

REAL WORLD C

Язык программирования C в реальном мире. Конвейер микропроцессора. Предсказания переходов. Пропуски по памяти.

К. Владимиров, Yadro, 2024
mail-to: konstantin.vladimirov@gmail.com

СЕМИНАР 7.1

Кеши и память (а также введение в бенчмаркинг)

Немного о бенчмаркинге

- Как выяснить сколько времени занимает цикл до $N * M$?

```
struct timespec t1, t2;  
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &t1);  
  
for (int i = 0; i < N; ++i)  
    for (int j = 0; j < M; ++j) {  
    }  
  
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &t2);
```

- Это будет всегда показывать ноль секунд. Почему?

As-if rule

Компилятор всегда работает так, чтобы наблюдаемое поведение оставалось таким, как если бы (as if) он не работал.

Что представляет собой наблюдаемое поведение?

- Accesses through volatile glvalues.
- Data written into files.
- The input and output dynamics of interactive devices.
- Компилятор имеет право делать с программой почти что угодно пока наблюдаемое поведение то же.
- Это даёт достаточную свободу оптимизаторам.

Компиляторы консервативны

```
struct timespec t1, t2;  
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &t1);  
  
for (int i = 0; i < N; ++i)  
    for (int j = 0; j < M; ++j) {  
        foo(i, j); // вызов в другой модуль  
    }  
  
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &t2);
```

- Компилятор тут ничего не сделает: он не уверен нет ли в другом модуле побочных эффектов при этом вызове.
- Проблема в том что вызов добавит нам оверхеда. Что делаем?

Компиляторы консервативны

```
struct timespec t1, t2;
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &t1);

for (int i = 0; i < N; ++i)
    for (int j = 0; j < M; ++j) {
        asm(""::"r"(i), "r"(j));
    }

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &t2);
```

- Компилятор тут ничего не сделает: он не анализирует ассемблерные вставки и не уверен что в такой вставке нет побочных эффектов.

Сила макросов

- Макрос определяет простую текстовую замену.

```
#define FIRST SECOND
```

- Заменит в тексте все вхождения токена FIRST на SECOND.

```
int FIRST = 5; // заменит имя на SECOND
```

- Но требуется полный токен или последовательность токенов.

```
int xFIRST = 5;
```

- Макросы могут брать аргументы.

```
#define ADD(X, Y) X + Y
```

Неожиданные свойства макросов

- Допустим у нас определён макрос:

```
#define ADD(X, Y) X + Y
```

- Что будет значить следующее выражение?

```
int x = ADD(2, 2) * 2;
```


Неожиданные свойства макросов

- Даже аккуратное определение макросов не спасает.

```
#define ADD(X, Y) ((X) + (Y))
```

- Тогда наивное использование кажется безопасным.

```
int x = ADD(2, 2) * 2; // Ok, yields 8
```

- Но мы можем улучшить далее.

```
#define MAX(X, Y) ((X) > (Y) ? (X) : (Y))
```

```
int z = MAX(++x, 2); // Чему тут равно z?
```

Макросы для бенчмаркинга

```
#define NOOPT(x) asm(""::"r"(x));  
for (int i = 0; i < N; ++i) {  
    NOOPT(i);  
    for (int j = 0; j < M; ++j) {  
        NOOPT(j);  
    }  
}
```

- Здесь удобный макрос NOOPT экранирует защиту от оптимизаций с помощью ассемблерной вставки.
- В итоге если мы решим сменить экранирование, нам не придётся менять код.

Простой эксперимент

```
double measure_loops(int N, int M) {  
    struct timespec time1, time2; int i, j;  
    simple_gettime(&time1);  
  
    for (int i = 0; i < N; ++i) {  
        NOOPT(i);  
        for (int j = 0; j < M; ++j) {  
            NOOPT(j);  
        }  
    }  
  
    simple_gettime(&time2);  
    return diff(time1, time2);  
}
```

Загадочный эксперимент

- Следующий код вычисляет сумму элементов двумерного массива.

```
for (j = 0; j < len; ++j)
    for (i = 0; i < ARRSZ; ++i)
        sum += arr[i][j];
```

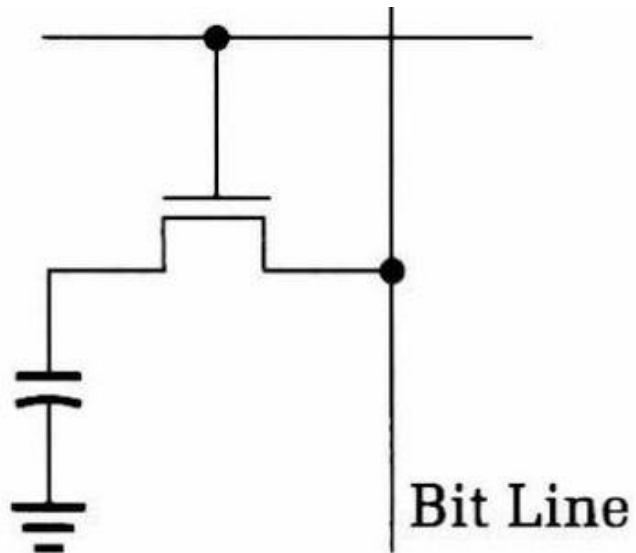
- И этот код тоже.

```
for (i = 0; i < ARRSZ; ++i)
    for (j = 0; j < len; ++j)
        sum += arr[i][j];
```

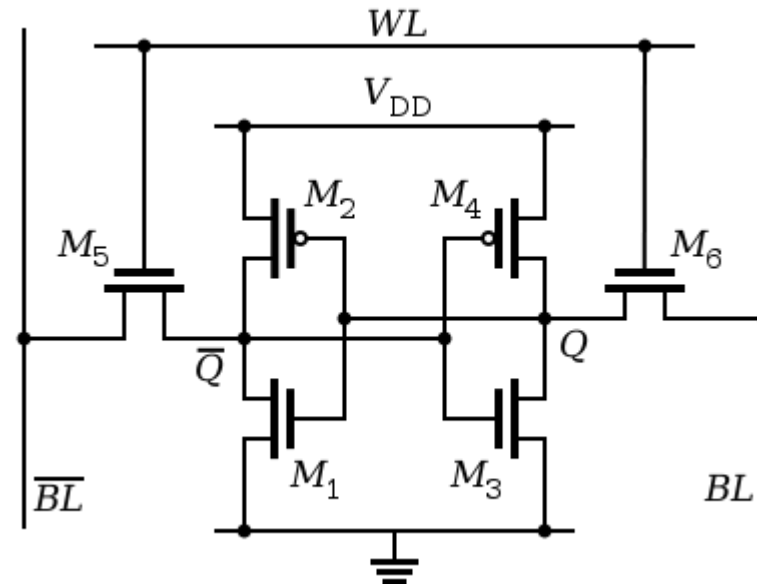
- Есть ли разница если мы забенчмаркаем? И если есть, то почему?

