Железные счётчики
на страже производительности

Sergey Kuksenko
sergey.kuksenko@oracle.com, @kuksenk0

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

Введение

Введение: стандартный дисклеймер

1. Computer Science → Software Engineering
   - Строим приложения по функциональным требованиям
   - В большой степени абстрактно, в “идеальном мире”
   - Рассуждения при помощи формальных методов

2. Performance Engineering
   - “Real world strikes back!”
   - Исследуем взаимодействия софта с железом на типичных данных
   - Эффективно предсказывается уже мало что
   - Рассуждения при помощи формальных методов

Методология

Где-то в идеальном мире

```
Customer customer;
Developer developer;
Product product = developer.implement();
customer.set(product);
Money bucks = customer.getMoney();
developer.set(bucks);
developer.drink();
```

На самом деле

```
Customer customer;
Developer developer;
Product product = developer.implement();
customer.set(product);
while( !customer.isHappy() ) {
   product = developer.fix(product);
   customer.set(product);
}
Money bucks = customer.getMoney();
developer.set(bucks);
developer.drink();
```

Оптимизируем

```
Customer customer;
Developer developer;
Product product = developer.implement();
customer.set(product);
while( !customer.isHappy() ) {
   while( !product.isGoodEnough() ) {
       product = developer.fix(product);
   }
   customer.set(product);
}
Money bucks = customer.getMoney();
developer.set(bucks);
developer.drink();
```

Методология: Performance Loop

```
...
while(true) {
    Score score = measure(product);
    if ( score.isGoodEnough() ) {
        break;
    }
    PerfData data = developer.gatherPerfData(product);
    Bottleneck issue = developer.analyse(data);
    Solution solution = developer.think(issue);
    product = developer.fix(product, solution);
}
...
```

Методология: ЧГК подход (WWH approach)

What?

Методология: ЧГК подход (WWH approach)

What? $\Rightarrow$ Where?

Методология: ЧГК подход (WWH approach)

What? $\Rightarrow$ Where? $\Rightarrow$ How?

Методология: ЧГК подход (WWH approach)

What? $\Rightarrow$ Where? $\Rightarrow$ How?

- Что мешает работать быстрее? (monitoring)
- Где это находится? (profiling)
- Как это исправить? (fixing)

Методология: Top-Down

- Системный уровень
  - OS, сеть, диск, процессор/память
- Уровень JVM
  - GC/Heap, JIT, classloading
- Уровень приложения
  - алгоритм, многопоточность, синхронизация, API
- Микроархитектурный уровень
  - caches, code/data alignment, pipeline stalls

Методология: Mindmap

Методология: Mindmap

http://shipilev.net/talks/
devoxx-Nov2012-perfMethodology-mindmap.pdf

Методология: Top-Down

Мониторим (e.g. mpstat)

Методология: Top-Down

Мониторим (e.g. mpstat)

- Много %sys $\Rightarrow$ . . .

$\Downarrow$

Методология: Top-Down

Мониторим (e.g. mpstat)

- Много %sys $\Rightarrow$ ...

  $\Downarrow$

- Много %irq, %soft $\Rightarrow$ ...

  $\Downarrow$

Методология: Top-Down

Мониторим (e.g. mpstat)

- Много %sys $\Rightarrow$ ...

  $\Downarrow$
- Много %irq, %soft $\Rightarrow$ ...

  $\Downarrow$
- Много %iowait $\Rightarrow$ ...

  $\Downarrow$

Методология: Top-Down

Мониторим (e.g. mpstat)

- Много %sys $\Rightarrow$ ...

  $\Downarrow$
- Много %irq, %soft $\Rightarrow$ ...

  $\Downarrow$
- Много %iowait $\Rightarrow$ ...

  $\Downarrow$
- Много %idle $\Rightarrow$ ...

