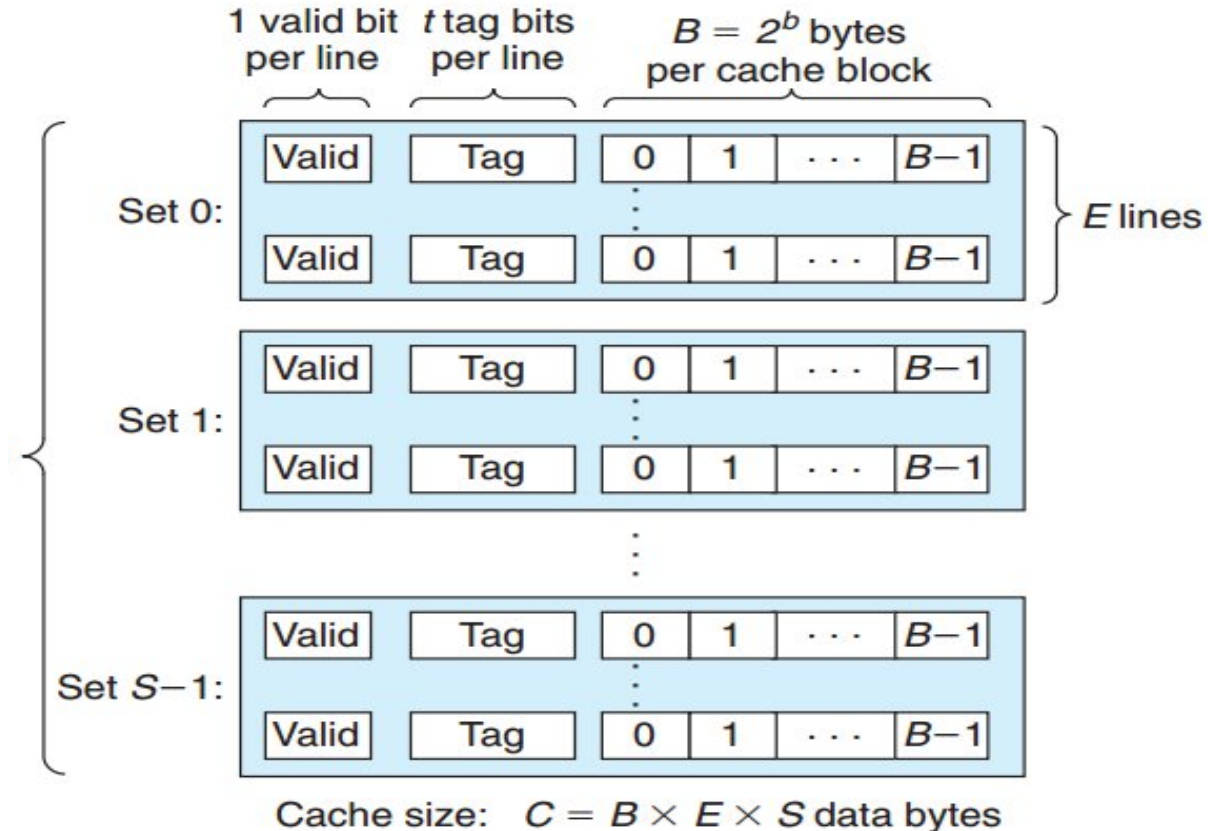
1. По полю *Set index* выбирается одно из S множеств
2. Среди E записей множества отыскивается строка с требуемым полем *Tag* и установленным битом *Valid*
 найдена – **cache hit**
 не найдена – **cache miss**
3. Данные из блока считываются с заданным смещением *Block offset*

Множественно-ассоциативная кеш-память



Методы отображения адресов (mapping)

- **Прямое отображение** (direct mapping) – в каждом мужестве по одной записи ($E = 1$, поле Tag не требуется, только Set index)
- **Полностью ассоциативное отображение** (full associative mapping) – одно множество ($S = 1$, поле Set index не требуется, только Tag)
- **Множественно-ассоциативное отображение** (set-associative mapping)



Алгоритмы замещения записей кеш-памяти

- **Алгоритм замещения строк кеш-памяти** (replacement policy) – выбирает строку и удаляет из кеш-памяти для размещения новой записи
- Алгоритмы требуют хранения вместе с каждой строкой кеш-памяти специализированного поля флагов/истории (age bits)
- **LRU** (Least Recently Used) – вытесняется наименее востребованную строку (2Q, LRU/K)
- **RR** (Random Replacement) – вытесняет случайную строку
- **Алгоритм L. Belady** – вытесняет запись, которая с большой вероятностью не понадобится в будущем (IBM Research, 1966)

Записи данных

- **Политика write-through** (сквозная запись) – запись в кеш-память влечет за собой немедленное обновление данных в кеш-памяти и оперативной памяти (кеш “отключается”)
- **Политика write-back** (отложенная запись, sory-back) – первоначально данные записываются только в кеш-память
- **Строки нет в кеш-памяти** (write miss)
 - В кеш-памяти выделяется строка
 - Из оперативной памяти загружается строка, соответствующая адресу
 - Необходимые байты изменяются в загруженной строке кеш-памяти
 - Строка помечается как *модифицированная* (dirty)
- **Строка присутствует в кеш-памяти** (write hit)
 - Необходимые байты изменяются в строке кеш-памяти
 - Строка помечается как *модифицированная* (грязная, dirty)
- Запись в память модифицированных строк осуществляется при их замещении

Пример: 4-way set associative cache

- Рассмотрим 4-х канальный (4-way) множественно-ассоциативный L1-кеш размером 32 KiB с длиной строки 64 байт (cache line)
- В какой записи кеш-памяти будут размещены данные для физического адреса длиной 52 бит: `0x00000FFFFAB64`?
- Количество записей в кеш-памяти (cache lines): $32 \text{ KiB} / 64\text{B} = 512$
- Количество множеств (sets): $512 / 4 = 128$
- Каждое множество содержит 4 канала (4-ways, 4 lines per set)
- Поле смещения (offset): $\log_2(64) = 6$ бит
- Поле номер множества (set index): $\log_2(128) = 7$ бит
- Поле tag: $52 - 7 - 6 = 39$ бит

Tag: 39 бит	Set Index: 7 бит	Offset: 6 бит
-------------	------------------	---------------

Пример: 4-way set associative cache

Physical Address (52 bits):

Tag (39 bit)	Set Index (7 bit)	Offset (6 bit)
--------------	-------------------	----------------

Cache (4-way set associative):

Set 0		Tag0	Cache line (64 bytes)
		Tag1	Cache line (64 bytes)
		Tag2	Cache line (64 bytes)
		Tag3	Cache line (64 bytes)
Set 1		Tag0	Cache line (64 bytes)
		Tag1	Cache line (64 bytes)
		Tag2	Cache line (64 bytes)
		Tag3	Cache line (64 bytes)
...			
Set 127		Tag0	Cache line (64 bytes)
		Tag1	Cache line (64 bytes)
		Tag2	Cache line (64 bytes)
		Tag3	Cache line (64 bytes)

Пример: 4-way set associative cache

Address: $0x0000FFFFAB64 = 11111111111111111010101101100100_2$

Tag: 1111111111111111101	Set Index: $0101101 = 45_{10}$	Offset: $100100 = 36_{10}$
----------------------------	--------------------------------	----------------------------

Cache (4-way set associative):

...	
Set 45		Tag0	Cache line (64 bytes)
		Tag1	Cache line (64 bytes)
		1111111111111111101	[Start from byte 36, ...
		Tag3	Cache line (64 bytes)
...	

- Тег 1111111111111111101 может находиться в любом из 4 каналов множества 45

Обращение к элементу массива

- В программе имеется массив `int v[100]`
- Обратились к элементу `v[17]` по физическому адресу:

$$0x00000FFFAB64 = 11111111111111111111010101101100100_2$$

Tag: 1111111111111111111101	Set Index: $0101101 = 45_{10}$	Offset: $100100 = 36_{10}$
-------------------------------	--------------------------------	----------------------------

- В кеш-память будет загружен блок из 64 байт с начальным адресом:
 $0x00000FFFAB40 = 11111111111111111111010101101000000_2$
- В строке кеш-памяти будут размещены 16 элементов по 4 байта (`int`):
`v[8], v[9], v[10], v[11], ..., v[17], ..., v[23]`

Cache (4-way set associative):

Set 45		Tag0	Cache line (64 bytes)
		Tag1	Cache line (64 bytes)
		1111111111111111111101	<code>v[8], v[9], ..., v[17], ..., v[23]</code>
		Tag3	Cache line (64 bytes)

Чтение через границу строки кеш-памяти (Cache Line Split)

- Чтение 4 байт начиная с физического адреса

$0x0000FFFA64 = 111111111111111111101010110111110_2$

Tag: 1111111111111111101	Set Index: $0101101 = 45_{10}$	Offset: $111110 = 62_{10}$
----------------------------	--------------------------------	----------------------------

Cache (4-way set associative):

...	
Set 45		Tag0	Cache line (64 bytes)
		Tag1	Cache line (64 bytes)
		1111111111111111101	[0, 1, ..., 62, 63]
		Tag3	Cache line (64 bytes)
Set 46	
		1111111111111111101	[0, 1, ..., 63]

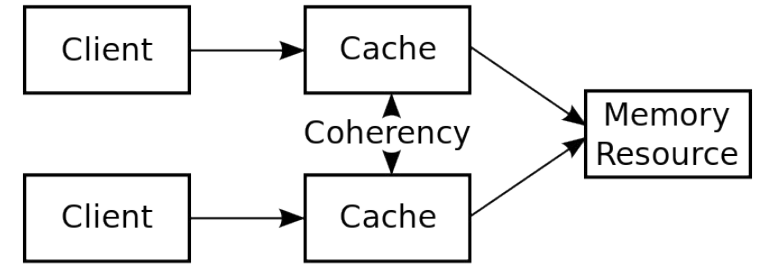
- 2 байта находятся в строке множества 45 (смещение 62)
- 2 байта в строке множества 46 (смещение 0)

Многоуровневая кеш-память

- Кеш-память уровня l называется **инклюзивной** (inclusive) по отношению к кеш-памяти уровня $l - 1$, если она включает все блоки содержащиеся в кеш-памяти уровня $l - 1$
- Кеш-память уровня l называется **экслюзивной** (exclusive, non-inclusive) по отношению к кеш-памяти уровня $l - 1$, если она содержит блоки не хранящиеся в кеш-памяти уровня $l - 1$
- **L2 инклюзивный для L1:** при промахе данные загружаются в L1 и L2
- **L2 экслюзивный для L1:** при промахе данные загружаются только в L1, при вытеснении из L1 строка перемещается в L2
- Инструкция `cruid()` // `$ cruid | grep -C10 inclu`
EDX Bit 01: Cache Inclusiveness
0 = Cache is not inclusive of lower cache levels
1 = Cache is inclusive of lower cache levels

Протокол обеспечения когерентности кеш-памяти

- **Когерентность кеш-памяти** (cache coherence) – свойство кеш-памяти, означающее согласованность данных, хранящихся в локальных кешах, с данными в оперативной памяти



