

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.

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 does prevent my application to work faster?
(monitoring)

Three magic questions

What? \Rightarrow Where?

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

Three magic questions

What? \Rightarrow Where? \Rightarrow How?

- What does prevent 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

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 ...
↓

Let's go (Top-Down)

Do system monitoring (e.g. mpstat)

- Lots of %sys \Rightarrow ...



- Lots of %irq, %soft \Rightarrow ...



Let's go (Top-Down)

Do system monitoring (e.g. mpstat)

- Lots of %sys \Rightarrow ...
 ↓
- Lots of %irq, %soft \Rightarrow ...
 ↓
- Lots of %iowait \Rightarrow ...
 ↓

Let's go (Top-Down)

Do system monitoring (e.g. mpstat)

- Lots of %sys \Rightarrow ...
↓
- Lots of %irq, %soft \Rightarrow ...
↓
- Lots of %iowait \Rightarrow ...
↓
- Lots of %idle \Rightarrow ...
↓

Let's go (Top-Down)

Do system monitoring (e.g. mpstat)

- Lots of %sys \Rightarrow ...
 ↓
- Lots of %irq, %soft \Rightarrow ...
 ↓
- Lots of %iowait \Rightarrow ...
 ↓
- Lots of %idle \Rightarrow ...
 ↓
- **Lots of %user**

Chapter 2

High CPU Load

and

the main question:

«who/what is to blame?»

CPU Utilization

- What does $\sim 100\%$ CPU Utilization mean?

CPU Utilization

- What does $\sim 100\%$ 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 \Rightarrow 100% CPU
- Pipeline stall due to memory load/stores \Rightarrow 100% CPU
- Pipeline flush due to mispredicted branch \Rightarrow 100% CPU
- Expensive instructions \Rightarrow 100% CPU
- Insufficient ILP* \Rightarrow 100% CPU
- etc. \Rightarrow 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_BLOCKOVERLAP_STORE	Loads that partially overlap an earlier store.	
04H	07H	SB_DRAINANY	All Store buffer stall cycles.	
05H	02H	MISALIGN_MEMORYSTORE	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 uncachable (UC or USWC) memory, load lock, and load with page table in UC or USWC memory region.	
06H	08H	STORE_BLOCKS_LTD_BLOCK	Cachable loads delayed with LTD 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 <code>ld_jat</code> facility.	In conjunction with <code>ld_jat</code> facility
0CH	01H	MEM_STORE_RETIRED_DTLB_MBS	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 count 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 (Cont'd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
08H	01H	UOPS_ISSUED_ANY	Counts the number of UOps issued by the Register Allocation Table to the Reservation Station, in the UOps issued from the front end to the back end.	
08H	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"
08H	02H	UOPS_ISSUED_FUSED	Counts the number of fused UOps that were issued from the Register Allocation Table to the Reservation Station.	
08H	01H	MEM_LINCORE_RETIRED_UNKNOWN_LL_MISS_SOURCE	Load instructions retired with unknown LLC miss (Precise Event).	Applicable to one and two sockets
08H	02H	MEM_LINCORE_RETIRED_HIT_REMOTE_CORE_L2_HIT	Load instructions retired that HIT modified data in sibling core (Precise Event).	Applicable to one and two sockets
08H	04H	MEM_LINCORE_RETIRED_HIT_REMOTE_SOCKET_HIT	Load instructions retired that HIT modified data in remote socket (Precise Event).	Applicable to two sockets only
08H	08H	MEM_LINCORE_RETIRED_LOCAL_DRAM_REMOTE_CACHE_HIT	Load instructions retired local dram and remote cache HIT data sources (Precise Event).	Applicable to one and two sockets
08H	10H	MEM_LINCORE_RETIRED_REMOTE_DRAM_REMOTE_CACHE_HIT	Load instructions retired remote DRAM and remote home-on-die cache HIT (Precise Event).	Applicable to two sockets only
08H	20H	MEM_LINCORE_RETIRED_REMOTE_LL_MISS	Load instructions retired other LLC miss (Precise Event).	Applicable to two sockets only
08H	80H	MEM_LINCORE_RETIRED_INVALID	Load instructions retired I/O (Precise Event).	Applicable to one and two sockets
10H	01H	FP_COMP_DPS_EXEEXB7	Counts the number of FP Computational UOps Executed. The number of FADD, FSUB, FCOP, FMUL, integer MMX and MMX2, FDMX, FPREP, FSCRTS, integer DIV, and DIVB. This event does not distinguish an FADD used in the middle of a transcendental flow from a separate FADD instruction.	
10H	02H	FP_COMP_DPS_EXEEXMX	Counts number of MMX UOps executed.	
10H	04H	FP_COMP_DPS_EXESSE_FP	Counts number of SSE and SSE2 FP uops executed.	
10H	08H	FP_COMP_DPS_EXESSE2_INTEGER	Counts number of SSE2 integer uops executed.	
10H	10H	FP_COMP_DPS_EXESSE_FP_PACKED	Counts number of SSE FP packed uops executed.	
10H	20H	FP_COMP_DPS_EXESSE_FP_SCALAR	Counts number of SSE FP scalar uops executed.	
10H	40H	FP_COMP_DPS_EXESSE_SINGLE_PRECISION	Counts number of SSE* FP single precision uops executed.	
10H	80H	FP_COMP_DPS_EXESSE_DOUBLE_PRECISION	Counts number of SSE* FP double precision uops executed.	
12H	01H	SMD_INT_128PACKED_MPY	Counts number of 128 bit SMD 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 (<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 XE
- etc. . .

