B4M36ESW: Efficient software

Lecture 1: Introduction

Michal Sojka
michal.sojka@cvut.cz

February 19, 2019

Outline

1. About the course
2. Basics
3. Hardware
4. Making the hardware faster
   - Caches
   - Instruction-level parallelism
   - Task parallelism
5. Energy
6. Exercise today
   - C/C++ compiler
   - Profiling

About this course

Teachers

Michal Sojka C/C++, embedded systems, operating systems

David Šišlák Java, servers, ...

Scope

- Writing fast programs
- Single (multi-core) computer, no distributed systems/cloud
- Interaction between software and **hardware**
- How general concepts apply to programs in both C/C++ and Java i.e. how to use **hardware** efficiently from C/C++ and Java
- The course is not about comparing C/C++ and Java,
  - but you should be able to make this comparison yourself at the end.

Grading

- Exercises
  - 7 small tasks
  - semestral work (both C/C++ and Java)
  - Maximum 60 points
  - **Minimum 30 points**
- Exam
  - Written test: max. 30 points
  - Voluntary oral exam: 10 points
  - **Minimum: 20 points**

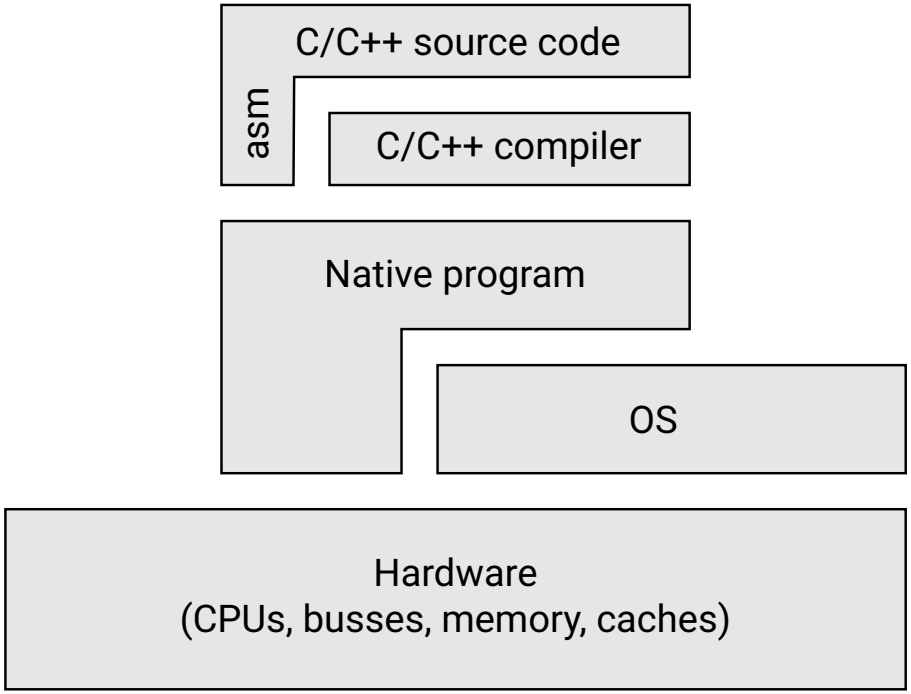
Outline

1. About the course
2. Basics
3. Hardware
4. Making the hardware faster
   - Caches
   - Instruction-level parallelism
   - Task parallelism
5. Energy
6. Exercise today
   - C/C++ compiler
   - Profiling

Efficient software

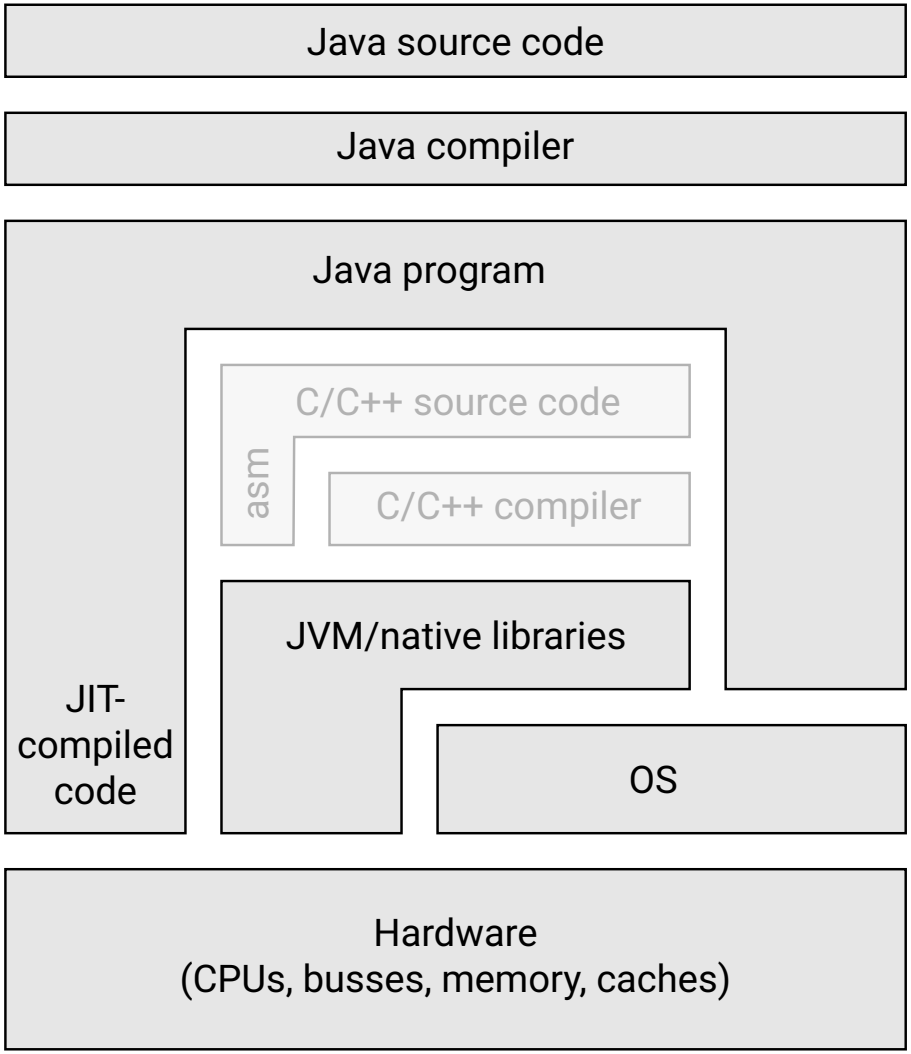
- There is no theory of how to write efficient software
- Writing efficient software is about:
  - Knowledge of all layers involved
  - Experience in knowing when and how performance can be a problem
  - Skill in detecting and zooming in on the problems
  - A good dose of common sense
- Best practices
  - Patterns that occur regularly
  - Typical mistakes

Layers involved in software execution



- In the end, everything is executed by hardware
  - Majority of this course is about how to tailor the code to use the hardware efficiently
- C/C++ source code is transformed into native (machine) code by the compiler
  - Compiler tries to optimize the generated code
  - Optimizations are often only heuristics
- Native code is executed directly or invokes OS services

Layers involved in software execution



- Java source code is also compiled
- Java program can execute
  - interpreted by Java Virtual Machine (JVM) or
  - natively after being just-it-time (JIT) compiled by JVM
- JVM is a native program
- Java program can use native libraries (JNI)
- ... long way from source to HW

Fundamental theorem of software engineering

All problems in computer science can be solved by another level of indirection

... except for the problem of too many layers of indirection.

—David Wheeler

Layers of indirection in today's systems

Hardware

- microcode, ISA
- virtual memory, MMU
- buses, arbiters

Software

- operating system kernel
- compiler
- language runtime
- application frameworks

Outline

1 About the course

2 Basics

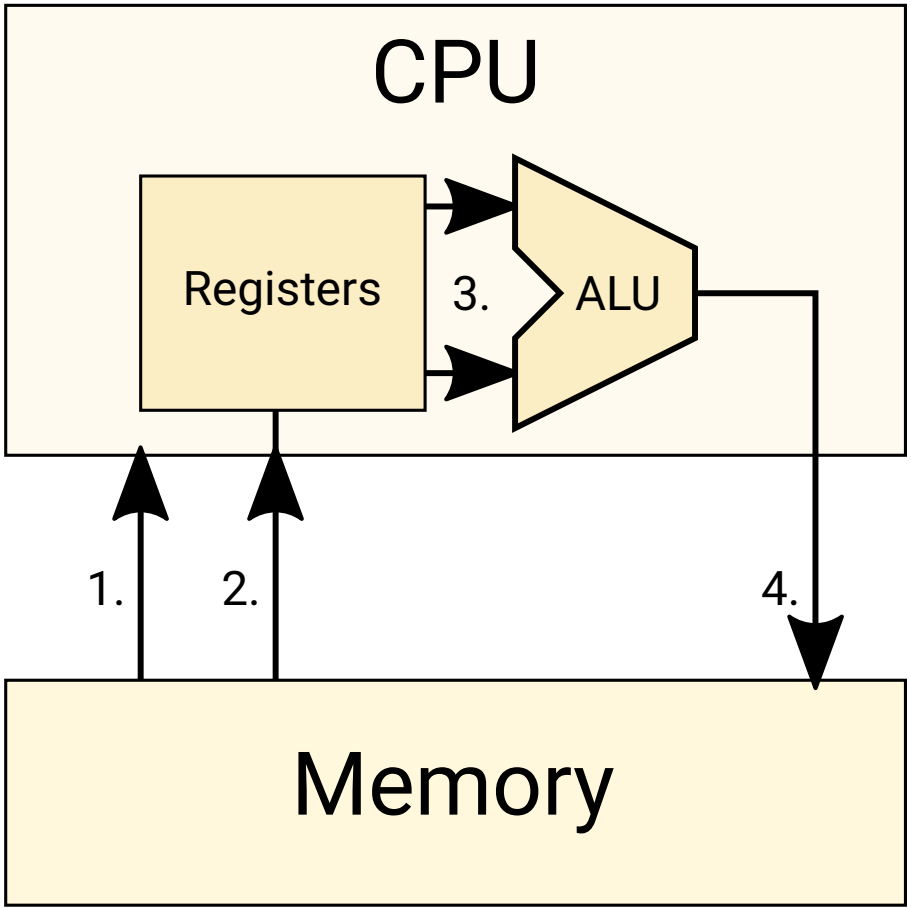
3 Hardware

4 Making the hardware faster
- Caches
- Instruction-level parallelism
- Task parallelism

5 Energy

6 Exercise today
- C/C++ compiler
- Profiling

CPU – principle of operation



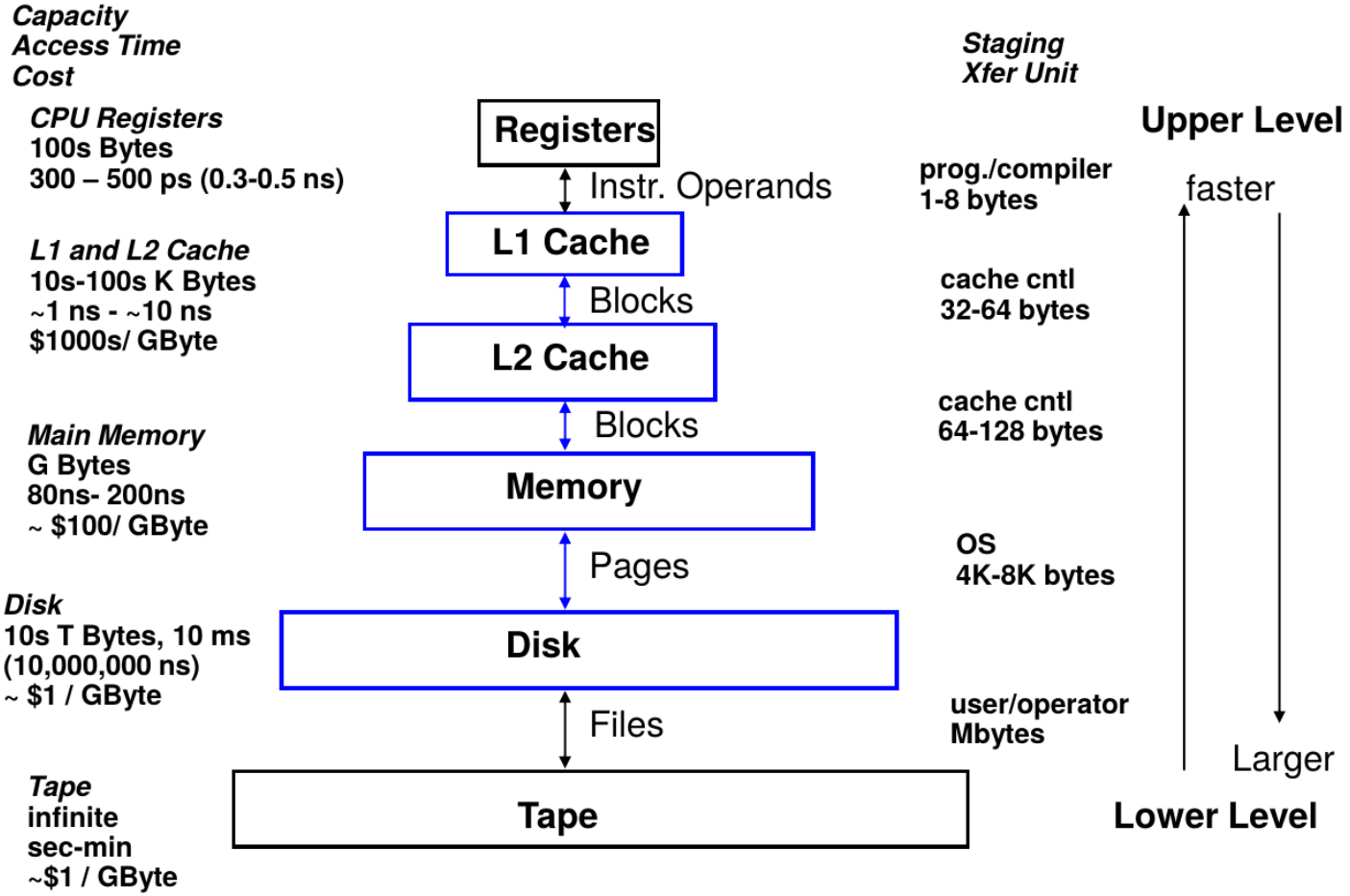
1. Fetch instruction from memory
2. Fetch data from memory
3. Perform computation
4. Store the result to memory

C code and machine code

```
int a, b, r;
void func() {
  r = a + b;
}
mov 0x100,%eax   ; load a
add 0x104,%eax   ; add b
mov %eax,0x108   ; store r
```

Memory

- Source of many performance problems in today's computers
- Reason: Memory is slow compared to CPUs!
- Solution: Caching ⇒ memory hierarchy



Latencies in computer systems

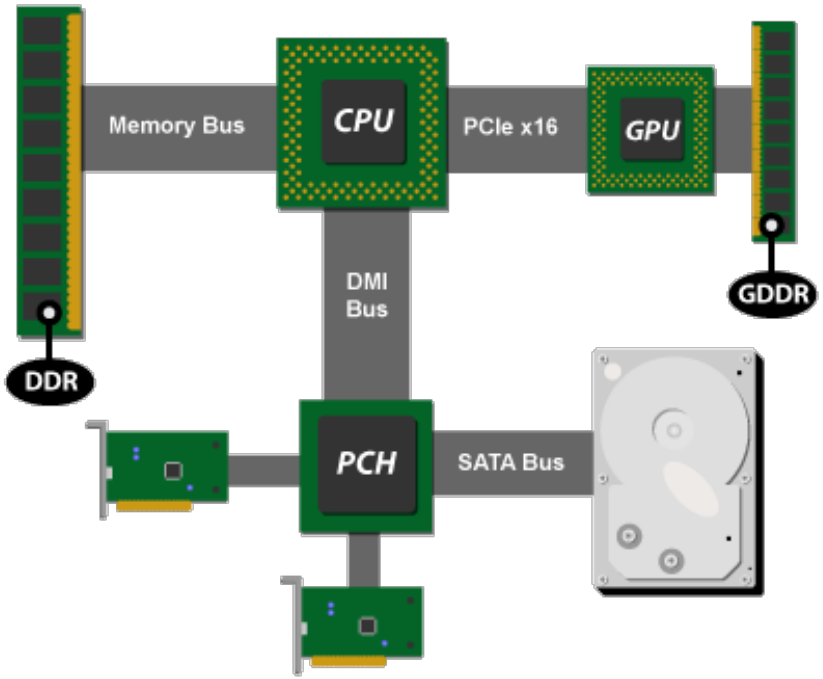
Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 µs	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–117 years
OS virtualization (container) system reboot	4 s	423 years
SCSI command timeout	30 s	3 millennia
HW virtualization system reboot	40 s	4 millennia
Physical server system reboot	5 m	32 millenia

Computer performance and laws of physics

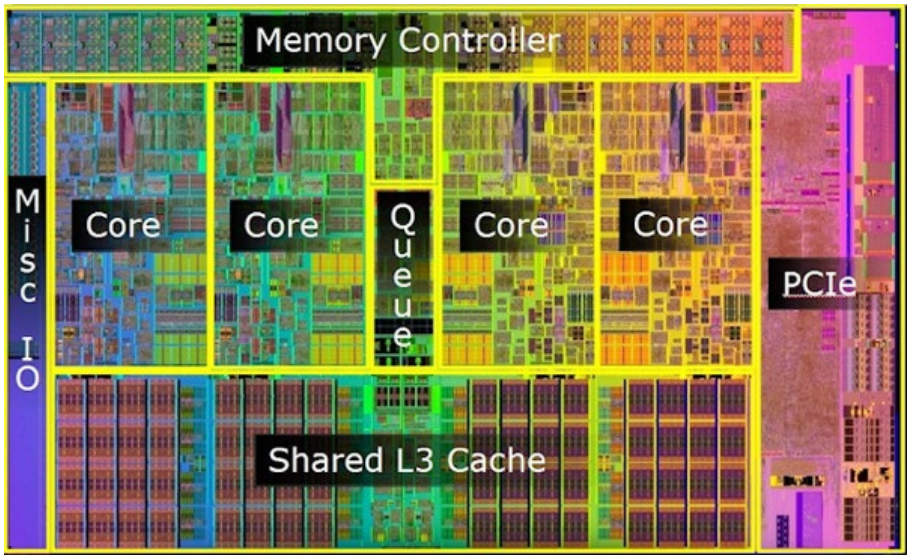
What distance travels light in vacuum during one 3 GHz CPU clock cycle?