Память с произвольным доступом

- Грубо можно классифицировать память на динамическую и статическую



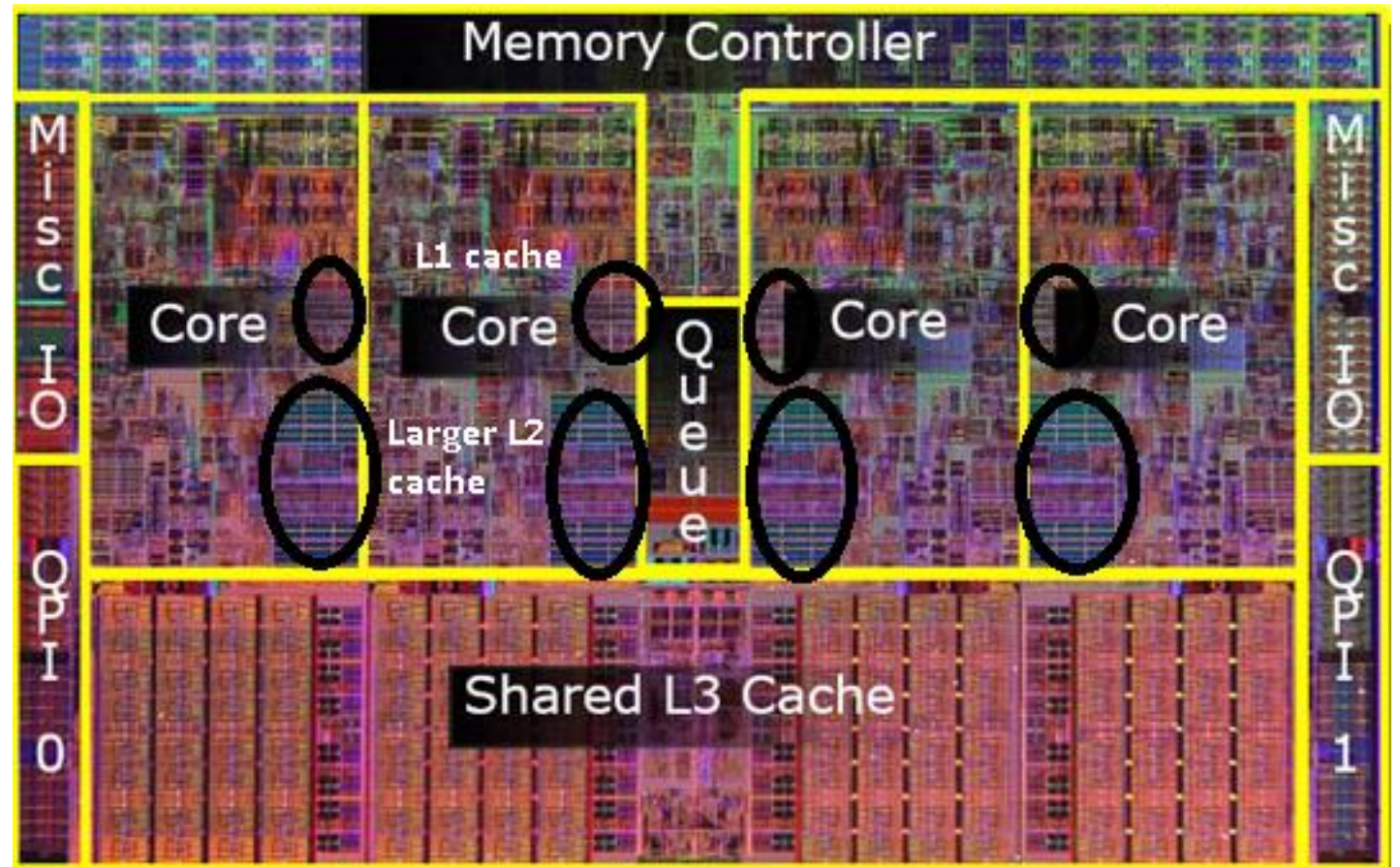
Ячейка Dynamic RAM



Ячейка Static RAM

Память с произвольным доступом

- Статическая память быстрее, но намного дороже. Поэтому то, что мы называем "оперативкой" это обычно DRAM
- В современных условиях это DDR, реже SDR
- SRAM используется, чтобы **кэшировать** недалеко от процессора часто используемые данные



Идея кэширования

- Допустим вы делаете обращение в память

```
int b = a[0]; // около 100 наносекунд
```

- Процессор предполагает что следующее обращение будет недалеко и загружает всю кэш-линию (около 64 байт) в L1 кэш

```
int c = a[1]; // около 0.5 наносекунд
```

- В современных процессорах есть много уровней кэшей и данные, которые не влезают (или вытесняются) из кэша L1 попадают в L2, а потом и в L3.
- В итоге чем активнее программа использует данные, тем быстрее у неё к ним доступ

Иерархия памяти

Вид памяти	Примерное время доступа	Примерный размер*
L1 cache	0.5 ns	256 Kb
L2 cache	7 ns	1 Mb
L3 cache	20 ns	8 Mb
RAM	100 ns	8 Gb
HDD (считать 4Kb)	150000 ns	1 Tb

Цена одного branch mispredict приблизительно равна цене одного cache miss с обращением в L2

*для coffee lake, i5-8300H

Локальность данных

- Теперь загадка развеивается. Двумерные массивы лежат в памяти непрерывным куском



- Следующий цикл обращается к каждому n-ному элементу

```
for (j = 0; j < len; ++j)
    for (i = 0; i < ARRSZ; ++i)
        sum += arr[i][j];
```

- Он делает cache miss **каждый** раз
- Разумеется, если его переписать, всё становится куда лучше

Более сложный пример

- Представьте, что у вас есть код, выполняющий перемножение матриц

```
void matrix_mult(const int *A, const int *B, int *C,
                 int AX, int AY, int BY) {
    for(int i = 0; i < AX; i++)
        for(int j = 0; j < BY; j++) {
            C[i * BY + j] = 0;
            for(int k = 0; k < AY; k++)
                C[i * BY + j] += A[i * AY + k] * B[k * BY + j];
        }
}
```

- Будет ли здесь влиять перестановка строчек внешних циклов? Если да то как, если нет, то почему?

