Java Performance:

Speedup your applications with hardware counters

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.

~~Intriguing~~ Introductory Example

Gotcha, here is my hot method

or

On Algorithmic O*ptimizations

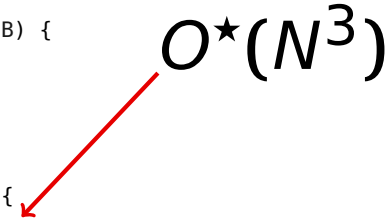
*just-O

Example 1: Very Hot Code

```
public int[][] multiply(int[][] A, int[][] B) {
    int size = A.length;
    int[][] R = new int[size][size];
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            int s = 0;
            for (int k = 0; k < size; k++) {
                s += A[i][k] * B[k][j];
            }
            R[i][j] = s;
        }
    }
    return R;
}
```

Example 1: Very Hot Code

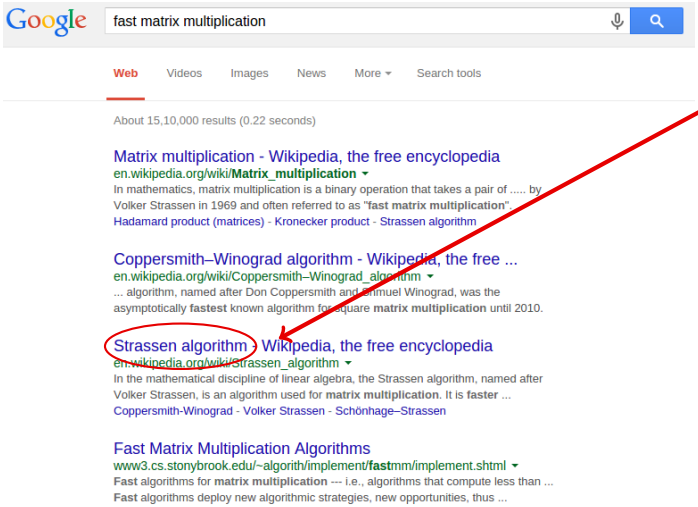
```java
public int[][] multiply(int[][] A, int[][] B) {
    int size = A.length;
    int[][] R = new int[size][size];
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            int s = 0;
            for (int k = 0; k < size; k++) {
                s += A[i][k] * B[k][j];
            }
            R[i][j] = s;
        }
    }
    return R;
}
```



---

* *big-O*

Example 1: "Ok Google"

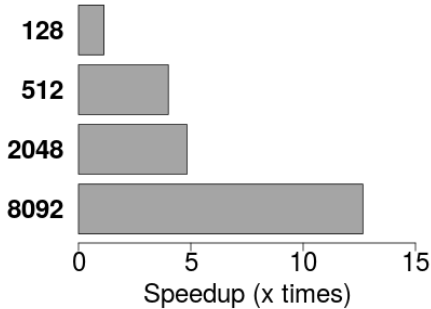


$O(N^{2.81})$

Example 1: Very Hot Code

N	multiply	strassen
128	0.0026	0.0023
512	0.48	0.12
2048	28	5.8
8192	3571	282
	time; seconds/op	

Example 1: Optimized speedup over baseline



Try the other way

Example 1: `JMH, -prof perfnorm, N=256`*

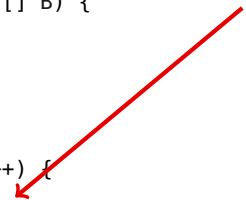
	per multiplication	per iteration
CPI	1.06	
cycles	146×10^6	8.7
instructions	137×10^6	8.2
L1-loads	68×10^6	4
L1-load-misses	33×10^6	2
L1-stores	0.56×10^6	
L1-store-misses	38×10^3	
LLC-loads	26×10^6	1.6
LLC-stores	11.6×10^3	

*$\sim$ *256Kb per matrix*

Example 1: Very Hot Code

```
public int[][] multiply(int[][] A, int[][] B) {
    int size = A.length;
    int[][] R = new int[size][size];
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            int s = 0;
            for (int k = 0; k < size; k++) {
                s += A[i][k] * B[k][j];
            }
            R[i][j] = s;
        }
    }
    return R;
}
```