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...

Oracle Studio events (e.g.)

- **cycles**
- **insts**
- branch-instruction-retired
- branch-misses-retired
- dtlbn
- 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

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

- 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 ⇒ 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

Examples

Example 1

Some «large» standard server-side
Java benchmark

Example 1: 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
```

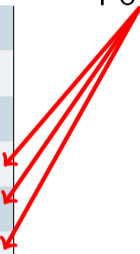
Example 1: Processed perf stat -d

IPC		0.54
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 1: Processed perf stat -d

IPC		0.54
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?



Example 1: Processed perf stat -d

IPC		0.54
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:

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

Example 1: Processed perf stat -d

IPC		0.54
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:

Superscalar in Action!

$$4 \times (338 + 38 + 25) \times 10^9 + 18 \times (25 \times 10^9) \approx 2408 \times 10^9 \text{ cycles ???}$$

Example 1: Processed perf stat -d

IPC		0.54
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

Issue!



Example 1: 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 1: 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

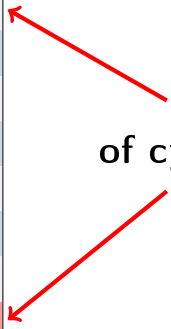
How to check?

- `dtlb_load_misses_miss_causes_a_walk`
- `dtlb_load_misses_walk_duration`

Example 1: dTLB misses

IPC		0.54
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 1: dTLB misses

Fixing it:

- Enable `-XX:+UseLargePages`
- try to shrink application working set

Example 1: -XX:+UseLargePages

		baseline	large pages
IPC		0.54	0.64
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

Boost – 20%

Example 1: normalized per transaction

		baseline	large pages
IPC		0.54	0.64
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 2

Gotcha, here is my hot method
or
On Algorithmic O^* ptimizations

*just-O

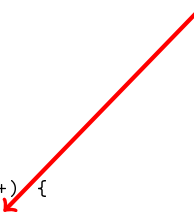
Example 2: Very Hot Code

```
public Matrix multiply(Matrix other) {
    int size = data.length;
    Matrix result = new Matrix(size);
    int[][] R = result.data;
    int[][] A = this.data;
    int[][] B = other.data;
    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 result;
}
```

Example 2: Very Hot Code

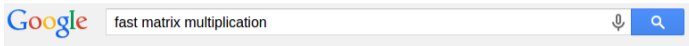
```
public Matrix multiply(Matrix other) {
    int size = data.length;
    Matrix result = new Matrix(size);
    int[][] R = result.data;
    int[][] A = this.data;
    int[][] B = other.data;
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            int s = 0;
      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

$O(N^3)$ *



*big-O

Example 2: "Ok Google"



Web Videos Images News More Search tools

About 15,10,000 results (0.22 seconds)

[Matrix multiplication - Wikipedia, the free encyclopedia](#)

en.wikipedia.org/wiki/Matrix_multiplication
In mathematics, matrix multiplication is a binary operation that takes a pair of ... by Volker Strassen in 1969 and often referred to as "fast matrix multiplication". Hadamard product (matrices) - Kronecker product - Strassen algorithm

[Coppersmith–Winograd algorithm - Wikipedia, the free ...](#)

en.wikipedia.org/wiki/Coppersmith–Winograd_algorithm
... algorithm, named after Don Coppersmith and Shmuel Winograd, was the asymptotically fastest known algorithm for square matrix multiplication until 2010.