- 10 cm
- Speed of light in silicon is even slower
- Each gate delays the information a bit
- It's already difficult to pass information quickly from one side of the chip to another
- Physical distance plays important role in the speed of computation

Example: Intel-based system (single socket, 2009)



Intel's P55 platform
Source: ArsTechnica



Lynnfield CPU
Source: Intel

Outline

1 About the course

2 Basics

3 Hardware

4 Making the hardware faster
- Caches
- Instruction-level parallelism
- Task parallelism

5 Energy

6 Exercise today
- C/C++ compiler
- Profiling

Making the hardware faster

... and more tricky to use it efficiently from software

- Hardware designers intensively optimize their hardware
- These optimizations improve performance in common (average) cases
- Using the HW in “uncommon” ways can drastically degrade the performance
- The layers between source code and hardware complicate understanding how is the hardware actually “used”
- What are the features that can be problematic from performance point of view?
- We will look at them in more detail in the rest of the lectures.

Outline

1 About the course

2 Basics

3 Hardware

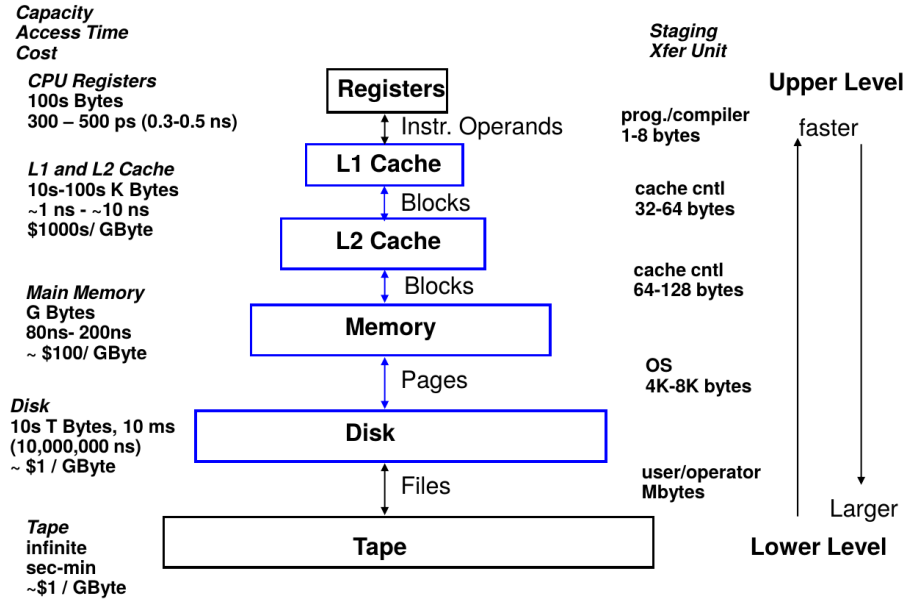
4 Making the hardware faster
- Caches
- Instruction-level parallelism
- Task parallelism

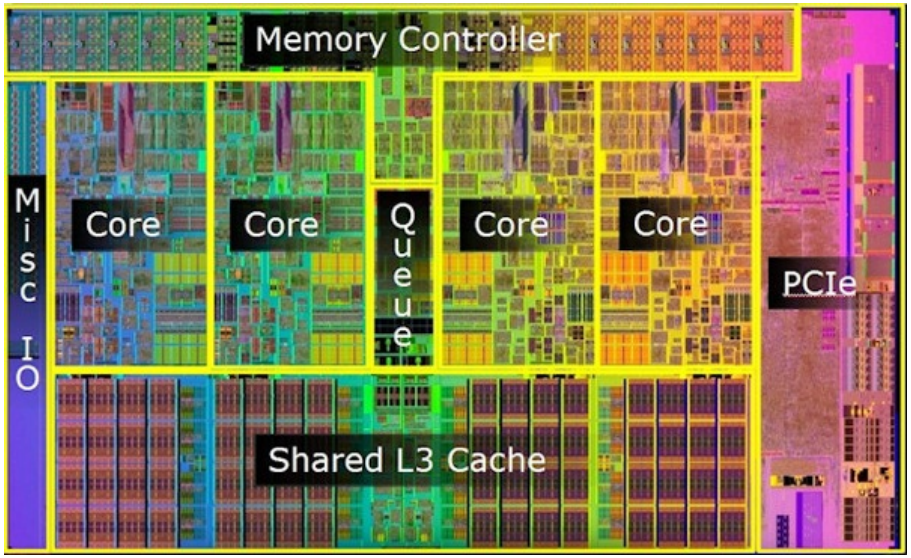
5 Energy

6 Exercise today
- C/C++ compiler
- Profiling

Caches

- Principle
  - Smaller but faster memory
  - Take advantage of spacial and temporal locality of memory accesses performed by the code.
- Problems
  - Random Access Memory (RAM) is no longer RAM from performance point of view
  - Management of multiple copies of a single data... (known as **cache coherence**)





Outline

1 About the course

2 Basics

3 Hardware

4 Making the hardware faster
- Caches
- Instruction-level parallelism
- Task parallelism

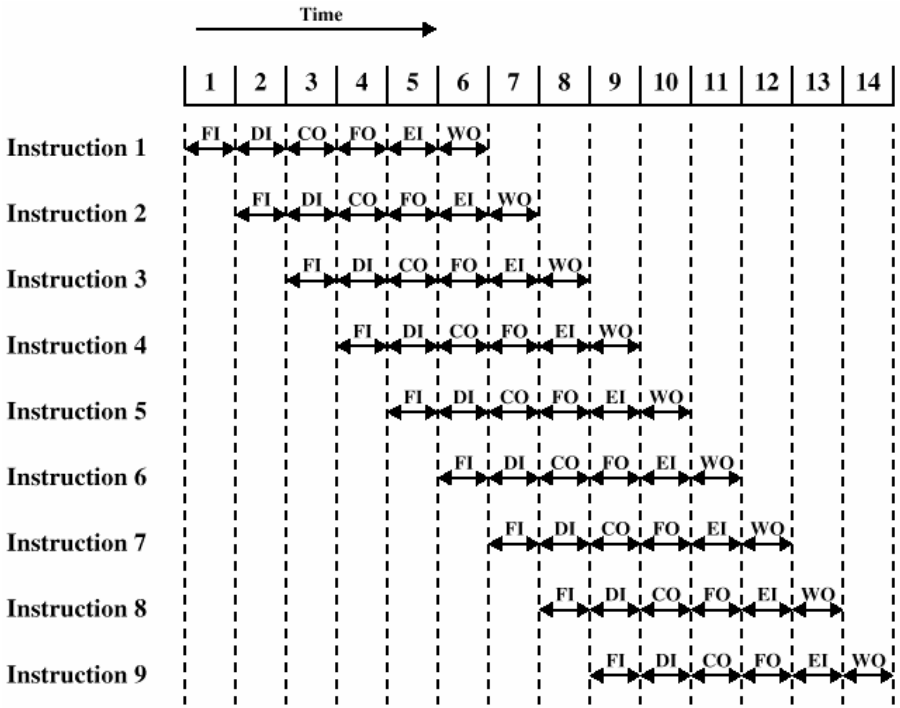
5 Energy

6 Exercise today
- C/C++ compiler
- Profiling

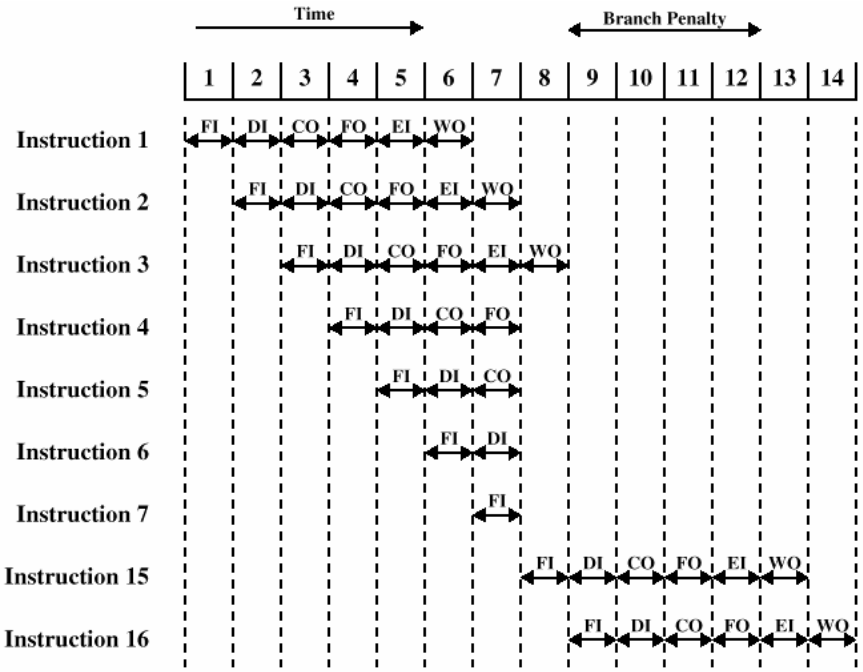
Pipelining, branch prediction

Branch = if/else

Branch not taken



Branch taken

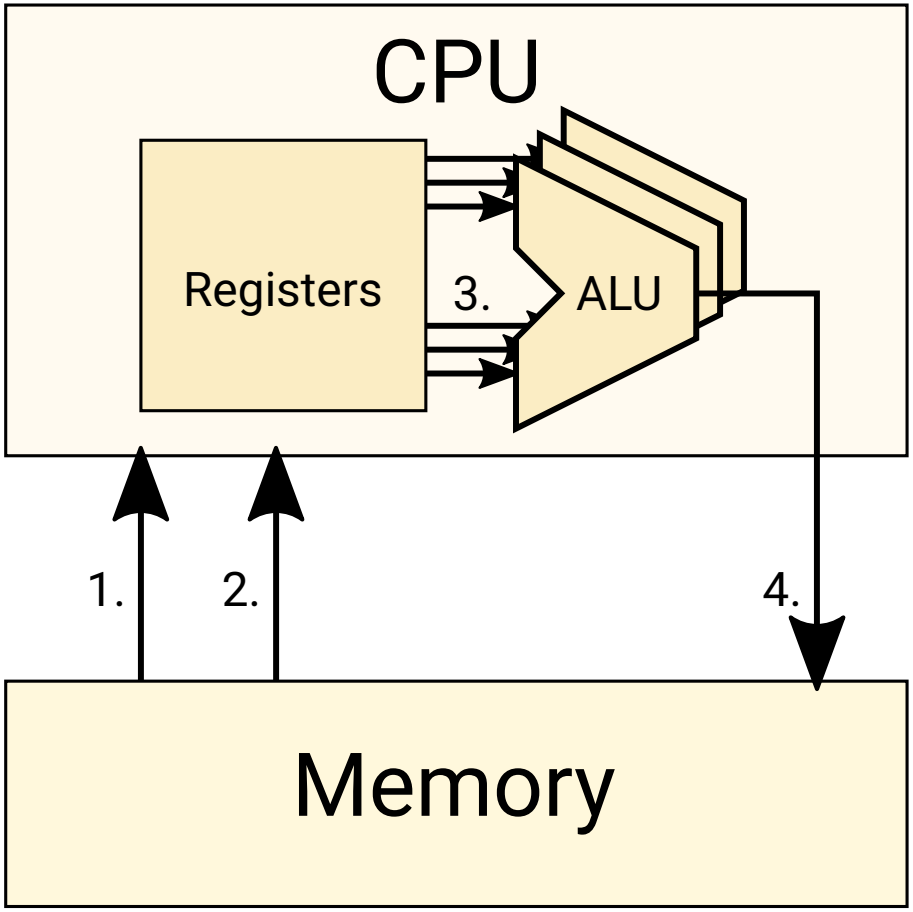


Example pipeline stages

1. Fetch instruction
2. Decode instruction
3. Calculate operands
4. Fetch operands
5. Execute instruction
6. Write output (result)

- Branch predictor tries to predict branch target and condition
- If it fails, we pay branch penalty
- Here, branch penalty is a *few cycles*, but it is much more severe with things like superscalar CPUs are involved.

Superscalar CPUs



Instruction stream

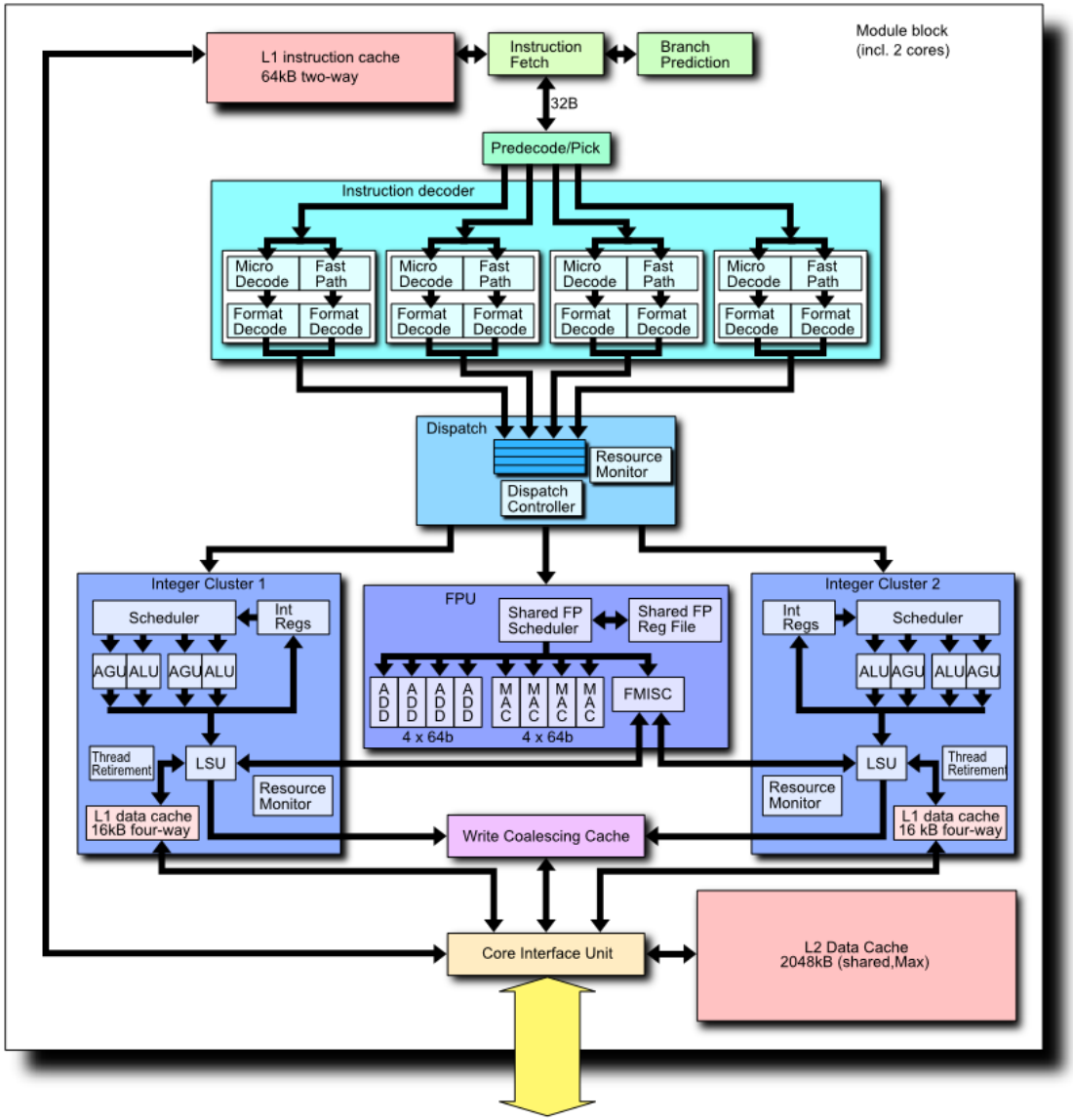
r = a + b
s = c + d
t = e + f
u = g + h
v = u + i

Superscalar execution

r = a + b; s = c + d; t = e + f
u = g + h
v = u + i

- Goal: Order instructions in a program to use all execution units (e.g. ALUs) in parallel
- Task for the compiler
- Complicates reading of assembler