L1-load-misses



Example 1: Very Hot Code

```
public int[][] multiply(int[][] A, int[][] B) {
    int size = A.length;
    int[][] R = new int[size][size];
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            int s = 0;
            for (int k = 0; k < size; k++) {
                s += A[i][k] * B[k][j];
            }
            R[i][j] = s;
        }
    }
    return R;
}
```

Example 1: Very Hot Code

```
public int[][] multiplyIKJ(int[][] A, int[][] B) {
    int size = A.length;
    int[][] R = 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++) {
                R[i][j] += aik * B[k][j];
            }
        }
    }
    return R;
}
```

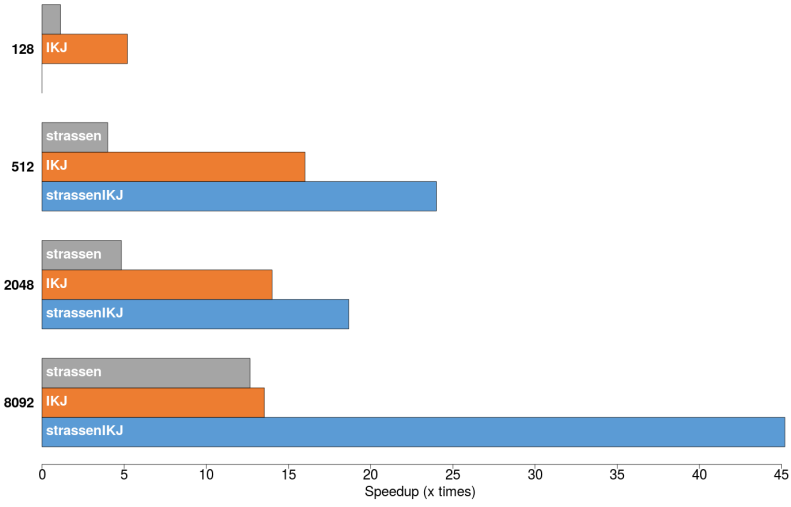
Example 1: Very Hot Code

	IJK		IKJ	
N	multiply	strassen	multiply	
128	0.0026	0.0023	0.0005	
512	0.48	0.12	0.03	
2048	28	5.8	2	
8192	3571	282	264	
	time; seconds/op			

Example 1: Very Hot Code

	IJK		IKJ	
N	multiply	strassen	multiply	strassen
128	0.0026	0.0023	0.0005	
512	0.48	0.12	0.03	0.02
2048	28	5.8	2	1.5
8192	3571	282	264	79
	time; seconds/op			

Example 1: Optimized speedup over baseline



Example 1: `JMH, -prof perfnorm, N=256`

	IJK		IKJ	
	multiply	iteration	multiply	iteration
CPI	1.06		0.51	
cycles	146×10^6	8.7	9.7×10^6	0.6
instructions	137×10^6	8.2	19×10^6	1.1
L1-loads	68×10^6	4	5.4×10^6	0.3
L1-load-misses	33×10^6	2	1.1×10^6	0.1
L1-stores	0.56×10^6		2.7×10^6	0.2
L1-store-misses	38×10^3		9×10^3	
LLC-loads	26×10^6	1.6	0.3×10^6	
LLC-stores	11.6×10^3		3.5×10^3	

Example 1: `JMH, -prof perfnorm, N=256`

	IJK		IKJ	
	multiply	iteration	multiply	iteration
CPI	1.06		0.51	
cycles	146×10^6	8.7	9.7×10^6	0.6
instructions	137×10^6	8.2	19×10^6	? 1.1
L1-loads	68×10^6	4	5.4×10^6	0.3
L1-load-misses	33×10^6	2	1.1×10^6	0.1
L1-stores	0.56×10^6		2.7×10^6	0.2
L1-store-misses	38×10^3		9×10^3	
LLC-loads	26×10^6	1.6	0.3×10^6	
LLC-stores	11.6×10^3		3.5×10^3	

