Caching In

Understand, measure and use your CPU Cache more effectively

jClarity

Richard Warburton

the hardware really wants to run fast and you only need to avoid getting in the way

- Luke Gorrie on Mechanical Sympathy

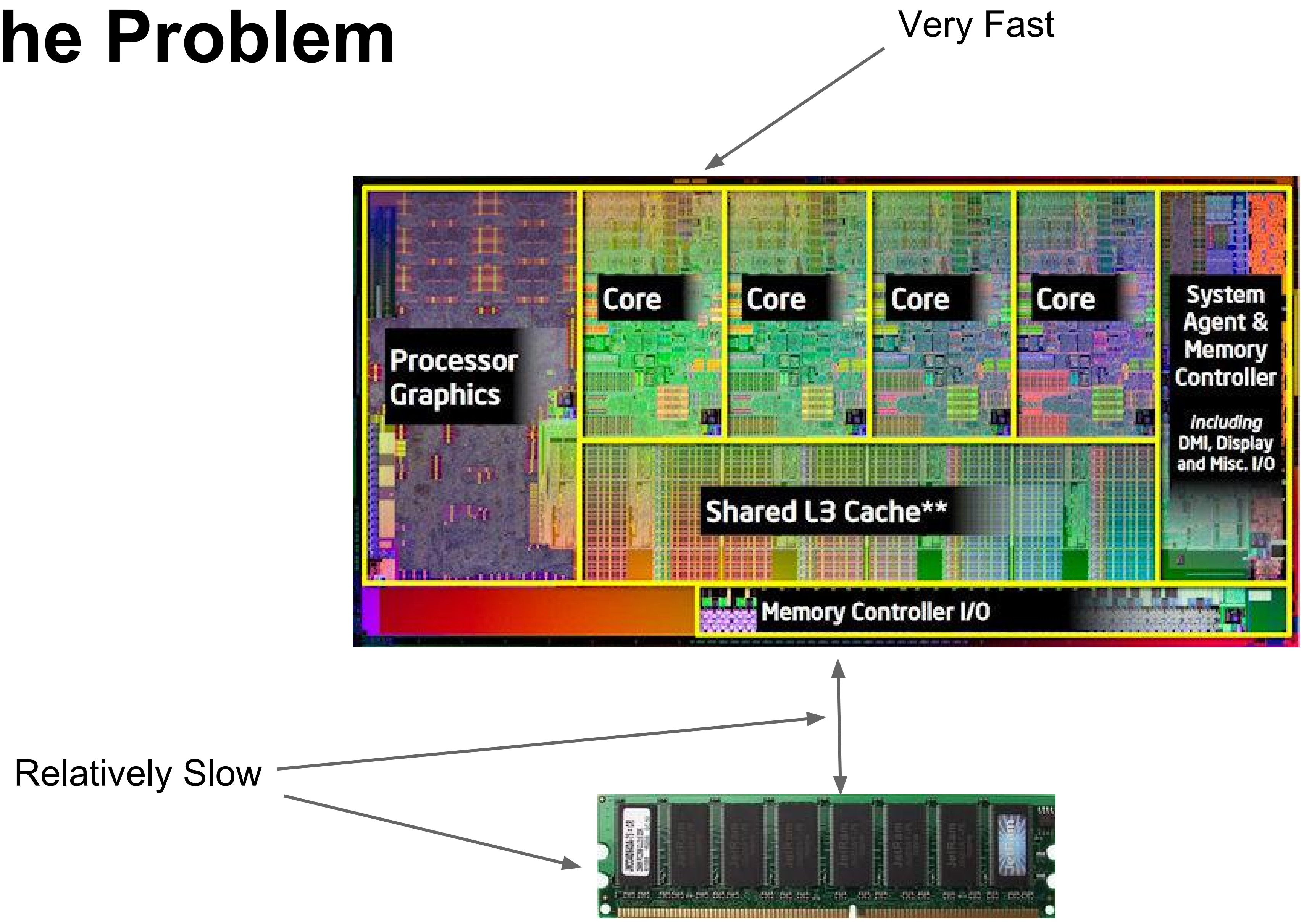
What are you talking about?

- Why do you care?
- Hardware Fundamentals
- Measurement
- Principles and Examples

Why do you care?

Perhaps why /should/ you care.