Математика идёт на помощь

- Запишем $(AB)_{ij} = \sum_{k=0}^{N-1} a_{ik} b_{jk}^T$
- Теперь можно заметить, что нелокальность вычислений проистекает из того факта, что обращения к второй матрице происходят в транспонированном виде
- Ради улучшения локальности мы можем завести дополнительную матрицу и транспонировать её

```
size_t bsz = BIG_BY * BIG_AY * sizeof(int);
int *tmp = (int *) malloc(bsz);
for(int i = 0; i < AY; i++)
    for(int j = 0; j < BY; j++)
        tmp[j * AY + i] = B[i * BY + j];

for(int i = 0; i < AX; i++)
    for(int j = 0; j < BY; j++) {
        C[i * BY + j] = 0;
        for(int k = 0; k < AY; k++)
            C[i * BY + j] +=
                A[i * AY + k] * tmp[j * AY + k];
    }
free(tmp);
```


Обсуждение

- Из-за предсказания бранчей, клеточное перемножение на замерах ведёт себя чуть хуже, чем подход с транспонированной матрицей
- Кроме того там накладывается важное ограничение: размеры матриц должны нацело делиться на SM иначе нужно писать чуть больше кода чтобы учесть краевые эффекты
- Зато этот метод не требует дополнительной памяти

Задача

- Предположим, что вы не знаете размер кэшей на вашем компьютере
- И у вас нет документации
- Интернета чтобы её поискать тоже нет
- Как бы вы его выяснили?

Метод Монте-Карло

- Предположим, у нас есть массив размера N
- Если к этому массиву обращаться (например инкрементировать элементы) последовательно и замерить время t_1
- А потом сделать такое же количество обращений по случайным адресам и замерить время t_2
- То что нам скажет соотношение t_1 и t_2 для разных N ?
- Можем ли мы использовать этот метод для оценки эффективного размера кэшей на своей машине?
- Проведите соотв. эксперименты и замеры

Эффекты кэшей и асимптотика

- Плохая кэш-локальность может снизить производительность в 20-30 раз
- Предположим, что у вас есть выбор между алгоритмом $O(N \log N)$ с хорошей локальностью данных и алгоритмом $O(N)$ с плохой локальностью
- Каким должен быть выбор для разных N ?

Кэш как структура данных

- Есть страницы по 64 байта, включая номер

```
struct page {  
    int index;        // page index: 1, 2, ... n  
    char data[60];    // page data  
};
```

- Также существует медленная функция

```
void slow_get_page(int n, struct page *p);
```

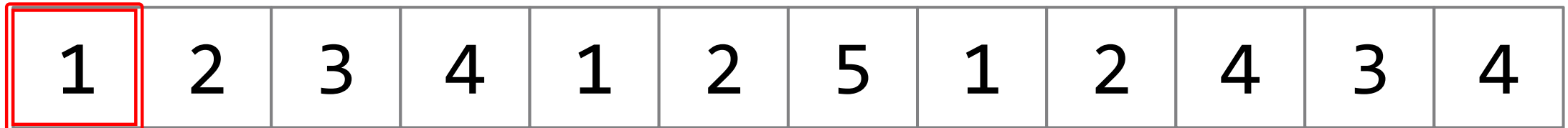
- Необходимо завести кэш для обращений к страницам
- Считаем, что всего в кэше места не больше, чем на m страниц, m много меньше n

Кэш как структура данных

- Какую структуру данных выбрать для кеша?
- Какую стратегию кэширования выбрать?
- Например у нас есть место на 3 страницы и к нам поступают запросы:
1, 2, 3, 4, 1, 2, 5, 1, 2, 4, 3, 4
- В какой момент страница **кэшируется**?
- В какой момент страница **вытесняется** из кэша?

Стратегия LRU (least recently used)

- Если запрошенный элемент найден, он перемещается вперёд.
- Если нет, он помещается вперёд а последний вытесняется.



Стратегия LRU (least recently used)

- Если запрошенный элемент найден, он перемещается вперёд.
- Если нет, он помещается вперёд а последний вытесняется.

1	2	3	4	1	2	5	1	2	4	3	4
---	---	---	---	---	---	---	---	---	---	---	---

2	1		
---	---	--	--

Стратегия LRU (least recently used)

- Если запрошенный элемент найден, он перемещается вперёд.
- Если нет, он помещается вперёд а последний вытесняется.

1	2	3	4	1	2	5	1	2	4	3	4
---	---	---	---	---	---	---	---	---	---	---	---

3	2	1	
---	---	---	--

Стратегия LRU (least recently used)

- Если запрошенный элемент найден, он перемещается вперёд.
- Если нет, он помещается вперёд а последний вытесняется.

1	2	3	4	1	2	5	1	2	4	3	4
---	---	---	---	---	---	---	---	---	---	---	---

4	3	2	1
---	---	---	---

Стратегия LRU (least recently used)

- Если запрошенный элемент найден, он перемещается вперёд.
- Если нет, он помещается вперёд а последний вытесняется.

1	2	3	4	1	2	5	1	2	4	3	4
---	---	---	---	---	---	---	---	---	---	---	---

1	4	3	2
---	---	---	---

Стратегия LRU (least recently used)

- Если запрошенный элемент найден, он перемещается вперёд.
- Если нет, он помещается вперёд а последний вытесняется.

1	2	3	4	1	2	5	1	2	4	3	4
---	---	---	---	---	---	---	---	---	---	---	---

2	1	4	3
---	---	---	---

Стратегия LRU (least recently used)

- Если запрошенный элемент найден, он перемещается вперёд.
- Если нет, он помещается вперёд а последний вытесняется.

1	2	3	4	1	2	5	1	2	4	3	4
---	---	---	---	---	---	---	---	---	---	---	---

5	2	1	4
---	---	---	---

Стратегия LRU (least recently used)

- Если запрошенный элемент найден, он перемещается вперёд.
- Если нет, он помещается вперёд а последний вытесняется.

1	2	3	4	1	2	5	1	2	4	3	4
---	---	---	---	---	---	---	---	---	---	---	---

1	5	2	4
---	---	---	---

Стратегия LRU (least recently used)

- Если запрошенный элемент найден, он перемещается вперёд.
- Если нет, он помещается вперёд а последний вытесняется.