Example 1: Free ~~beer~~ benefits!*

```
 cycles  insts
...
 0.15%   0.17%  <addr>: vmovdqu  0x10(%rax,%rdx,4),%ymm5
 8.83%   9.02%          vpmulld  %ymm4,%ymm5,%ymm5
43.60%  42.03%          vpaddd   0x10(%r9,%rdx,4),%ymm5,%ymm5
 6.26%   6.24%          vmovdqu  %ymm5,0x10(%r9,%rdx,4)      ;*iastore
19.82%  20.82%          add      $0x8,%edx                   ;*iinc
 1.46%   2.75%          cmp      %ecx,%edx
                        jl       <addr>                      ;*if_icmpge
...
```

how to get asm: JMH, `-prof perfasm`

Example 1: Free ~~beer~~ benefits!*

```
 cycles  insts
...
 0.15%   0.17%  <addr>:  vmovdqu  0x10(%rax,%rdx,4),%ymm5
 8.83%   9.02%           vpmulld  %ymm4,%ymm5,%ymm5
43.60%  42.03%           vpaddd   0x10(%r9,%rdx,4),%ymm5,%ymm5
 6.26%   6.24%           vmovdqu  %ymm5,0x10(%r9,%rdx,4)      ;*iastore
19.82%  20.82%           add      $0x8,%edx                   ;*iinc
 1.46%   2.75%           cmp      %ecx,%edx
                         jl       <addr>                      ;*if_icmpge
...
```

Vectorization (SSE/AVX)!

how to get asm: JMH, `-prof perfasm`

Chapter 1

Performance Optimization Methodology:
A Very Short Introduction.

Three magic questions and the direction of the journey

Three magic questions

Three magic questions

What?

- What prevents my application to work faster? (monitoring)

Three magic questions

What? ⇒ Where?

- What prevents my application to work faster?
  (monitoring)
- Where does it hide?
  (profiling)

Three magic questions

What? ⇒ Where? ⇒ How?

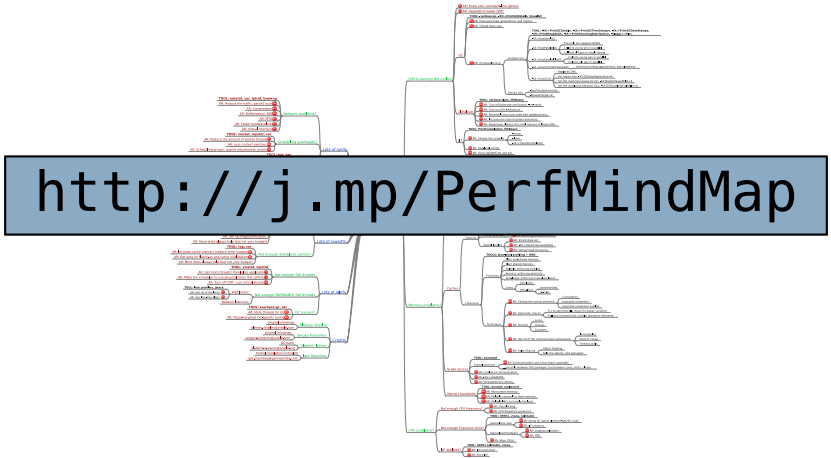
- What prevents my application to work faster?
  (monitoring)
- Where does it hide?
  (profiling)
- How to stop it messing with performance?
  (tuning/optimizing)

Top-Down Approach

- System Level
  - Network, Disk, OS, CPU/Memory
- JVM Level
  - GC/Heap, JIT, Classloading
- Application Level
  - Algorithms, Synchronization, Threading, API
- Microarchitecture Level
  - Caches, Data/Code alignment, CPU Pipeline Stalls

Methodology in Essence

Methodology in Essence



Let's go (Top-Down)

Do system monitoring (e.g. mpstat)

Let's go (Top-Down)

Do system monitoring (e.g. mpstat)

- Lots of %sys $\Rightarrow$ ...

$\Downarrow$

Let's go (Top-Down)

Do system monitoring (e.g. mpstat)

- Lots of %sys $\Rightarrow \ldots$
  $\Downarrow$
- Lots of %irq, %soft $\Rightarrow \ldots$
  $\Downarrow$

Let's go (Top-Down)

Do system monitoring (e.g. mpstat)

- Lots of %sys $\Rightarrow \ldots$
- $\Downarrow$
- Lots of %irq, %soft $\Rightarrow \ldots$
- $\Downarrow$
- Lots of %iowait $\Rightarrow \ldots$
- $\Downarrow$

Let's go (Top-Down)

Do system monitoring (e.g. mpstat)