The Problem

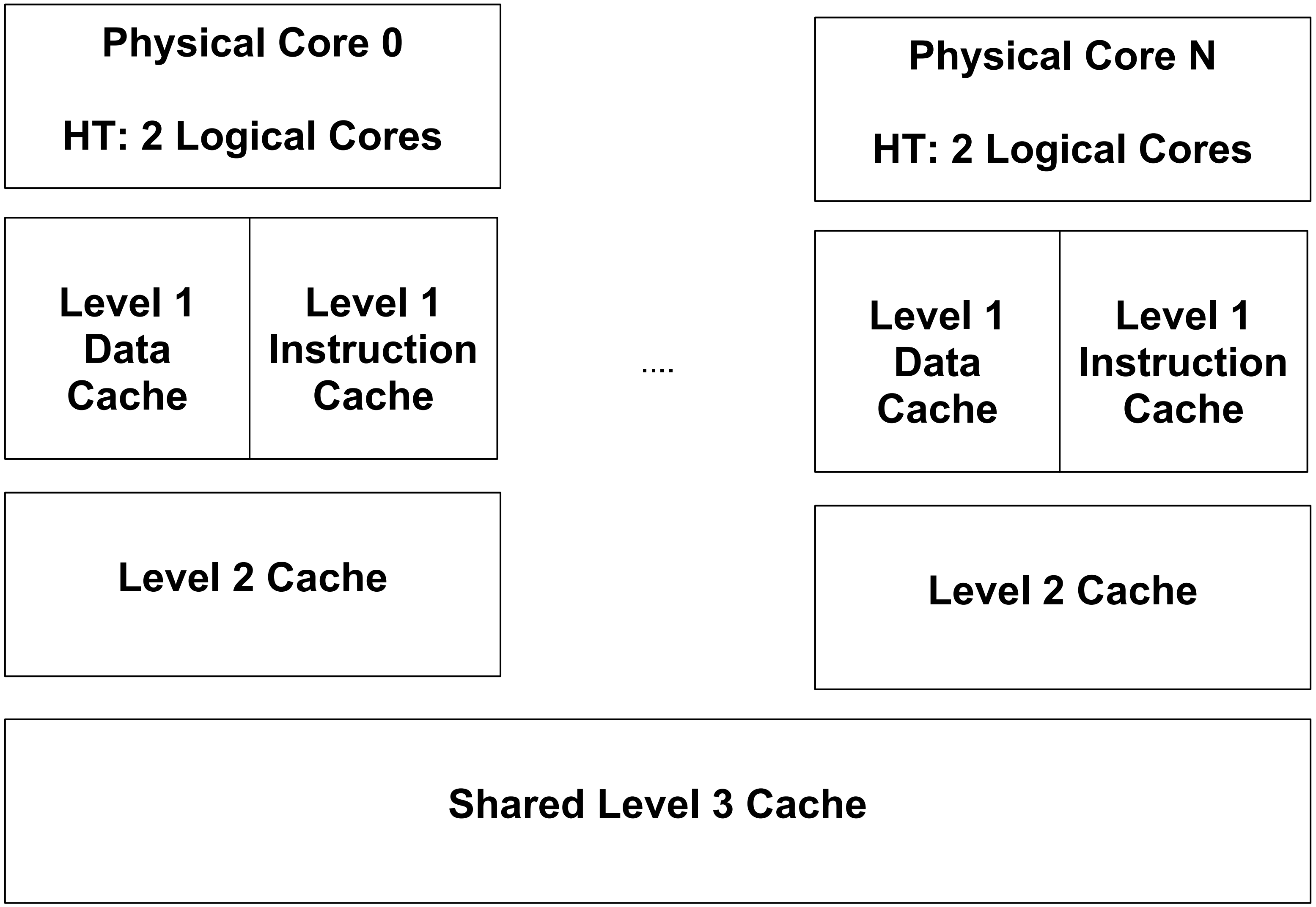

Very Fast
Processor Graphics
Core
Core
Core
Core
System Agent & Memory Controller
Including DMI, Display and Misc. I/O
Shared L3 Cache**
Memory Controller I/O
Relatively Slow

The Solution: CPU Cache

- Core Demands Data, looks at its cache
  - If present (a "hit") then data returned to register
  - If absent (a "miss") then data looked up from memory and stored in the cache

- Fast memory is expensive, a small amount is affordable

Multilevel Cache: Intel Sandybridge



CPU Stalls

- Want your CPU to execute instructions

- **Pipeline Stall** = CPU can't execute because its waiting on memory

- **Stalls displace potential computation**

- Busy-work Strategies
  - Out-of-Order Execution
  - Simultaneous MultiThreading (SMT) / HyperThreading (HT)

How bad is a miss?

Location	Latency in Clockcycles
Register	0
L1 Cache	3
L2 Cache	9
L3 Cache	21
Main Memory	**150-400**

NB: Sandybridge Numbers

Cache Lines

- Data transferred in cache lines
- Fixed size block of memory
- Usually 64 bytes in current x86 CPUs
  - Between 32 and 256 bytes
- Purely hardware consideration

You don't need to know everything

Architectural Takeaways

- Modern CPUs have large multilevel Caches
- Cache misses cause Stalls
- Stalls are expensive

Measurement

Barometer, Sun Dial ... and CPU Counters

Do you care?

- **DON'T** look at CPU Caching behaviour first

- Many problems not Execution Bound
  - Networking
  - Database or External Service
  - I/O
  - Garbage Collection
  - Insufficient Parallelism

- Consider caching behaviour when you're execution bound and know your hotspot.

CPU Performance Event Counters

- CPU Register which can count events
  - Eg: the number of Level 3 Cache Misses

- Model Specific Registers
  - not instruction set standardised by x86
  - differ by CPU (eg Penryn vs Sandybridge) and manufacturer

- Don't worry - leave details to tooling

Measurement: Instructions Retired

- The number of instructions which executed until completion
    - ignores branch mispredictions
- When stalled you're not retiring instructions
- Aim to maximise instruction retirement when reducing cache misses

Cache Misses

- Cache Level
  - Level 3
  - Level 2
  - Level 1 (Data vs Instruction)