- **Протокол MESI**

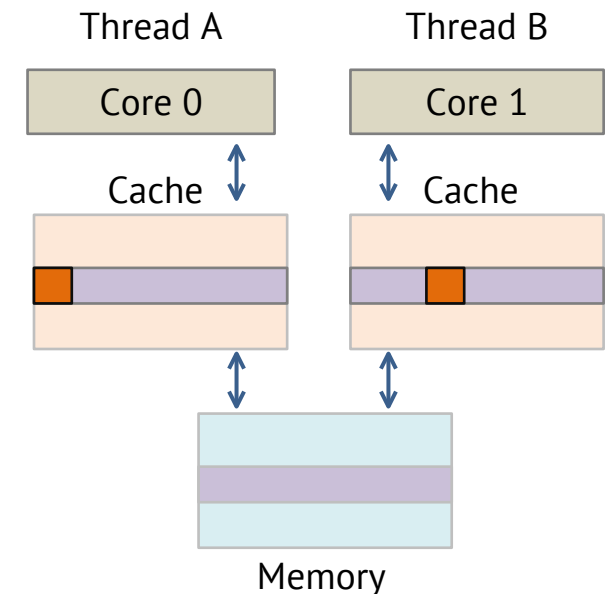
- Каждая строка кеш-памяти содержит 2 флага состояния MESI
- Отслеживаемый объект – строка кеш-памяти

Table 11-4. MESI Cache Line States

Cache Line State	M (Modified)	E (Exclusive)	S (Shared)	I (Invalid)
This cache line is valid?	Yes	Yes	Yes	No
The memory copy is...	Out of date	Valid	Valid	—
Copies exist in caches of other processors?	No	No	Maybe	Maybe
A write to this line ...	Does not go to the system bus.	Does not go to the system bus.	Causes the processor to gain exclusive ownership of the line.	Goes directly to the system bus.

Ложное разделение данных (false sharing)

- **Ложное разделение данных – потоки одновременно выполняют чтение/запись в пределах одной строки кеш-памяти, но в непересекающиеся участки**
 - Потоки одновременно записывают в одну строку кеш-памяти
 - Один поток записывает в строку, остальные читают
 - Один поток записывает, в ядрах остальных потоков включилась аппаратная предвыборка (prefetcher)
- **Потоки непрерывно меняют MESI-состояние строки**
 - Поток-писатель: переводит свою строку в состояние MODIFIED, у остальных в состояние INVALID
 - Поток-читатель получает копию из кеш-памяти LLC, кеш-памяти другого потока или основной памяти

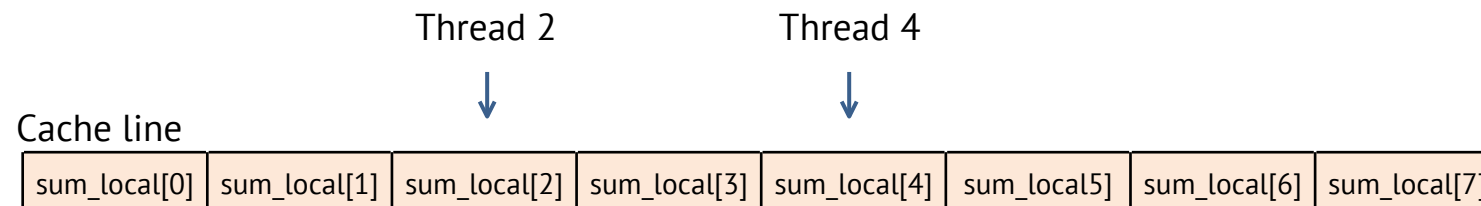


Ложное разделение данных (false sharing)

```
double sum = 0.0, sum_local[omp_get_max_threads()];
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    sum_local[tid] = 0.0;

    #pragma omp for nowait
    for (int i = 0; i < N; i++) {
        sum_local[tid] += x[i] * y[i];
    }

    #pragma omp atomic
    sum += sum_local[tid];
}
```



Устранение ложного разделения данных (false sharing)

```
struct tparam {  
    double sum;  
    uint8_t padding[64 - sizeof(double)];  
};
```

```
void fun()  
{  
    double sum = 0.0;  
    struct tparam sum_local[omp_get_max_threads()] __attribute__((aligned(64)));  
    #pragma omp parallel  
    {  
        int tid = omp_get_thread_num();  
        sum_local[tid].sum = 0.0;  
        #pragma omp for nowait  
        for (int i = 0; i < N; i++) {  
            sum_local[tid].sum += x[i] * y[i];  
        }  
        #pragma omp atomic  
        sum += sum_local[tid].sum;  
    }  
}
```

- Отдельная строка кеш-памяти для данных каждого потока

Cache line



Подходы к эффективному использованию кеш-памяти в программах

Подходы к эффективному использованию кеш-памяти

- **Оптимизация доступа к данным** — улучшение временной локальности кода
 - Слияние циклов (loop fusion)
 - Перестановка циклов (loop interchange)
 - Блочное выполнение гнезда циклов (loop blocking/tailing)
 - Предвыборка данных (prefetching)
 - Реорганизация кода для сокращения вытеснения данных из кеш-памяти (cache pollution)
- **Оптимизация размещения данных в памяти** — улучшение пространственной локальности
 - Выравнивание адресов размещения данных (alignment)
 - Слияние массивов (array merging)
 - Ложное разделение данных (false sharing)

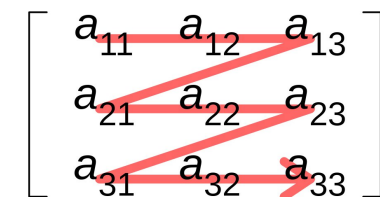
Перестановка циклов (loop interchange)

- Два смежных цикла переставляются местами
- Ограничения: перестановка не должна влиять на семантику программы
- Плюсы: уменьшается шаг обращения к данным (stride), улучшается пространственная локальность

```
double matrix_sum(double a[N][N])
{
    double sum = 0;
    for (int j = 0; j < N; j++) {
        for (int i = 0; i < N; i++) {
            sum += a[i][j]; /* stride-N read, uses N new cache lines, evicts a[0][], a[1][], ... */
        }
    }
    return sum;
}
```

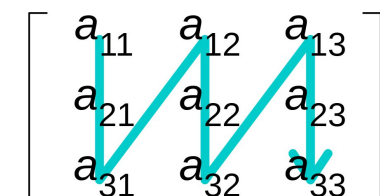
```
double matrix_sum_interchanged(double a[N][N])
{
    double sum = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            sum += a[i][j]; /* stride-1 read, uses N / 8 cache lines */
        }
    }
    return sum;
}
```

Row-major order



C/C++

Column-major order



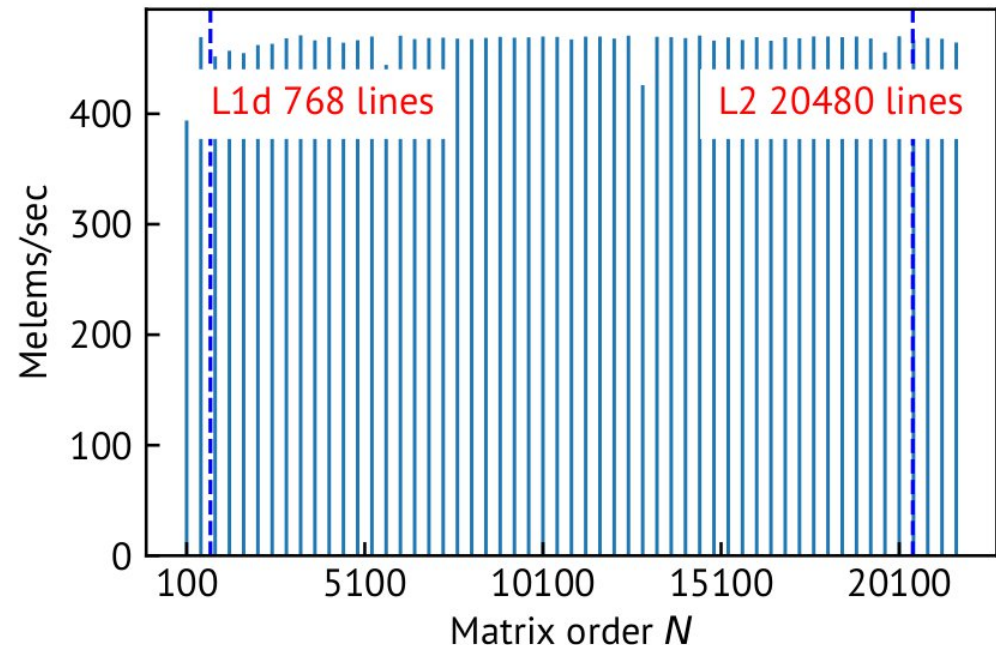
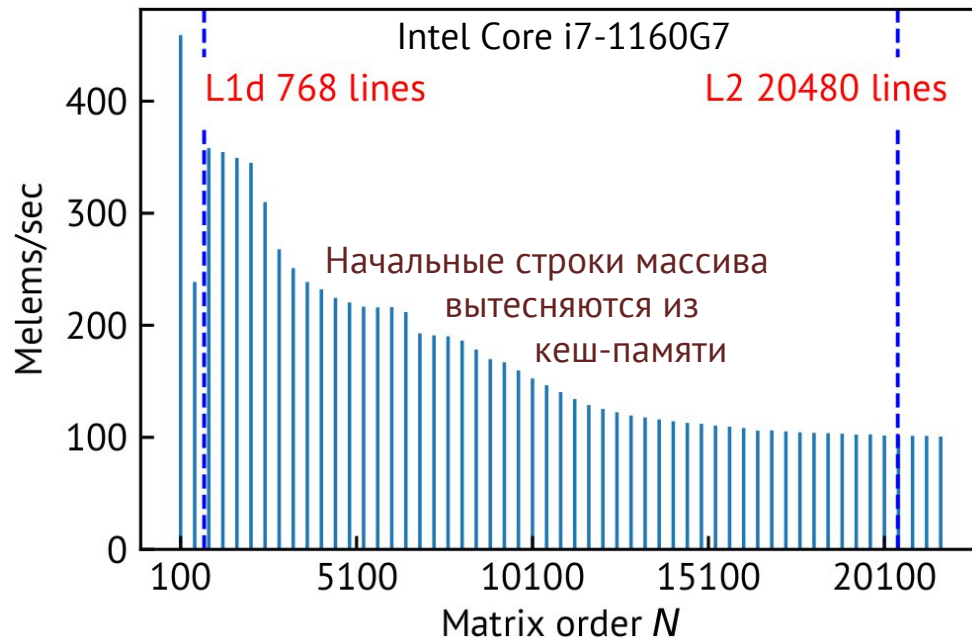
Fortran

Перестановка циклов (loop interchange)

- Два смежных цикла переставляются местами

```
double matrix_sum(double a[N][N]) {  
    double sum = 0;  
    for (int j = 0; j < N; j++) {  
        for (int i = 0; i < N; i++)  
            sum += a[i][j];  
    }  
    return sum;  
}
```

```
double matrix_sum_interchanged(double a[N][N]) {  
    double sum = 0;  
    for (int i = 0; i < N; i++) {  
        for (int j = 0; j < N; j++)  
            sum += a[i][j];  
    }  
    return sum;  
}
```