- Lots of %sys $\Rightarrow \ldots$
  $\Downarrow$
- Lots of %irq, %soft $\Rightarrow \ldots$
  $\Downarrow$
- Lots of %iowait $\Rightarrow \ldots$
  $\Downarrow$
- Lots of %idle $\Rightarrow \ldots$
  $\Downarrow$

Let's go (Top-Down)

Do system monitoring (e.g. mpstat)

- Lots of %sys $\Rightarrow \ldots$
- $\Downarrow$
- Lots of %irq, %soft $\Rightarrow \ldots$
- $\Downarrow$
- Lots of %iowait $\Rightarrow \ldots$
- $\Downarrow$
- Lots of %idle $\Rightarrow \ldots$
- $\Downarrow$
- **Lots of %user**

Chapter 2

High CPU Load

and
the main question:
«who/what is to blame?»

CPU Utilization

- What does ∼100% CPU Utilization mean?

CPU Utilization

- What does ∼100% CPU Utilization mean?
  - OS has enough tasks to schedule

CPU Utilization

- What does ∼100% CPU Utilization mean?
  - OS has enough tasks to schedule

Can profiling help?

CPU Utilization

- What does ~100% CPU Utilization mean?
  - OS has enough tasks to schedule

Can profiling help?

Profiler* shows «WHERE» application time is spent,
but there's no answer to the question «WHY».

*traditional profiler

Who/What is to blame?

Complex CPU microarchitecture:

- Inefficient algorithm ⇒ 100% CPU
- Pipeline stall due to memory load/stores ⇒ 100% CPU
- Pipeline flush due to mispredicted branch ⇒ 100% CPU
- Expensive instructions ⇒ 100% CPU
- Insufficient ILP* ⇒ 100% CPU
- etc. ⇒ 100% CPU

*Instruction Level Parallelism

Chapter 3

Hardware Counters

HWC, PMU - WTF?

PMU: Performance Monitoring Unit

Performance Monitoring Unit - profiles hardware activity, built into CPU.

PMU: Performance Monitoring Unit

Performance Monitoring Unit - profiles hardware activity,
built into CPU.

PMU Internals (in less than 21 seconds) :

Hardware counters (HWC) count hardware performance events (performance monitoring events)

Events

Vendor's documentation! (e.g. 2 pages from 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 "inv=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.	

Events: Issues

- Hundreds of events
- Microarchitectural experience is required
- Platform dependent
  - vary from CPU vendor to vendor
  - may vary when CPU manufacturer introduces new microarchitecture
- How to work with HWC?

HWC

HWC modes:

- Counting mode
  - `if (event_happened) ++counter;`
  - general monitoring (answering «WHAT?»)
- Sampling mode
  - ```
    if (event_happened)
        if (++counter < threshold) INTERRUPT;
    ```
  - profiling (answering «WHERE?»)
    ```

HWC: Issues

So many events, so few counters

e.g. “Nehalem”:

- over 900 events
- 7 HWC (3 fixed + 4 programmable)

- «multiple-running» (different events)
  - Repeatability
- «multiplexing» (only a few tools are able to do that)
  - Steady state

HWC: Issues (cont.)

Sampling mode:

- «instruction skid»
  (hard to correlate event and instruction)
- Uncore events
  (hard to bind event and execution thread;
  e.g. shared L3 cache)

HWC: typical usages

- hardware validation
- performance analysis

HWC: typical usages

- hardware validation
- performance analysis
- run-time tuning (e.g. JRockit, etc.)
- security attacks and defenses
- test code coverage
- etc.

HWC: tools

- `Oracle Solaris Studio Performance Analyzer`

  http://www.oracle.com/technetwork/server-storage/solarisstudio

- `perf/perf_events`

  http://perf.wiki.kernel.org

- `JMH` (Java Microbenchmark Harness)

  http://openjdk.java.net/projects/code-tools/jmh/)

  `-prof perf`

  `-prof perfnorm` = `perf`, normalized per operation

  `-prof perfasm` = `perf` + `-XX:+PrintAssembly`

HWC: tools (cont.)

- AMD CodeXL
- Intel® Vtune™ Amplifier
- etc. . .

perf events (e.g.)

- **cycles**
- **instrustions**
- 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
- etc...

Oracle Studio events (e.g.)

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

Chapter 4

I've got HWC data. What's next?

or

Introduction to microarchitecture performance analysis

Execution time

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

Optimization is. . .

reducing the (spent) cycle count!*

*everything else is overclocking

Microarchitecture Equation

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

- $PathLength$ - number of instructions
- CPI - cycles per instruction
- IPC - instructions per cycle

$PathLength * CPI$

- $PathLength \sim$ algorithm efficiency (the smaller the better)
- $CPI \sim$ CPU efficiency (the smaller the better)
  - $CPI = 4$ – bad!
  - $CPI = 1$
    - Nehalem – just good enough!
    - SandyBridge and later – not so good!
  - $CPI = 0.4$ – good!
  - $CPI = 0.2$ – ideal!

What to do?

- low *CPI*
  - reduce *PathLength* → «tune algorithm»
- high *CPI* ⇒ CPU stalls
  - memory stalls → «tune data structures»
  - branch stalls → «tune control logic»
  - instruction dependency → «break dependency chains»
  - long latency ops → «use more simple operations»
  - etc. . .

High CPI: Memory bound

- dTLB misses
- L1,L2,L3,...,LN misses
- NUMA: non-local memory access
- memory bandwidth
- false/true sharing
- cache line split *(not in Java world, except...)*
- store forwarding *(unlikely, hard to fix on Java level)*
- 4K aliasing

High CPI: Core bound

- long latency operations: DIV, SQRT
- FP assist: floating points denormal, NaN, inf
- bad speculation *(caused by mispredicted branch)*
- port saturation *(forget it)*

High CPI: Front-End bound

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

solvable by HotSpot tweaking

Example 2

Some «large» standard server-side Java benchmark

Example 2: **perf stat -d**