- What to measure
  - Hits
  - Misses
  - Reads vs Writes
  - Calculate Ratio

Cache Profilers

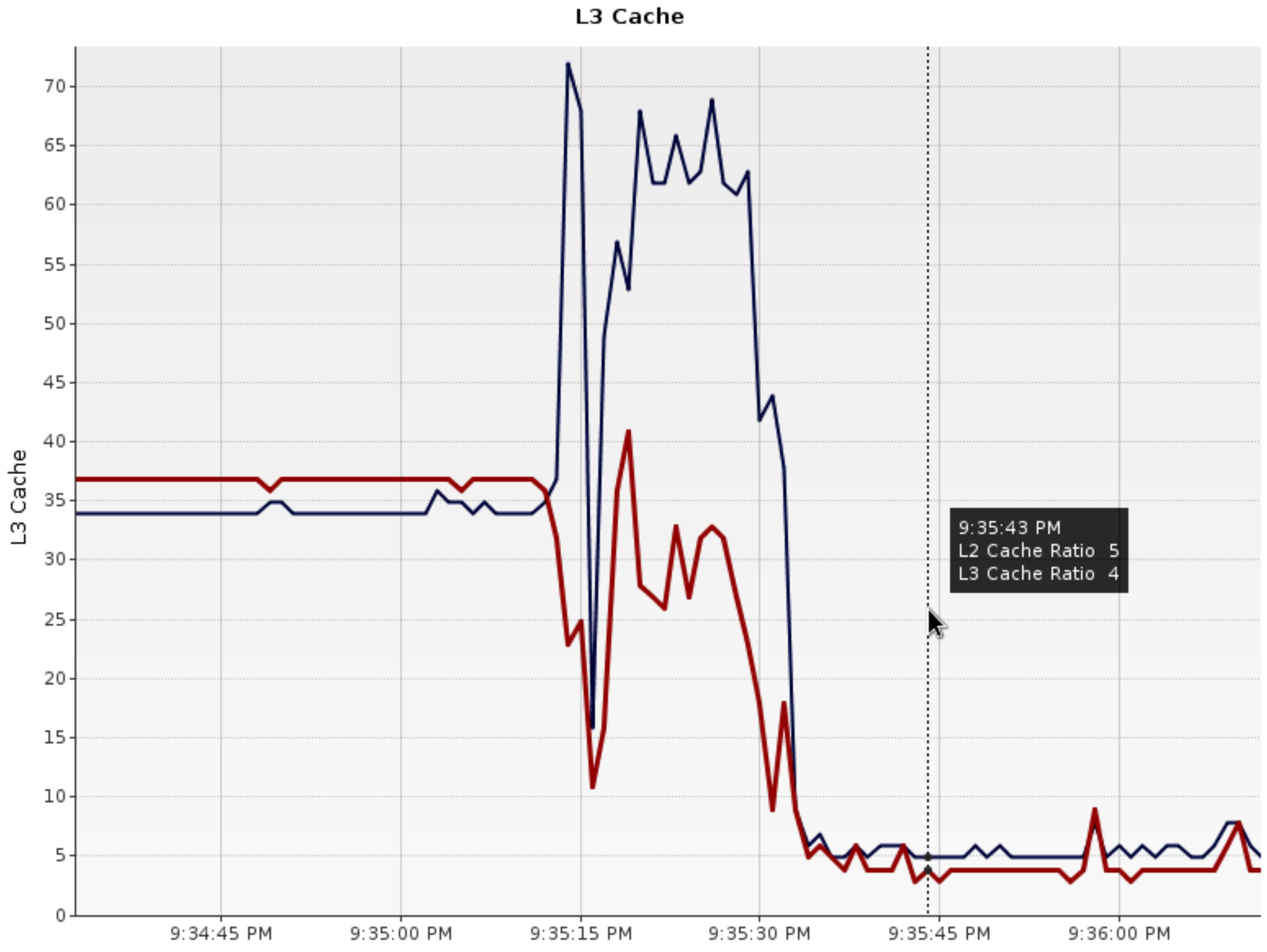
- Open Source
  - perf
  - linux (rdmsr/wrmsr)
  - Linux Kernel (via /proc)

- Proprietary
  - jClarity jMSR
  - Intel VTune
  - AMD Code Analyst
  - Visual Studio 2012