Слияние циклов (loop fusion)

- Два смежных цикла объединятся в один
- Ограничения: одинаковое пространство итераций циклов
- Плюсы: повышается локальность обращения к данным, сокращаются накладные расходы на поддержание цикла (проверка условия, переход), увеличивается количество независимых по данным инструкций в теле цикла (параллелизм уровня инструкций)

```
void vec_sum(double a[N], double b[N], double c[N])
{
    for (int i = 0; i < N; i++) {
        b[i] = a[i] + 1.0;
    } /* For large N, the initial elems of the a[i], b[i] will be evicted */

    for (int i = 0; i < N; i++) {
        c[i] = b[i] * 4.0;
    }
}
```

```
void vec_sum_fusion(double a[N], double b[N], double c[N])
{
    for (int i = 0; i < N; i++) {
        b[i] = a[i] + 1.0;
        c[i] = b[i] * 4.0; /* uses cached b[i] */
    }
}
```

Нежелательное вытеснение строк из кеш-памяти (cache pollution)

- Обработка второстепенного массива в цикле может вытеснить из кеш-памяти ранее подготовленные данные

```
void pollute_cache(double b[N]) {
    for (int i = 0; i < CACHE_SIZE; i++) {
        cache_buf[i] = 1;
    }
}

int main()
{
    /* Prepare data */
    for (int i = 0; i < N; i++) {
        a[i] = i;
    } /* Uses N / 8 cache lines */

    /* Evict data from cache */
    pollute_cache(b);

    /* Read prepared data    <— a[] may be evicted from cache !!! */
    double aa = 0;
    for (int i = 0; i < N; i += 8) {
        aa += a[i];
    }
    /* Read N / 8 lines */

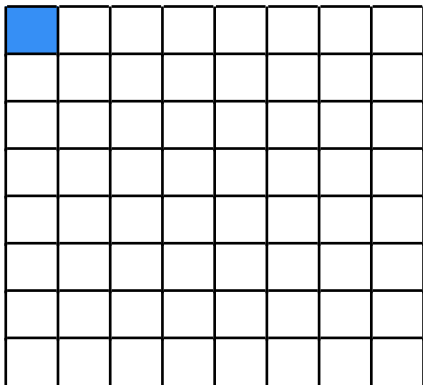
    return 0;
}
```

Варианты модификации кода:

1. Переставить первый цикл и вызов `pollute_cache()`
2. Использовать обход записи в кеш-памяти в `pollute_cache()` – non-temporal stores

Умножение матриц (DGEMM)

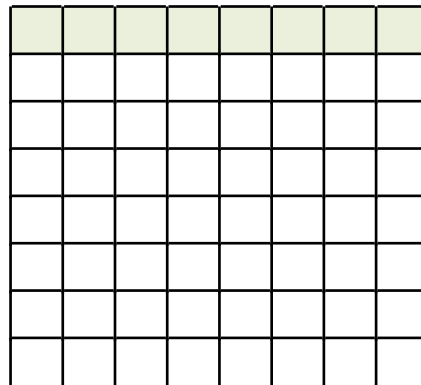
double c[N][N]



Чтение/запись с шагом 1
[0][0], [0][1], [0][2], ...,
[N-1][N-1]

=

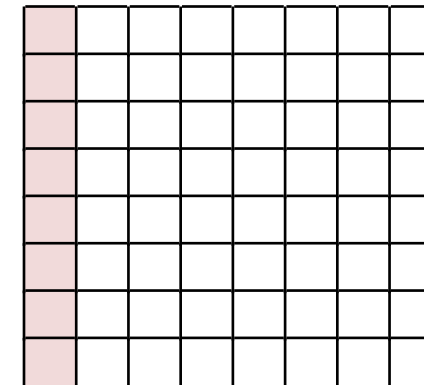
double a[N][N]



Чтение с шагом 1
[0][0], [0][1], [0][2], ...,
[N-1][N-1]

*

double b[N][N]

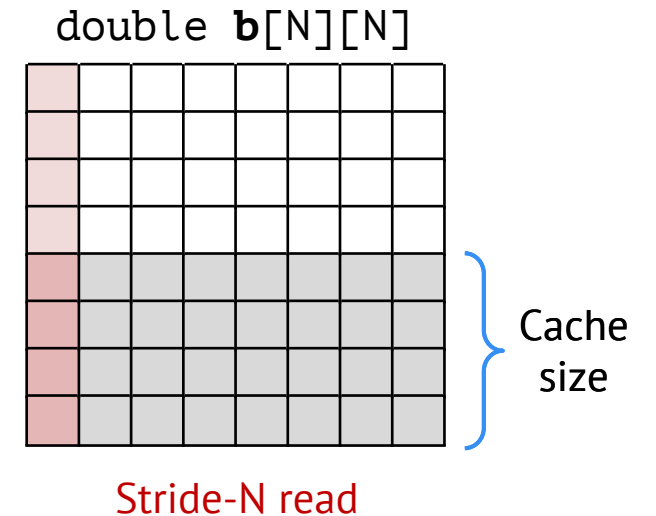


Чтение с шагом N
[0][0], [1][0], ..., [N-1][0], ...,
[N-1][N-1]

```
/* Cache line 64B, sizeof(double) = 8 */  
void dgemm_def(double a[N][N], double b[N][N], double c[N][N]) {  
    for (int i = 0; i < N; i++) {  
        for (int j = 0; j < N; j++) {  
            for (int k = 0; k < N; k++) {  
                c[i][j] += a[i][k] * b[k][j];  
            } /* Uses: 1 cacheline for c[i][j], N/8 lines a[i][*], N lines b[*][j] */  
        }  
    }  
}
```

Умножение матриц (DGEMM)

```
/* Cache line 64B, sizeof(double) = 8 */
void dgemm_def(double a[N][N], double b[N][N], double c[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```



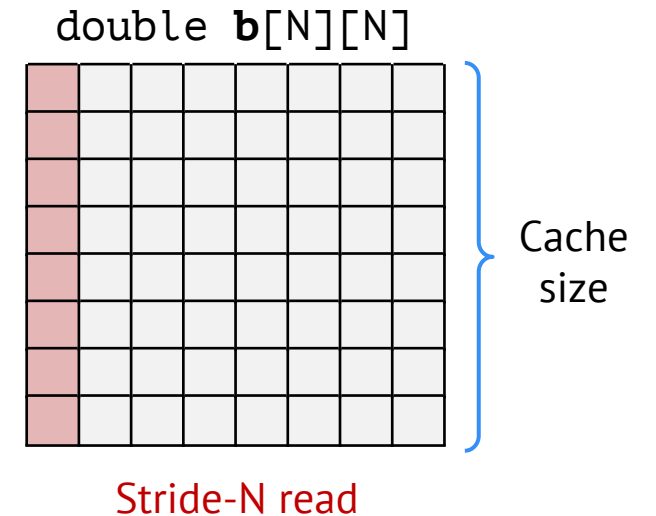
$CACHE_LINES = CACHE_SIZE / CACHE_LINE_SIZE$

Худший случай $N \gg CACHE_LINES$

- Первый проход по циклу k ($i = 0, j = 0$):
в кеш-памяти использована 1 строка для $c[0][0]$, $N/8$ строк для $a[0][k]$,
 N строк для $b[k][0]$
- На итерации $k \geq CACHE_LINES$ начнется вытеснение из кеш-памяти строк
с элементами $b[0][0], b[1][0], b[2][0], \dots$, а также возможно $c[0][0], a[0][0], a[0][1], \dots, a[0][N-1]$

Умножение матриц (DGEMM)

```
/* Cache line 64B, sizeof(double) = 8 */  
void dgemm_def(double a[N][N], double b[N][N], double c[N][N]) {  
    for (int i = 0; i < N; i++) {  
        for (int j = 0; j < N; j++) {  
            for (int k = 0; k < N; k++) {  
                c[i][j] += a[i][k] * b[k][j];  
            }  
        }  
    }  
}
```



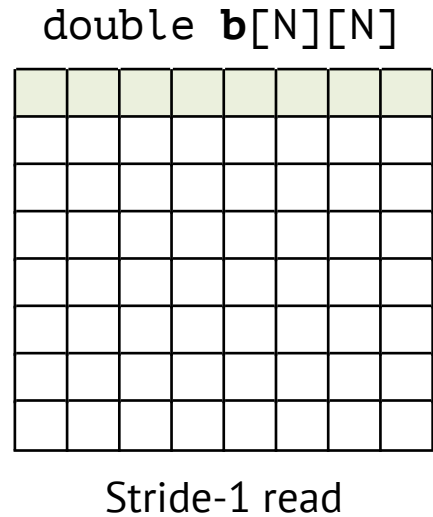
CACHE_LINES = CACHE_SIZE / CACHE_LINE_SIZE

Лучший случай – три матрицы помещаются в кеш-память

- Каждая матрица помещается в кеш-память, вытеснения строк не происходит
- Для трех матриц требуется $3 * \text{sizeof}(\text{double}) * N^2$ строк кеш-памяти:
 $24N^2 \leq \text{CACHE_SIZE}$

Умножение матриц (DGEMM): перестановка циклов $ijk \rightarrow ikj$

```
/* Cache line 64B, sizeof(double) = 8 */  
void dgemm_interchanged(double a[N][N], double b[N][N], double c[N][N])  
{  
    for (int i = 0; i < N; i++) {  
        for (int k = 0; k < N; k++) {  
            for (int j = 0; j < N; j++) {  
                c[i][j] += a[i][k] * b[k][j];  
            }  
        }  
    }  
}
```



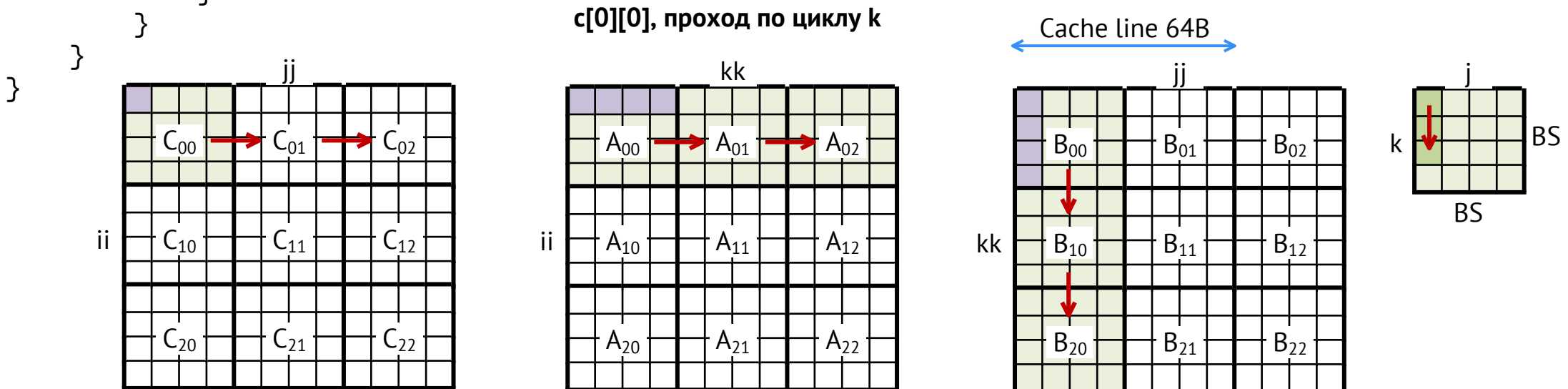
- Чтение элементов матрицы $b[N][N]$ с шагом 1

Блочное умножение матриц (blocked GEMM)