```
    880851.993237 task-clock (msec)
          39,318 context-switches
             437 cpu-migrations
           7,931 page-faults
2,277,063,113,376 cycles
1,226,299,634,877 instructions           #    0.54  insns per cycle
  229,500,265,931 branches               #  260.544 M/sec
    4,620,666,169 branch-misses          #    2.01% of all branches
  338,169,489,902 L1-dcache-loads        #  383.912 M/sec
   37,937,596,505 L1-dcache-load-misses  #   11.22% of all L1-dcache hits
   25,232,434,666 LLC-loads              #   28.645 M/sec
    7,307,884,874 L1-icache-load-misses  #    0.00% of all L1-icache hits
  337,730,278,697 dTLB-loads             #  382.846 M/sec
    6,094,356,801 dTLB-load-misses       #    1.81% of all dTLB cache hits
   12,210,841,909 iTLB-loads             #   13.863 M/sec
      431,803,270 iTLB-load-misses       #    3.54% of all iTLB cache hits

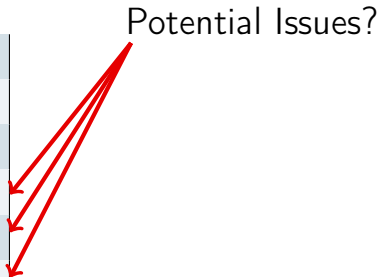
    301.557213044 seconds time elapsed
```

Example 2: Processed **`perf stat -d`**

CPI		1.85
cycles	$\times 10^9$	2277
instructions	$\times 10^9$	1226
L1-dcache-loads	$\times 10^9$	338
L1-dcache-load-misses	$\times 10^9$	38
LLC-loads	$\times 10^9$	25
dTLB-loads	$\times 10^9$	338
dTLB-load-misses	$\times 10^9$	6

Example 2: Processed `perf stat -d`

CPI		1.85
cycles	$\times 10^9$	2277
instructions	$\times 10^9$	1226
L1-dcache-loads	$\times 10^9$	338
L1-dcache-load-misses	$\times 10^9$	38
LLC-loads	$\times 10^9$	25
dTLB-loads	$\times 10^9$	338
dTLB-load-misses	$\times 10^9$	6



Example 2: Processed `perf stat -d`

CPI		1.85
cycles	$\times 10^9$	2277
instructions	$\times 10^9$	1226
L1-dcache-loads	$\times 10^9$	338
L1-dcache-load-misses	$\times 10^9$	38
LLC-loads	$\times 10^9$	25
dTLB-loads	$\times 10^9$	338
dTLB-load-misses	$\times 10^9$	6

Potential Issues?

Let's count:

$$4\times(338\times 10^9)+$$
$$12\times((38-25)\times 10^9)+$$
$$36\times(25\times 10^9)\approx$$
$$2408\times 10^9 \text{ cycles ???}$$

Cache latencies (L1/L2/L3) = 4/12/36 cycles

Example 2: Processed **perf stat -d**

CPI		1.85
cycles	$\times 10^9$	2277
instructions	$\times 10^9$	1226
L1-dcache-loads	$\times 10^9$	338
L1-dcache-load-misses	$\times 10^9$	38
LLC-loads	$\times 10^9$	25
dTLB-loads	$\times 10^9$	338
dTLB-load-misses	$\times 10^9$	6

Let's count:

$4 \times (338 \times 10^$

$12 \times ((3$ $\times 10^9) +$

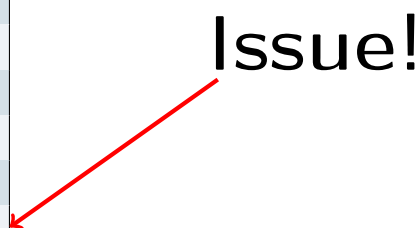
3 $\times 10^9) \approx$

2408×10^9 cycles ???

Superscalar in Action!

Example 2: Processed **perf stat -d**