  $\Downarrow$

Методология: Top-Down

Мониторим (e.g. mpstat)

- Много %sys $\Rightarrow$ ...
  $\Downarrow$
- Много %irq, %soft $\Rightarrow$ ...
  $\Downarrow$
- Много %iowait $\Rightarrow$ ...
  $\Downarrow$
- Много %idle $\Rightarrow$ ...
  $\Downarrow$
- **Много %user**

CPU Utilization

- О чём говорит ∼100% CPU Utilization?

CPU Utilization

- О чём говорит ∼100% CPU Utilization?
  - Как часто OS ставит процессы на исполнение.

CPU Utilization

- О чём говорит ∼100% CPU Utilization?
  - Как часто OS ставит процессы на исполнение.

Профилирование покажет «ГДЕ» программа проводит время, без ответа на вопрос «ПОЧЕМУ».

Кто виноват?

Кто виноват?

Сложная микроархитектура современных CPU:

- http://www.slideshare.net/SergeyKuksenko/quantum-performance-effects
- https://www.youtube.com/watch?v=RGFJjQKChNQ

Кто виноват?

Сложная микроархитектура современных CPU:

- Плохой алгоритм $\Rightarrow$ 100% CPU

Кто виноват?

Сложная микроархитектура современных CPU:

- Плохой алгоритм $\Rightarrow$ 100% CPU
- Ждем данные из памяти $\Rightarrow$ 100% CPU

Кто виноват?

Сложная микроархитектура современных CPU:

- Плохой алгоритм $\Rightarrow$ 100% CPU
- Ждем данные из памяти $\Rightarrow$ 100% CPU
- Откат из-за непредсказанного перехода $\Rightarrow$ 100% CPU

Кто виноват?

Сложная микроархитектура современных CPU:

- Плохой алгоритм $\Rightarrow$ 100% CPU
- Ждем данные из памяти $\Rightarrow$ 100% CPU
- Откат из-за непредсказанного перехода $\Rightarrow$ 100% CPU
- Сложные и дорогие операции $\Rightarrow$ 100% CPU

Кто виноват?

Сложная микроархитектура современных CPU:

- Плохой алгоритм $\Rightarrow$ 100% CPU
- Ждем данные из памяти $\Rightarrow$ 100% CPU
- Откат из-за непредсказанного перехода $\Rightarrow$ 100% CPU
- Сложные и дорогие операции $\Rightarrow$ 100% CPU
- Не хватает ILP $\Rightarrow$ 100% CPU

Кто виноват?

Сложная микроархитектура современных CPU:

- Плохой алгоритм $\Rightarrow$ 100% CPU
- Ждем данные из памяти $\Rightarrow$ 100% CPU
- Откат из-за непредсказанного перехода $\Rightarrow$ 100% CPU
- Сложные и дорогие операции $\Rightarrow$ 100% CPU
- Не хватает ILP $\Rightarrow$ 100% CPU
- и т.д. $\Rightarrow$ 100% CPU

PMU

PMU: Performance Monitoring Unit

Performance Monitoring Unit - это встроенный хардверный профайлер внутри процессора.

- Performance monitoring events
- Hardware counters (HWC)

PMU: События

Читаем документацию от производителя! (e.g. 2 страницы из 32)