1	2	3	4	1	2	5	1	2	4	3	4
---	---	---	---	---	---	---	---	---	---	---	---

2	1	5	4
---	---	---	---

Стратегия LRU (least recently used)

- Если запрошенный элемент найден, он перемещается вперёд.
- Если нет, он помещается вперёд а последний вытесняется.

1	2	3	4	1	2	5	1	2	4	3	4
---	---	---	---	---	---	---	---	---	---	---	---

4	2	1	5
---	---	---	---

Стратегия LRU (least recently used)

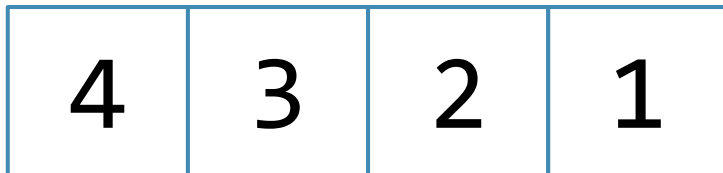
- Если запрошенный элемент найден, он перемещается вперёд.
- Если нет, он помещается вперёд а последний вытесняется.

1	2	3	4	1	2	5	1	2	4	3	4
---	---	---	---	---	---	---	---	---	---	---	---

3	4	2	1
---	---	---	---

Стратегия LRU (least recently used)

- Если запрошенный элемент найден, он перемещается вперёд.
- Если нет, он помещается вперёд а последний вытесняется.

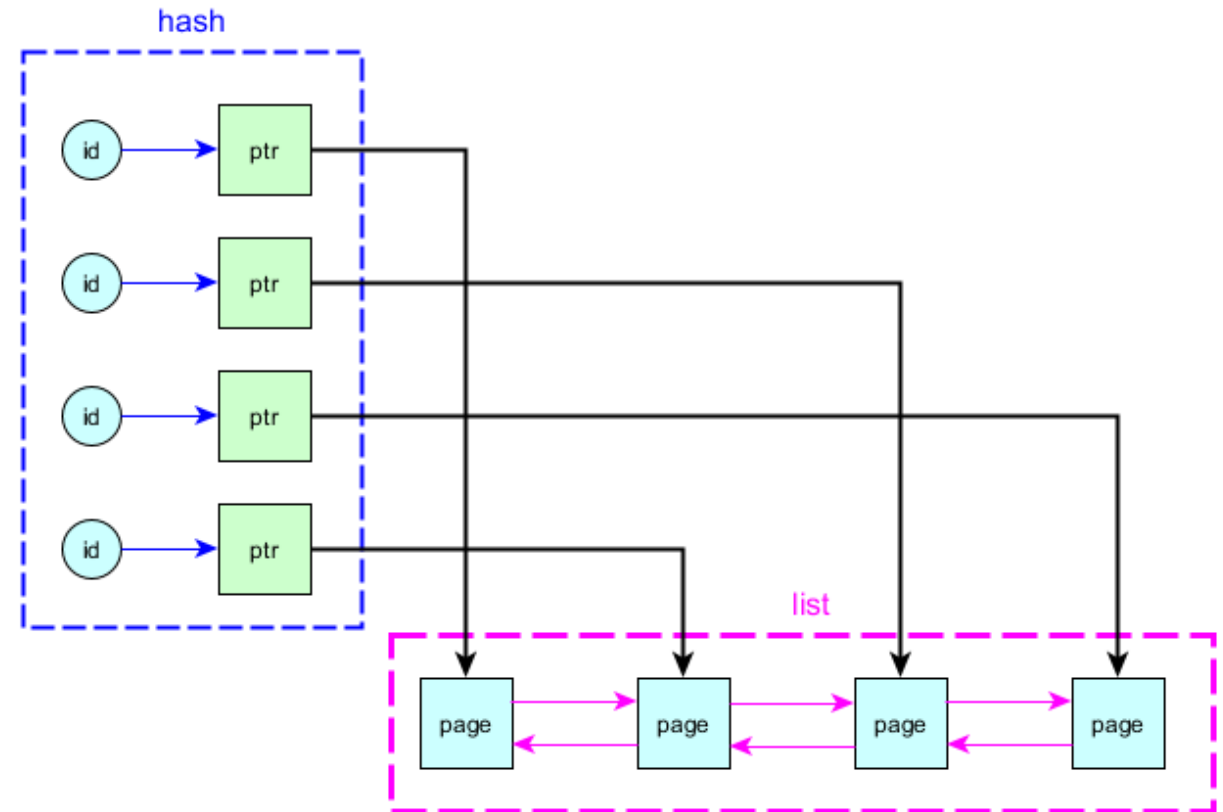


Обсуждение

- Какие структуры данных нам понадобятся чтобы сделать LRU cache?

Обсуждение

- Какие структуры данных нам понадобятся чтобы сделать LRU cache?
- Двусвязный список для собственно кеша.
- Хеш-таблица для того, чтобы быстро определять кеширован ли элемент.



Problem LC – LRU кэш

- Необходимо написать код функции

```
struct page {  
    int index;        // page index: 1, 2, ... n  
    char data[60];    // page data  
};  
  
void slow_get_page(int n, struct page *p);  
  
void get_page(int id, struct page *p) {  
    // TODO: заполнить структуру *p используя кэш  
}
```

- Функция должна обеспечивать переиспользование страниц не худшее, чем при стратегии LRU

Командный проект

- В группах по 3 человека
- Реализуйте LRU, пройдите Problem LC.
- Реализуйте один из более сложных алгоритмов, сравните производительность с LRU.
- Подготовьте слайды к защите проекта.

Чек-лист на проект

- Проверьте что вы до защиты послали мне репозиторий, там есть код, его можно скачать, он скомпилируется как минимум под Windows и под Linux.
- Проверьте, что в вашем репозитории есть коммиты от всех членов вашей команды и работа каждого видна.
- Проверьте, что вы принимаете входные данные в формате problem LC.
- Проверьте, что вы правильно разбили проект по модулям.
- Проверьте, что у вас есть система сборки и достаточное количество тестов, при этом среди ваших тестов есть и написанные руками и случайно сгенерированные.

Чек-лист на проект

- Проверьте, что ваш код отформатирован в едином стиле и каждый модуль откомментирован хотя бы в заголовке.
- Проверьте, что вы реализуете именно тот алгоритм который вам выдан и при этом понимаете что вы сделали и как работает алгоритм.
- Проверьте, что ваша реализация эффективна (вы должны быть не менее эффективны чем базовый LRU)
- Проверьте, что у вас есть тесты эффективности на разных паттернах и сравнение с базовым LRU.
- Проверьте, что вы проходите valgrind и у вас нет очевидных ошибок при работе с динамической памятью.