Example: AMD Bulldozer CPU



Outline

1 About the course

2 Basics

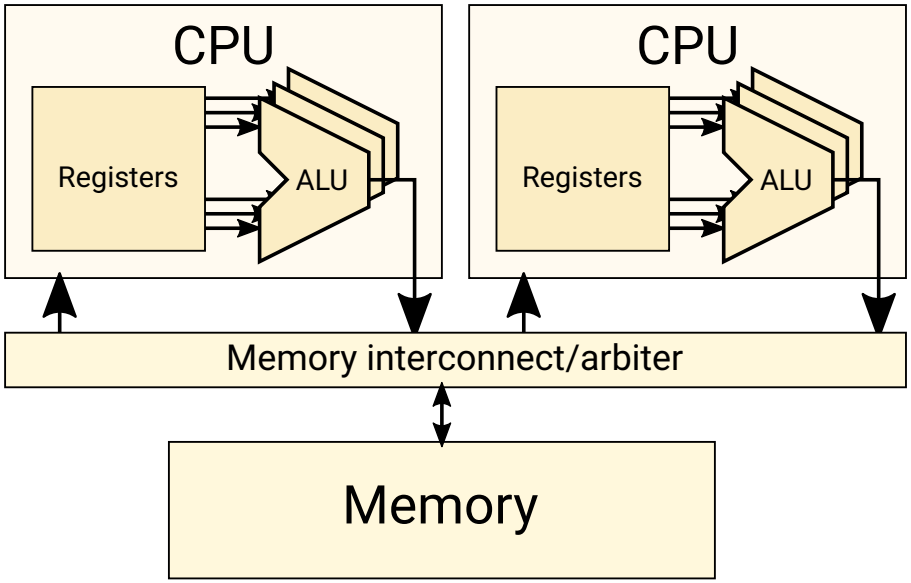
3 Hardware

4 Making the hardware faster
- Caches
- Instruction-level parallelism
- Task parallelism

5 Energy

6 Exercise today
- C/C++ compiler
- Profiling

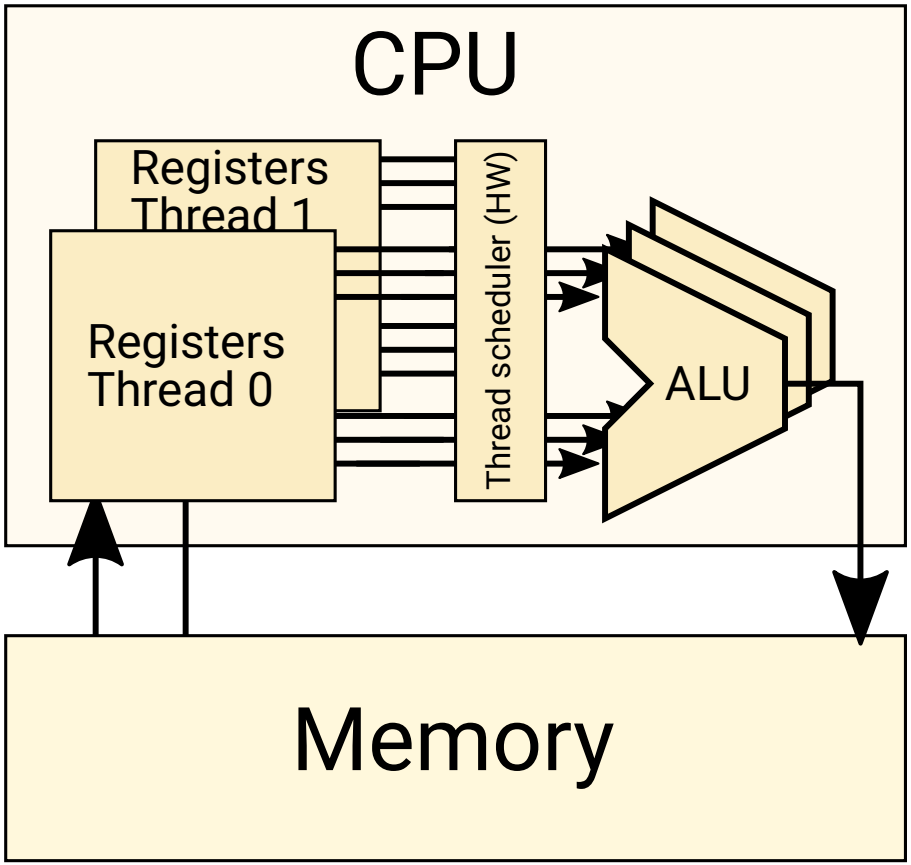
Multiple CPUs



- Computers usually run multiple programs simultaneously
- Let's execute them simultaneously on two CPUs
- The CPUs can be on
  - single chip ⇒ multi-core
  - multiple chips ⇒ multi-socket

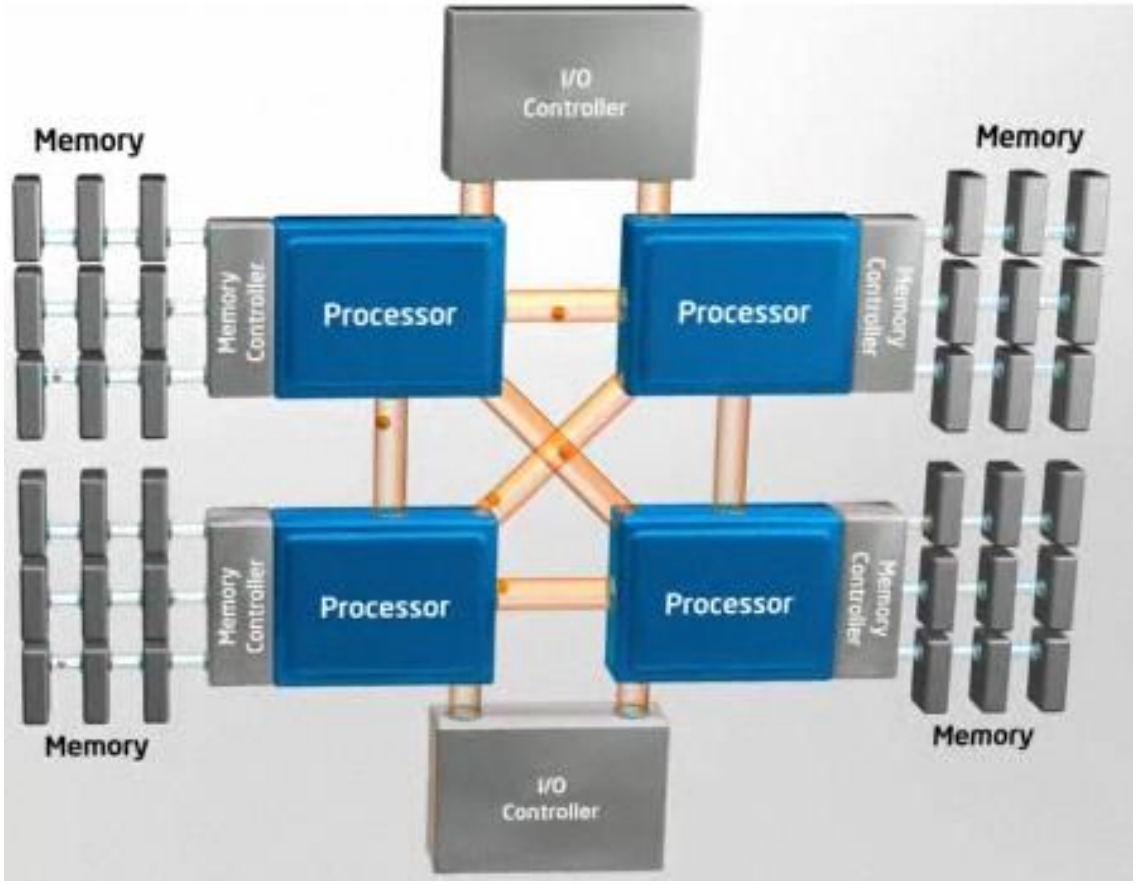
- Performance problems: synchronization
  - Communication between cores (via shared cache or memory interconnect) is slow
  - What is communication?
    - Access to shared data in memory
    - Mutex – e.g. to ensure mutually exclusive access to shared data structure in memory
    - synchronized keyword in Java

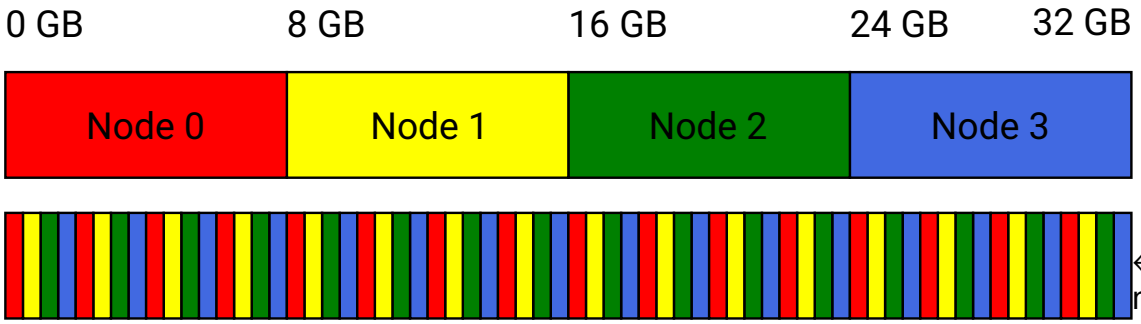
Hyper-threaded CPU



- “Cheaper variant”
- Duplicate just the registers, not the execution engines (ALU)
- Add HW scheduler to simulate parallel execution
- When one HW thread waits for memory, the other can execute
- From SW point of view, it looks like a multi-core CPU
- Imperfect instruction-level-parallelism (superscalar CPU) is improved by task-parallelism

Non-Uniform Memory Access (NUMA)





←Two possible mappings of memory addresses to memory location

- Multi-socket system
- Each socket has locally connected memory
- Other sockets access the memory via inter-socket interconnects (slower, ca 15%)
- Software sees all memory
- SW (OS) should allocate memory local to where it runs, apps could help

Out-of-order execution

Instruction stream

```
r = a + b
s = c + d
t = e + f
u = g + h
v = u + i
```

a and **c** are not cached, the rest is:

Superscalar, out-of-order execution

```
t = e + f; u = g + h
r = a + b; s = c + d; v = u + i
```

From a single CPU point of view, everything is correct

- Complicates synchronization
- Other CPUs can see results of computations in different order

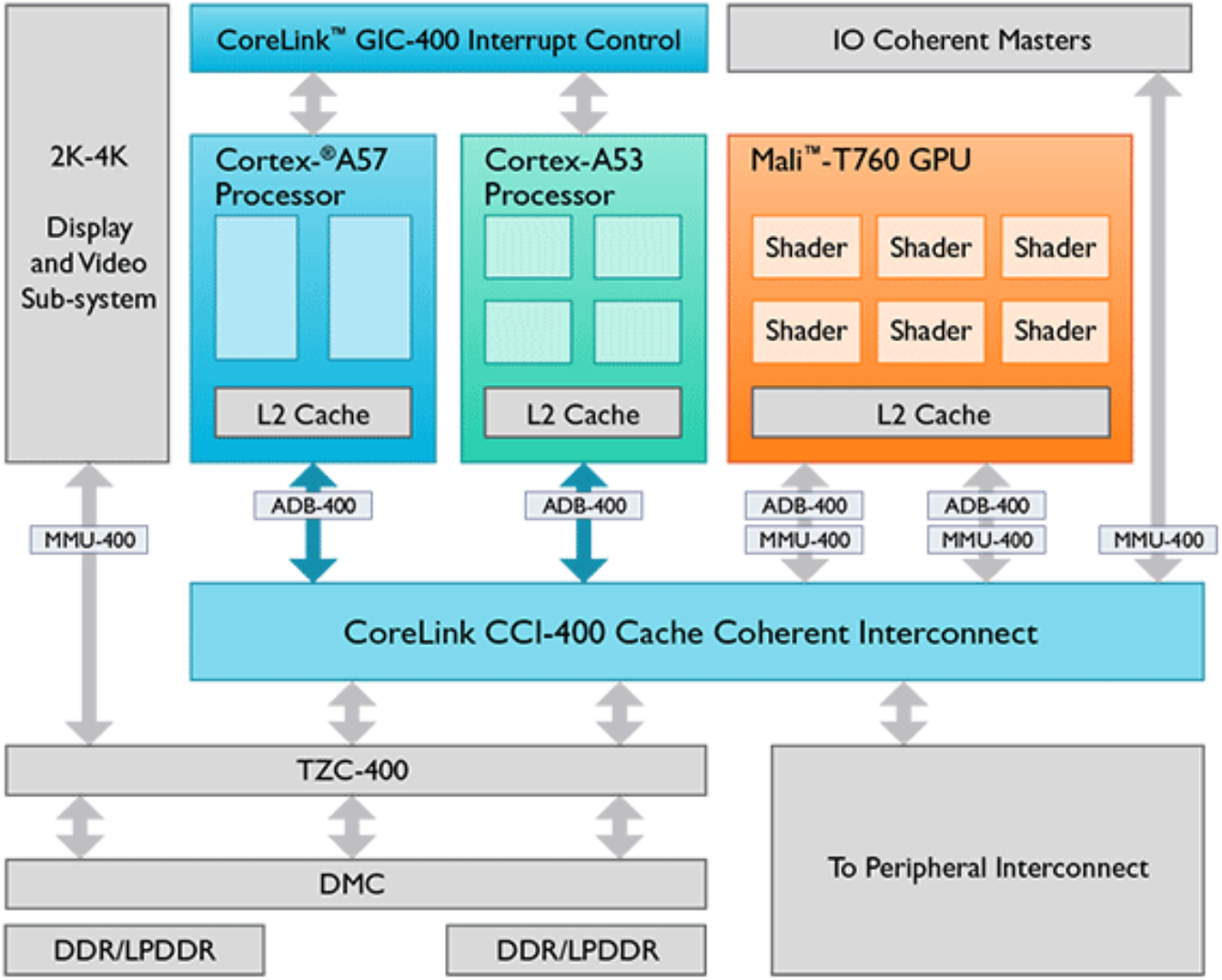
When order matters?

```
lock = 1
r = a + b
s = a - b
lock = 0
```

The above example will likely not work, because accesses to “lock” may be reordered.

Embedded heterogeneous systems

Different CPUs/GPUs on a single chip



Source: ARM

Outline

1 About the course

2 Basics

3 Hardware

4 Making the hardware faster
- Caches
- Instruction-level parallelism
- Task parallelism

5 Energy

6 Exercise today
- C/C++ compiler
- Profiling

Energy is the new spped

- Today, we no longer want just fast software
- We also care about heating and battery life of our mobile phones
- Good news: Fast software is also energy efficient

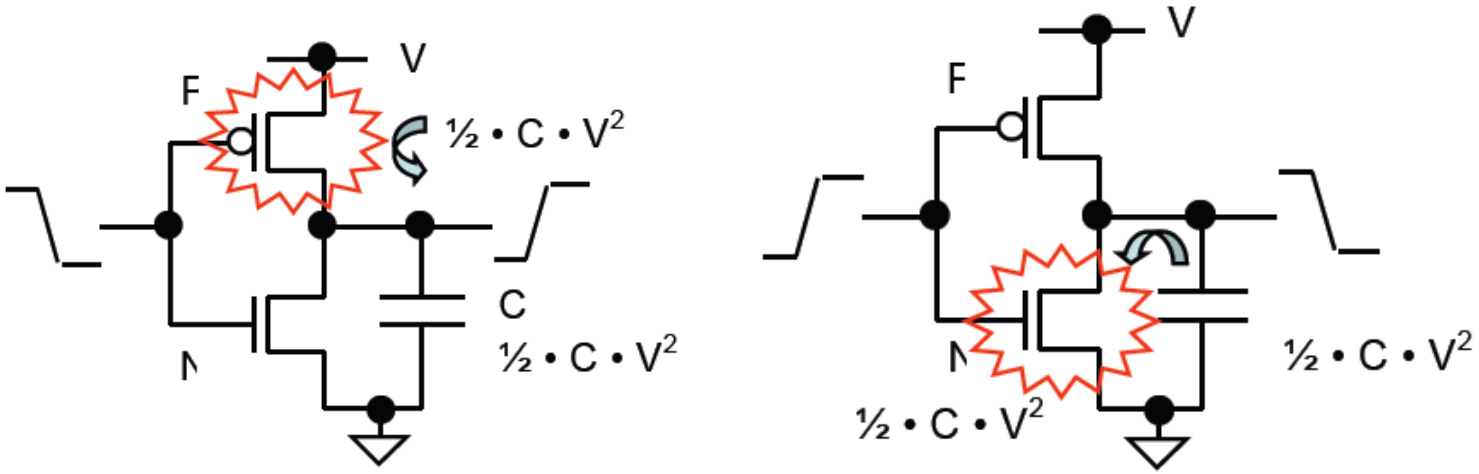
Power consumption of CMOS circuits

Two components:

- Static dissipation
  - leakage current through P-N junctions etc.
  - higher voltage $\rightarrow$ higher static dissipation
- Dynamic dissipation
  - charging and discharging of load capacitance (useful + parasitic)
  - short-circuit current

$$P_{total} = P_{static} + P_{dyn}$$

Dynamic power consumption and gate delay



Charging the parasite capacities needs energy

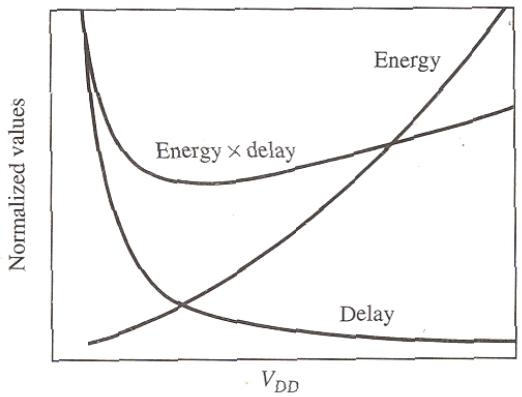
Power consumption

$$P_{dyn} = a \cdot C \cdot V_{dd}{}^2 \cdot f$$

Gate delay

$$t = \frac{\gamma \cdot C \cdot V_{dd}}{(V_{dd} - V_T)^2} \approx \frac{1}{V_{dd}}$$