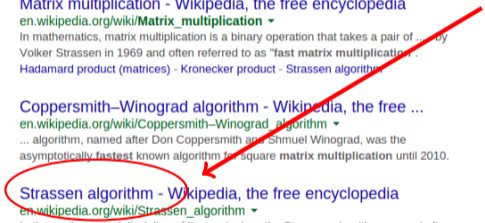
[Strassen algorithm - Wikipedia, the free encyclopedia](#)

en.wikipedia.org/wiki/Strassen_algorithm
In the mathematical discipline of linear algebra, the Strassen algorithm, named after Volker Strassen, is an algorithm used for matrix multiplication. It is faster ... Coppersmith–Winograd - Volker Strassen - Schönhage–Strassen

[Fast Matrix Multiplication Algorithms](#)

www3.cs.stonybrook.edu/~algorithm/implementation/fastmm/implementation.shtml
Fast algorithms for matrix multiplication --- i.e., algorithms that compute less than ... Fast algorithms deploy new algorithmic strategies, new opportunities, thus ...

$$O(N^{2.81})$$



Example 2: Very Hot Code

N	multiply	strassen
32	33 μs	
64	252 μs	
128	3019 μs	2119 μs
256	56 ms	15 ms
512	488 ms	111 ms
1024	3270 ms	785 ms
2048	32 s	6 s
4096	309 s	42 s
8192	2611* s	293 s
<i>time/op</i>		

Let's try the other way

Example 2: JMH, -prof perfnorm, N=256*

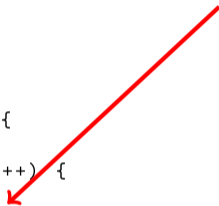
	per multiplication	per iteration
CPI	1.06	
cycles	146×10^6	8.7
instructions	137×10^6	8.2
L1-loads	68×10^6	4
L1-load-misses	33×10^6	2
L1-stores	0.56×10^6	
L1-store-misses	38357	
LLC-loads	26×10^6	1.6
LLC-stores	11595	

*~ 256Kb per matrix

Example 2: Very Hot Code


```
public Matrix multiply(Matrix other) {
    int size = data.length;
    Matrix result = new Matrix(size);
    int[][] R = result.data;
    int[][] A = this.data;
    int[][] B = other.data;
    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 result;
}
```

L1-dcache-load-misses



Example 2: Very Hot Code

```
public Matrix multiply(Matrix other) {
    int size = data.length;
    Matrix result = new Matrix(size);
    int[][] R = result.data;
    int[][] A = this.data;
    int[][] B = other.data;
    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 result;
}
```



Example 2: Very Hot Code

```
public Matrix multiplyIKJ(Matrix other) {
    int size = data.length;
    Matrix result = new Matrix(size);
    int[][] R = result.data;
    int[][] A = this.data;
    int[][] B = other.data;
    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 result;
}
```

Example 2: Very Hot Code

	IJK		IKJ	
N	multiply	strassen	multiply	
32	33 μs		13 μs	
64	252 μs		80 μs	
128	3019 μs	2119 μs	464 μs	
256	56 ms	15 ms	4 ms	
512	488 ms	111 ms	29 ms	
1024	3270 ms	785 ms	369 ms	
2048	32 s	6 s	3.4 s	
4096	309 s	42 s	25 s	
8192	2611 s	293 s	210 s	

time/op

Example 2: Very Hot Code

N	IJK		IKJ	
	multiply	strassen	multiply	strassen
32	33 μs		13 μs	
64	252 μs		80 μs	
128	3019 μs	2119 μs	464 μs	
256	56 ms	15 ms	4 ms	
512	488 ms	111 ms	29 ms	25 ms
1024	3270 ms	785 ms	369 ms	192 ms
2048	32 s	6 s	3.4 s	1.4 s
4096	309 s	42 s	25 s	10 s
8192	2611 s	293 s	210 s	73 s

time/op

Example 2: 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	38357		8959	
LLC-loads	26×10^6	1.6	0.3×10^6	
LLC-stores	11595		3532	

Example 2: Free beer benefits!

```
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
...
```

Example 2: Free beer benefits!

```
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
...
```

Vectorization (SSE/AVX)!

Example 3

~~«A plague on both your houses»~~
«++ on both your threads»

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: Let's collect HWC

All values are normalized per 10^3 operations

	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

Example 3: Let's collect HWC (cont.)

All values are normalized per 10^3 operations

	Same Core		Diff Cores		Diff Sockets	
	sharing	padded	sharing	padded	sharing	padded
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

Example 3: «on a core and a prayer»

in case of the single core(HT) we have to look into
MACHINE_CLEAR.SMEMORY_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
CLEAR.S	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

Conclusion:

Conclusion: 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!**

Conclusion: «Here be dragons»

To a



Viktor Klang

@viktorklang



Following

• A full stack developer is one who is close to
• overflow?



RETWEETS

408

FAVORITES

214



12:49 PM - 12 Mar 2015

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 ?