```
#define BS (CACHELINE_SIZE / sizeof(double))

void dgemm_block(double a[N][N], double b[N][N], double c[N][N]) {
    for (int ii = 0; ii < N; ii += BS) {
        for (int jj = 0; jj < N; jj += BS) {
            for (int kk = 0; kk < N; kk += BS) {
                for (int i = ii; i < IMIN(N, ii + BS); i++) {
                    for (int j = jj; j < IMIN(N, jj + BS); j++) {
                        for (int k = kk; k < IMIN(N, kk + BS); k++)
                            c[i][j] += a[i][k] * b[k][j];
                    }
                }
            }
        }
    }
}
```

- Матрицы разбиваются на подматрицы $BS \times BS$
- Каждая подматрица (блок) полностью помещается в кеш-память
- По каждой подматрице C выполняется N/BS проходов

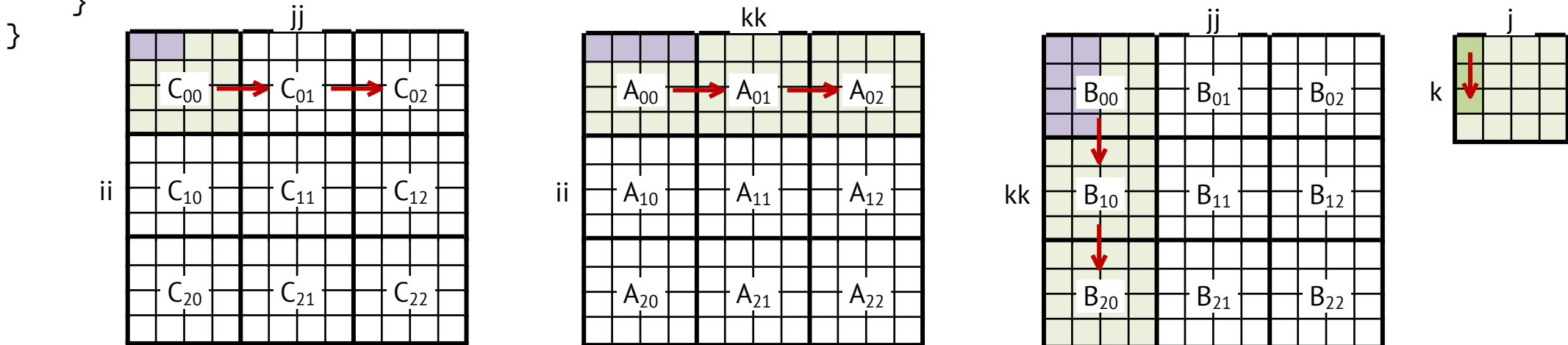


Блочное умножение матриц (blocked GEMM)

```
#define BS (CACHELINE_SIZE / sizeof(double))

void dgemm_block(double a[N][N], double b[N][N], double c[N][N]) {
    for (int ii = 0; ii < N; ii += BS) {
        for (int jj = 0; jj < N; jj += BS) {
            for (int kk = 0; kk < N; kk += BS) {
                for (int i = ii; i < IMIN(N, ii + BS); i++) {
                    for (int j = jj; j < IMIN(N, jj + BS); j++) {
                        for (int k = kk; k < IMIN(N, kk + BS); k++)
                            c[i][j] += a[i][k] * b[k][j];
                    }
                }
            }
        }
    }
}
```

- Матрицы разбиваются на подматрицы $BS \times BS$
- Каждая подматрица (блок) полностью помещается в кеш-память

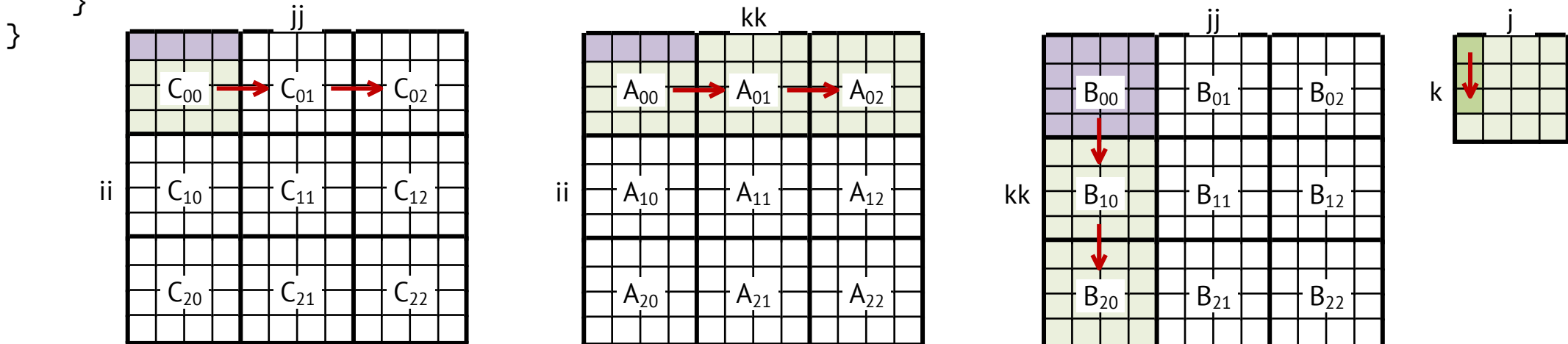


Блочное умножение матриц (blocked GEMM)

```
#define BS (CACHELINE_SIZE / sizeof(double))

void dgemm_block(double a[N][N], double b[N][N], double c[N][N]) {
    for (int ii = 0; ii < N; ii += BS) {
        for (int jj = 0; jj < N; jj += BS) {
            for (int kk = 0; kk < N; kk += BS) {
                for (int i = ii; i < IMIN(N, ii + BS); i++) {
                    for (int j = jj; j < IMIN(N, jj + BS); j++) {
                        for (int k = kk; k < IMIN(N, kk + BS); k++)
                            c[i][j] += a[i][k] * b[k][j];
                    }
                }
            }
        }
    }
}
```

- Матрицы разбиваются на подматрицы $BS \times BS$
- Каждая подматрица (блок) полностью помещается в кеш-память

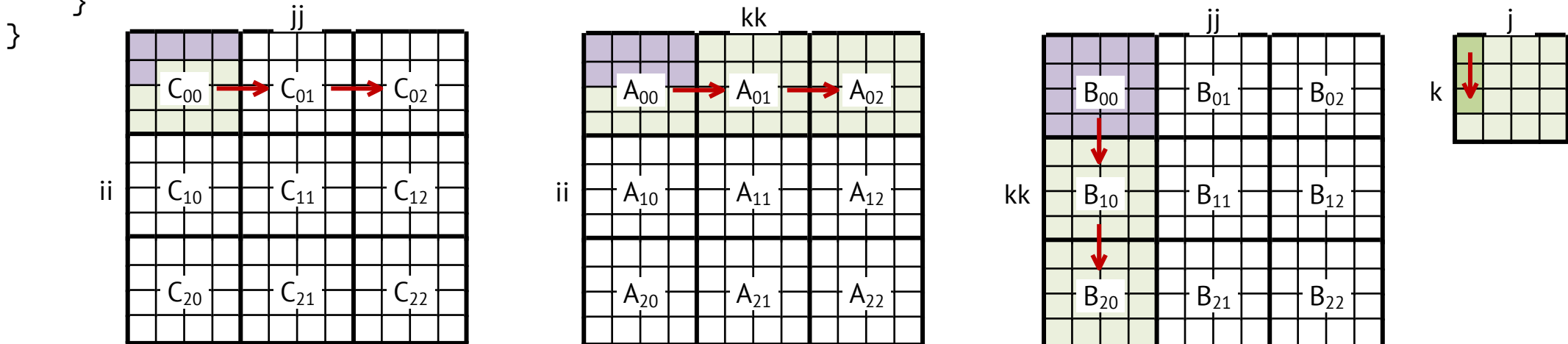


Блочное умножение матриц (blocked GEMM)

```
#define BS (CACHELINE_SIZE / sizeof(double))

void dgemm_block(double a[N][N], double b[N][N], double c[N][N]) {
    for (int ii = 0; ii < N; ii += BS) {
        for (int jj = 0; jj < N; jj += BS) {
            for (int kk = 0; kk < N; kk += BS) {
                for (int i = ii; i < IMIN(N, ii + BS); i++) {
                    for (int j = jj; j < IMIN(N, jj + BS); j++) {
                        for (int k = kk; k < IMIN(N, kk + BS); k++)
                            c[i][j] += a[i][k] * b[k][j];
                    }
                }
            }
        }
    }
}
```

- Матрицы разбиваются на подматрицы $BS \times BS$
- Каждая подматрица (блок) полностью помещается в кеш-память

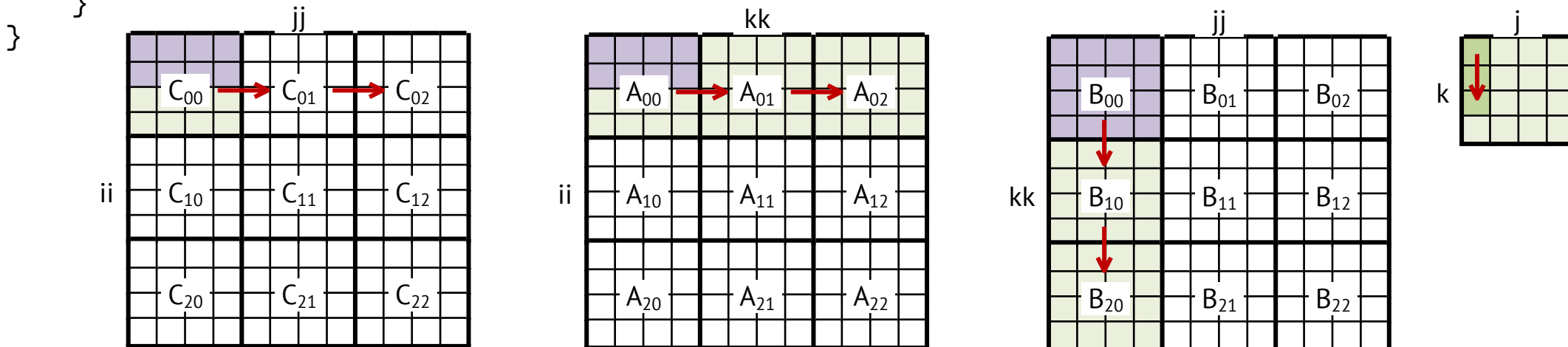


Блочное умножение матриц (blocked GEMM)