Low power $\Rightarrow$ slow



Methods to reduce power/energy consumption

- use better technology/smaller gates (HW engineers)
- use better placing and routing on the chip (HW engineers)
- reduce power supply V_{DD} and/or frequency = Dynamic voltage and frequency scaling (OS job – apps can help)
  - raising it back takes time (rump-up latency)
  - deciding optimal sleep state to take requires knowing the future
  - recent Android versions have API for “predicting future”
- reduce activity (clock gating = switch off parts of the chip that are not used) [job for OS and HW, apps can help]
- **use better algorithms and/or data structures** (SW engineers)

Outline

1 About the course

2 Basics

3 Hardware

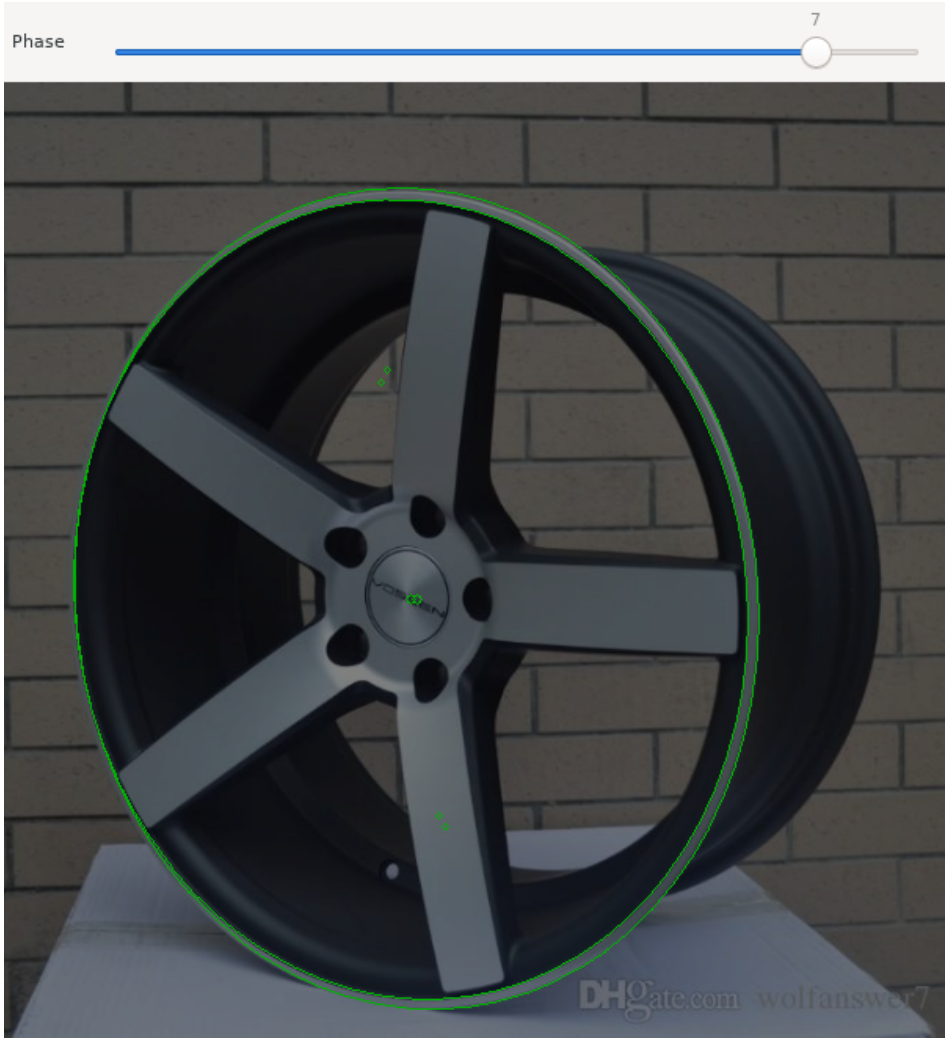
4 Making the hardware faster
- Caches
- Instruction-level parallelism
- Task parallelism

5 Energy

6 Exercise today
- C/C++ compiler
- Profiling

Exercises example

- Ellipse detection using RANSAC algorithm



Outline

1 About the course

2 Basics

3 Hardware

4 Making the hardware faster
- Caches
- Instruction-level parallelism
- Task parallelism

5 Energy

6 Exercise today
- C/C++ compiler
- Profiling

C/C++ compiler

- Generates native code from C/C++ source code
- Popular compilers: GCC, Clang (LLVM), icc, MSVC, ...
- Perform many “optimization passes”
  - Those will be covered in a separate lecture
- For now, very brief overview of what you might need today

Compiler flags (gcc, clang)

- Documentation is your friend:
  - Command `(p)info gcc`
  - `https://gcc.gnu.org/onlinedocs/`
  - Clang's flags are mostly compatible with gcc
- Generate debugging information: `-g`
- Optimization level: `-O0`, `-O1`, `-O2`, `-O3`, `-Os` (size)
  - `-O2` is considered "safe", `-O3` may be buggy
  - Individual optimization passes:
    `-free-ccp`, `-fast-math`, `-fomit-frame-pointer`, `-free-vectorize`, ...
  - Find out which optimizations passes are active for given optimization level:
    `g++ -Q -O2 --help=optimizers`
- Code generation
  - `-fpic`, `-fpack-struct`, `-fshort-enums`
  - Machine dependent:
    - Generate instructions for given micro-architecture: `-march=haswell`, `-march=skylake` (will not run on older hardware)
    - Use only "older" instructions, but schedule them for for given µarch:
      `-mtune=haswell`, `-mtune=native`,
    - `-m32`, `-minline-all-stringops`, ...

Outline

1 About the course

2 Basics

3 Hardware

4 Making the hardware faster
- Caches
- Instruction-level parallelism
- Task parallelism

5 Energy

6 Exercise today
- C/C++ compiler
- Profiling

Profiling

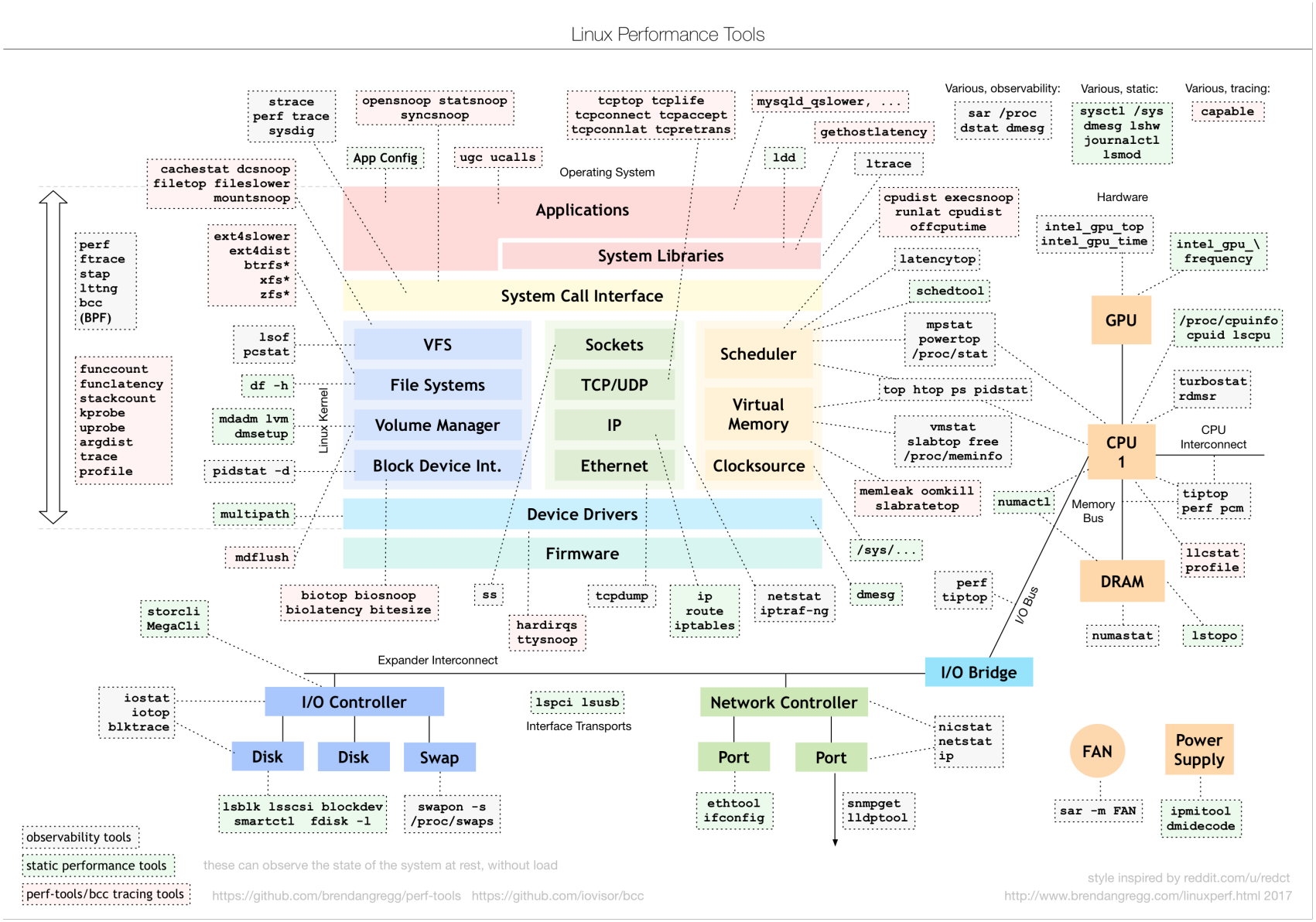
- Profiling: Identifies where your code is slow
- “Premature optimization is the root of all evil”
  — D. Knuth
- Software is complex!
- We want to optimize the bottlenecks, not all code
- Real world codebases are big: Reading all the code is a waste of time (for optimizing)

Bottlenecks

Sources:

- code
- memory
- network
- disk
- ...

Linux Performance Tools



Profiling tools

In order to do:	You can use:
Manual instrumentation	printf and similar
Static instrumentation	gprof
Dynamic instrumentation	callgrind, cachegrind
Performance counters	oprofile, perf
Heap profiling	massif, google-perftools

- **Instrumentation = modifying the code the perform measurements**

Static instrumentation: gprof

- **gcc -pg ... -o program**
  - Adds profiling code to every function/basic block
- **./program**
  - generates gmon.out
- **gprof program**

```
Flat profile:

Each sample counts as 0.01 seconds.
%    cumulative self          self   total
time seconds    seconds calls s/call s/call name
33.86 15.52     15.52    1    15.52  15.52  func2
33.82 31.02     15.50    1    15.50  15.50  new_func1
33.29 46.27     15.26    1    15.26  30.75  func1
0.07  46.30     0.03                        main
```

Event sampling

- **Basic idea**
  - when an interesting event occurs, look at where program executes
  - result is histogram of addresses and event counts
- **Events**
  - time, cache miss, branch-prediction miss, page fault
- **Implementation**
  - timer interrupt → upon entry, program address is stored on stack
  - each event has counting register
    - when threshold is reached, an interrupt is generated

Performance counters

- Hardware inside the CPU (Intel, ARM, ...)
- Software can configure which events to count and when/whether to generate interrupts
- In many cases can be accessed from application code
- Documentation:
  - Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide
  - Intel® 64 and IA-32 Architectures Optimization Reference Manual
  - ARM® Architecture Reference Manual ARMv8, for ARMv8-A architecture profile

perf

- **linux-tools package**
- **Can monitor both HW and SW events**
- **Can analyze:**
  - single application
  - whole system
  - ...
- **https://perf.wiki.kernel.org/**

perf usage

- perf list
- perf stat -e cycles -e branch-misses -e branches -e cache-misses -e cache-references ./vecadd

```
Performance counter stats for './vecadd':

    1,898,543,656      cycles                                                  (79.98%)
          267,572      branch-misses             #    0.08% of all branches    (79.97%)
      348,090,074      branches                                                (79.95%)
       20,232,628      cache-misses              #   75.588 % of all cache refs (80.51%)
       26,767,103      cache-references                                        (80.09%)

      0.619472916 seconds time elapsed
```

perf usage II.

- perf record -e cycles -e branch-misses ./vecadd
- perf report

Useful resources

- Denis Bakhvalov's blog: `https://dendibakh.github.io/notes/`
- `https://dendibakh.github.io/blog/2019/02/16/Performance-optimization-contest-1`

Effective Software

Lecture 2: Virtual machine, JVM bytecode, (de-)compilers, disassembler, profiling

David Šišlák
david.sislak@fel.cvut.cz

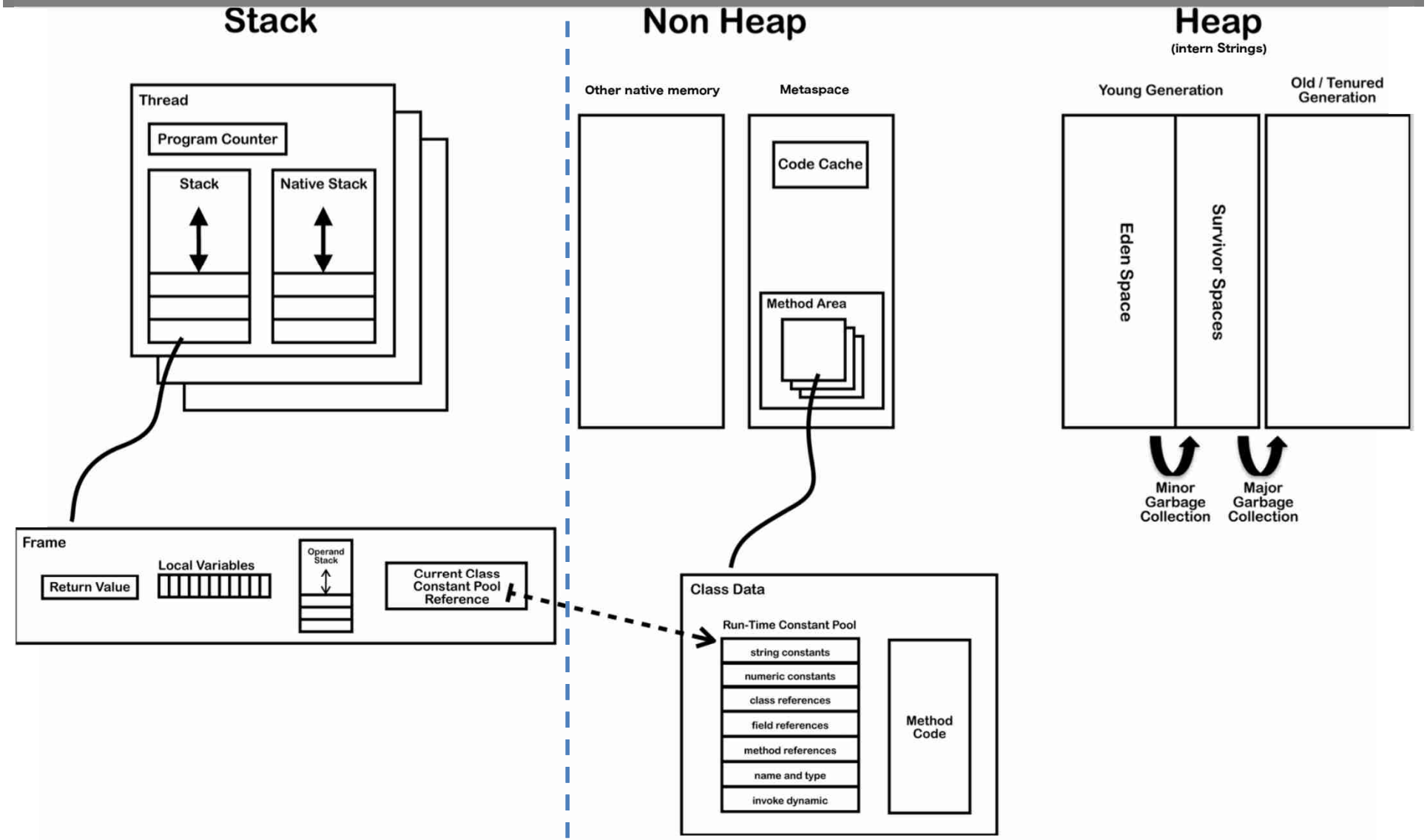
[1] Oaks, S.: Java Performance: The Definitive Guide. O'Reilly, USA 2014.
[2] Fog, A.: The microarchitecture of Intel, AMD and VIA CPU, 2016.
[3] JVM source code - http://openjdk.java.net

Introduction – Virtual Machine

» **Virtual machine** model (.NET, JVM – Scala, Jython, JRuby, Clojure, …)
- source code
- compiled into VM *bytecode*
- hybrid run-time environment (platform dependent VM implementation)
  - interpreted *bytecode*
  - *complied assembly-code* (native CPU code)
  - automated platform capability optimizations (e.g. use of SIMD)