СЕМИНАР 7.2

Предсказание переходов и конвейер процессора

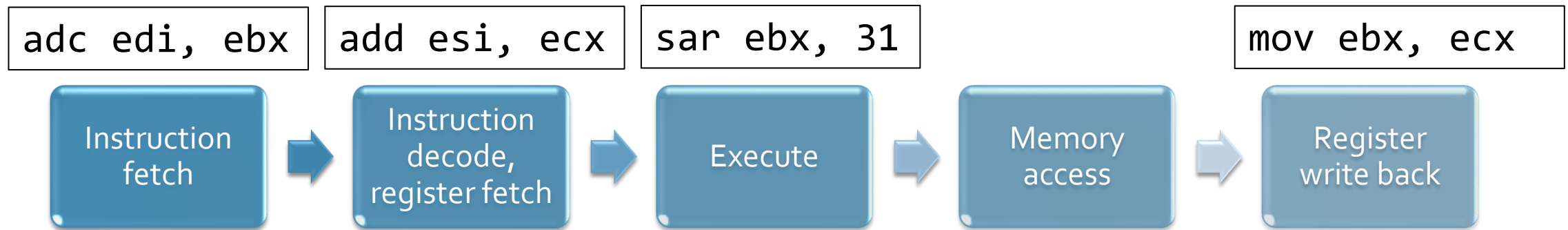
Загадочная проблема

- Следующий код суммирует все элементы массива, меньшие, чем 128.

```
for (j = 0; j < len; ++j)
    if (arr[j] > 128)
        sum += arr[j];
```

- Проблема в том, что для несортированных массивов он (без изменений в самом коде) работает почти в четыре раза медленнее, чем для сортированных.
- См. пример **bmystery.c** в файлах к семинару.
- Значит надо понять **почему** происходит замедление и **как** это исправить.

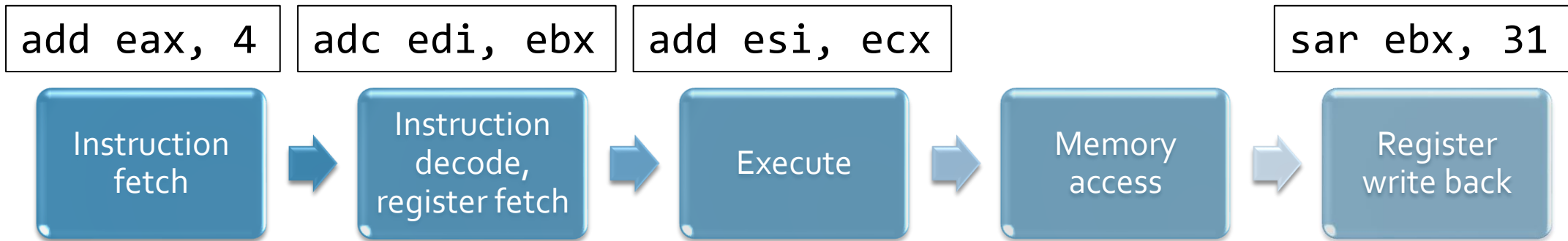
Конвейер микропроцессора



```
mov ebx, ecx
sar ebx, 31
add esi, ecx
adc edi, ebx
add eax, 4
cmp eax, edx
```

executed, writes ebx back
executing (waiting ebx)
decoded
fetch

Конвейер микропроцессора



```
mov    ebx, ecx
sar    ebx, 31
add    esi, ecx
adc    edi, ebx
add    eax, 4
cmp    eax, edx
```

executed, writes ebx back
executing (waiting ebx)
decoded
fetch

Обсуждение

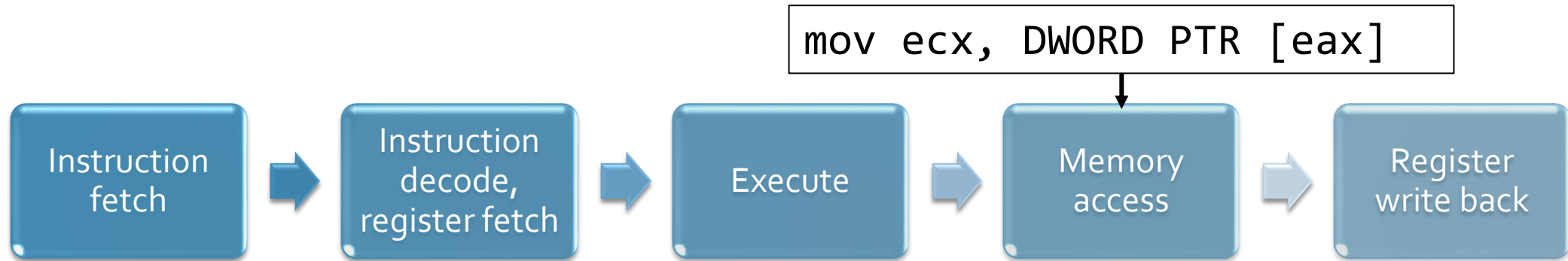
- Что если в конвейере встретится инструкция условного перехода?

```
    mov     ecx, DWORD PTR [eax]
    cmp     ecx, 128
    jle     L3
    mov     ebx, ecx
    sar     ebx, 31
    add     esi, ecx
    adc     edi, ebx
L3:
    mov     ecx, edx
```

doing memory access (LSQ)
executing (waiting ecx)
decoded (known to be jump)
fetched?

fetched?

Конвейер микропроцессора



```
mov    ecx, DWORD PTR [eax]
cmp    ecx, 128
jle    L3
mov    ebx, ecx
```

doing memory access (LSQ)
executing (waiting ecx)
decoded (known to be jump)
fetch?

Предсказание переходов

- Теперь странности исчезают

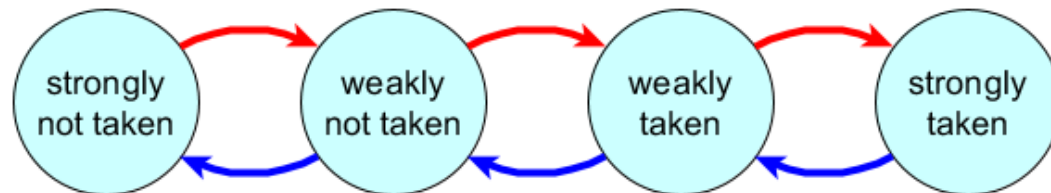
```
for (j = 0; j < len; ++j)
    if (arr[j] > 128) // тут случайное значение
        sum += arr[j];
```

- Вероятность правильного предсказания перехода теперь... а кстати, какая?
- Это очень сильно зависит от конкретного механизма предсказания переходов.