```
#define BS (CACHELINE_SIZE / sizeof(double))

void dgemm_block(double a[N][N], double b[N][N], double c[N][N]) {
    for (int ii = 0; ii < N; ii += BS) {
        for (int jj = 0; jj < N; jj += BS) {
            for (int kk = 0; kk < N; kk += BS) {
                for (int i = ii; i < IMIN(N, ii + BS); i++) {
                    for (int j = jj; j < IMIN(N, jj + BS); j++) {
                        for (int k = kk; k < IMIN(N, kk + BS); k++)
                            c[i][j] += a[i][k] * b[k][j];
                    }
                }
            }
        }
    }
}
```

- Матрицы разбиваются на подматрицы $BS \times BS$
- Каждая подматрица (блок) полностью помещается в кеш-память

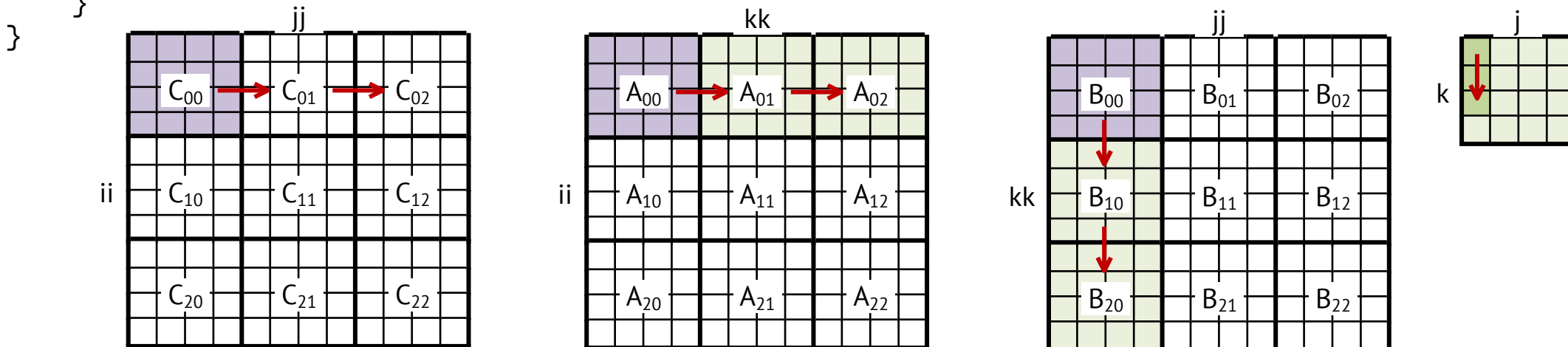


Блочное умножение матриц (blocked GEMM)

```
#define BS (CACHELINE_SIZE / sizeof(double))

void dgemm_block(double a[N][N], double b[N][N], double c[N][N]) {
    for (int ii = 0; ii < N; ii += BS) {
        for (int jj = 0; jj < N; jj += BS) {
            for (int kk = 0; kk < N; kk += BS) {
                for (int i = ii; i < IMIN(N, ii + BS); i++) {
                    for (int j = jj; j < IMIN(N, jj + BS); j++) {
                        for (int k = kk; k < IMIN(N, kk + BS); k++)
                            c[i][j] += a[i][k] * b[k][j];
                    }
                }
            }
        }
    }
}
```

- Матрицы разбиваются на подматрицы $BS \times BS$
- Каждая подматрица (блок) полностью помещается в кеш-память

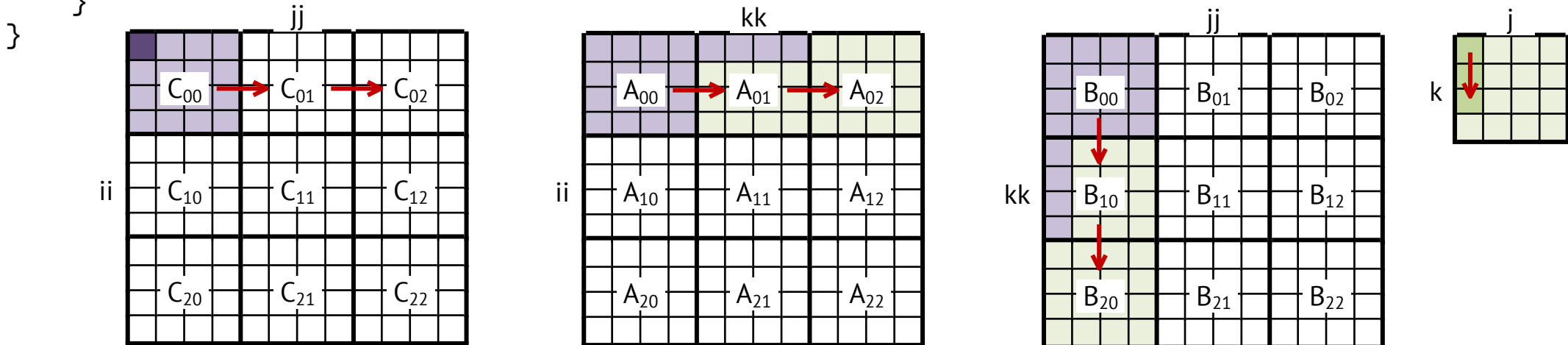


Блочное умножение матриц (blocked GEMM)

```
#define BS (CACHELINE_SIZE / sizeof(double))

void dgemm_block(double a[N][N], double b[N][N], double c[N][N]) {
    for (int ii = 0; ii < N; ii += BS) {
        for (int jj = 0; jj < N; jj += BS) {
            for (int kk = 0; kk < N; kk += BS) {
                for (int i = ii; i < IMIN(N, ii + BS); i++) {
                    for (int j = jj; j < IMIN(N, jj + BS); j++) {
                        for (int k = kk; k < IMIN(N, kk + BS); k++)
                            c[i][j] += a[i][k] * b[k][j];
                    }
                }
            }
        }
    }
}
```

- Матрицы разбиваются на подматрицы $BS \times BS$
- Каждая подматрица (блок) полностью помещается в кеш-память



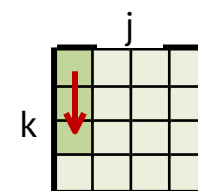
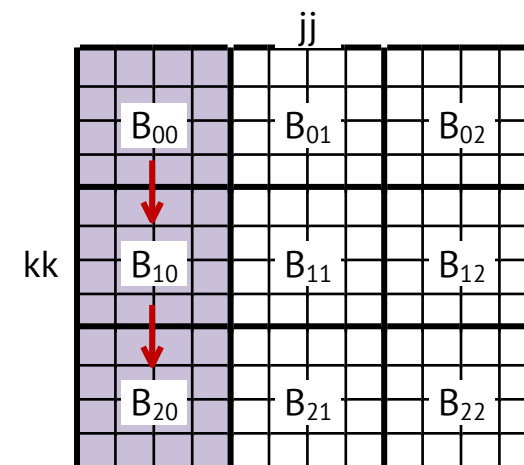
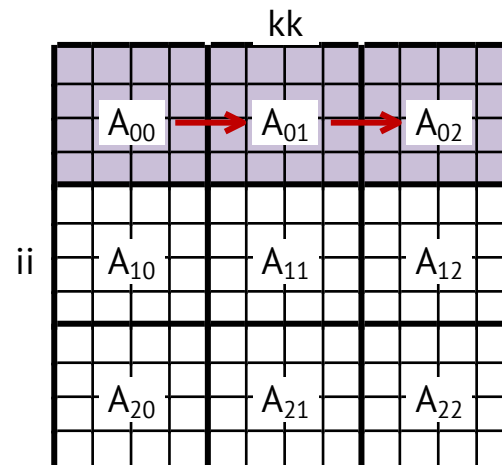
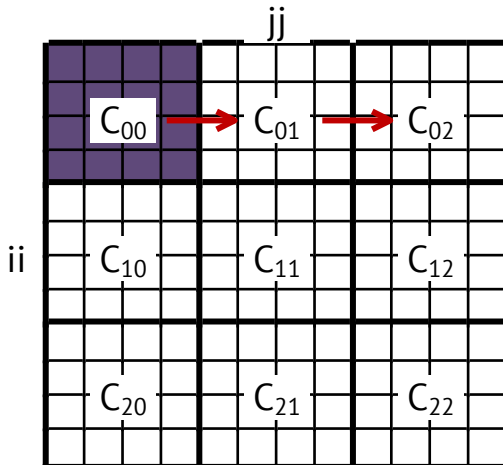
Блочное умножение матриц (blocked GEMM)

```
#define BS (CACHELINE_SIZE / sizeof(double))

void dgemm_block(double a[N][N], double b[N][N], double c[N][N]) {
    for (int ii = 0; ii < N; ii += BS) {
        for (int jj = 0; jj < N; jj += BS) {
            for (int kk = 0; kk < N; kk += BS) {
                for (int i = ii; i < IMIN(N, ii + BS); i++) {
                    for (int j = jj; j < IMIN(N, jj + BS); j++) {
                        for (int k = kk; k < IMIN(N, kk + BS); k++)
                            c[i][j] += a[i][k] * b[k][j];
                    }
                }
            }
        }
    }
}
```

- Матрицы разбиваются на подматрицы $BS \times BS$
- Каждая подматрица (блок) полностью помещается в кеш-память

По подматрице C_{00} выполнено N/BS проходов



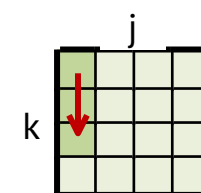
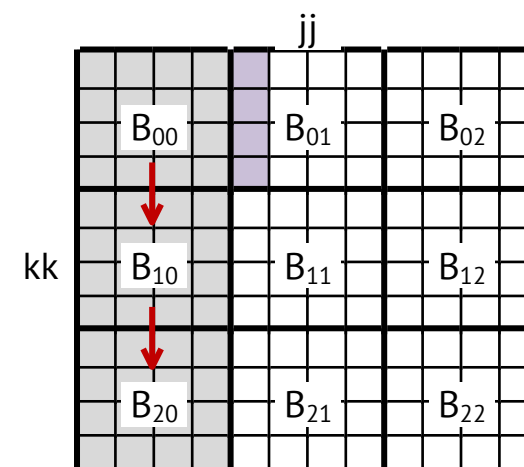
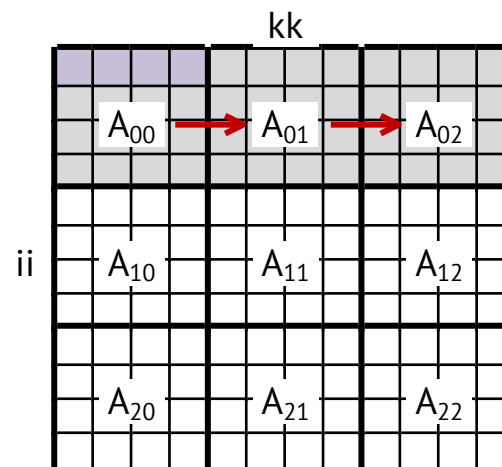
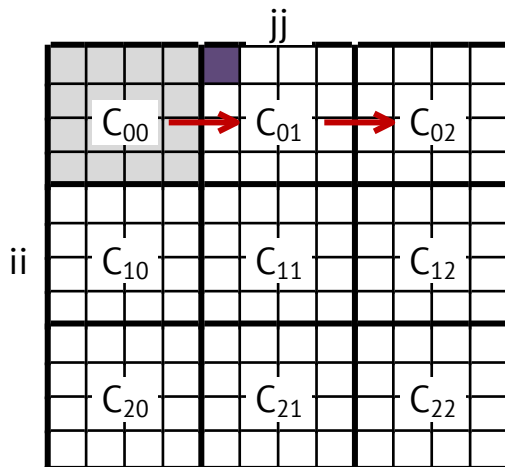
Блочное умножение матриц (blocked GEMM)