» comparison of **bytecode** to **assembly-code**
- (+) platform independence (portable) – architecture (RISC/CISC, bits), OS
- (+) reflection – observe, modify own structure at run-time
- (+) small size
- (-) slower execution – interpreted mode, compilation latencies
- (-) less control on assembly code – less options for custom optimization

JAVA Virtual Machine – Memory Layout



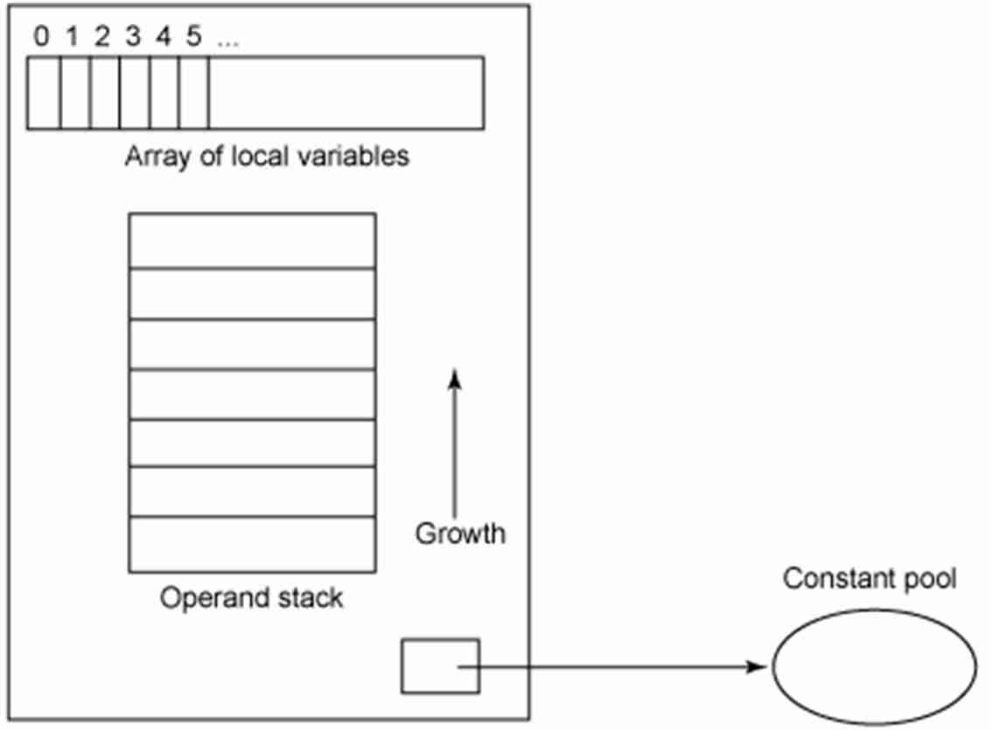
Thread specific

Shared by many threads

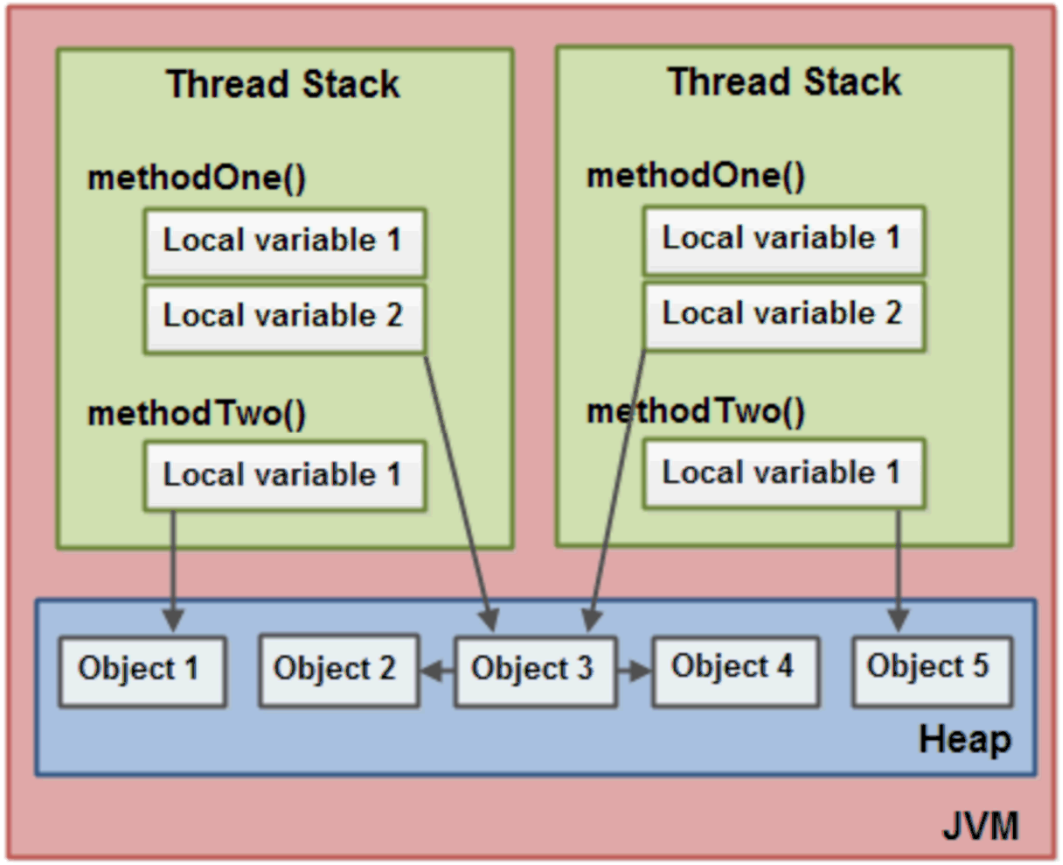
JAVA Virtual Machine - Frame

- **frame**
  - each thread has stack with frames (outside of heap, fixed length)

    StackOverflowError vs. OutOfMemoryError
  - frame is **created** each time method is invoked (**destroyed** after return)
    - ***interpreted frame*** per exactly one method
    - ***complied frame*** includes all in-lined methods
  - frame **size** determined at compile-time (in class file for interpreted)
  - **variables** (any type)
    - {this} – *instance call only!*
    - {method arguments}
    - {local variables}
  - **operand stack** (any type)
    - LIFO
  - **reference to run-time** constant pool (class def)
  - method + class is associated



JAVA Virtual Machine – Memory Layout



» References in JVM are called Ordinary Object Pointers (OOP)

- **compressed** – 32 bit – able to address 32GB heap (using object alignment)
- **regular** – 64 bit

JAVA Virtual Machine – Stack-oriented Machine

» **stack-oriented** - stack machine model for passing parameters and output for instructions

$$(2 + 3) \times 11 + 1$$

timeline →

Input	2	3	add	11	mul	1	add
Stack	2	3 2	5	11 5	55	1 55	56

» **JVM bytecode** – sequence of ***instructions*** composed of

- *opcode* – operation code, what should be done
- opcode specific *parameters* – some has no params, some multiple

JAVA Virtual Machine – Opcodes

- **JVM opcode** (1 Byte only always):
  - load and store (aload_0, istore, aconst_null, …)
  - arithmetic and logic (ladd, fcmpl, …)
  - type conversion (i2b, d2i, …)
  - object manipulation (new, putfield, getfield, …)
  - stack management (swap, dup2, …)
  - control transfer (ifeq, goto, …)
  - method invocation (invokespecial, areturn, …) – frame manipulation
  - exceptions and monitor concurrency (athrow, monitorenter, …)
- prefix/suffix – i, l, s, b, c, f, d and a (reference)
- variables as registers – e.g. istore_1 (variable 0 is **this** for instance method)

```
mov     %rax,%r8
shl     $0x5,%eax
sub     %r8d,%eax
add     %ecx,%eax
inc     %edx
```

CPU assembly-code
AT&T syntax

VS.

```
iconst_0
istore_3
iload_3
bipush        100
```

JVM bytecode

JAVA Virtual Machine – Object Oriented Language

» **Class file** – product of source code compilation

- one per each class
- method bytecode is included

```
ClassFile {
    u4              magic;
    u2              minor_version;
    u2              major_version;
    u2              constant_pool_count;
    cp_info         contant_pool[constant_pool_count - 1];
    u2              access_flags;
    u2              this_class;
    u2              super_class;
    u2              interfaces_count;
    u2              interfaces[interfaces_count];
    u2              fields_count;
    field_info      fields[fields_count];
    u2              methods_count;
    method_info     methods[methods_count];
    u2              attributes_count;
    attribute_info  attributes[attributes_count];
}
```

JAVA Virtual Machine – Example 1 – Source Code

```
public class Employee<Type> {
    private Type data;
    public int id;

    public Employee(Type data, int id) {
        update(data,id);
    }

    private void update(Type data, int id) {
        this.data = data;
        this.id = id;
    }

    public Type employeeData() {
        return data;
    }
}
```

notice usage of Generics

JAVA Virtual Machine – Example 1 – Class File Content

```
00000000  ca fe ba be 00 00 00 34  00 20 0a 00 06 00 19 0a  |.......4. ......|
00000010  00 05 00 1a 09 00 05 00  1b 09 00 05 00 1c 07 00  |................|
00000020  1d 07 00 1e 01 00 04 64  61 74 61 01 00 12 4c 6a  |.......data...Lj|
00000030  61 76 61 2f 6c 61 6e 67  2f 4f 62 6a 65 63 74 3b  |ava/lang/Object;|
00000040  01 00 09 53 69 67 6e 61  74 75 72 65 01 00 06 54  |...Signature...T|
00000050  54 79 70 65 3b 01 00 02  69 64 01 00 01 49 01 00  |Type;...id...I..|
00000060  06 3c 69 6e 69 74 3e 01  00 16 28 4c 6a 61 76 61  |.<init>...(Ljava|
00000070  2f 6c 61 6e 67 2f 4f 62  6a 65 63 74 3b 49 29 56  |/lang/Object;I)V|
00000080  01 00 04 43 6f 64 65 01  00 0f 4c 69 6e 65 4e 75  |...Code...LineNu|
00000090  6d 62 65 72 54 61 62 6c  65 01 00 0a 28 54 54 79  |mberTable...(TTy|
000000a0  70 65 3b 49 29 56 01 00  06 75 70 64 61 74 65 01  |pe;I)V...update.|
000000b0  00 0c 65 6d 70 6c 6f 79  65 65 44 61 74 61 01 00  |..employeeData..|
000000c0  14 28 29 4c 6a 61 76 61  2f 6c 61 6e 67 2f 4f 62  |.()Ljava/lang/Ob|
000000d0  6a 65 63 74 3b 01 00 08  28 29 54 54 79 70 65 3b  |ject;...()TType;|
000000e0  01 00 2b 3c 54 79 70 65  3a 4c 6a 61 76 61 2f 6c  |..+<Type:Ljava/l|
000000f0  61 6e 67 2f 4f 62 6a 65  63 74 3b 3e 4c 6a 61 76  |ang/Object;>Ljav|
00000100  61 2f 6c 61 6e 67 2f 4f  62 6a 65 63 74 3b 01 00  |a/lang/Object;..|
00000110  0a 53 6f 75 72 63 65 46  69 6c 65 01 00 0d 45 6d  |.SourceFile...Em|
00000120  70 6c 6f 79 65 65 2e 6a  61 76 61 0c 00 0d 00 1f  |ployee.java.....|
00000130  0c 00 12 00 0e 0c 00 07  00 08 0c 00 0b 00 0c 01  |................|
00000140  00 11 65 6d 70 6c 6f 79  65 65 2f 45 6d 70 6c 6f  |..employee/Emplo|
00000150  79 65 65 01 00 10 6a 61  76 61 2f 6c 61 6e 67 2f  |yee...java/lang/|
00000160  4f 62 6a 65 63 74 01 00  03 28 29 56 00 21 00 05  |Object...()V.!..|
00000170  00 06 00 00 00 02 00 02  00 07 00 08 00 01 00 09  |................|
00000180  00 00 00 02 00 0a 00 01  00 0b 00 0c 00 00 00 03  |................|
00000190  00 01 00 0d 00 0e 00 02  00 0f 00 00 00 2b 00 03  |.............+..|
000001a0  00 03 00 00 00 0b 2a b7  00 01 2a 2b 1c b7 00 02  |......*...*+....|
000001b0  b1 00 00 00 01 00 10 00  00 00 0e 00 03 00 00 00  |................|
000001c0  07 00 04 00 08 00 0a 00  09 00 09 00 00 00 02 00  |................|
000001d0  11 00 02 00 12 00 0e 00  02 00 0f 00 00 00 2b 00  |.............+.|
000001e0  02 00 03 00 00 00 0b 2a  2b b5 00 03 2a 1c b5 00  |.......*+...*...|
000001f0  04 b1 00 00 00 01 00 10  00 00 00 0e 00 03 00 00  |................|
00000200  00 0c 00 05 00 0d 00 0a  00 0e 00 09 00 00 00 02  |................|
00000210  00 11 00 01 00 13 00 14  00 02 00 0f 00 00 00 1d  |................|
00000220  00 01 00 01 00 00 00 05  2a b4 00 03 b0 00 00 00  |........*.......|
00000230  01 00 10 00 00 00 06 00  01 00 00 00 11 00 09 00  |................|
00000240  00 00 02 00 15 00 02 00  09 00 00 00 02 00 16 00  |................|
00000250  17 00 00 00 02 00 18                              |.......|
```

class constant pool

JAVA Virtual Machine – Example 1 – Disassembled Constants

» **javap** – JAVA disassembler included in JDK (readable form of class file)

```
public class employee.Employee<Type extends java.lang.Object> extends java.lang.Object
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #6.#25         // java/lang/Object."<init>":()V
   #2 = Methodref          #5.#26         // employee/Employee.update:(Ljava/lang/Object;I)V
   #3 = Fieldref           #5.#27         // employee/Employee.data:Ljava/lang/Object;
   #4 = Fieldref           #5.#28         // employee/Employee.id:I
   #5 = Class              #29            // employee/Employee
   #6 = Class              #30            // java/lang/Object
   #7 = Utf8               data
   #8 = Utf8               Ljava/lang/Object;
   #9 = Utf8               Signature
  #10 = Utf8               TType;
  #11 = Utf8               id
  #12 = Utf8               I
  #13 = Utf8               <init>
  #14 = Utf8               (Ljava/lang/Object;I)V
  #15 = Utf8               Code
  #16 = Utf8               LineNumberTable
  #17 = Utf8               (TType;I)V
  #18 = Utf8               update
  #19 = Utf8               employeeData
  #20 = Utf8               ()Ljava/lang/Object;
  #21 = Utf8               ()TType;
  #22 = Utf8               <Type:Ljava/lang/Object;>Ljava/lang/Object;
  #23 = Utf8               SourceFile
  #24 = Utf8               Employee.java
  #25 = NameAndType        #13:#31        // "<init>":()V
  #26 = NameAndType        #18:#14        // update:(Ljava/lang/Object;I)V
  #27 = NameAndType        #7:#8          // data:Ljava/lang/Object;
  #28 = NameAndType        #11:#12        // id:I
  #29 = Utf8               employee/Employee
  #30 = Utf8               java/lang/Object
  #31 = Utf8               ()V
{
}
Signature: #22                          // <Type:Ljava/lang/Object;>Ljava/lang/Object;
```

Generics used only for compilation

JAVA Virtual Machine – Example 1 – Disassembled Fields

```
{
  private Type data;
    descriptor: Ljava/lang/Object;
    flags: ACC_PRIVATE
    Signature: #10                          // TType;

  public int id;
    descriptor: I
    flags: ACC_PUBLIC
```

» **descriptor** is used by VM – no generics included

» **signature** is used for compilation – contain Generics

JAVA Virtual Machine – Example 1 – Disassembled Method

```
public Type employeeData();
  descriptor: ()Ljava/lang/Object;
  flags: ACC_PUBLIC
  Code:
    stack=1, locals=1, args_size=1
       0: aload_0
       1: getfield      #3                  // Field data:Ljava/lang/Object;
       4: areturn
    LineNumberTable:
      line 17: 0
  Signature: #21                           // ()TType;
```

```
16      public Type employeeData() {
17          return data;
18      }
```

opcode offset in bytecode
for the method employeeData

- » getfield
  - takes 1 ref from stack
  - build an index into runtime pool of class instance by reference **this**
- » areturn
  - takes 1 ref from stack
  - push onto the stack of calling method

0	1	2	3	4
aload_0	getfield	00	03	areturn

0	1	2	3	4
2A	B4	00	03	B0

JAVA Virtual Machine – Example 1 – Disassembled Constructor

```
public employee.Employee(Type, int);
  descriptor: (Ljava/lang/Object;I)V
  flags: ACC_PUBLIC
  Code:
    stack=3, locals=3, args_size=3
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: aload_0
       5: aload_1
       6: iload_2
       7: invokespecial #2                  // Method update:(Ljava/lang/Object;I)V
      10: return
    LineNumberTable:
      line 7: 0
      line 8: 4
      line 9: 10
  Signature: #17                          // (TType;I)V

private void update(Type, int);
  descriptor: (Ljava/lang/Object;I)V
  flags: ACC_PRIVATE
  Code:
    stack=2, locals=3, args_size=3
       0: aload_0
       1: aload_1
       2: putfield      #3                  // Field data:Ljava/lang/Object;
       5: aload_0
       6: iload_2
       7: putfield      #4                  // Field id:I
      10: return
    LineNumberTable:
      line 12: 0
      line 13: 5
      line 14: 10
  Signature: #17                          // (TType;I)V
```

```
7    public Employee(Type data, int id) {
8        update(data,id);
9    }
10
11   private void update(Type data, int id) {
12       this.data = data;
13       this.id = id;
14   }
```