Table 19-14. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	LOAD_BLOCK.OVERLAP_STORE	Loads that partially overlap an earlier store.	
04H	07H	SB_DRAIN.ANY	All Store buffer stall cycles.	
05H	02H	MISALIGN_MEMORY.STORE	All store referenced with misaligned address.	
06H	04H	STORE_BLOCKS.AT_RET	Counts number of loads delayed with at-Retirement block code. The following loads need to be executed at retirement and wait for all senior stores on the same thread to be drained: load splitting across 4K boundary (page split), load accessing uncacheable (UC or USWC) memory, load lock, and load with page table in UC or USWC memory region.	
06H	08H	STORE_BLOCKS.L1D_BLOCK	Cacheable loads delayed with L1D block code.	
07H	01H	PARTIAL_ADDRESS_ALIAS	Counts false dependency due to partial address aliasing.	
08H	01H	DTLB_LOAD_MISSES.ANY	Counts all load misses that cause a page walk.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED	Counts number of completed page walks due to load miss in the STLB.	
08H	04H	DTLB_LOAD_MISSES.WALK_CYCLES	Cycles PMH is busy with a page walk due to a load miss in the STLB.	
08H	10H	DTLB_LOAD_MISSES.STLB_HIT	Number of cache load STLB hits.	
08H	20H	DTLB_LOAD_MISSES.PDE_MISS	Number of DTLB cache load misses where the low part of the linear to physical address translation was missed.	
08H	01H	MEM_INST_RETIRED.LOADS	Counts the number of instructions with an architecturally-visible load retired on the architected path.	
08H	02H	MEM_INST_RETIRED.STORES	Counts the number of instructions with an architecturally-visible store retired on the architected path.	
08H	10H	MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD	Counts the number of instructions exceeding the latency specified with ld_lat facility.	In conjunction with ld_lat facility
0CH	01H	MEM_STORE_RETIRED.DTLB_MISS	The event counts the number of retired stores that missed the DTLB. The DTLB miss is not counted if the store operation causes a fault. Does not counter prefetches. Counts both primary and secondary misses to the TLB.	

Table 19-14. Non-Architectural Performance Events in the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
0EH	01H	UOPS_ISSUED.ANY	Counts the number of Uops issued by the Register Allocation Table to the Reservation Station, i.e. the UOPs issued from the front end to the back end.	
0EH	01H	UOPS_ISSUED.STALLED_CYCLES	Counts the number of cycles no Uops issued by the Register Allocation Table to the Reservation Station, i.e. the UOPs issued from the front end to the back end.	set "invert=1, cmask = 1"
0EH	02H	UOPS_ISSUED.FUSED	Counts the number of fused Uops that were issued from the Register Allocation Table to the Reservation Station.	
0FH	01H	MEM_UNCORE_RETIRED.UNKNOWN_SOURCE	Load instructions retired with unknown LLC miss (Precise Event).	Applicable to one and two sockets
0FH	02H	MEM_UNCORE_RETIRED.OTHER_CORE_L2_HIT	Load instructions retired that HIT modified data in sibling core (Precise Event).	Applicable to one and two sockets
0FH	04H	MEM_UNCORE_RETIRED.REMOTE_HITM	Load instructions retired that HIT modified data in remote socket (Precise Event).	Applicable to two sockets only
0FH	08H	MEM_UNCORE_RETIRED.LOCAL_DRAM_AND_REMOTE_CACHE_HIT	Load instructions retired local dram and remote cache HIT data sources (Precise Event).	Applicable to one and two sockets
0FH	10H	MEM_UNCORE_RETIRED.REMOTE_DRAM	Load instructions retired remote DRAM and remote home-remote cache HITM (Precise Event).	Applicable to two sockets only
0FH	20H	MEM_UNCORE_RETIRED.OTHER_LLC_MISS	Load instructions retired other LLC miss (Precise Event).	Applicable to two sockets only
0FH	80H	MEM_UNCORE_RETIRED.UNCACHEABLE	Load instructions retired I/O (Precise Event).	Applicable to one and two sockets
10H	01H	FP_COMP_OPS_EXE.X87	Counts the number of FP Computational Uops Executed. The number of FADD, FSUB, FCOM, FMULs, integer MUL and IMULs, FDIVs, FPREMs, FSQRTS, integer DIVs, and IDIVs. This event does not distinguish an FADD used in the middle of a transcendental flow from a separate FADD instruction.	
10H	02H	FP_COMP_OPS_EXE.MMX	Counts number of MMX Uops executed.	
10H	04H	FP_COMP_OPS_EXE.SSE_FP	Counts number of SSE and SSE2 FP uops executed.	
10H	08H	FP_COMP_OPS_EXE.SSE2_INTEGER	Counts number of SSE2 integer uops executed.	
10H	10H	FP_COMP_OPS_EXE.SSE_FP_PACKED	Counts number of SSE FP packed uops executed.	
10H	20H	FP_COMP_OPS_EXE.SSE_FP_SCALAR	Counts number of SSE FP scalar uops executed.	
10H	40H	FP_COMP_OPS_EXE.SSE_SINGLE_PRECISION	Counts number of SSE* FP single precision uops executed.	
10H	80H	FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION	Counts number of SSE* FP double precision uops executed.	
12H	01H	SIMD_INT_128.PACKED_MPY	Counts number of 128 bit SIMD integer multiply operations.	