CPI		1.85
cycles	$\times 10^9$	2277
instructions	$\times 10^9$	1226
L1-dcache-loads	$\times 10^9$	338
L1-dcache-load-misses	$\times 10^9$	38
LLC-loads	$\times 10^9$	25
dTLB-loads	$\times 10^9$	338
dTLB-load-misses	$\times 10^9$	6



Example 2: dTLB misses

TLB = Translation Lookaside Buffer

- a memory cache that stores recent translations of virtual memory addresses to physical addresses
- each memory access → access to TLB
- TLB miss may take hundreds cycles

Example 2: dTLB misses

TLB = Translation Lookaside Buffer

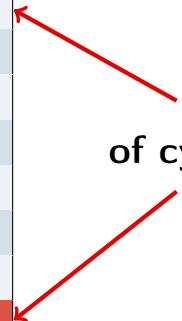
- a memory cache that stores recent translations of virtual memory addresses to physical addresses
- each memory access → access to TLB
- TLB miss may take hundreds cycles

How to check?

- `dtlb_load_misses_miss_causes_a_walk`
- `dtlb_load_misses_walk_duration`

Example 2: dTLB misses

CPI		1.85
cycles	$\times 10^9$	2277
instructions	$\times 10^9$	1226
L1-dcache-loads	$\times 10^9$	338
L1-dcache-load-misses	$\times 10^9$	38
LLC-loads	$\times 10^9$	25
dTLB-loads	$\times 10^9$	338
dTLB-load-misses	$\times 10^9$	6
dTLB-walks-duration*	$\times 10^9$	296



13%

of cycles (time)

*in cycles

Example 2: dTLB misses

Fixing it:

- Try to shrink application working set
  (modify you application)
- Enable `-XX:+UseLargePages`
  (modify you execution scripts)

Example 2: dTLB misses

Fixing it:

- Enable `-XX:+UseLargePages`
  (modify you execution scripts)

20% boost on the benchmark!

Example 2: `-XX:+UseLargePages`

		baseline	large pages
CPI		1.85	1.56
cycles	$\times 10^9$	2277	2277
instructions	$\times 10^9$	1226	1460
L1-dcache-loads	$\times 10^9$	338	401
L1-dcache-load-misses	$\times 10^9$	38	38
LLC-loads	$\times 10^9$	25	25
dTLB-loads	$\times 10^9$	338	401
dTLB-load-misses	$\times 10^9$	6	0.24
dTLB-walks-duration	$\times 10^9$	296	2.6

Example 2: normalized per transaction

		baseline	large pages
CPI		1.85	1.56
cycles	$\times 10^6$	23.4	19.5
instructions	$\times 10^6$	12.5	12.5
L1-dcache-loads	$\times 10^6$	3.45	3.45
L1-dcache-load-misses	$\times 10^6$	0.39	0.33
LLC-loads	$\times 10^6$	0.26	0.21
dTLB-loads	$\times 10^6$	3.45	3.45
dTLB-load-misses	$\times 10^6$	0.06	0.002
dTLB-walks-duration	$\times 10^6$	3.04	0.022

Example 3

~~«A plague on both your houses»~~

«++ on both your threads»

Example 3: False Sharing

- False Sharing:
  `https://en.wikipedia.org/wiki/False_sharing`
- `@Contended` (JEP 142)
  `http://openjdk.java.net/jeps/142`

Example 3: False Sharing