Бранч предиктор исходит из истории

```
.....  
0x09c .....  
0x100 cmp ecx, 128  
0x104 jle +14  
0x108 mov ebx, ecx  
0x10c sar ebx, 31  
0x110 add esi, ecx  
0x114 adc edi, ebx  
0x118 mov ecx, edx  
0x11c .....  
.....
```

Адрес	История
0x60	00000000
0x240	11
0x104	11001010
0x24	11100
0x3ae	10000000
0x304	0



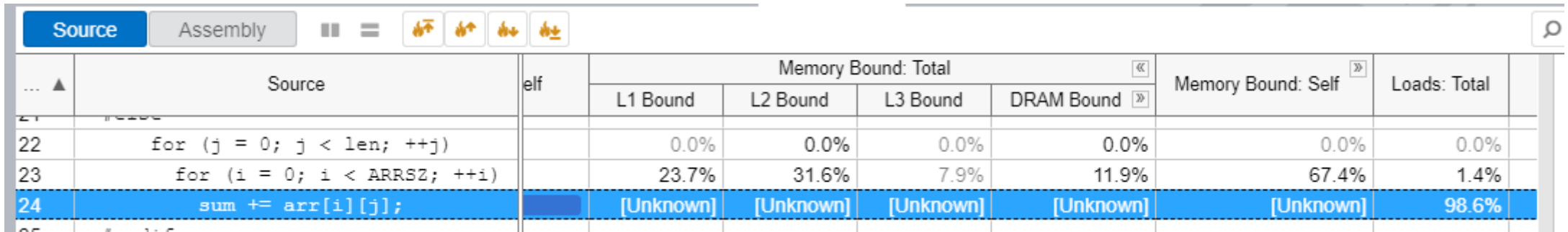
Профилировщики

- Существует большое количество программ, которые помогают в анализе производительности, включая микроархитектурные эффекты.

... ▲ ...	Source						
		«			Bad Speculation «		Back-End Bound x
		«			Branch Mispredict	Ma... Cle...	
		D	(Info) DSB Coverage	(Info) ...			
15	for (i = 0; i < NITER; ++i)						
16	for (j = 0; j < len; ++j)	0%	7.9%	0.0%	11.5%	0.0%	6.4%
17	if (arr[j] > 128)	0%	18.9%	0.0%	0.0%	0.0%	12.5%
18	sum += arr[j];	0%	73.0%	0.0%	23.5%	0.0%	11.2%
19							
20	qsum = sum;						

Профилировщики

- Они же могут помочь и с кешами



The screenshot shows a performance profiler interface with a 'Source' tab selected. The interface includes a toolbar with icons for source, assembly, and various performance metrics. Below the toolbar is a table with columns for source code, memory bound statistics, and loads. The table has a header row with 'Source', 'elf', 'Memory Bound: Total' (subdivided into L1 Bound, L2 Bound, L3 Bound, and DRAM Bound), 'Memory Bound: Self', and 'Loads: Total'. The table contains three rows of data. The third row, which is highlighted in blue, shows the source code 'sum += arr[i][j];' and memory bound statistics: L1 Bound [Unknown], L2 Bound [Unknown], L3 Bound [Unknown], DRAM Bound [Unknown], Memory Bound: Self [Unknown], and Loads: Total 98.6%.

...	Source	elf	Memory Bound: Total				Memory Bound: Self	Loads: Total
			L1 Bound	L2 Bound	L3 Bound	DRAM Bound		
22	for (j = 0; j < len; ++j)		0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
23	for (i = 0; i < ARRSZ; ++i)		23.7%	31.6%	7.9%	11.9%	67.4%	1.4%
24	sum += arr[i][j];		[Unknown]	[Unknown]	[Unknown]	[Unknown]	[Unknown]	98.6%

- К сожалению обучение использованию профилировщика выходит за пределы курса, но вы можете разобраться самостоятельно.
- Рекомендую специализированные (Inte VTune и подобные).

Давайте посмотрим ассемблер

```
    movsx rsi, ebp                // rsi = len
    test  ebp, ebp                // if (len == 0)
    jle   .LoopExit               // goto LoopExit;
    xor   eax, eax                // j = 0;
.L1: movsx rdx, DWORD PTR [rbx+rax*4] // rdx = arr[j];
    cmp   edx, 128                // if (arr[j] <= 128)
    jle   .L2                     // goto L2;
    add   rcx, rdx                 // sum += rdx;
.L2: add   rax, 1                  // j += 1;
    cmp   rsi, rax                 // if (j != len)
    jne   .L1                      // goto L1;
```

- Есть ли идеи что тут можно улучшить?

Хитрая оптимизация

- Используя знание о происходящем, можно вообще убрать переход.

```
for (j = 0; j < len; ++j) {  
    if (arr[j] > 128)  
        sum += arr[j];  
}
```

```
for (j = 0; j < len; ++j) {  
    int tmp = (arr[j] > 128);  
    sum += (arr[j] * tmp);  
}
```


