

REAL WORLD C

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

К. Владимиров, Intel, 2019
mail-to: konstantin.vladimirov@gmail.com

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
- Предположим, необходимо эффективно обрабатывать именно несортированные массивы
- 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
fetches

Проблема переходов



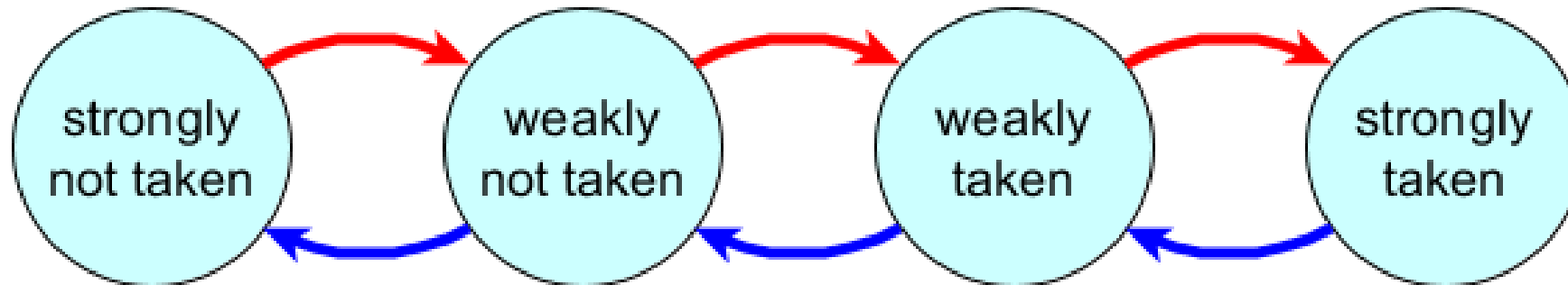
```
mov    ecx, DWORD PTR [eax]
cmp    ecx, 128
jle    L3
mov   ebx, ecx
sar    ebx, 31
add    esi, ecx
adc    edi, ebx
```

L3:

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

Предсказание переходов

- Используется конечный автомат, работающий на истории переходов



- Проследим работу на простом цикле

```
for (j = 0; j < 100; ++j) {  
    x = (x + j) & 78;  
}
```

- Предсказание для достаточно больших циклов почти всегда верное

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?
- Точный ответ дать тяжело. К тому же, модель автомата на прошлом слайде очень условная: в современных процессорах стоят **гораздо** более сложные схемы
- Но всегда можно посмотреть, используя профайлер

Предсказание переходов

- Следующий скриншот получен с Intel Vtune

...	Source	Bad Speculation					Back-End Bound
		D	(Info) DSB Coverage	(Info) ...	Branch Mispredict	Ma... Cle...	
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;						

- Итак, четверть бранчей не угадывается. Что делать?

Хитрая оптимизация

- Используя знание ассемблера, 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);  
}
```


Хитрая оптимизация

- Используя знание ассемблера, можно вообще убрать переход

L4:

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

L3:

```
add    eax, 4
cmp    eax, edx
jne    L4
```

L3:

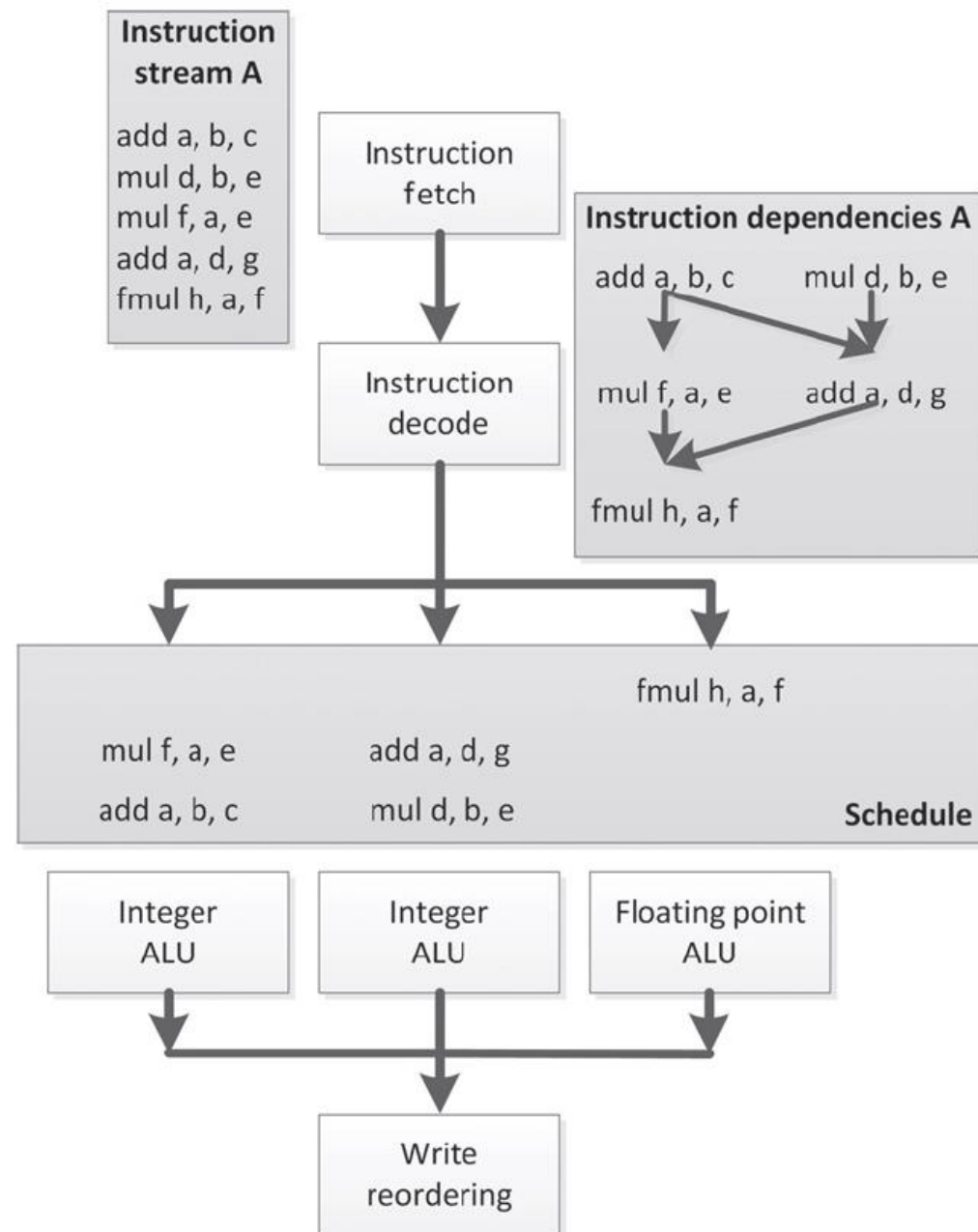
```
mov    edx, DWORD PTR [ecx]
xor    eax, eax
cmp    edx, 128
setg  al
imul  eax, edx
cdq
add    esi, eax
adc    edi, edx
add    ecx, 4
cmp    ebx, ecx
jne    L3
```

Обсуждение

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

Интермедия: 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**



Загадочная проблема #2

- Следующий код вычисляет сумму элементов двумерного массива

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