PMU: События

Проблемы:

- Тысяча их!
- Требуют знания микроархитектуры
- Платформозависимые
  - сильно различаются у разных производителей CPU
  - могут сильно меняться у одного производителя при смене микроархитектуры
- Как с ними вообще работать?

PMU: HWC

HWC имеют два режима работы:

- Counting mode
  - `if (случилось событие) ++counter;`
  - общая диагностика (ответ на вопрос «ЧТО»)
- Sampling mode
  - ```
    if (случилось событие)
        if (++counter < threshold ) INTERRUPT;
    ```
  - позволяет профилировать (ответ на вопрос «ГДЕ»)
    ```

HWC: Проблемы

Событий много, счетчиков мало

(e.g. Nehalem: 3 fixed HWC, 4 programmable)

- «multiple-running» (с разными событиями)
  - повторяемость приложения
- multiplexing (если тулы позволяют)
  - стабильность приложения (steady state)

HWC: Проблемы

Sampling mode:

- «instruction skid»
  (невозможность точного связывания события и инструкции)
- Uncore events
  (e.g. у нас общий L3 cache на несколько ядер)

HWC: использование

- валидация железа
- анализ производительности

HWC: использование

- валидация железа
- анализ производительности
- run-time tuning (e.g. JRockit, etc.)
- security attacks и защита от них
- test code coverage
- etc.

HWC: tools

- Oracle Solaris Studio Performance Analyzer
- perf (perf_events) (`http://perf.wiki.kernel.org`)
- JMH
  - perf mode
  - perfasm mode (perf + `-XX:+PrintAssembly`)

HWC: tools(cont.)

- AMD CodeXL
- Intel Vtune Amplifier XE
- etc...

HWC: События perf_events (e.g.)

- **cycles**
- **instructions**
- cache-references
- cache-misses
- branches
- branch-misses
- bus-cycles
- ref-cycles

- dTLB-loads
- dTLB-load-misses
- L1-dcache-loads
- L1-dcache-load-misses
- L1-dcache-stores
- L1-dcache-store-misses
- LLC-loads
- LLC-load-misses
- etc...

HWC: События Oracle Studio (e.g.)

- **cycles**
- **insts**
- branch-instruction-retired
- branch-misses-retired
- dtlbm
- l1h
- l1m
- l2h
- l2m
- l3h
- l3m
- etc...

μArch Performance Analysis (μAPA)

μAPA: ГУМ

$$time = \frac{cycles}{frequency}$$

Оптимизация это

уменьшение количества тактов (cycles)![1]

[1] все остальное - overclocking

μAPA: ГУМ

Главное Уравнение Микроархитектуры:

$$cycles = PathLength * CPI = PathLength * \frac{1}{IPC}$$

- PathLength - количество инструкций
- CPI - cycles per instruction
- IPC - instructions per cycle

μAPA: $PathLength * CPI$

- PathLength - мера эффективности алгоритма (чем меньше, тем лучше)
- CPI - мера эффективности работы CPU (чем меньше, тем лучше)
  - $CPI = 4$ – плохо!
  - $CPI = 0.4$ – хорошо!
  - $CPI = 0.2$ – идеально!
  - $CPI = 1$
    - Nehalem – сойдет!
    - SandyBridge и позже – плоховатенько!

μAPA: Что делать?

- низкий CPI
  - уменьшаем PathLength – «улучшаем эффективность алгоритма»
- большой CPI $\Rightarrow$ stalls
  - memory stalls – «улучшаем эффективность структур данных»
  - branch stalls – «улучшаем логику передачи управления»
  - instruction dependency – «разрываем зависимости по данным»
  - long latency ops – «заменяем на полегче»
  - и т.д.

μAPA: Проблемы?

Нам нужна классификация проблем!

Проблемы: Memory bound

- dTLB misses
- L1,L2,L3,...,LN misses
- NUMA (чужая память)
- memory bandwidth
- sharing (false/true)
- cache line split (не бывает в Java, почти)
- store forwarding
  (маловероятно, да и все равно не исправить на Java уровне)
- 4k aliasing (попробуй поймать)

Проблемы: Core bound

- long latency operations (DIV, SQRT)
- FP assist (floating points denormal, NaN, inf)
- bad speculation (вызывается непредсказанным переходом)
- port saturation (нам не нужно)

Проблемы: Front-End bound

- iTLB miss
- iCache miss
- branch mispredict
- LSD (loop stream decoder)

Можно решить, только если залезть в исходники HotSpot

Примеры

Пример 1

Пример 1: perf stat -d