```
@State(Scope.Group)
public static class StateBaseline {
    int field0;
    int field1;
}

@Benchmark
@Group("baseline")
public int justDoIt(StateBaseline s) {
    return s.field0++;
}

@Benchmark
@Group("baseline")
public int doItFromOtherThread(StateBaseline s) {
    return s.field1++;
}
```

Example 3: Measure

	resource*	sharing	padded
Same Core (HT)	L1	9.5	4.9
Diff Cores (within socket)	L3 (LLC)	10.6	2.8
Diff Sockets	nothing	18.2	2.8
		average time, ns/op	

*shared between threads

Example 3: What do HWC tell us?

	Same Core		Diff Cores		Diff Sockets	
	sharing	padded	sharing	padded	sharing	padded
CPI	1.3	0.7	1.4	0.4	1.7	0.4
cycles	33130	17536	36012	9163	46484	9608
instructions	26418	25865	26550	25747	26717	25768
L1-loads	12593	9467	9696	8973	9672	9016
L1-load-misses	10	5	12	4	33	3
L1-stores	4317	7838	7433	4069	6935	4074
L1-store-misses	5	2	161	2	55	1
LLC-loads	4	3	58	1	32	1
LLC-load-misses	1	1	53	≈ 0	35	≈ 0
LLC-stores	1	1	183	≈ 0	49	≈ 0
LLC-store-misses	1	≈ 0	182	≈ 0	48	≈ 0

* All values are normalized per 10^3 operations

Example 3: «on a core and a prayer»

in case of the single core(HT) we have to look into
`MACHINE_CLEARS.MEMORY_ORDERING`

	Same Core		Diff Cores		Diff Sockets	
	sharing	padded	sharing	padded	sharing	padded
CPI	1.3	0.7	1.4	0.4	1.7	0.4
cycles	33130	17536	36012	9163	46484	9608
instructions	26418	25865	26550	25747	26717	25768
CLEARS	238	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0

Example 3: Diff Cores

L2_STORE_LOCK_RQSTS - L2 RFOs breakdown

	Diff Cores		Diff Sockets	
	sharing	padded	sharing	padded
CPI	1.4	0.4	1.7	0.4
LLC-stores	183	≈ 0	49	≈ 0
LLC-store-misses	182	≈ 0	48	≈ 0
L2_STORE_LOCK_RQSTS.MISS	134	≈ 0	33	≈ 0
L2_STORE_LOCK_RQSTS.HIT_E	≈ 0	≈ 0	≈ 0	≈ 0
L2_STORE_LOCK_RQSTS.HIT_M	≈ 0	≈ 0	≈ 0	≈ 0
L2_STORE_LOCK_RQSTS.ALL	183	≈ 0	49	≈ 0

Question!

To the audience

Example 3: Diff Cores

	Diff Cores		Diff Sockets	
	sharing	padded	sharing	padded
CPI	1.4	0.4	1.7	0.4
LLC-stores	183	≈ 0	49	≈ 0
LLC-store-misses	182	≈ 0	48	≈ 0
`L2_STORE_LOCK_RQSTS.MISS`	134	≈ 0	33	≈ 0
`L2_STORE_LOCK_RQSTS.ALL`	183	≈ 0	49	≈ 0

Why $183 > 49$ & $134 > 33$,
but the same socket case is faster?

Example 3: Some events count duration

For example:

`OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DEMAND_RFO`

	Diff Cores		Diff Sockets	
	sharing	padded	sharing	padded
CPI	1.4	0.4	1.7	0.4
cycles	36012	9163	46484	9608
instructions	26550	25747	26717	25768
`O_R_O.CYCLES_W_D_RFO`	21723	10	29601	56

Summary

Summary: "Performance is easy"

To achieve high performance:

- **You have to know your Application!**
- **You have to know your Frameworks!**
- **You have to know your Virtual Machine!**
- **You have to know your Operating System!**
- **You have to know your Hardware!**

Enlarge your knowledge with these simple tricks!

Reading list:

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

Thanks!

Q & A ?