```
#define BS (CACHELINE_SIZE / sizeof(double))

void dgemm_block(double a[N][N], double b[N][N], double c[N][N]) {
    for (int ii = 0; ii < N; ii += BS) {
        for (int jj = 0; jj < N; jj += BS) {
            for (int kk = 0; kk < N; kk += BS) {
                for (int i = ii; i < IMIN(N, ii + BS); i++) {
                    for (int j = jj; j < IMIN(N, jj + BS); j++) {
                        for (int k = kk; k < IMIN(N, kk + BS); k++)
                            c[i][j] += a[i][k] * b[k][j];
                    }
                }
            }
        }
    }
}
```

- Матрицы разбиваются на подматрицы $BS \times BS$
- Каждая подматрица (блок) полностью помещается в кеш-память

Вычисление подматрицы C_{01}



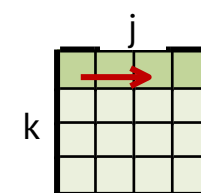
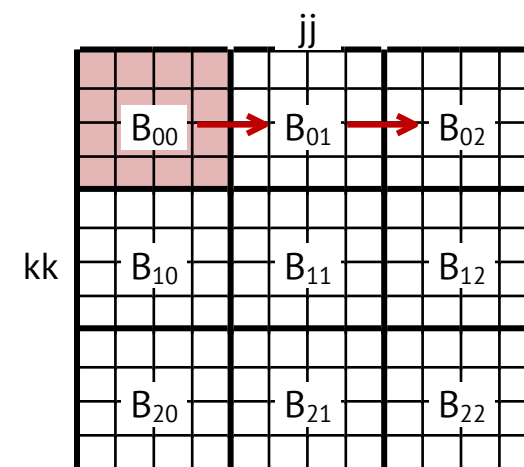
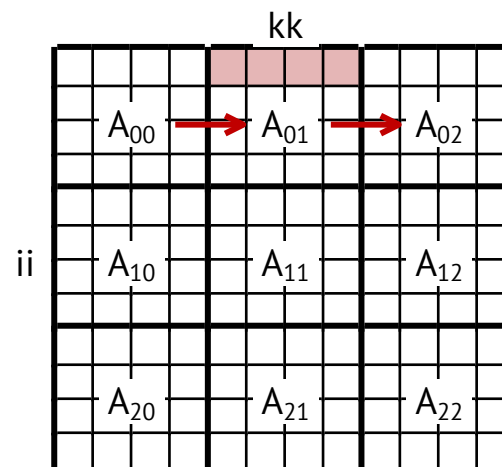
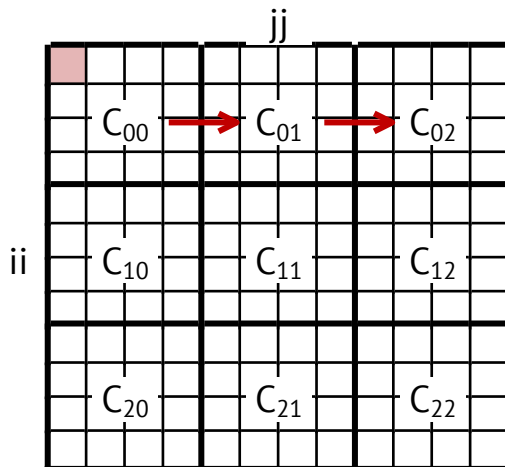
Блочное умножение матриц (blocked GEMM)

```
#define BS (CACHELINE_SIZE / sizeof(double))

void dgemm_block(double a[N][N], double b[N][N], double c[N][N]) {
    for (int ii = 0; ii < N; ii += BS) {
        for (int kk = 0; kk < N; kk += BS) {
            for (int jj = 0; jj < N; jj += BS) {
                for (int i = ii; i < IMIN(N, ii + BS); i++) {
                    for (int k = kk; k < IMIN(N, kk + BS); k++) {
                        for (int j = jj; j < IMIN(N, jj + BS); j++)
                            c[i][j] += a[i][k] * b[k][j];
                    }
                }
            }
        }
    }
}
```

- Матрицы разбиваются на подматрицы BSxBS
- Каждая подматрица (блок) полностью помещается в кеш-память

Loop interchange: **ii, kk, jj, i, k, j**



Non-temporal data (cache bypass)

Intel64 non-temporal stores (streaming stores)

- **non-temporal data** — область памяти с данными, к которым не будет обращений в ближайшее время (отсутствует временная локальность)
- Если при записи строка отсутствует в кеш-памяти (write miss), то она сперва загружается в кеш, а затем в нее записываются данные (write-allocate policy)
- Если операция записи обновляет содержимое всей строки, то операция чтения из памяти является избыточной
- При промахе операции записи в многоядерной системе протокол обеспечения когерентности кеш-памяти (MESI, MESIF) выполняет чтение строки из памяти и отправляет широковещательное уведомление об изменении её состояния (Read For Ownership – RFO, read + invalidate broadcast)
- **SSE2 Streaming stores**
- MOVNTDQ (Store Packed Int. Using Non-Temporal Hint) записывает 16 байт в память в обход кеш-памяти
- `_mm_stream_si128(__m128i *p, __m128i a)`
- `_mm256_stream_si256(__m256i *p, __m256i a) // VMOVNTDQ AVX`
- `_mm512_stream_si512(__m512i *p, __m512i a) // VMOVNTDQ AXV-512`
- *The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy.*
- *Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTDQ instructions.*

Non-temporal stores (streaming stores)

```
double data[N];
int idx[N];

void vec_shuffle() {
    srand(0);
    for (int i = 0; i < N; i++) {
        idx[i] = i;
        data[i] = i;
    }
    for (int i = 0; i < N; i++) {
        int left = N - i;
        int swap = rand() % left;
        int temp = idx[i];
        idx[i] = idx[swap];
        idx[swap] = temp;
    }
}

void vec_sum() {
    /* Iterate in random order */
    double s = 0;
    for (int i = 0; i < N; i++) {
        s += data[idx[i]];
    }
    x = s;
}
```

```
void pollute_cache()
{
    /* Evict lines from cache */
    for (int i = 0; i < CACHE_SIZE;
        i += CACHE_LINE_SIZE)
    {
        cache_buf[i] = 1;
    }
}

void main()
{
    /* Prepare data */
    vec_shuffle();

    /* Evict data from cache */
    pollute_cache();

    /* Read prepared data */
    vec_sum();

    _mm_sfence();
}
```

Вытесняет из кеш-памяти массивы idx[N], data[N]

```
# N=1000, elapsed time (tsc) 5765
```

Non-temporal stores (streaming stores)

```
// Write 16 bytes to dest
static inline void ntstore(uint8_t *dest, uint8_t a)
{
    /* dest must be 16 byte aligned */
    __m128i i = _mm_set_epi8(a, a, a, a, a, a, a, a,
                             a, a, a, a, a, a, a, a);
    _mm_stream_si128((__m128i *)&dest[0], i); /* MOVNTDQ */
    _mm_stream_si128((__m128i *)&dest[16], i);
    _mm_stream_si128((__m128i *)&dest[32], i);
    _mm_stream_si128((__m128i *)&dest[48], i);
    /*
    _mm_stream_si128(__m128i *p, __m128i a)
    stores the data in a to the address p without polluting
    the caches. If the cache line containing address p is already
    in the cache, the cache will be updated.
    Address p must be 16-byte aligned.
    */
}

void pollute_cache_nta()
{
    for (int i = 0; i < CACHE_SIZE; i += CACHE_LINE_SIZE) {
        ntstore(&cache_buf[i], 1);
    }
    _mm_sfence();
}
```

Полностью заполняет буфер
Write Combining (64B)
с использованием
4 векторных инструкций
записи по 16 байт

```
void main()
{
    /* Prepare data */
    vec_shuffle();

    /* Evict data from cache */
    pollute_cache();

    /* Read prepared data */
    vec_sum();

    _mm_sfence();
}
```

Запись в массив 64 байт в обход
кеш-памяти – NT stores

N=1000, elapsed time (tsc) 4522

Предвыборка данных в кеш-память (prefetching)

Предвыборка данных в кеш-память (prefetching)

- Предвыборка позволяет амортизировать накладные расходы на промахи кеш-памяти
- «Подсказка» процессору (hint), асинхронное выполнение (опциональное)

PREFETCHh—Prefetch Data Into Caches (minimum of 32B is prefetched)

«Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by a locality hint»

Instruction	Description	
PREFETCHT0 <i>m8</i>	Move data specified by address closer to the processor using T0 hint.	T0 (temporal data) — prefetch data into all cache levels.
PREFETCHT1 <i>m8</i>	Move data specified by address closer to the processor using T1 hint.	T1 (temporal data with respect to first level cache) — prefetch data in all cache levels except 0th cache level
PREFETCHT2 <i>m8</i>	Move data specified by address closer to the processor using T2 hint.	T2 (temporal data with respect to second level cache) — prefetch data in all cache levels, except 0th and 1st cache levels.
PREFETCHNTA <i>m8</i>	Move data specified by address closer to the processor using NTA hint.	NTA (non-temporal data with respect to all cache levels) — prefetch data into non-temporal cache structure (this hint can be used to minimize pollution of caches)

Предвыборка данных в кеш-память (prefetching)

- Предвыборка позволяет амортизировать накладные расходы на промахи кеш-памяти
- «Подсказка» процессору (hint), асинхронное выполнение (опциональное)

```
void _mm_prefetch(char *p, int i)
```

- p – address of the byte (and corresponding cache line) to be prefetched
- i – type of prefetch operation: _MM_HINT_T0, _MM_HINT_T1, _MM_HINT_T2, _MM_HINT_NTA

```
void __builtin_prefetch(const void *addr, ...)
```

GNU extension to prefetch memory

Предвыборка данных в кеш-память (prefetching) (предвыборка на каждой итерации цикла)