JAVA Virtual Machine – Example 1 – Decompiler

» **procyon** – open-source JAVA decompiler

```java
//
// Decompiled by Procyon v0.5.30
//

package employee;

public class Employee<Type>
{
    private Type data;
    public int id;

    public Employee(final Type type, final int n) {
        this.update(type, n);
    }

    private void update(final Type data, final int id) {
        this.data = data;
        this.id = id;
    }

    public Type employeeData() {
        return this.data;
    }
}
```

De-compiled source code

```java
public class Employee<Type> {
    private Type data;
    public int id;

    public Employee(Type data, int id) {
        update(data,id);
    }

    private void update(Type data, int id) {
        this.data = data;
        this.id = id;
    }

    public Type employeeData() {
        return data;
    }
}
```

Original source code

JAVA Virtual Machine – Example 2 – Source Code

```java
private static Integer daysInMonth(int month, int year)
{
    int retVal;
    switch (month)
    {
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
            retVal=31;
            break;
        case 2:
            retVal = (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0)) ? 29 : 28;
            break;
        case 4:
        case 6:
        case 9:
        case 11:
            retVal = 30;
            break;
        default:
            throw new IllegalArgumentException("Unknown month: " + month);
    }
    return new Integer(retVal);
}

private static int compute() {
    int month = 4;
    int year = 2000;
    int o=0;
    for (int i=0; i<1_000_000; i++) {
        o+=daysInMonth(month, year);
    }
    return o;
}
```

JAVA Virtual Machine – Example 2 – daysInMonth Bytecode

```
private static java.lang.Integer daysInMonth(int, int);
  descriptor: (II)Ljava/lang/Integer;
  flags: ACC_PRIVATE, ACC_STATIC
  Code:
    stack=4, locals=3, args_size=2
       0: iload_0
       1: tableswitch   { // 1 to 12
                     1: 64
                     2: 70
                     3: 64
                     4: 102
                     5: 64
                     6: 102
                     7: 64
                     8: 64
                     9: 102
                    10: 64
                    11: 102
                    12: 64
                default: 108
          }
      64: bipush        31
      66: istore_2
      67: goto          135
      70: iload_1
      71: iconst_4
      72: irem
      73: ifne          96
      76: iload_1
      77: bipush        100
      79: irem
      80: ifne          91
      83: iload_1
      84: sipush        400
      87: irem
      88: ifne          96
      91: bipush        29
      93: goto          98
      96: bipush        28
      98: istore_2
      99: goto          135
     102: bipush        30
     104: istore_2
     105: goto          135
```

```java
int retVal;
switch (month)
{
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        retVal=31;
        break;
    case 2:
        retVal = (year % 4 == 0 &&
                    (year % 100 != 0 ||
                     year % 400 == 0)) ?
                 29 : 28;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        retVal = 30;
        break;
    default:
        throw new IllegalArgumentException(
                "Unknown month: " + month);
}
return new Integer(retVal);
```

JAVA Virtual Machine – Example 2 – daysInMonth Bytecode

```
 108: new           #2                  // class java/lang/IllegalArgumentException
 111: dup
 112: new           #3                  // class java/lang/StringBuilder
 115: dup
 116: invokespecial #4                  // Method java/lang/StringBuilder."<init>":()V
 119: ldc           #5                  // String Unknown month:
 121: invokevirtual #6                  // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
 124: iload_0
 125: invokevirtual #7                  // Method java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
 128: invokevirtual #8                  // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
 131: invokespecial #9                  // Method java/lang/IllegalArgumentException."<init>":(Ljava/lang/String;)V
 134: athrow
 135: new           #10                 // class java/lang/Integer
 138: dup
 139: iload_2
 140: invokespecial #11                 // Method java/lang/Integer."<init>":(I)V
 143: areturn
LineNumberTable:
  line 7: 0
  line 16: 64
  line 17: 67
  line 19: 70
  line 20: 99
  line 25: 102
  line 26: 105
  line 28: 108
  line 30: 135
StackMapTable: number_of_entries = 8
  frame_type = 251 /* same_frame_extended */
    offset_delta = 64
  frame_type = 5 /* same */
  frame_type = 20 /* same */
  frame_type = 4 /* same */
  frame_type = 65 /* same_locals_1_stack_item */
    stack = [ int ]
  frame_type = 3 /* same */
  frame_type = 5 /* same */
  frame_type = 252 /* append */
    offset_delta = 26
    locals = [ int ]
```

```
        default:
            throw new IllegalArgumentException(
                    "Unknown month: " + month);
    }
    return new Integer(retVal);
```

StackMapTable – define variable types in Frame (variables and stack) at every bytecode jump target position

- first stack map automatically determined from method descriptor
- others are differential updates

JAVA Virtual Machine – Example 2 – compute Bytecode

```
private static int compute();
  descriptor: ()I
  flags: ACC_PRIVATE, ACC_STATIC
  Code:
    stack=3, locals=4, args_size=0
       0: iconst_4
       1: istore_0
       2: sipush        2000
       5: istore_1
       6: iconst_0
       7: istore_2
       8: iconst_0
       9: istore_3
      10: iload_3
      11: ldc           #12                 // int 1000000
      13: if_icmpge     33
      16: iload_2
      17: iload_0
      18: iload_1
      19: invokestatic  #13                 // Method daysInMonth:(II)Ljava/lang/Integer;
      22: invokevirtual #14                 // Method java/lang/Integer.intValue:()I
      25: iadd
      26: istore_2
      27: iinc          3, 1
      30: goto          10
      33: iload_2
      34: ireturn
    LineNumberTable:
      line 34: 0
      line 35: 2
      line 36: 6
      line 37: 8
      line 38: 16
      line 37: 27
      line 40: 33
    StackMapTable: number_of_entries = 2
      frame_type = 255 /* full_frame */
        offset_delta = 10
        locals = [ int, int, int, int ]
        stack = []
      frame_type = 250 /* chop */
        offset_delta = 22
```

for cycle

```
private static int compute() {
    int month = 4;
    int year = 2000;
    int o=0;
    for (int i=0; i<1_000_000; i++) {
        o+=daysInMonth(month, year);
    }
    return o;
}
```

No optimization during source code compilation !

Interpreted code execution is as inefficient as your source code !!!

bytecode offset 10 is related to for cycle bytecode start where there are 4 ints as local variables and no stack

JAVA Virtual Machine – Source Code Compilation

» **source code compilation** (*source code => bytecode*)
  » *bytecode* is not better than your *source code*
    » invariants in loop are not removed
  » no optimizations like
    » loop unrolling
    » algebraic simplification
    » strength reduction

obfuscation = make code difficult to be understood by humans but with the same functionality

» optionally *bytecode* could be modified before execution by JVM
  - e.g. **ProGuard** – *obfuscator* including bytecode optimizations
    – shrinker – **compact code**, remove dead code
    – optimizer
      - modify access pattern (private, static, final)
      - **inline** bytecode
    – obfuscator – renaming, layout changes
    – preverifier – ensure class loading

Test yourself
- compute method is simplified
- faster interpretation
- better JIT output

JAVA Virtual Machine – Bytecode Compilation in run-time

- **Just-in-time** (JIT)
  - converts bytecode into assembly code in run-time
  - check OpenJDK sources for very detailed information

    http://openjdk.java.net
- JIT includes **adaptive optimization** (adaptive tiered compilation since version 7)
  - balance trade-off between JIT and interpreting instructions
  - monitors frequently executed parts "hot spots" **including data** on caller-callee relationship for virtual method invocation
  - triggers dynamic re-compilation based on current execution profile
  - inline expansion to remove context switching
  - optimize branches
  - can make risky assumption (e.g. skip code) ->
    - unwind to valid state
    - deoptimize previously JITed code even if code is already executed

- Ahead-of-Time Compilation (AOT) – remove warm-up phase
  - compile into assembly code prior to launching the virtual machine

JAVA Virtual Machine – JIT Compilation

- *Just-in-time (JIT) compilers* – asynchronous (3 C1, 7 C2 threads for 32 cores)
  - **C1 compiler** – much faster than C2
    - simplified inlining, using CPU registers
    - window-based optimization over small set of instructions
    - intrinsic functions with vector operations **SIMD** (Math, arraycopy, …)
  - **C2 compiler** – high-end fully optimizing compiler
    - dead code elimination, loop unrolling, loop invariant hoisting, common sub-expression elimination, constant propagation
    - full inlining, full deoptimization (back to level 0)
    - escape analysis, null check elimination,
    - pattern-based loop vectorization and super word packing (**SIMD**)
- **JIT compilation tiers** – adaptive compilation levels in JVM

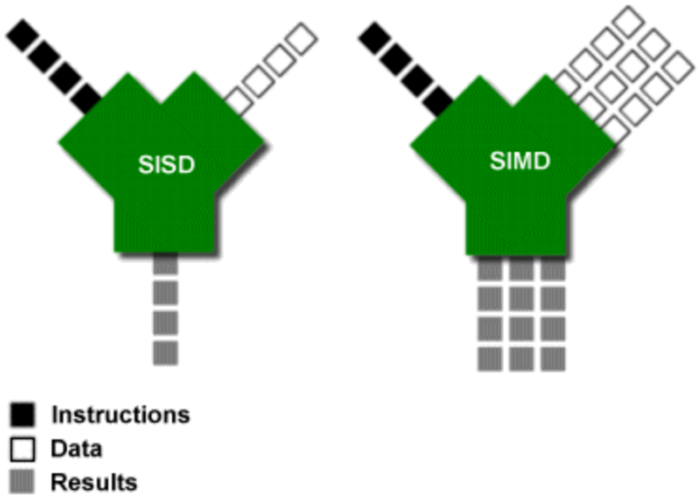
```
CompLevel_none              = 0,        // Interpreter
CompLevel_simple            = 1,        // C1
CompLevel_limited_profile   = 2,        // C1, invocation & backedge counters
CompLevel_full_profile      = 3,        // C1, invocation & backedge counters + mdo
CompLevel_full_optimization = 4,        // C2
```

- **on-stack replacement** (OSR) – optimization during execution of a method
  - start at bytecode jump targets (goto, if_)

Assembly Code

» reasons to study **assembly code** (both Java and C/C++)

- educational reasons
  - predict efficient coding techniques
- debugging and verification
  - how well the code looks like
- optimize code



1. **for speed**
   - avoid poorly compiled patterns
   - data fits into cache
   - predictable branches or no branches
   - use <u>vector programing</u> if possible (**SIMD**)
     - » 256bit registers with AVX2 since Intel Sandy Bridge
     - » 512bit AVX-512 since Intel Knight Landing (Xeon Phi)
2. **for size**
   - primarily code cache efficiency

JAVA Virtual Machine – Example 2 – Tiered Compilation

» -XX:+PrintCompilation (-XX:+PrintInlining)

{millis from start} {compilation_task_id} {flags} {tier} {class:method} (bytecode size)@OSR {removing not rentrant/zombie}

```
67    1       3    java.lang.String::hashCode (55 bytes)
68    2       3    java.lang.String::charAt (29 bytes)
69    3       3    java.lang.String::length (6 bytes)
74    4       3    java.lang.String::indexOf (70 bytes)
74    5     n 0    java.lang.System::arraycopy (native)   (static)
74    6       3    java.lang.String::equals (81 bytes)
75    8       3    java.lang.Object::<init> (1 bytes)
75    9       3    java.lang.Math::min (11 bytes)
75    7       3    java.lang.AbstractStringBuilder::ensureCapacityInternal (16 bytes)
75   10       3    java.lang.AbstractStringBuilder::append (50 bytes)
76   11       3    java.lang.String::getChars (62 bytes)
81   12       1    java.lang.ref.Reference::get (5 bytes)
81   13       3    java.lang.StringBuilder::append (8 bytes)
82   14       3    java.lang.String::indexOf (7 bytes)
83   16       3    java.lang.Number::<init> (5 bytes)
83   19       1    java.lang.Object::<init> (1 bytes)
84    8       3    java.lang.Object::<init> (1 bytes)   made not entrant
84   18       3    SwitchTest::daysInMonth (144 bytes)
84   17       3    java.lang.Integer::<init> (10 bytes)
84   15       1    java.lang.Integer::intValue (5 bytes)
84   20       4    SwitchTest::daysInMonth (144 bytes)
86   18       3    SwitchTest::daysInMonth (144 bytes)   made not entrant
88   21 %     3    SwitchTest::compute @ 10 (35 bytes)
88   22       3    SwitchTest::compute (35 bytes)
89   23 %     4    SwitchTest::compute @ 10 (35 bytes)
91   21 %     3    SwitchTest::compute @ -2 (35 bytes)   made not entrant
91   23 %     4    SwitchTest::compute @ -2 (35 bytes)   made not entrant
92   24 %     4    SwitchTest::compute @ 10 (35 bytes)
94   25       4    SwitchTest::compute (35 bytes)
95   22       3    SwitchTest::compute (35 bytes)   made not entrant
```

Notice **standard compilation path 0 -> 3 -> 4**

JVM – Example 2 – daysInMonth Assembly Code – Tier 3

» -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly

» **all examples are in JVM 8 64-bit, Intel Haswell CPU, AT&T syntax**

tier 3 - C1 with invocation & backedge counters + MethodDataOop counter

because: count="256" iicount="256" **hot_count="256"**

stack initialization, **invocation counter** in MDO (0xDC) + **trigger** C2 (tier 4)

```
    127   17    b  3       SwitchTest::daysInMonth (144 bytes)
Decoding compiled method 0x0000000108a95190:
Code:
[Entry Point]
[Verified Entry Point]
[Constants]
  # {method} {0x000000012169d568} 'daysInMonth' '(II)Ljava/lang/Integer;' in 'SwitchTest'
  # parm0:    rsi       = int
  # parm1:    rdx       = int
  #           [sp+0x90]  (sp of caller)
  0x0000000108a95380: mov    %eax,-0x14000(%rsp)
  0x0000000108a95387: push   %rbp
  0x0000000108a95388: sub    $0x80,%rsp
  0x0000000108a9538f: mov    %rdx,%rdi
  0x0000000108a95392: movabs $0x12169db40,%rax  ;   {metadata(method data for {method} {0x000000012169d568} 'daysInMonth' '(II)Ljava/lang/Integer;' in 'SwitchTest')}
  0x0000000108a9539c: mov    0xdc(%rax),%edx
  0x0000000108a953a2: add    $0x8,%edx
  0x0000000108a953a5: mov    %edx,0xdc(%rax)
  0x0000000108a953ab: movabs $0x12169d568,%rax  ;   {metadata({method} {0x000000012169d568} 'daysInMonth' '(II)Ljava/lang/Integer;' in 'SwitchTest')}
  0x0000000108a953b5: and    $0x1ff8,%edx
  0x0000000108a953bb: cmp    $0x0,%edx
  0x0000000108a953be: je     0x0000000108a95996  ;*iload_0
                                                 ; - SwitchTest::daysInMonth@0 (line 7)
```

RSP – current stack position

R15 – current threat meta information

RAX – return value

month, year

stacking banging technique, StackOverflowException

stack frame allocation, saving registers

0x1ff8 >> 3 = 1024 invocations **trigger tier 4 (C2)**

JVM – Example 2 – daysInMonth Assembly Code – Tier 3

ESI is month input

```
0x0000000108a953c4: cmp    $0x1,%esi
0x0000000108a953c7: je     0x0000000108a95597
0x0000000108a953cd: cmp    $0x2,%esi
0x0000000108a953d0: je     0x0000000108a95435
0x0000000108a953d6: cmp    $0x3,%esi
0x0000000108a953d9: je     0x0000000108a95597
0x0000000108a953df: cmp    $0x4,%esi
0x0000000108a953e2: je     0x0000000108a9557d
0x0000000108a953e8: cmp    $0x5,%esi
0x0000000108a953eb: je     0x0000000108a95597
0x0000000108a953f1: cmp    $0x6,%esi
0x0000000108a953f4: je     0x0000000108a9557d
0x0000000108a953fa: cmp    $0x7,%esi
0x0000000108a953fd: je     0x0000000108a95597
0x0000000108a95403: cmp    $0x8,%esi
0x0000000108a95406: je     0x0000000108a95597
0x0000000108a9540c: cmp    $0x9,%esi
0x0000000108a9540f: je     0x0000000108a9557d
0x0000000108a95415: cmp    $0xa,%esi
0x0000000108a95418: je     0x0000000108a95597
0x0000000108a9541e: cmp    $0xb,%esi
0x0000000108a95421: je     0x0000000108a9557d
0x0000000108a95427: cmp    $0xc,%esi
0x0000000108a9542a: je     0x0000000108a95597
0x0000000108a95430: jmpq   0x0000000108a956d0  ;*tableswitch
                                               ; - SwitchTest::daysInMonth@1 (line 7)
```

default jump

JVM – Example 2 – daysInMonth Assembly Code – Tier 3

target for month=4, **backedge counter** tracking in MDO (0x290):

```
0x0000000108a9557d: movabs $0x12169db40,%rdx  ;   {metadata(method data for {method} {0x000000012169d568} 'daysInMonth' '(II)Ljava/lang/Integer;' in 'SwitchTest')}
0x0000000108a95587: incl   0x290(%rdx)
0x0000000108a9558d: mov    $0x1e,%ebx
0x0000000108a95592: jmpq   0x0000000108a955ac  ;*goto
                                               ; - SwitchTest::daysInMonth@105 (line 26)
```