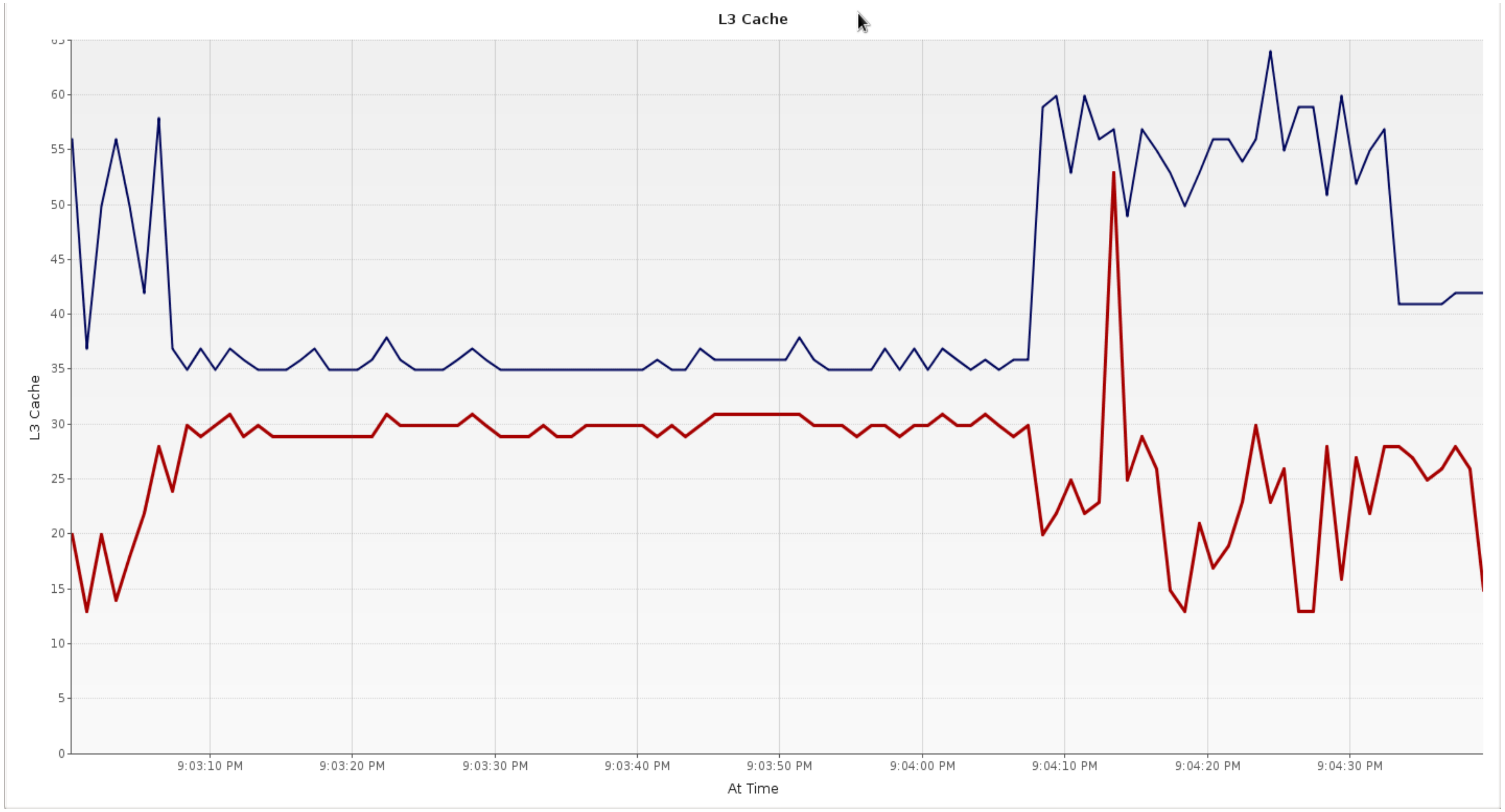
Good Benchmarking Practice

- Warmups
- Measure and rule-out other factors
- Specific Caching Ideas ...

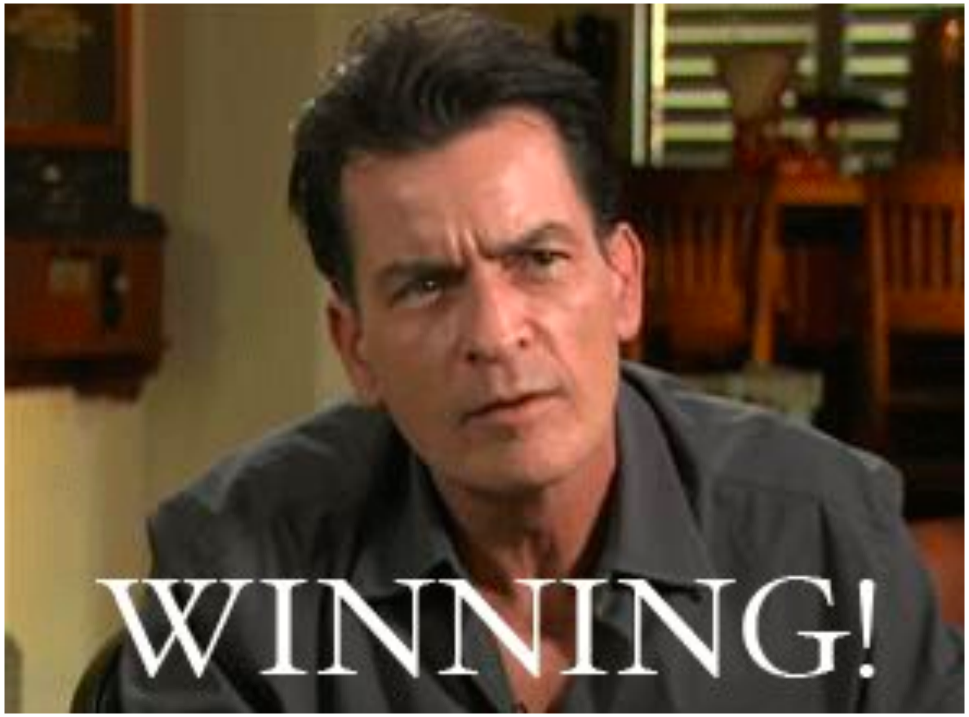
Take Care with Working Set Size



Good Benchmark has Low Variance



You can measure Cache behaviour



Principles & Examples

Sequential Code

Prefetching

- Prefetch = Eagerly load data
- Adjacent Cache Line Prefetch
- Data Cache Unit (Streaming) Prefetch
- Problem: CPU Prediction isn't perfect
- Solution: Arrange Data so accesses are predictable

Temporal Locality

Repeatedly referring to same data in a short time span

Spatial Locality

Referring to data that is close together in memory

Sequential Locality

Referring to data that is arranged linearly in memory

General Principles

- Use smaller data types (`-XX:+UseCompressedOops`)
- Avoid 'big holes' in your data
- Make accesses as linear as possible