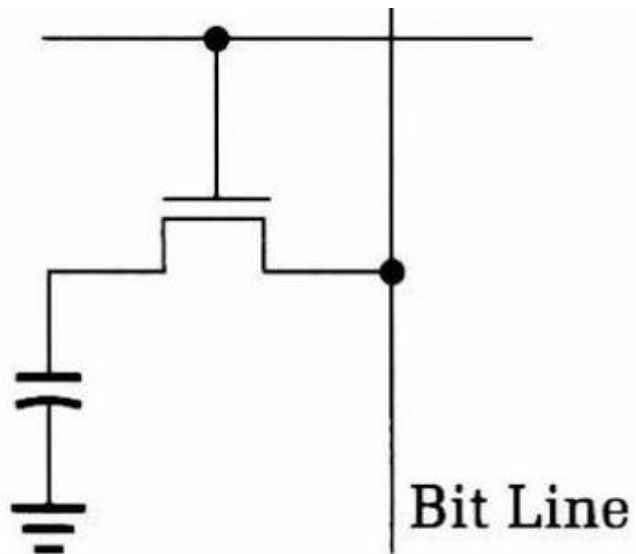
- Он начинает работать в несколько раз (!) быстрее, если переставить местами циклы

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

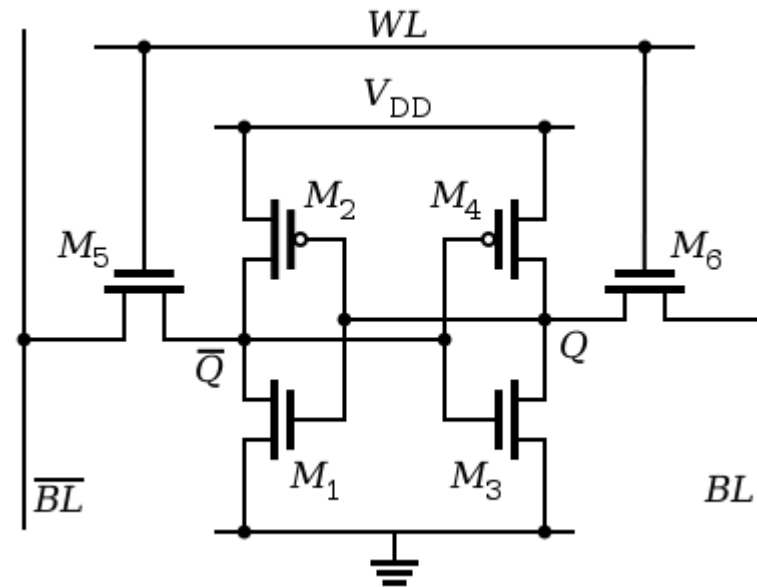
- Почему? Ответ будет долгий.

Память с произвольным доступом

- Грубо можно классифицировать память на динамическую и статическую



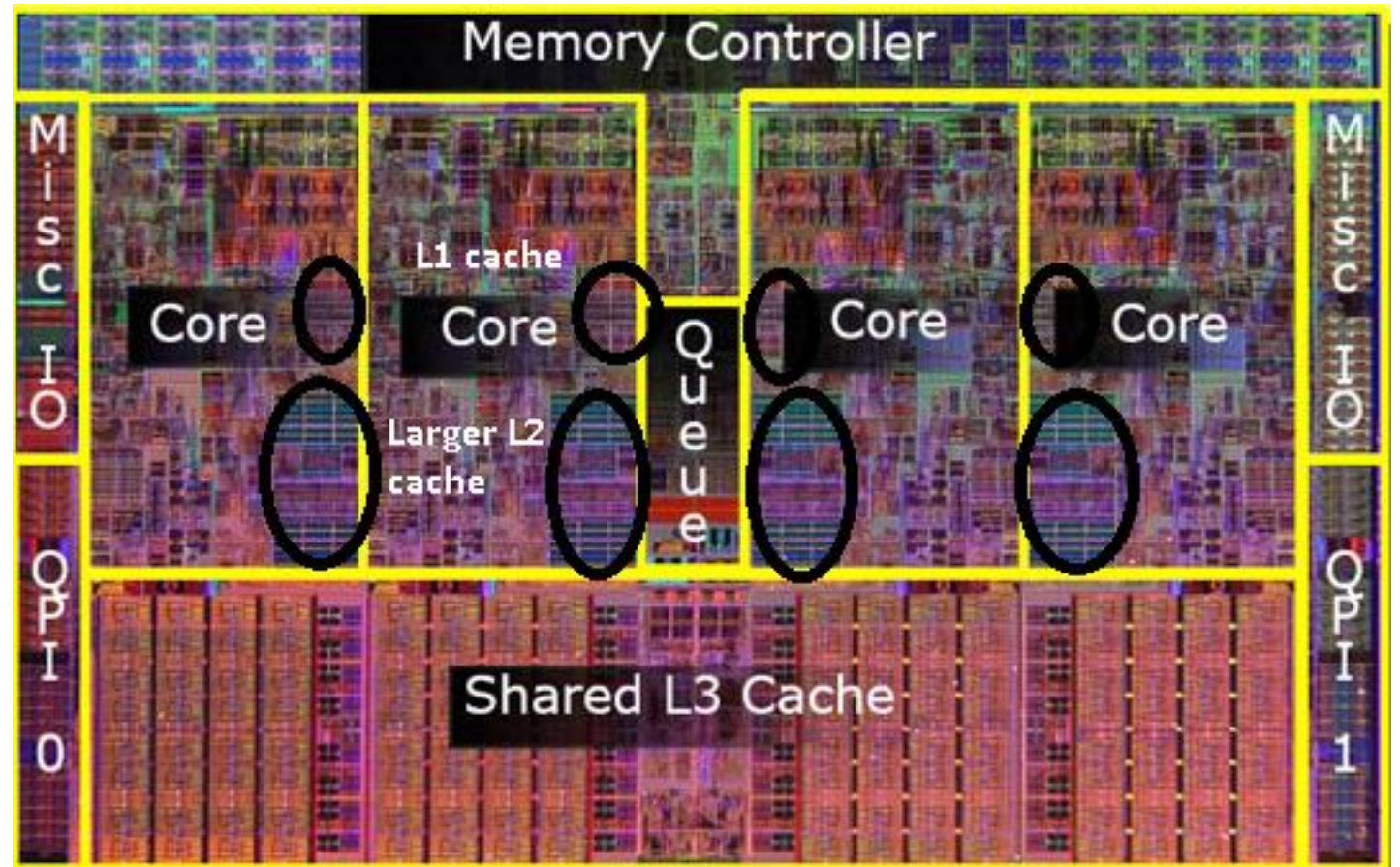
Ячейка Dynamic RAM



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



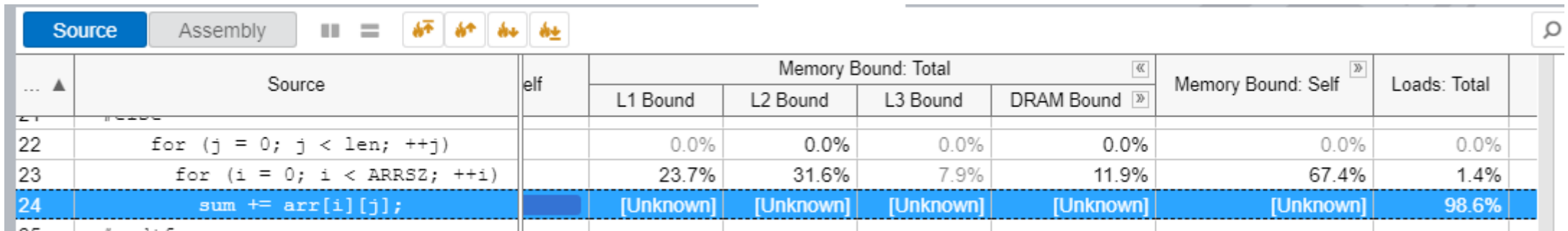
- Следующий цикл обращается к каждому n-ному элементу