```
double dotp(double *x, double *y, int n)
{
    double dp = 0;
    for (int i = 0; i < n; i++) {
        dp += x[i] * y[i];
    }
    return dp;
}
```

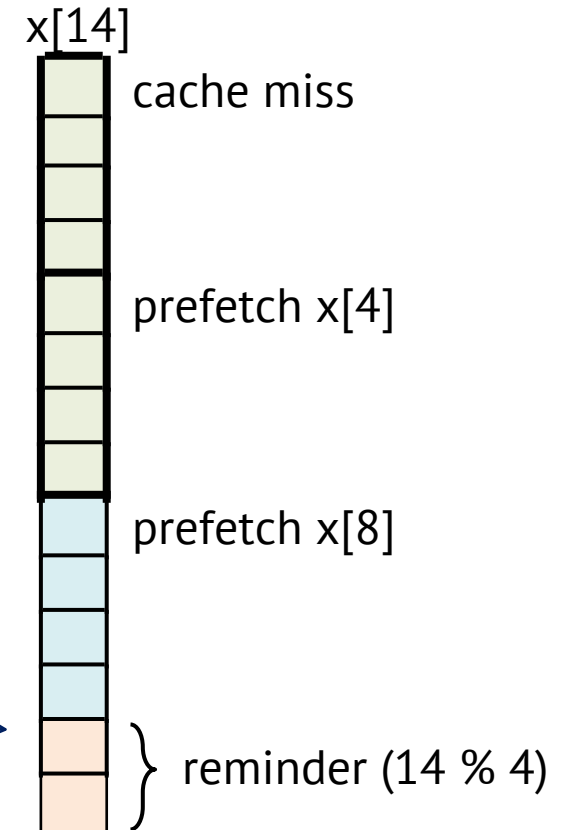
```
double dotp_opt_v1(double *x, double *y, int n)
{
    /* Naive prefetching on each iteration */
    double dp = 0;
    for (int i = 0; i < n; i++) {
        _mm_prefetch(&x[i + 1], _MM_HINT_T0);
        _mm_prefetch(&y[i + 1], _MM_HINT_T0);
        dp += x[i] * y[i];
    }
    /* Might generate exception on last iteration &x[i + 1] - invalid address */
    return dp;
}
```

- Избыточная предвыборка => вытеснение строк из кеш-памяти
- Возможна ошибка при попытке предвыборки &x[n], &y[n] – недействительный адресс

Предвыборка данных в кеш-память (prefetching) (предвыборка на k итераций)

```
double dotp_opt_v2(double *x, double *y, int n)
{
    /* Unroll loop and prefetch for 4 iterations */
    double dp = 0;
    int i;
    for (i = 0; i < n - 4; i += 4) {
        _mm_prefetch(&x[i + 4], _MM_HINT_T0);
        _mm_prefetch(&y[i + 4], _MM_HINT_T0);
        dp += x[i] * y[i];
        dp += x[i + 1] * y[i + 1];
        dp += x[i + 2] * y[i + 2];
        dp += x[i + 3] * y[i + 3];
    }

    for (; i < n; i++) {
        dp += x[i] * y[i];
    }
    /* Cache miss during first iteration */
    return dp;
}
```



Предвыборка данных в кеш-память (prefetching) (предвыборка на k итераций)

```
double dotp_opt_v3(double *x, double *y, int n)
{
    /* Unroll loop and prefetch for 4 iterations */
    double dp = 0, t1 = 0, t2 = 0, t3 = 0;
    int i;

    _mm_prefetch(&x[0], _MM_HINT_T0);
    _mm_prefetch(&y[0], _MM_HINT_T0);
    for (i = 0; i < n - 4; i += 4) {
        _mm_prefetch(&x[i + 4], _MM_HINT_T0);
        _mm_prefetch(&y[i + 4], _MM_HINT_T0);
        dp += x[i] * y[i];
        t1 += x[i + 1] * y[i + 1];
        t2 += x[i + 2] * y[i + 2];
        t3 += x[i + 3] * y[i + 3];
    }
    dp = dp + t1 + t2 + t3;

    for (; i < n; i++) {
        dp += x[i] * y[i];
    }
    return dp;
}
```

- Предвыборка x[0], y[0]
- Устранение зависимости по данным между инструкциями цикла (по переменной dp)

Оптимизация размещения данных в памяти

Разбиение структур (struct split)

```

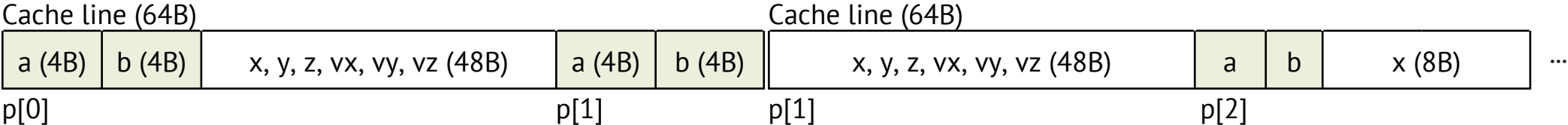
struct node {
    float a;
    float b;
    double x, y, z;
    double vx, vy, vz;
}; // sizeof(struct node) = 56

struct node p[N] __attribute__((aligned(CACHELINE_SIZE)));

void run()
{
    vec_init();
    float s = 0;
    for (int i = 0; i < N; i++) {
        s += p[i].a * p[i].b;
    }
}
    
```

Критический цикл

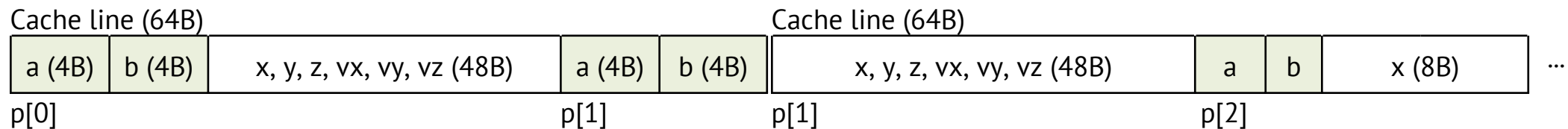
- Чтение полей a и b
- x, y, z, vx, vy, vz занимают место в строке кеш-памяти и не используются



Разбиение структур (struct split)

```
$ pahole ./prog
struct node {
    float      a;          /*      0      4 */
    float      b;          /*      4      4 */
    double     x;          /*      8      8 */
    double     y;          /*     16      8 */
    double     z;          /*     24      8 */
    double     vx;         /*     32      8 */
    double     vy;         /*     40      8 */
    double     vz;         /*     48      8 */

    /* size: 56, cachelines: 1, members: 8 */
    /* last cacheline: 56 bytes */
};
```



Разбиение структур (struct split)

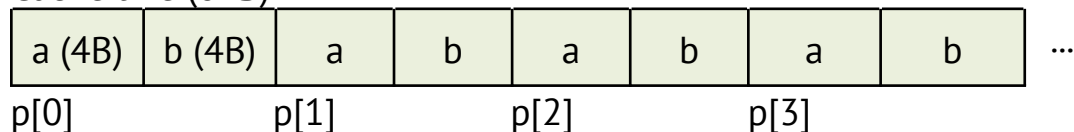
```
struct node1 {
    float a, b;
};
struct node2 {
    double x, y, z, vx, vy, vz;
};

struct node1 p1[N] __attribute__((aligned(CACHELINE_SIZE)));
struct node2 p2[N] __attribute__((aligned(CACHELINE_SIZE)));

void run()
{
    vec_init();
    float s = 0;
    for (int i = 0; i < N; i++) {
        s += p1[i].a * p1[i].b;
    }
}
```

- Разбили структуру на две: node1, node2
- Массив p1 содержит только элемент a и b
- В строке кеш-памяти помещается 4 элемента массива p1 (сокращается число промахов кеш-памяти)

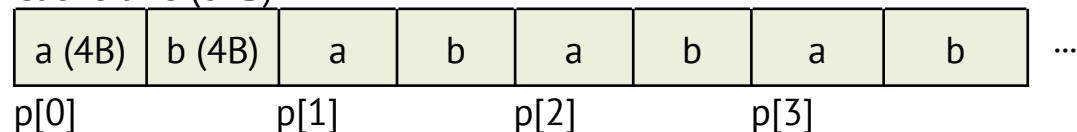
Cache line (64B)



Разбиение структур

```
$ pahole ./prog-split
struct node1 {
    float      a;          /*      0      4 */
    float      b;          /*      4      4 */
    /* size: 8, cachelines: 1, members: 2 */
    /* last cacheline: 8 bytes */
};
struct node2 {
    double     x;          /*      0      8 */
    double     y;          /*      8      8 */
    double     z;          /*     16      8 */
    double     vx;         /*     24      8 */
    double     vy;         /*     32      8 */
    double     vz;         /*     40      8 */
    /* size: 48, cachelines: 1, members: 6 */
    /* last cacheline: 48 bytes */
};
```

Cache line (64B)



Выравнивание структур

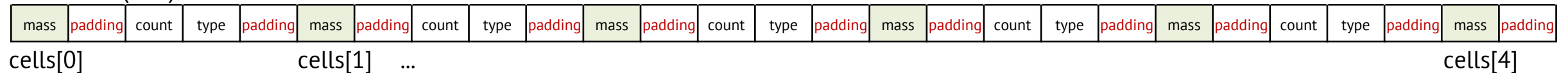
```
struct cell {
    char mass;
    // padding 3 bytes
    int count;
    char type;
    // padding 3 bytes
}; // sizeof(cell) = 12
```

- Адрес поля должен быть выравнен на границу, соответствующую его размеру
- Размер структуры должен быть кратен размеру самого большого поля
- Компилятор добавляет поля выравнивания (padding), которые занимают место в строке кеш-памяти

```
struct cell cells[N] __attribute__((aligned(CACHELINE_SIZE)));
```

```
void run()
{
    init();
    for (int i = 0; i < N; i++) {
        cells[i].mass++;
    }
}
```

Cache line (64B)



Выравнивание структур

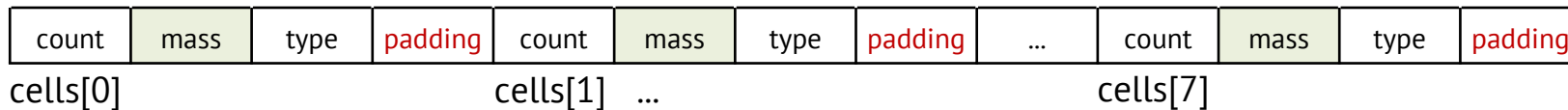
```
struct cell {  
    int count;  
    char mass;  
    char type;  
    // padding 2 bytes  
}; // sizeof(cell) = 8
```

```
struct cell cells[N] __attribute__((aligned(CACHELINE_SIZE)));
```

```
void run()  
{  
    init();  
    for (int i = 0; i < N; i++) {  
        cells[i].mass++;  
    }  
}
```

- Переставили местами поля mass и count
- Выравнивание только на границу размера наибольшего поля (count)
- В строку кеш-памяти помещается 8 элементов cell

Cache line (64B)



Выравнивание структур

x86 (32-bit)

- **char** (one byte) will be 1-byte aligned
- **short** (two bytes) will be 2-byte aligned
- **int** (four bytes) will be 4-byte aligned
- **long** (four bytes) will be 4-byte aligned
- **float** (four bytes) will be 4-byte aligned
- **double** (eight bytes) will be 8-byte aligned on Windows and 4-byte aligned on Linux
- **pointer** (four bytes) will be 4-byte aligned

x86-64 (LP64)

- **long** (eight bytes) will be 8-byte aligned
- **double** (eight bytes) will be 8-byte aligned
- **pointer** (eight bytes) will be 8-byte aligned

Размер структуры должен быть кратен размеру самого большого поля

Array of Structures (AoS) – массив структур

```
struct particle {  
    double x, y;  
    double vx, vy;  
    double m;  
    double e;  
};  
  
struct particle p[N];  
double dist[N];  
  
void run()  
{  
    init();  
    for (int i = 0; i < N; i++) {  
        dist[i] = sqrt(p[i].x * p[i].x + p[i].y * p[i].y);  
    }  
}
```

- **Критический участок кода** – обращение только к элементам x, y
- Поля vx, vy, m, e занимают место в строке кеш-памяти

Structure of Arrays (SoA) – структура из массивов