```
    718977.872294 task-clock (msec)
           35,609 context-switches
              338 cpu-migrations
            5,956 page-faults
1,857,021,243,903 cycles
  916,928,198,201 instructions              #    0.49  insns per cycle
  168,191,439,904 branches                  #  233.931 M/sec
    2,297,813,287 branch-misses             #    1.37% of all branches
  252,425,639,125 L1-dcache-loads           #  351.090 M/sec
   27,250,789,142 L1-dcache-load-misses     #   10.80% of all L1-dcache hits
   19,555,293,888 LLC-loads                 #   27.199 M/sec
    5,959,388,213 L1-icache-load-misses:HG  #
  252,370,072,667 dTLB-loads:HG             #  351.012 M/sec
    4,724,764,893 dTLB-load-misses:HG       #    1.87% of all dTLB cache hits
    9,903,086,687 iTLB-loads:HG             #   13.774 M/sec
      355,234,855 iTLB-load-misses:HG       #    3.59% of all iTLB cache hits
```

Пример 1: `perf stat -d`

```
    718977.872294 task-clock (msec)
           35,609 context-switches
              338 cpu-migrations
            5,956 page-faults
1,857,021,243,903 cycles
  916,928,198,201 instructions              #    0.49  insns per cycle
  168,191,439,904 branches                  #  233.931 M/sec
    2,297,813,287 branch-misses             #    1.37% of all branches
  252,425,639,125 L1-dcache-loads           #  351.090 M/sec
   27,250,789,142 L1-dcache-load-misses     #   10.80% of all L1-dcache hits
   19,555,293,888 LLC-loads                 #   27.199 M/sec
    5,959,388,213 L1-icache-load-misses:HG  #
  252,370,072,667 dTLB-loads:HG             #  351.012 M/sec
    4,724,764,893 dTLB-load-misses:HG       #    1.87% of all dTLB cache hits
    9,903,086,687 iTLB-loads:HG             #   13.774 M/sec
      355,234,855 iTLB-load-misses:HG       #    3.59% of all iTLB cache hits
```

Пример 1: dTLB misses

TLB = Translation Lookaside Buffer

- Кэш транслятора виртуальных адресов в физические
- Каждое обращение к памяти – обращение к TLB
- TLB miss стоит сотни тактов

Пример 1: dTLB misses

TLB = Translation Lookaside Buffer

- Кэш транслятора виртуальных адресов в физические
- Каждое обращение к памяти – обращение к TLB
- TLB miss стоит сотни тактов

Что делать:

- LargePages (Huge pages)
- попытаться уменьшить «Working Set»

Пример 1: `perf stat -d`

-XX:-UseLargePages

```
             0.49 insns per cycle
1,857,021,243,903 cycles
  916,928,198,201 instructions
  168,191,439,904 branches
    2,297,813,287 branch-misses
  252,425,639,125 L1-dcache-loads
   27,250,789,142 L1-dcache-load-misses
   19,555,293,888 LLC-loads
    5,959,388,213 L1-icache-load-misses
  252,370,072,667 dTLB-loads:HG
    4,724,764,893 dTLB-load-misses:HG
    9,903,086,687 iTLB-loads:HG
      355,234,855 iTLB-load-misses:HG
```

-XX:+UseLargePages