```
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

Использование профилировщика

- Как обычно профилировка позволяет показать довольно точное распределение промахов по уровням кэшей для каждой строчки



The screenshot shows a profiler interface with a table of memory bound statistics. The table has columns for Source, elf, Memory Bound: Total (L1 Bound, L2 Bound, L3 Bound, DRAM Bound), Memory Bound: Self, and Loads: Total. Line 24 is highlighted in blue, showing a high percentage of cache misses (98.6% in Loads: Total).

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%

- После этого можно смотреть код и делать выводы
- Обратите внимание: на скриншоте из-за погрешности отладочной информации мы видим смещение на одну строчку

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 ветвей, 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, 3, 4, 3
- В какой момент страница **кэшируется**?
- В какой момент страница **вытесняется** из кэша?

Кэш как структура данных: LRU

- Стратегия LRU (least recently used) подразумевает очередь в которой элементы индексируются хеш-таблицей
- Если запрошенный элемент найден, он передвигается вперёд, если не найден, то добавляется спереди, а задний вытесняется
- Например у нас есть место на 4 страницы и к нам поступают запросы:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 3

- Тогда кэш меняется так

1 → 2, 1 → 3, 2, 1 → 4, 3, 2, 1 → 1, 4, 3, 2 → 2, 1, 4, 3
→ 5, 2, 1, 4 → 1, 5, 2, 4 → 2, 1, 5, 4 → 3, 2, 1, 5
→ 4, 3, 2, 1 → 3, 4, 2, 1

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

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?

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