```
struct particle {
    double x[N]    __attribute__((aligned(CACHELINE_SIZE)));
    double y[N]    __attribute__((aligned(CACHELINE_SIZE)));
    double vx[N]   __attribute__((aligned(CACHELINE_SIZE)));
    double vy[N]   __attribute__((aligned(CACHELINE_SIZE)));
    double m[N]    __attribute__((aligned(CACHELINE_SIZE)));
    double e[N]    __attribute__((aligned(CACHELINE_SIZE)));
} sys;

double dist[N] __attribute__((aligned(CACHELINE_SIZE)));

void run()
{
    init();
    for (int i = 0; i < N; i++) {
        dist[i] = sqrt(sys.x[i] * sys.x[i] + sys.y[i] * sys.y[i]);
    }
}
```

- Больше элементов x, y помещается в кеш-память
- Возможность векторизации кода

Профилирование обращения к памяти

```
$ perf list | grep cache
cache-misses           [Hardware event]
cache-references       [Hardware event]
L1-dcache-load-misses [Hardware cache event]
L1-dcache-loads        [Hardware cache event]
L1-dcache-stores       [Hardware cache event]
L1-icache-load-misses [Hardware cache event]
LLC-load-misses        [Hardware cache event]
LLC-loads              [Hardware cache event]
LLC-store-misses       [Hardware cache event]
LLC-stores             [Hardware cache event]
...
```

```
$ perf stat -e cache-misses taskset --cpu-list 0 ./loop
```

```
Performance counter stats for 'taskset --cpu-list 0 ./loop':
```

```
110916792      cache-misses
```

```
2,811556255 seconds time elapsed
```

```
2,806984000 seconds user
```

```
0,004004000 seconds sys
```

Профилирование обращения к памяти

```
$ perf record -e cache-misses taskset --cpu-list 0 ./loop
[ perf record: Woken up 2 times to write data ]
[ perf record: Captured and wrote 0,450 MB perf.data (11748 samples) ]
```

```
$ perf report
```

```
Samples: 11K of event 'cache-misses', Event count (approx.): 120738902
Overhead Command Shared Object Symbol
99,34% loop loop [.] vec_sum
0,07% loop [unknown] [k] 0xfffffffffb41d225a
0,06% taskset [unknown] [k] 0xfffffffffb3fcde4b
0,05% loop [unknown] [k] 0xfffffffffb41cc8f9
0,02% loop [unknown] [k] 0xfffffffffb3fcde55
0,02% loop [unknown] [k] 0xfffffffffb41d221c
0,02% loop [unknown] [k] 0xfffffffffb41d05ae
0,02% loop [unknown] [k] 0xfffffffffb3f4ec66
0,02% loop [unknown] [k] 0xfffffffffb3fce3b6
0,02% loop [unknown] [k] 0xfffffffffb3f393ac
0,02% loop [unknown] [k] 0xfffffffffb3f003fe
0,02% loop [unknown] [k] 0xfffffffffc0dddedc
0,01% loop [unknown] [k] 0xfffffffffb41d2c00
0,01% taskset [unknown] [k] 0xfffffffffb41ab762
0,01% loop [unknown] [k] 0xfffffffffb3fcde53
0,01% loop [unknown] [k] 0xfffffffffb3f24369
0,01% loop [unknown] [k] 0xfffffffffb3fafd1c
0,01% loop [unknown] [k] 0xfffffffffb3f488dd
0,01% loop loop [.] main
```



```
for (int i = 0; i < N; i++) {
    movl    $0x0, -0x4(%rbp)
    ↓ jmp    af
    c[i] = b[i] * 4.0;
0,24 6f:   mov     -0x4(%rbp),%eax
0,04   cltq
0,25   lea    0x0(,%rax,8),%rdx
0,21   mov    -0x20(%rbp),%rax
5,34   add    %rdx,%rax
0,13   movsd  (%rax),%xmm1
13,37  mov    -0x4(%rbp),%eax
0,01   cltq
0,07   lea    0x0(,%rax,8),%rdx
0,03   mov    -0x28(%rbp),%rax
3,27   add    %rdx,%rax
0,05   movsd  _IO_stdin_used+0x58,%xmm0
2,85   mulsd  %xmm1,%xmm0
18,66  movsd  %xmm0,(%rax)
for (int i = 0; i < N; i++) {
5,84   addl   $0x1, -0x4(%rbp)
0,30  af:   cmpl   $0xaae5f, -0x4(%rbp)
0,53   jle    6f
}
```


Профилирование обращения к памяти (perf-mem)

```
Available samples
5K cpu/mem-loads,ldlat=30/P
5K cpu/mem-stores/P
```

\$ perf mem report

Дополнительная информация:

- TLB – обращение к TLB
- Snoop – обращение к межпроцессорной шине

```
36 27:   mov     -0xc(%rbp),%eax
      cltq
      imul  $0xfa00,%rax,%rdx
16   mov     -0x18(%rbp),%rax
      add     %rax,%rdx
      mov     -0x10(%rbp),%eax
      cltq
617  movsd   (%rdx,%rax,8),%xmm0
700  movsd   -0x8(%rbp),%xmm1
      addsd  %xmm1,%xmm0
      movsd  %xmm0,-0x8(%rbp)
      for (int i = 0; i < N; i++) {
17   addl   $0x1,-0xc(%rbp)
75 56:   cmpl   $0x1f3f,-0xc(%rbp)
      jle   27
      for (int j = 0; j < N; j++) {
      addl   $0x1,-0x10(%rbp)
63:  cmpl   $0x1f3f,-0x10(%rbp)
      jle   1e
      }
      }
```



Overhead	Samples	Local	Weight	Memory access	Symbol	Shared Object	Data Symbol	Data Object	Snoop	TLB access
11,55%	729	197		L1 or L1 hit	[.] __random	libc.so.6	[.] 0x00007fe256f18748	anon	None	L1 or L2 hit
9,92%	623	198		L1 or L1 hit	[.] __random	libc.so.6	[.] 0x00007fe256f18748	anon	None	L1 or L2 hit
9,90%	628	196		L1 or L1 hit	[.] __random	libc.so.6	[.] 0x00007fe256f18748	anon	None	L1 or L2 hit
7,14%	446	199		L1 or L1 hit	[.] __random	libc.so.6	[.] 0x00007fe256f18748	anon	None	L1 or L2 hit
6,17%	393	195		L1 or L1 hit	[.] __random	libc.so.6	[.] 0x00007fe256f18748	anon	None	L1 or L2 hit
4,04%	251	200		L1 or L1 hit	[.] __random	libc.so.6	[.] 0x00007fe256f18748	anon	None	L1 or L2 hit
2,72%	174	194		L1 or L1 hit	[.] __random	libc.so.6	[.] 0x00007fe256f18748	anon	None	L1 or L2 hit
1,89%	117	201		L1 or L1 hit	[.] __random	libc.so.6	[.] 0x00007fe256f18748	anon	None	L1 or L2 hit
0,78%	50	193		L1 or L1 hit	[.] __random	libc.so.6	[.] 0x00007fe256f18748	anon	None	L1 or L2 hit
0,52%	32	202		L1 or L1 hit	[.] __random	libc.so.6	[.] 0x00007fe256f18748	anon	None	L1 or L2 hit
0,32%	11	367		L1 or L1 hit	[.] matrix_sum	loop	[.] 0x00007ffcd8188938	[stack]	None	L1 or L2 hit
0,25%	6	528		L1 or L1 hit	[.] matrix_sum	loop	[.] 0x00007ffcd8188938	[stack]	None	L1 or L2 hit
0,23%	7	411		L1 or L1 hit	[.] matrix_sum	loop	[.] 0x00007ffcd8188938	[stack]	None	L1 or L2 hit
0,21%	13	203		L1 or L1 hit	[.] __random	libc.so.6	[.] 0x00007fe256f18748	anon	None	L1 or L2 hit
0,21%	6	437		L1 or L1 hit	[.] matrix_sum	loop	[.] 0x00007ffcd8188938	[stack]	None	L1 or L2 hit
0,20%	7	351		L1 or L1 hit	[.] matrix_sum	loop	[.] 0x00007ffcd8188938	[stack]	None	L1 or L2 hit
0,19%	6	403		L1 or L1 hit	[.] matrix_sum	loop	[.] 0x00007ffcd8188938	[stack]	None	L1 or L2 hit
0,19%	6	399		L1 or L1 hit	[.] matrix_sum	loop	[.] 0x00007ffcd8188938	[stack]	None	L1 or L2 hit
0,19%	1	2379		L1 or L1 hit	[.] matrix_sum	loop	[.] 0x00007ffcd8188938	[stack]	None	L1 or L2 hit
0,19%	6	386		L1 or L1 hit	[.] matrix_sum	loop	[.] 0x00007ffcd8188938	[stack]	None	L1 or L2 hit

Cannot load tips.txt file, please install perf!

Perf Shared Data C2C/HITM Analyzer (perf-c2c)

- **Анализатор разделения строк кеш-памяти** (cache line sharing) – выводит статистику по разделяемым строкам кеш-памяти
- C2C – Cache to Cache
- Local HITM – загрузка в локальную строку, находящуюся в состоянии MODIFIED
- Remote HITM – загрузка в строку, находящуюся в состоянии MODIFIED (удаленного NUMA-узла)

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    sum_local[tid] = 0.0;
    #pragma omp for nowait
    for (int i = 0; i < N; i++) {
34:        sum_local[tid] += x[i] * y[i];
    }
    #pragma omp atomic
    sum += sum_local[tid];
}
```

```
$ perf c2c record --call-graph dwarf,8192 -F max --all-user ./fs
info: Using a maximum frequency rate of 40250 Hz
[ perf record: Woken up 230 times to write data ]
[ perf record: Captured and wrote 385,035 MB perf.data (47655 samples) ]
```

```
$ perf c2c report --call-graph
```

Shared Data Cache Line Table (1 entries, sorted on Total HITMs)																						
----- Cacheline -----				Tot	----- Load Hitm -----				Total	Total	Total	----- Stores -----		----- Core Load Hit -----			- LLC Load Hit - -		- RMT Load Hit - -		--- Load Dram ----	
Index	Address	Node	PA cnt	Hitm	Total	LclHitm	RmtHitm	records	Loads	Stores	L1Hit	L1Miss	FB	L1	L2	LclHit	LclHitm	RmtHit	RmtHitm	Lcl	Rmt	
+	0	0x7ffd3abb9940	0	35232	100,00%	3	3	0	3458	3458	0	0	0	0	3454	0	1	3	0	0	0	0



Cacheline 0x7ffd3abb9940																							
----- HITM -----				-- Store Refs --				----- CL -----				----- cycles -----				Total	cpu	Symbol		Shared	Source:Line Node		
RmtHitm	LclHitm	L1 Hit	L1 Miss	Off	Node	PA cnt	Code address		rmt hitm	lcl hitm	load	records	cnt			Object							
+	0,00%	100,00%	0,00%	0,00%	0x8	0	1	0x563526f8942b	0	164	48	549	1	[.]	fun._omp_fn.0	fs						fs.c:34	0