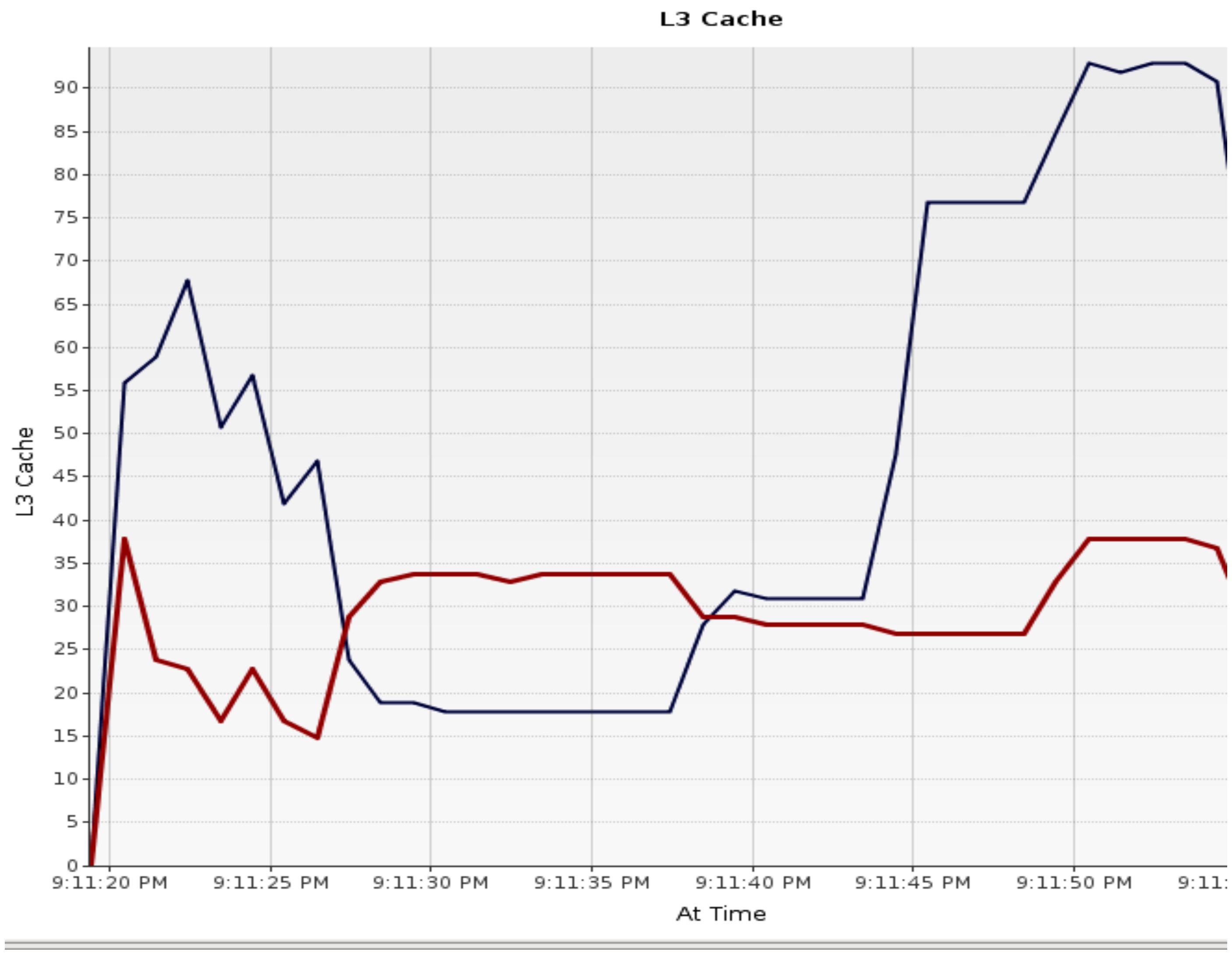
Primitive Arrays

```
// Sequential Access = Predictable

for (int i=0; i<someArray.length; i++)
   someArray[i]++;
```

Primitive Arrays - Skipping Elements

```
// Holes Hurt

for (int i=0; i<someArray.length; i += SKIP)
   someArray[i]++;
```

Primitive Arrays - Skipping Elements

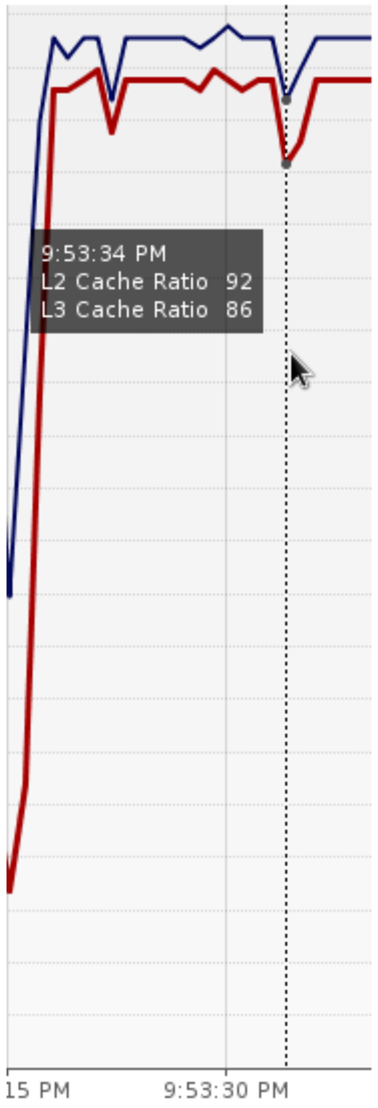


Multidimensional Arrays

- Multidimensional Arrays are really Arrays of Arrays in Java. (Unlike C)
- Some people realign their accesses:

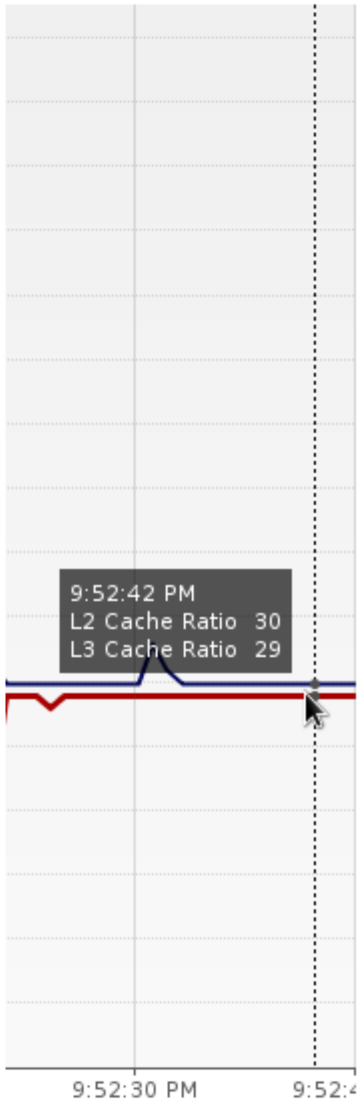
```
for (int col=0; col<COLS; col++) {
 for (int row=0; row<ROWS; row++) {
  array[ROWS * col + row]++;
 }
}
```

Bad Access Alignment



Strides the wrong way, bad locality.

```
array[COLS * row + col]++;
```



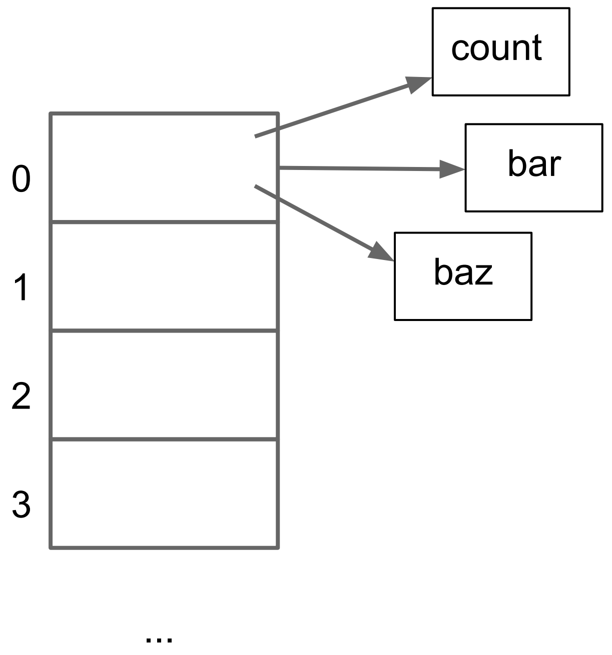
Strides the right way, good locality.

```
array[ROWS * col + row]++;
```

Data Layout Principles

- Primitive Collections (GNU Trove, etc.)
- Arrays > Linked Lists
- Hashtable > Search Tree
- Avoid Code bloating (Loop Unrolling)
- Custom Data Structures
  - Judy Arrays
    - an associative array/map
  - kD-Trees
    - generalised Binary Space Partitioning
  - Z-Order Curve
    - multidimensional data in one dimension

Data Locality vs Java Heap Layout



```
class Foo {
   Integer count;
   Bar bar;
   Baz baz;
}

// No alignment guarantees
for (Foo foo : foos) {
   foo.count = 5;
   foo.bar.visit();
}
```

Data Locality vs Java Heap Layout

- Serious Java Weakness
- Location of objects in memory hard to guarantee.
- GC also interferes
    - Copying
    - Compaction

Alternative Approach

- Use `sun.misc.Unsafe` to directly allocate memory
- Manage mapping yourself
- Use with care
  - Maybe slower
  - Maybe error prone
  - Definitely hard work

Game Event Server

```
class PlayerAttack {
  private long playerId;
  private int weapon;
  private int energy;
  ...
  private int getWeapon() {
    return weapon;
  }
  ...
```

Flyweight Unsafe (1)

```
static final Unsafe unsafe = ... ;

long space = OBJ_COUNT * ATTACK_OBJ_SIZE;

address = unsafe.allocateMemory(space);

static PlayerAttack get(int index) {
   long loc = address +
                  (ATTACK_OBJ_SIZE * index)
   return new PlayerAttack(loc);
}
```

Flyweight Unsafe (2)

```
class PlayerAttack {
  static final long WEAPON_OFFSET = 8
  private long loc;
  public int getWeapon() {
    long address = loc + WEAPON_OFFSET
    return unsafe.getInt(address);
  }
  public void setWeapon(int weapon) {
    long address = loc + WEAPON_OFFSET
    unsafe.setInt(address, weapon);
  }
}
```

SLAB

- Manually writing this is:
    - time consuming
    - error prone
    - boilerplate-ridden

- Prototype library:
    - http://github.com/RichardWarburton/slab
    - Maps an interface to a flyweight

Slab (2)

```
public interface GameEvent extends Cursor {
  public int getId();
  public void setId(int value);
  public long getStrength();
  public void setStrength(long value);
}

Allocator<GameEvent> eventAllocator =
                                        Allocator.of(GameEvent.class);

GameEvent event = eventAllocator.allocate(100);

event.move(50);

event.setId(6);
assertEquals(6, event.getId());
```