Снова посмотрим ассемблер

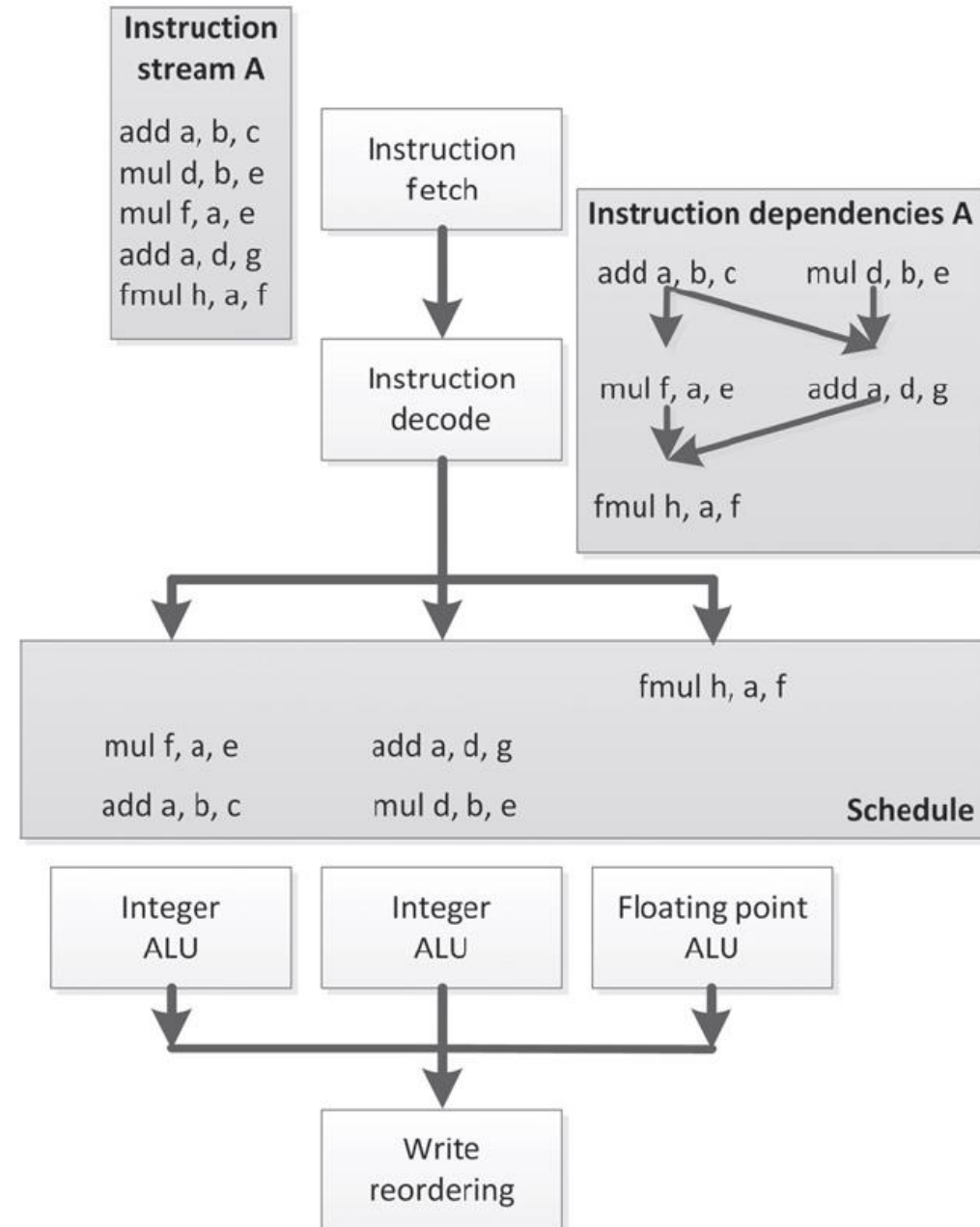
```
    movsx rsi, ebp
    test  ebp, ebp
    jle   .LoopExit
    xor   edx, edx
.Loop: mov  ecx, DWORD PTR [rbx+rdx*4]
    xor   eax, eax
    cmp   ecx, 128
    setg  al
    imul  eax, ecx
    cdqe
    add   rdx, 1
    cmp   rsi, rdx
    jne   .Loop
```

Обсуждение

- В данном случае стало в несколько раз лучше
- Должны ли мы рассматривать возможность делать такого рода оптимизации?
- Не опасно ли их делать?

Out of order

- Слайд с конвейером мог ввести в заблуждение
- На самом деле инструкции исполняются не так уж линейно
- Важная часть конвейера – **планировщик** который раскидывает инструкции по ALU, учитывая их специфику и взаимосвязи
- Это делает mispredict **ещё хуже**



Обсуждение



- Очень долгая и сложная стадия это доступ к памяти.
- Что если бы мы смогли заранее туда сбегать пока конвейер делает что-то другое?

Prefetch

- Техника **предвыборки (prefetch)** служит для того, чтобы подкачать в кэш данные

```
for (int i = 0; i < ARRSZ; ++i) {  
    a[i] = a[i] + b[i];  
    __builtin_prefetch(a[i+1]);  
    __builtin_prefetch(b[i+1]);  
}
```

- Здесь до перехода будут подкачаны значения для вычисления следующей итерации цикла