EBX=30 is retVal

jump target, **inlined TLAB allocation** of Integer object:

```
0x0000000108a955ac: movabs $0x7c0011320,%rdx  ;   {metadata('java/lang/Integer')}
0x0000000108a955b6: mov    0x60(%r15),%rax
0x0000000108a955ba: lea    0x10(%rax),%rdi
0x0000000108a955be: cmp    0x70(%r15),%rdi
0x0000000108a955c2: ja     0x0000000108a959bc
0x0000000108a955c8: mov    %rdi,0x60(%r15)
0x0000000108a955cc: mov    0xa8(%rdx),%rcx
0x0000000108a955d3: mov    %rcx,(%rax)
0x0000000108a955d6: mov    %rdx,%rcx
0x0000000108a955d9: shr    $0x3,%rcx
0x0000000108a955dd: mov    %ecx,0x8(%rax)
0x0000000108a955e0: xor    %rcx,%rcx
0x0000000108a955e3: mov    %ecx,0xc(%rax)
0x0000000108a955e6: xor    %rcx,%rcx          ;*new  ; - SwitchTest::daysInMonth@135 (line 30)
```

0x10 **Integer** instance size

no space in **TLAB** -> new TLAB + external allocation with header init returns after the inlined allocation

object initialization, header filed with prototype mark

RAX Integer instance address

Heap object structure (64-bit JVM):

- header 12 or 16 Bytes
- object data super class first, type grouped

8B - mark word
4B / 8B – Klass ref.
… object data

Heap array object structure (64-bit JVM):

- header 16 or 20 Bytes
- sequence of array values

8B - mark word
4B / 8B – Klass ref.
4B – array length
sequence of values

JVM – Example 2 – daysInMonth Assembly Code – Tier 3

inlined Integer constructor with supers, invocation counts in MDOs (0xDC)

Integer::<init>, Number::<init>, Object::<init>

- currently in tier 3 (C1 counters in MDO)

```
0x0000000108a955e9: mov    %rax,%rdx
0x0000000108a955ec: movabs $0x12169db40,%rsi   ;  {metadata(method data for {method} {0x000000012169d568} 'daysInMonth' '(II)Ljava/lang/Integer;' in 'SwitchTest')}
0x0000000108a955f6: addq   $0x1,0x358(%rsi)
0x0000000108a955fe: movabs $0x1214df850,%rdx   ;  {metadata(method data for {method} {0x0000000121341738} '<init>' '(I)V' in 'java/lang/Integer')}
0x0000000108a95608: mov    0xdc(%rdx),%esi
0x0000000108a9560e: add    $0x8,%esi
0x0000000108a95611: mov    %esi,0xdc(%rdx)
0x0000000108a95617: movabs $0x121341738,%rdx   ;  {metadata({method} {0x0000000121341738} '<init>' '(I)V' in 'java/lang/Integer')}
0x0000000108a95621: and    $0x7ffff8,%esi
0x0000000108a95627: cmp    $0x0,%esi
0x0000000108a9562a: je     0x0000000108a959c9
0x0000000108a95630: mov    %rax,%rdx
0x0000000108a95633: movabs $0x1214df850,%rsi   ;  {metadata(method data for {method} {0x0000000121341738} '<init>' '(I)V' in 'java/lang/Integer')}
0x0000000108a9563d: addq   $0x1,0x108(%rsi)
0x0000000108a95645: movabs $0x1214df720,%rdx   ;  {metadata(method data for {method} {0x000000012133a9d8} '<init>' '()V' in 'java/lang/Number')}
0x0000000108a9564f: mov    0xdc(%rdx),%esi
0x0000000108a95655: add    $0x8,%esi
0x0000000108a95658: mov    %esi,0xdc(%rdx)
0x0000000108a9565e: movabs $0x12133a9d8,%rdx   ;  {metadata({method} {0x000000012133a9d8} '<init>' '()V' in 'java/lang/Number')}
0x0000000108a95668: and    $0x7ffff8,%esi
0x0000000108a9566e: cmp    $0x0,%esi
0x0000000108a95671: je     0x0000000108a959e0
0x0000000108a95677: mov    %rax,%rdx
0x0000000108a9567a: movabs $0x1214df720,%rsi   ;  {metadata(method data for {method} {0x000000012133a9d8} '<init>' '()V' in 'java/lang/Number')}
0x0000000108a95684: addq   $0x1,0x108(%rsi)
0x0000000108a9568c: movabs $0x12140ddf8,%rdx   ;  {metadata(method data for {method} {0x000000012129d480} '<init>' '()V' in 'java/lang/Object')}
0x0000000108a95696: mov    0xdc(%rdx),%esi
0x0000000108a9569c: add    $0x8,%esi
0x0000000108a9569f: mov    %esi,0xdc(%rdx)
0x0000000108a956a5: movabs $0x12129d480,%rdx   ;  {metadata({method} {0x000000012129d480} '<init>' '()V' in 'java/lang/Object')}
0x0000000108a956af: and    $0x7ffff8,%esi
0x0000000108a956b5: cmp    $0x0,%esi
0x0000000108a956b8: je     0x0000000108a959f7
0x0000000108a956be: mov    %ebx,0xc(%rax)      ;*putfield value
                                               ; - java.lang.Integer::<init>@6 (line 850)
                                               ; - SwitchTest::daysInMonth@140 (line 30)
```

invocation cnt of Integer::<init> in daysInMonth for inline

invocation cnt in Integer::<init> + trigger its C2 (tier 4)

invocation cnt of Number::<init> in Int::<init> for inline

invocation cnt in Number::<init> + trigger its C2 (tier 4)

invocation cnt of Object::<init> in Numb::<init> for inline

invocation cnt in Object::<init> + trigger its C2 (tier 4)

RAX.value = EBX (retVal)

JVM – Example 2 – daysInMonth Assembly Code – Tier 3

final cleanup and return, RAX contains return value (pointer to Integer instance)

```
0x0000000108a956c1: add    $0x80,%rsp
0x0000000108a956c8: pop    %rbp
0x0000000108a956c9: test   %eax,-0x214c5cf(%rip)        # 0x0000000106949100
                                                ;   {poll_return}
0x0000000108a956cf: retq                        ;*areturn
                                                ; - SwitchTest::daysInMonth@143 (line 30)
```

stack dealocation, reload register

safepoint poll check

» **Ordinary Object Pointer** (Oop) – flexible reference to an object

» **safepoint** – Oops in perfectly described state by OopMap (GCmaps)

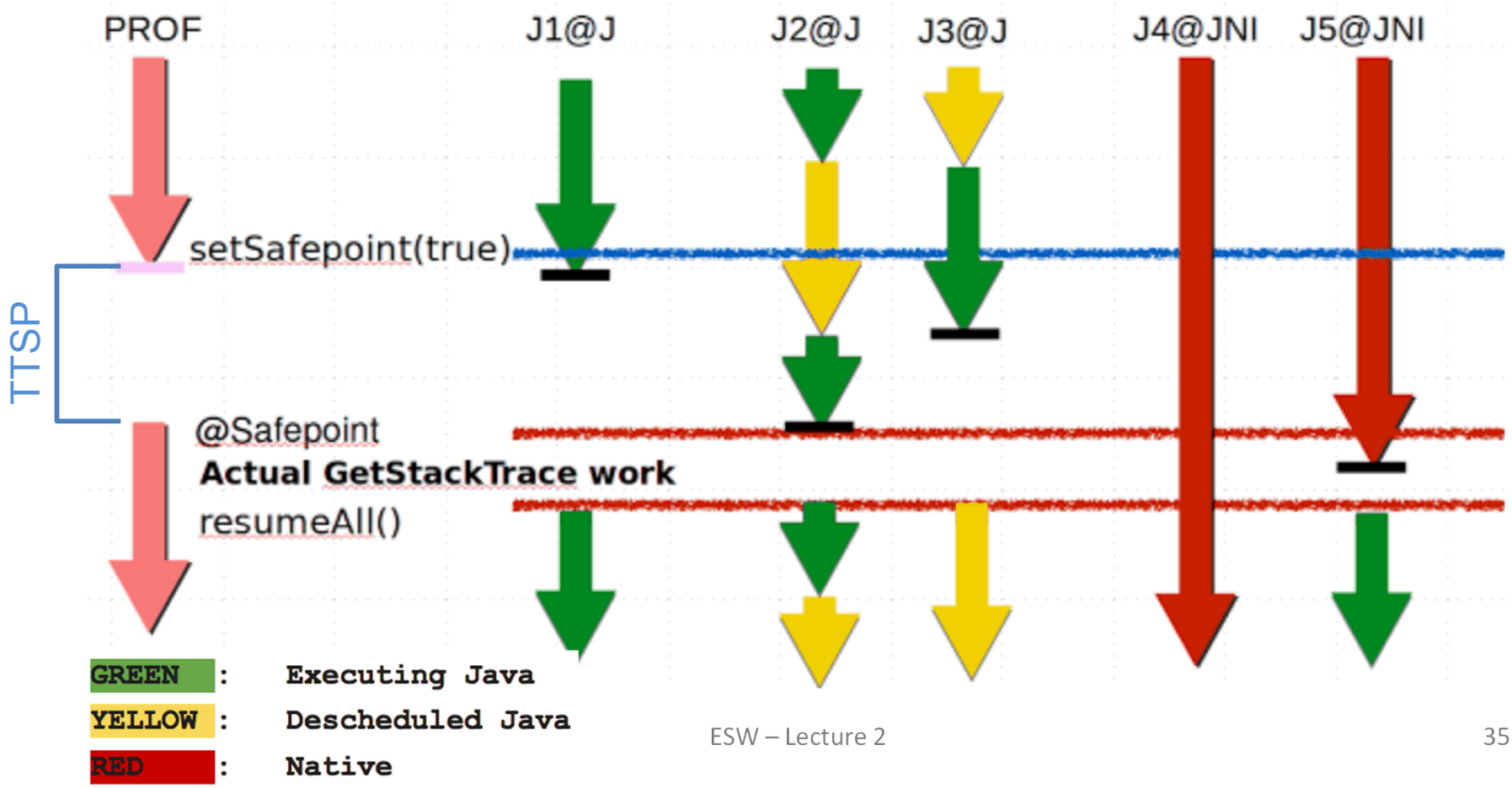
- Oop can be **safely manipulated externally** while thread is suspended
- in interpreted mode – between any 2 byte codes
- in C1/C2 compiled – end of all methods (not in-lined), non-counted loop back edge, during JVM run-time call
- parked, blocked on IO, monitor or lock
- while running JNI (do not need thread suspension)
- **global safepoint (all threads)** – stop the world
  - GC, print threads, thread dumps, heap dump, get all stack trace
  - enableBiasedLocking, RevokeBias
  - class redefinition (e.g. instrumentation), debug
- **local safepoint (just executing thread)**
  - de-optimization, enable/revoke bias locking, OSR

JVM – Time To Safe Point

» **Time To Safe Point** (TTSP) – how long it takes to enter safepoint

-XX:+PrintSafepointStatistics -XX:+PrintGCApplicationStoppedTime -XX:PrintSafepointStatisticsCount=1

TTSP overhead in profiler while calling GetStackTrace example with 5 threads:



JVM – Example 2 – daysInMonth Assembly Code – Tier 4

tier 4 – C2 compiler – no profile counters

because: count="5376" iicount="5376" **hot_count="5376"**

stack initialization, **use lookup table jump** for table switch

```
[Entry Point]
[Verified Entry Point]
  # {method} {0x000000012169d568} 'daysInMonth' '(II)Ljava/lang/Integer;' in 'SwitchTest'
  # parm0:    rsi       = int
  # parm1:    rdx       = int
  #           [sp+0x20]  (sp of caller)
  0x0000000108a97020: mov    %eax,-0x14000(%rsp)  ;   {no_reloc}
  0x0000000108a97027: push   %rbp
  0x0000000108a97028: sub    $0x10,%rsp         ;*synchronization entry
                                                ; - SwitchTest::daysInMonth@-1 (line 7)

  0x0000000108a9702c: mov    %esi,%r11d
  0x0000000108a9702f: dec    %r11d
  0x0000000108a97032: cmp    $0xc,%r11d
  0x0000000108a97036: jae    0x0000000108a9704a
  0x0000000108a97038: movslq %esi,%r10
  0x0000000108a9703b: movabs $0x108a96fc0,%r11  ;   {section_word}
  0x0000000108a97045: jmpq   *-0x8(%r11,%r10,8)  ;*tableswitch
                                                ; - SwitchTest::daysInMonth@1 (line 7)
```

month, year

default (>=12)

```
    135   18    b  4       SwitchTest::daysInMonth (144 bytes)
Decoding compiled method 0x0000000108a96e50:
Code:
[Constants]
  0x0000000108a96fc0 (offset:    0): 0x08a97083   0x0000000108a97083
  0x0000000108a96fc4 (offset:    4): 0x00000001
  0x0000000108a96fc8 (offset:    8): 0x08a9706c   0x0000000108a9706c
  0x0000000108a96fcc (offset:   12): 0x00000001
  0x0000000108a96fd0 (offset:   16): 0x08a97083   0x0000000108a97083
  0x0000000108a96fd4 (offset:   20): 0x00000001
  0x0000000108a96fd8 (offset:   24): 0x08a9708a   0x0000000108a9708a
  0x0000000108a96fdc (offset:   28): 0x00000001
  0x0000000108a96fe0 (offset:   32): 0x08a97083   0x0000000108a97083
  0x0000000108a96fe4 (offset:   36): 0x00000001
  0x0000000108a96fe8 (offset:   40): 0x08a9708a   0x0000000108a9708a
  0x0000000108a96fec (offset:   44): 0x00000001
  0x0000000108a96ff0 (offset:   48): 0x08a97083   0x0000000108a97083
  0x0000000108a96ff4 (offset:   52): 0x00000001
  0x0000000108a96ff8 (offset:   56): 0x08a97083   0x0000000108a97083
  0x0000000108a96ffc (offset:   60): 0x00000001
  0x0000000108a97000 (offset:   64): 0x08a9708a   0x0000000108a9708a
  0x0000000108a97004 (offset:   68): 0x00000001
  0x0000000108a97008 (offset:   72): 0x08a97083   0x0000000108a97083
  0x0000000108a9700c (offset:   76): 0x00000001
  0x0000000108a97010 (offset:   80): 0x08a9708a   0x0000000108a9708a
  0x0000000108a97014 (offset:   84): 0x00000001
  0x0000000108a97018 (offset:   88): 0x08a97083   0x0000000108a97083
  0x0000000108a9701c (offset:   92): 0x00000001
```

JVM – Example 2 – daysInMonth Assembly Code – Tier 4

target for month=4

Integer.<init>, Number.<init>, Object.<init> - iicount="5376" -> **Inline (hot)**

optimized branching, inlined TLAB allocation, inlined constructors, no nulling, caching optimization

```
0x0000000108a9708a: mov     $0x1e,%ebp          ;*goto
                                                ; - SwitchTest::daysInMonth@105 (line 26)

0x0000000108a9708f: mov     0x60(%r15),%rax
0x0000000108a97093: mov     %rax,%r10
0x0000000108a97096: add     $0x10,%r10
0x0000000108a9709a: cmp     0x70(%r15),%r10
0x0000000108a9709e: jae     0x0000000108a97124
0x0000000108a970a4: mov     %r10,0x60(%r15)
0x0000000108a970a8: prefetchnta 0xc0(%r10)
0x0000000108a970b0: mov     $0xf8002264,%r10d   ;   {metadata('java/lang/Integer')}
0x0000000108a970b6: shl     $0x3,%r10
0x0000000108a970ba: mov     0xa8(%r10),%r10
0x0000000108a970c1: mov     %r10,(%rax)
0x0000000108a970c4: movl    $0xf8002264,0x8(%rax)  ;*new
                                                ; - SwitchTest::daysInMonth@135 (line 30)
                                                ;   {metadata('java/lang/Integer')}
0x0000000108a970cb: mov     %ebp,0xc(%rax)      ;*synchronization entry
                                                ; - SwitchTest::daysInMonth@-1 (line 7)

0x0000000108a970ce: add     $0x10,%rsp
0x0000000108a970d2: pop     %rbp
0x0000000108a970d3: test    %eax,-0x214e0d9(%rip)        # 0x0000000106949000
                                                ;   {poll_return}
0x0000000108a970d9: retq
```

EBP=30 is retVal

TLAB Integer object allocation, ref in **RAX**

cache optimization 3 cache lines ahead

MarkWord fetch from class and then store

compressed OOP to Integer class

RAX.value = EBX (retVal)

final cleanup

RAX contains **return value** (pointer to Integer instance)

JVM – Example 2 – daysInMonth Assembly Code – Tier 4

target for default

class IllegalArgumentException no profile -> uncommon -> reinterpret

remap inputs, return back to reinterpreter

```
0x0000000108a9704a: mov    %esi,%ebp
0x0000000108a9704c: mov    $0x2,%esi
0x0000000108a97051: xchg   %ax,%ax
0x0000000108a97053: callq  0x000000010898b1a0  ; OopMap{off=56}
                                              ;*new  ; - SwitchTest::daysInMonth@108 (line 28)
                                              ;   {runtime_call}
0x0000000108a97058: callq  0x0000000107e7e33c  ;*new
                                              ; - SwitchTest::daysInMonth@108 (line 28)
                                              ;   {runtime_call}
```

then tier 3 code version is discarded

```
138   17       3       SwitchTest::daysInMonth (144 bytes)   made not entrant
```