Data Alignment

- An address `a` is `n`*-byte aligned* when:
  - `n` is a power of two
  - `a` is divisible by `n`
- Cache Line Aligned:
  - byte aligned to the size of a cache line

Rules and Hacks

- Java (Hotspot) ♥ 8-byte alignment:
  - `new` Java Objects
  - `Unsafe.allocateMemory`
  - `Bytebuffer.allocateDirect`

- Get the object address:
  - Unsafe gives it to you directly
  - Direct `ByteBuffer` has a field called 'address'

Cacheline Aligned Access

- Performance Benefits:
  - 3-6x on Core2Duo
  - ~1.5x on Nehalem Class
  - Measured using direct ByteBuffers
  - Mid Aligned vs Straddling a Cacheline

- http://psy-lob-saw.blogspot.co.uk/2013/01/direct-memory-alignment-in-java.html

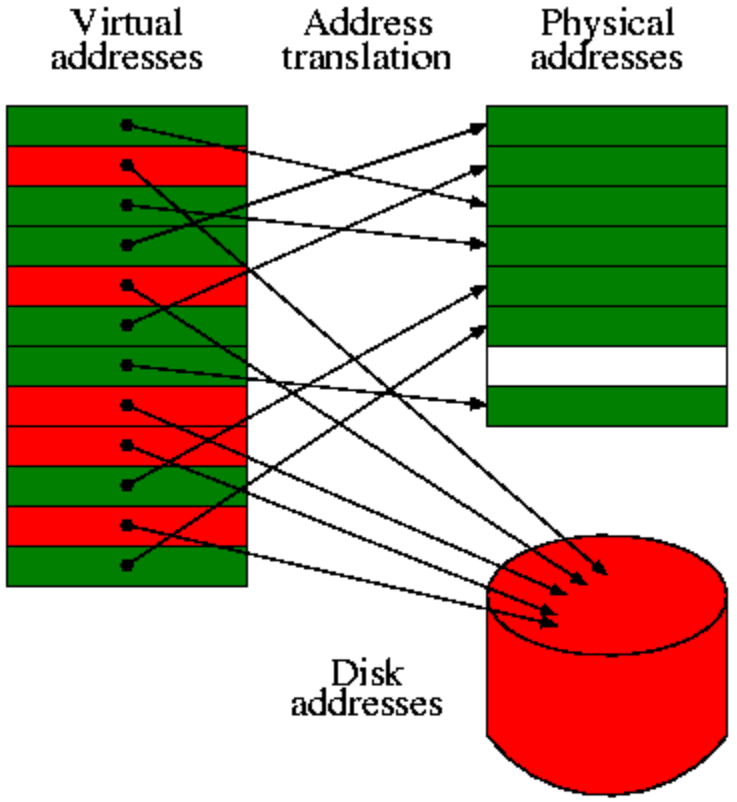
Today I learned ...

Access and Cache Aligned, Contiguous Data

Translation Lookaside Buffers

TLB, not TLAB!

Virtual Memory



- RAM is finite
- RAM is fragmented
- Multitasking is nice
- Programming easier with a contiguous address space

Page Tables

- Virtual Address Space split into pages

- Page has two Addresses:
  - Virtual Address: Process' view
  - Physical Address: Location in RAM/Disk

- Page Table
  - Mapping Between these addresses
  - OS Managed
  - Page Fault when entry is missing
  - OS Pages in from disk

Translation Lookaside Buffers

- TLB = Page Table Cache
- Avoids memory bottlenecks in page lookups
- CPU Cache is multi-level => TLB is multi-level
- Miss/Hit rates measurable via CPU Counters
    - Hit/Miss Rates
    - Walk Duration

TLB Flushing

- Process context switch => change address space
- Historically caused "flush" =

- Avoided with an Address Space Identifier (ASID)

Page Size Tradeoff:

Bigger Pages
- = less pages
- = quicker page lookups

vs

Bigger Pages
- = wasted memory space
- = more disk paging

Page Sizes

- Page Size is adjustable
- Common Sizes: 4KB, 2MB, 1GB
- Potential Speedup: 10%-30%
- `-XX:+UseLargePages`
- Oracle DBs particularly

Too Long, Didn't Listen

1. Virtual Memory + Paging have an overhead

2. Translation Lookaside Buffers used to reduce cost

3. Page size changes important

Principles & Examples (2)

Concurrent Code

Context Switching

- Multitasking systems 'context switch' between processes

- Variants:
  - Thread
  - Process
  - Virtualized OS

Context Switching Costs

- Direct Cost
  - Save old process state
  - Schedule new process
  - Load new process state

- Indirect Cost
  - TLB Reload (Process only)
  - CPU Pipeline Flush
  - Cache Interference
  - Temporal Locality

Indirect Costs (cont.)

- "Quantifying The Cost of Context Switch" by Li et al.
  - 5 years old - principle still applies
  - Direct = 3.8 microseconds
  - Indirect = ~0 - 1500 microseconds
  - indirect dominates direct when working set > L2 Cache size
- Cooperative hits also exist
- Minimise context switching if possible

Locking

- Locks require kernel arbitration
    - Not a context switch, but a mode switch
    - Lots of lock contention causes context switching
- Has a cache-pollution cost
- Can use try lock-free algorithms

GC Safepointing

- GC needs to pause program
  - Safe points
- Safe Points cause a page fault
- Also generates Context Switch between Mutator and GC
  - Not guaranteed to be scheduled back on the same core

Java Memory Model

- JSR 133 Specs the memory model
- Turtles Example:
    - Java Level: `volatile`
    - CPU Level: `lock`
    - Cache Level: MESI
- Has a cost!