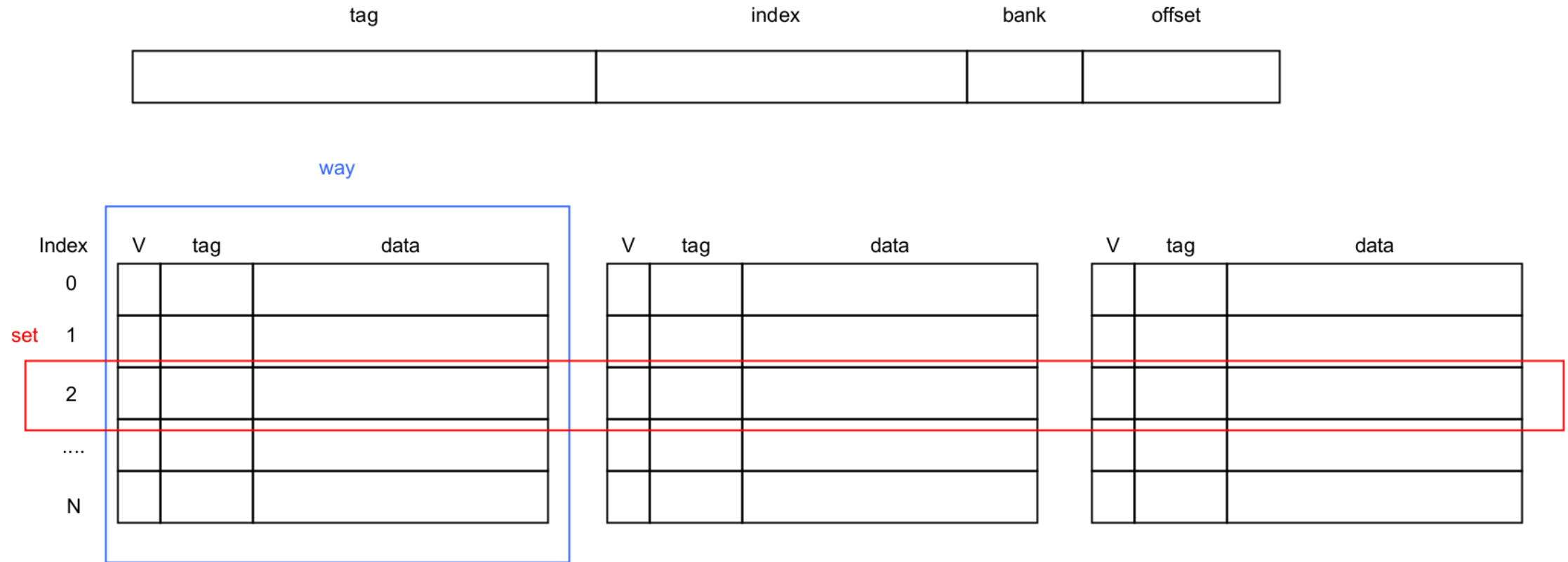
Instruction cache

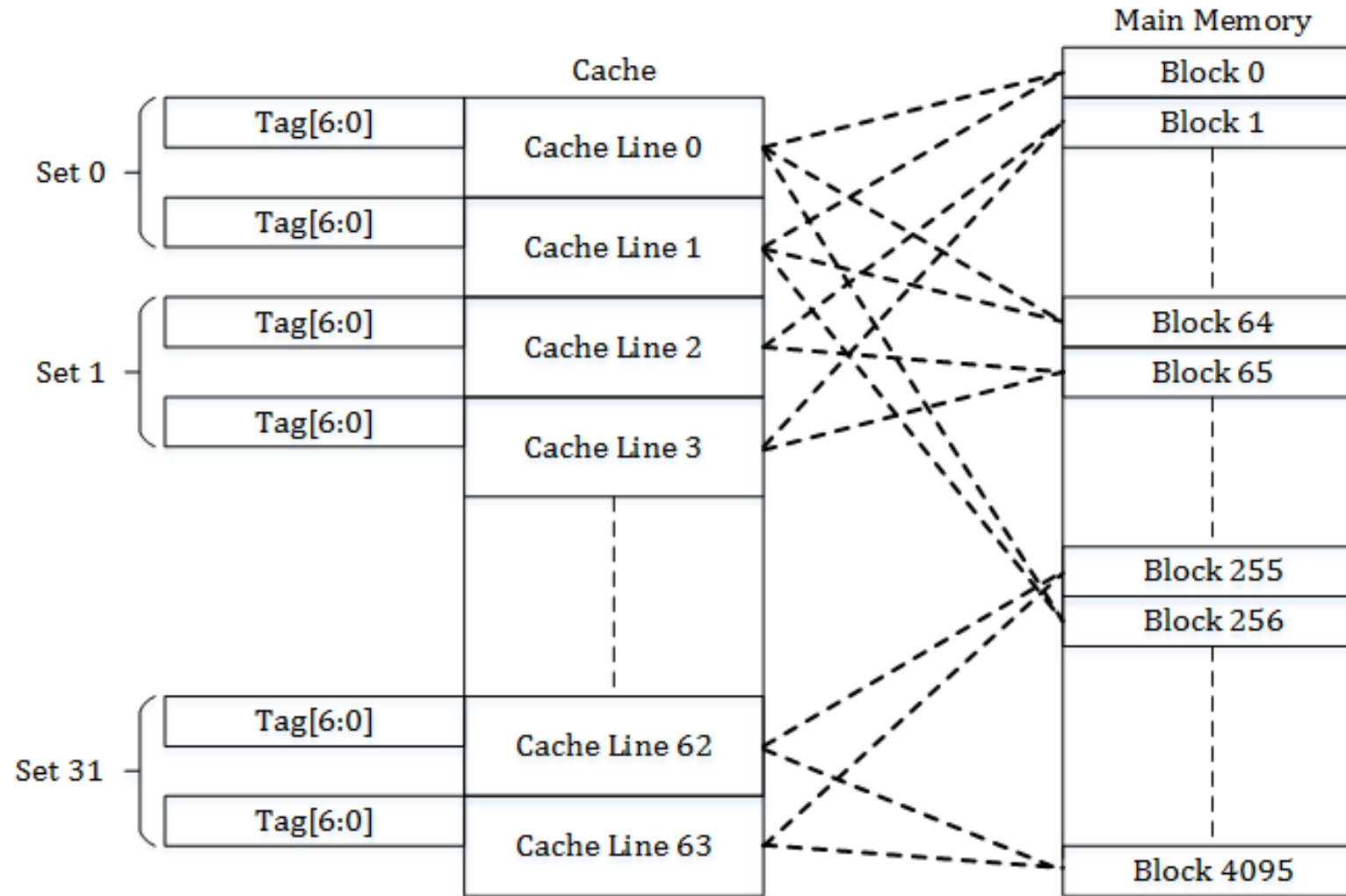
- Инструкции это тоже данные.
- Конвейер декодировав инструкцию сохраняет её в кэш инструкций.
- Таким образом, кроме branch mispredict можно рассматривать instruction cache miss.
- Но обычно в процессоре достаточно большой кэш инструкций: речь идёт о чём-то около 32 килобайт на каждое ядро и поэтому наглядно увидеть эффекты на простом приложении сложно.
- Сможете ли вы самостоятельно придумать эксперимент на кэш инструкций?

Загадочная проблема

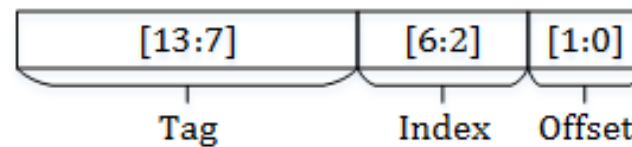
- Что быстрее:
- бинарный поиск в массиве из 2^{25} элементов
- бинарный поиск в массиве из $2^{25} + 2^{10}$ элементов
- Кажется бы несомненно чем меньше массив тем быстрее в среднем бинарный поиск?

Немного о настоящих кешах





Memory Size = 16Kbytes
 Memory Block Size = 4 bytes
 Cache Size = 256 bytes
 Block Size = 4 bytes
 Associativity = 2
 Number of Sets = 32



Очевидное решение

- Бинарный поиск с небольшим смещением будет попадать в разные кеш-линии из-за **ассоциативности** кеша.

Литература

- [C11] ISO/IEC – "Information technology – Programming languages – C", 2011
- [K&R] Brian W. Kernighan, Dennis Ritchie – The C programming language, 1988
- [SDM] Intel Software Developer Manual: [intel-sdm](#)
- [Patterson] John Hennesy, David Patterson – Computer Architecture: A Quantitative Approach, 2011
- [Drepper] Ulrich Drepper – What Every Programmer Should Know About Memory, 2007
- [Linden] Peter van der Linden – Expert C Programming: Deep C Secrets, 1994