```
             0.63 insns per cycle
1,858,183,899,070 cycles
1,162,494,638,281 instructions
  211,381,201,267 branches
    2,733,847,675 branch-misses
  319,502,351,264 L1-dcache-loads
   29,480,701,672 L1-dcache-load-misses
   20,080,211,489 LLC-loads
    7,323,312,881 L1-icache-load-misses
  319,165,268,237 dTLB-loads:HG
      186,034,647 dTLB-load-misses:HG
    1,063,297,490 iTLB-loads:HG
       21,423,723 iTLB-load-misses:HG
```

Ускорение – 28%

Пример 1: `perf stat -d`

-XX:-UseLargePages

```
             0.49  insns per cycle
1,857,021,243,903  cycles
  916,928,198,201  instructions
  168,191,439,904  branches
    2,297,813,287  branch-misses
  252,425,639,125  L1-dcache-loads
   27,250,789,142  L1-dcache-load-misses
   19,555,293,888  LLC-loads
    5,959,388,213  L1-icache-load-misses
  252,370,072,667  dTLB-loads:HG
    4,724,764,893  dTLB-load-misses:HG
    9,903,086,687  iTLB-loads:HG
      355,234,855  iTLB-load-misses:HG
```

-XX:+UseLargePages

```
             0.63  insns per cycle
1,858,183,899,070  cycles
1,162,494,638,281  instructions
  211,381,201,267  branches
    2,733,847,675  branch-misses
  319,502,351,264  L1-dcache-loads
   29,480,701,672  L1-dcache-load-misses
   20,080,211,489  LLC-loads
    7,323,312,881  L1-icache-load-misses
  319,165,268,237  dTLB-loads:HG
      186,034,647  dTLB-load-misses:HG
    1,063,297,490  iTLB-loads:HG
       21,423,723  iTLB-load-misses:HG
```

Ускорение – 28%

Пример 2

Пример 2: решето

```
@Benchmark
public boolean[] directWrite() {
    boolean[] s = new boolean[SIZE];
    for (int i = 2; i <= SQRT_SIZE; i++) {
        if (!s[i]) {
            for (int j = i * i; j < SIZE; j += i) {
                s[j] = true;
            }
        }
    }
    return s;
}
```

Пример 2: `perf stat (SIZE=16777216)`

	directWrite
throuhput (ops/sec)	5.4
CPI	1.2
cycles	78,943,190,741
instructions	65,360,777,246
L1-dcache-loads	6,590,053,617
L1-dcache-load-misses	2,529,995,709
L1-dcache-stores	13,186,522,910
L1-dcache-store-misses	4,701,044,439
branches	13,139,649,942
branch-misses	2,705,112

Пример 2: опять решето

```
@Benchmark
public boolean[] testAndWrite() {
    boolean[] s = new boolean[SIZE];
    for (int i = 2; i <= SQRT_SIZE; i++) {
        if (!s[i]) {
            for (int j = i * i; j < SIZE; j += i) {
                if (!s[j]) {
                    s[j] = true;
                }
            }
        }
    }
    return s;
}
```

Пример 2: Сравниваем (SIZE=16777216)

	directWrite	testAndWrite
throuhput (ops/sec)	5.4	6.8
CPI	1.2	0.87
cycles	78,943,190,741	79,987,335,051
instructions	65,360,777,246	91,441,204,021
L1-dcache-loads	6,590,053,617	16,418,119,354
L1-dcache-load-misses	2,529,995,709	3,237,485,830
L1-dcache-stores	13,186,522,910	7,560,513,159
L1-dcache-store-misses	4,701,044,439	1,700,509,942
branches	13,139,649,942	29,660,113,381
branch-misses	2,705,112	506,189,014

Пример 2: А если (SIZE=65536) ?

Пример 2: А если (SIZE=65536) ?

	directWrite	testAndWrite
throuhput (ops/sec)	6762	4769
CPI	0.3	0.4
cycles	78,570,302,779	78,450,742,371
instructions	258,297,155,155	199,283,466,155
L1-dcache-loads	25,569,610,439	35,826,879,999
L1-dcache-load-misses	5,557,518,092	3,950,327,363
L1-dcache-stores	54,520,084,320	20,132,957,601
L1-dcache-store-misses	11,408,146,649	3,891,330,319
branches	51,623,896,048	63,167,232,412
branch-misses	17,645,765	914,376,450

Пример 3

Пример 3: There is no spoon