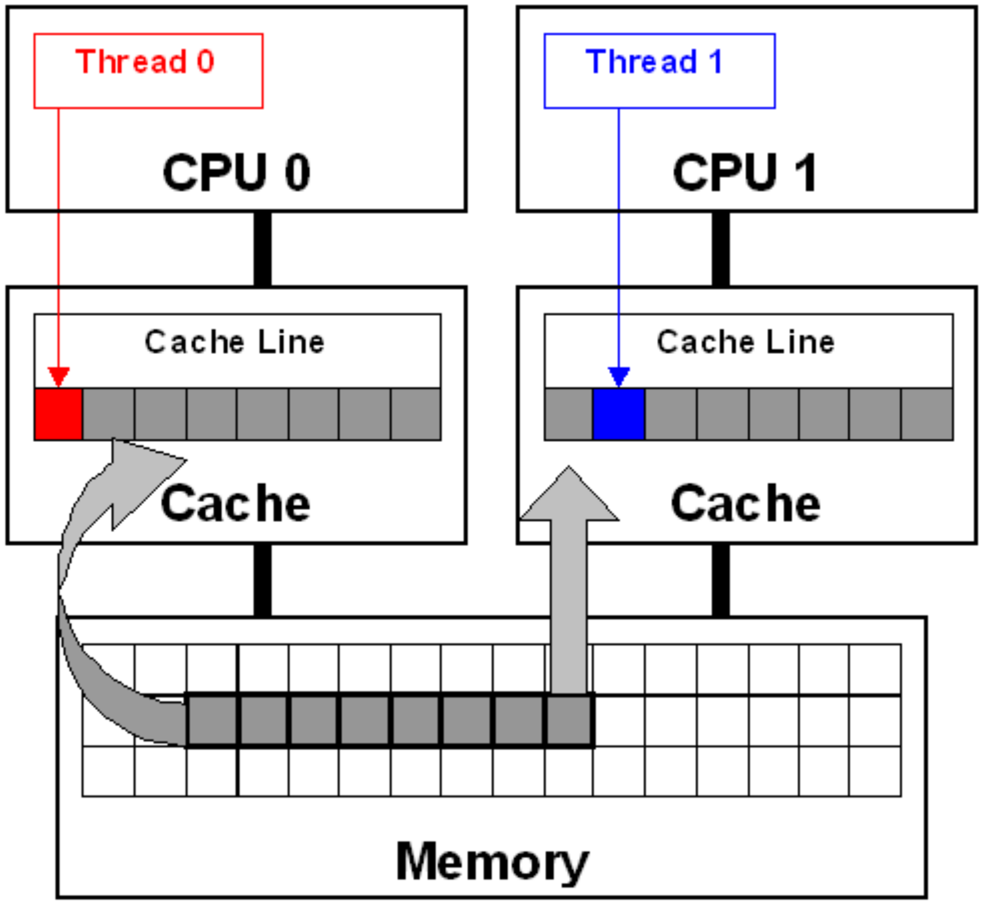
False Sharing

- Data can share a cache line

- Not always good
    - Some data 'accidentally' next to other data.
    - Causes Cache lines to be evicted prematurely

- Serious Concurrency Issue

Concurrent Cache Line Access



Current Solution: Field Padding

```
public volatile long value;
public long pad1, pad2, pad3, pad4,
pad5, pad6;
```

8 bytes(long) * 7 fields + header = 64 bytes = Cacheline

NB: fields aligned to 8 bytes even if smaller

Real Solution: JEP 142

```
class UncontendedFoo {

   int someValue;

   @Contended volatile long foo;

   Bar bar;
}
```

http://openjdk.java.net/jeps/142

False Sharing in Your GC

- Card Tables
  - Split RAM into 'cards'
  - Table records which parts of RAM are written to
  - Avoid rescanning large blocks of RAM

- BUT ... optimise by writing to the byte on every write

- -XX:+UseCondCardMark
  - Use With Care: 15-20% sequential slowdown

Buyer Beware:

- Context Switching
- False Sharing
- Visibility/Atomicity/Locking Costs

Too Long, Didn't Listen

1. Caching Behaviour has a performance effect

2. The effects are measurable

3. There are common problems and solutions

Useful Links

- My blog
  - insightfullogic.com
- Lots about memory
  - akkadia.org/drepper/cpumemory.pdf
- jClarity
  - jclarity.com/friends/
- Martin Thompson's blog
  - mechanical-sympathy.blogspot.co.uk
- Concurrency Interest
  - g.oswego.edu/dl/concurrency-interest/

Questions?

@RichardWarburto

Garbage Collection Impact

- Young Gen Copying Collector
  - Copying changes location
  - Sequential Allocation helps

- Concurrent Mark Sweep
  - lack of compaction until CMF hurts

- Parallel Old GC
  - Compaction at Full GC helps

A Brave New World

(hopefully, sort of)

Arrays 2.0

- Proposed JVM Feature
- Library Definition of Array Types
- Incorporate a notion of 'flatness'
- Semi-reified Generics
- Loads of other goodies (final, volatile, resizing, etc.)
- Requires Value Types

Value Types

- Assign or pass the object you copy the value and not the reference
- Allow control of the memory layout
- Control of layout = Control of Caching Behaviour
- Idea, initial prototypes in mlvm

The future will solve all problems!