```
public int SIZE=2048;
int[][] A,B;

@Benchmark
public int[][] multIJK() {
    int[][] C = new int[SIZE][SIZE];
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            int sum = 0;
            for (int k = 0; k < SIZE; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
    return C;
}
```

Пример 3: There is no fork

```
public int SIZE=2048;
int[][] A,B;

@Benchmark
public int[][] multIKJ() {
    int[][] C = new int[SIZE][SIZE];
    for (int i = 0; i < SIZE; i++) {
        for (int k = 0; k < SIZE; k++) {
            int aik = A[i][k];
            for (int j = 0; j < SIZE; j++) {
                C[i][j] += aik * B[k][j];
            }
        }
    }
    return C;
}
```

Пример 3: Результаты

- multIJK $= 32\ seconds/op$
- multIKJ $= 3.4\ seconds/op$

Как и ожидалось!

Пример 3: Результаты

- multIJK $= 32\ seconds/op$
- multIKJ $= 3.4\ seconds/op$

А проверить?

Пример 3: Сравниваем

	`multIJK`	`multIKJ`
time (secs/op)	32	3.4
IPC	0.8	0.84
cycles	859,506,439,753	96,363,199,858
instructions	690,598,102,605	80,686,481,893
L1-dcache-loads	342,142,677,561	22,914,025,095
L1-dcache-load-misses	189,212,903,367	5,507,038,542

Пример 3: down the rabbit hole (multIKJ)

```
cycles insts
...
6.42%  5.54%  0x00007ff6591d20c5: vmovdqu %ymm1,0x10(%r14,%rbx,4) ;*iastore

9.49% 11.91%  0x00007ff6591d20cc: add    $0x8,%ebx                ;*iinc

0.15%  0.05%  0x00007ff6591d20cf: cmp    %r11d,%ebx

              0x00007ff6591d20d2: jl     0x00007ff6591d20b2       ;*if_icmpge
...
```

Пример 3: down the rabbit hole (multIKJ)

```
cycles insts
...
6.42%  5.54%  0x00007ff6591d20c5: vmovdqu %ymm1,0x10(%r14,%rbx,4) ;*iastore

9.49% 11.91%  0x00007ff6591d20cc: add    $0x8,%ebx                ;*iinc

0.15%  0.05%  0x00007ff6591d20cf: cmp    %r11d,%ebx

              0x00007ff6591d20d2: jl     0x00007ff6591d20b2       ;*if_icmpge
...
```

Векторизация (SSE/AVX)!

Пример 3: `-XX:UseSSE=0 -XX:UseAVX=0`

	`multIKJ(with SSE)`	`multIKJ(no SSE)`
time (secs/op)	3.4	6.3
IPC	0.84	1.66
cycles	96,363,199,858	191,825,787,188
instructions	80,686,481,893	318,140,467,924
L1-dcache-loads	22,914,025,095	179,849,909,115
L1-dcache-load-misses	5,507,038,542	5,811,906,774

Заключение:

Заключение: Учиться, учиться и учиться!

Изучаем микроархитектуру:

- “Computer Architecture: A Quantitative Approach” John L. Hennessy, David A. Patterson
- `http://www.agner.org/optimize/`
- `http://www.google.com/search?q=Hardware+performance+counter`

Q & A ?