JVM – Example 2 – compute Assembly Code – Tier 4 OSR

OSR @10 – On Stack Replacement at bytecode 10

tier 4 – C2 (before there was tier 3 OSR @10 because 60416 loops and tier 3)

because: backedge_count="101376" **hot_count="101376"**

```
147   21 %  b  4       SwitchTest::compute @ 10 (35 bytes)
```

copy 4 locals on stack from tier3 OSR @10 to regs

```
StackMapTable: number_of_entries = 2
  frame_type = 255 /* full_frame */
    offset_delta = 10
    locals = [ int, int, int, int ]
    stack = []
  frame_type = 250 /* chop */
    offset_delta = 22
```

```
private static int compute() {
    int month = 4;
    int year = 2000;
    int o=0;
    for (int i=0; i<1_000_000; i++) {
        o+=daysInMonth(month, year);
    }
    return o;
}
```

```
0x0000000108a98370: mov    %eax,-0x14000(%rsp)
0x0000000108a98377: push   %rbp
0x0000000108a98378: sub    $0x20,%rsp
0x0000000108a9837c: mov    (%rsi),%ebx
0x0000000108a9837e: mov    0x18(%rsi),%ebp
0x0000000108a98381: mov    0x10(%rsi),%r13d
0x0000000108a98385: mov    0x8(%rsi),%r14d
0x0000000108a98389: mov    %rsi,%rdi
```

RSI compiled stack of tier 3 **OSR** @10

```
 0: iconst_4
 1: istore_0
 2: sipush        2000
 5: istore_1
 6: iconst_0
 7: istore_2
 8: iconst_0
 9: istore_3
10: iload_3
11: ldc           #12
13: if_icmpge     33
16: iload_2
17: iload_0
18: iload_1
19: invokestatic  #13
22: invokevirtual #14
25: iadd
26: istore_2
27: iinc          3, 1
30: goto          10
33: iload_2
34: ireturn
```

JVM – Example 2 – compute Assembly Code – Tier 4 OSR

loop criteria

```
for (int i=0; i<1_000_000; i++) {
    o+=daysInMonth(month, year);
}
```

EBX is local I; 0xF4240 = 1 000 000

```
0x0000000108a98423: cmp    $0xf4240,%ebx
0x0000000108a98429: jge    0x0000000108a98450  ;*if_icmpge
                                               ; - SwitchTest::compute@13 (line 37)

0x0000000108a9842b: inc    %ebx                ;*iinc
                                               ; - SwitchTest::compute@27 (line 37)
```

then there is **inlined** tier 4 daysOfMonth (lookup jump) because the call is **hot**

ending with addition into accumulator o

```
0x0000000108a9841a: add    %r8d,%r14d          ; OopMap{off=189}
                                               ;*goto
                                               ; - SwitchTest::compute@30 (line 37)
```

reinterpret on end of cycle jump (unstable if bytecode), save 3 locals to stack

```
0x0000000108a98450: mov    $0xffffff65,%esi
0x0000000108a98455: mov    %r13d,(%rsp)
0x0000000108a98459: mov    %r14d,0x4(%rsp)
0x0000000108a9845e: mov    %ebx,0xc(%rsp)
0x0000000108a98462: nop
0x0000000108a98463: callq  0x000000010898b1a0  ; OopMap{off=264}
                                               ;*if_icmpge
                                               ; - SwitchTest::compute@13 (line 37)
                                               ;   {runtime_call}
0x0000000108a98468: callq  0x0000000107e7e33c  ;   {runtime_call}
```

```
StackMapTable: number_of_entries = 2
  frame_type = 255 /* full_frame */
    offset_delta = 10
    locals = [ int, int, int, int ]
    stack = []
  frame_type = 250 /* chop */
    offset_delta = 22
```

JVM – Example 2 – compute Assembly Code – Tier 4

tier 4 – C2

because: count="2" **backedge_count="150528"**

use combination of **full inline**, **dead code elimination**, **object escape**, **loop invariant hoisting**, **strength reduction**

```
   157   23    b  4       SwitchTest::compute (35 bytes)
Decoding compiled method 0x0000000108a97f90:
Code:
[Entry Point]
[Verified Entry Point]
[Constants]
  # {method} {0x000000012169d638} 'compute' '()I' in 'SwitchTest'
  #           [sp+0x20]  (sp of caller)
  0x0000000108a980c0: sub    $0x18,%rsp
  0x0000000108a980c7: mov    %rbp,0x10(%rsp)    ;*synchronization entry
                                                ; - SwitchTest::compute@-1 (line 34)

  0x0000000108a980cc: mov    $0x1c9c380,%eax
  0x0000000108a980d1: add    $0x10,%rsp
  0x0000000108a980d5: pop    %rbp
  0x0000000108a980d6: test   %eax,-0x214f0dc(%rip)        # 0x0000000106949000
                                                ;   {poll_return}
  0x0000000108a980dc: retq
```

30 000 000

RAX contains **return value** (primitive int)

Java Virtual Machine – Performance

» **requires warm-up to utilize benefits of C2 (or C1)**

» compilers cannot do all magic -> **write better algorithms**

» **32-bit vs 64 bits JVMs**

- 32-bit (max ~3GB heap)
  - smaller memory footprint
  - slower long & double operations
- 64-bit max 32GB virtual memory (with default ObjectAlignmentInBytes)
  - faster performance for long&double
  - slight increase of memory footprint
  - compressed OOPs are slightly slower for references upon usage
  - compressed OOPs less memory -> less frequent GC -> faster program
- 64-bit >32GB virtual memory (large heap)
  - fast reference usage
  - wasting a lot of memory (48GB ~32GB with compressed OOPs)

Java Virtual Machine – CPU and Memory Profiling

» **profiling**
- CPU – time spent in methods
- memory – usage, allocations

» **modes**
- sampling
  - periodic sampling of stacks of running threads
  - no invocation counts, no 100% accuracy (various sampling errors)
  - no bytecode (& assembly code) modifications
  - 1-2% impact to standard performance (TTSP, thread dumps, analysis)
- tracing (instrumetation) - method entry, exit, traceObjAllocations
  - **instrumented bytecode** -> **affected performance -> affected compiler optimizations**

» **jvisualvm**
- JVM monitoring, troubleshooting and profiling tool
- included in all JDKs

JVM – Example 2 – CPU Tracing of daysOfMonth

assembly code of tier 4 – C2 (before there was very complex tier 3)

inlined daysInMonth **rootMethodEntry** tracking

```
# {method} {0x000000012489e838} 'daysInMonth' '(II)Ljava/lang/Integer;' in 'SwitchTest'
# parm0:    rsi       = int
# parm1:    rdx       = int
#           [sp+0x70]  (sp of caller)
0x000000010c08aa80: mov    %eax,-0x14000(%rsp)  ;   {no_reloc}
0x000000010c08aa87: push   %rbp
0x000000010c08aa88: sub    $0x60,%rsp         ;*synchronization entry
                                              ; - SwitchTest::daysInMonth@-1 (line 7)

0x000000010c08aa8c: mov    %edx,0x4(%rsp)
0x000000010c08aa90: mov    %esi,(%rsp)
0x000000010c08aa93: movabs $0x76c73a180,%r10  ;   {oop(a 'java/lang/Class' = 'org/netbeans/lib/profiler/server/ProfilerRuntimeCPU')}
0x000000010c08aa9d: movzbl 0x82(%r10),%r11d   ;*getstatic recursiveInstrumentationDisabled
                                              ; - org.netbeans.lib.profiler.server.ProfilerRuntimeCPUFullInstr::rootMethodEntry@0 (line 189)
                                              ; - SwitchTest::daysInMonth@3 (line 7)

0x000000010c08aaa5: test   %r11d,%r11d
0x000000010c08aaa8: jne    0x000000010c08b075  ;*ifeq
                                              ; - org.netbeans.lib.profiler.server.ProfilerRuntimeCPUFullInstr::rootMethodEntry@3 (line 189)
                                              ; - SwitchTest::daysInMonth@3 (line 7)

0x000000010c08aaae: movabs $0x76c73e220,%r10  ;   {oop(a 'java/lang/Class' = 'org/netbeans/lib/profiler/server/ThreadInfo')}
0x000000010c08aab8: mov    0x78(%r10),%r8d    ;*getstatic lastThreadInfo
                                              ; - org.netbeans.lib.profiler.server.ThreadInfo::getThreadInfo@4 (line 244)
                                              ; - org.netbeans.lib.profiler.server.ProfilerRuntimeCPUFullInstr::rootMethodEntry@7 (line 193)
                                              ; - SwitchTest::daysInMonth@3 (line 7)

0x000000010c08aabc: mov    0x40(%r12,%r8,8),%ebp  ;*getfield thread
                                              ; - org.netbeans.lib.profiler.server.ThreadInfo::getThreadInfo@9 (line 246)
                                              ; - org.netbeans.lib.profiler.server.ProfilerRuntimeCPUFullInstr::rootMethodEntry@7 (line 193)
```

749 Bytes of assembly code for each **rootMethodEntry**

fc9

JVM – Example 2 – CPU Tracing of daysOfMonth

additional **rootMethodEntry** and **rootMethodExit** trackings for

Integer::<init> and Number::<init>

inlined rootMethodExit after Integer instance.value = retVal

```
0x000000010c08b73a: mov    0x8(%rsp),%r11
0x000000010c08b73f: mov    %r10d,0xc(%r11)    ;*synchronization entry
                                              ; - org.netbeans.lib.profiler.server.ProfilerRuntimeCPUFullInstr::methodExit@-1 (line 147)
                                              ; - java.lang.Integer::<init>@20 (line 851)
                                              ; - SwitchTest::daysInMonth@148 (line 30)

0x000000010c08b743: movabs $0x76c73a180,%r10  ;   {oop(a 'java/lang/Class' = 'org/netbeans/lib/profiler/server/ProfilerRuntimeCPU')}
0x000000010c08b74d: movzbl 0x82(%r10),%ebp    ;*getstatic recursiveInstrumentationDisabled
                                              ; - org.netbeans.lib.profiler.server.ProfilerRuntimeCPUFullInstr::methodExit@0 (line 147)
                                              ; - java.lang.Integer::<init>@20 (line 851)
                                              ; - SwitchTest::daysInMonth@148 (line 30)

0x000000010c08b755: test   %ebp,%ebp
0x000000010c08b757: jne    0x000000010c08bdd1  ;*ifeq
                                              ; - org.netbeans.lib.profiler.server.ProfilerRuntimeCPUFullInstr::methodExit@3 (line 147)
                                              ; - java.lang.Integer::<init>@20 (line 851)
                                              ; - SwitchTest::daysInMonth@148 (line 30)

0x000000010c08b75d: movabs $0x76c73e220,%r10  ;   {oop(a 'java/lang/Class' = 'org/netbeans/lib/profiler/server/ThreadInfo')}
0x000000010c08b767: mov    0x78(%r10),%ecx    ;*getstatic lastThreadInfo
                                              ; - org.netbeans.lib.profiler.server.ThreadInfo::getThreadInfo@4 (line 244)
                                              ; - org.netbeans.lib.profiler.server.ProfilerRuntimeCPUFullInstr::methodExit@7 (line 151)
                                              ; - java.lang.Integer::<init>@20 (line 851)
                                              ; - SwitchTest::daysInMonth@148 (line 30)

0x000000010c08b76b: mov    0x40(%r12,%rcx,8),%ebp  ;*invokestatic currentThread
                                              ; - org.netbeans.lib.profiler.server.ThreadInfo::getThreadInfo@0 (line 243)
                                                                                timeCPUFullInstr::rootMethodEntry@7 (line 19
```

313 Bytes of assembly code for each **rootMethodEntry**

JVM – Example 2 – CPU Tracing Outcome

Java VisualVM

Applications
- Local
  - VisualVM
  - IntelliJ Platform (pid 72840)
- Remote
- VM Coredumps
- Snapshots

Start Page | SwitchTest (pid 84116) | SwitchTest (pid 84774) | SwitchTest (pid 84916)

Overview | Monitor | Threads | Sampler | Profiler | Buffer Pools | Visual GC | Tracer | [snapshot] 09:39:39

SwitchTest (pid 84916)

Profiler Snapshot

View: Methods

Call Tree – Method	Total Time [%]	Total Time	Invocations
RMI TCP Connection(idle)		83,120 ms (100%)	1
RMI TCP Connection(idle)		38,035 ms (100%)	1
main		8,444 ms (100%)	1
SwitchTest.compute ()		8,444 ms (100%)	100
SwitchTest.daysInMonth (int, int)		5,059 ms (59.9%)	100000000
java.lang.Integer.<init> (int)		2,808 ms (33.3%)	100000000
Self time		1,750 ms (20.7%)	100000000
java.lang.Number.<init> ()		1,058 ms (12.5%)	100000000
Self time		2,251 ms (26.7%)	100000000
Self time		3,384 ms (40.1%)	100
SwitchTest.waitForAnyInputLine ()		0.000 ms (0%)	1

Method Name Filter (Contains)

JVM – Example 2 – Profiling Performance

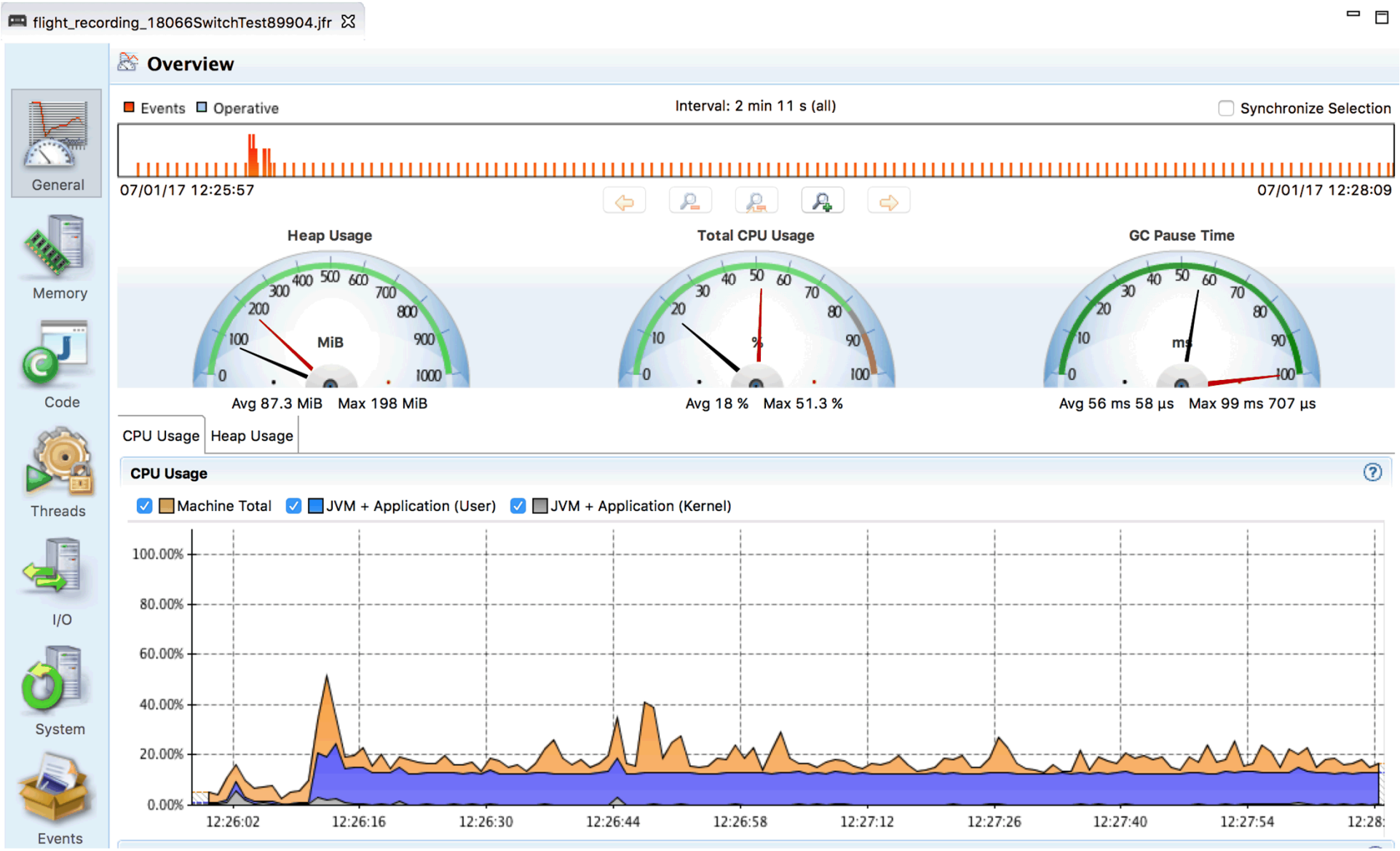
» CPU tracing of **compute** results into **much slower code**
- no object escape from daysInMonth call
- no invariant hoisting
- no strength reduction (full loop remains there)

» object allocation tracing is similar with **traceObjAlloc** injected calls

» **recommended approach**
- do sampling first
- identify performance bottlenecks (where most time is spent)
  - it could be outside of JVM (e.g. latency of external DB, file system)
- focus with tracing just to identified parts

JVM – Java Mission Control

jmc – JRockit JVM, included in commercial JDKs, sampling in Flight recorder



Approach to Performance Testing

» **test real application** – ideally the way it is used

- **microbenchmarks** – measure very small units
  - warm-up – to measure real code, not compilers itself, biased locks
    - keep in mind caching
  - beware of compilers – use results, reordering of operations
  - synchronization – multi-threaded benchmarks
  - vary pre-calculated parameters affecting complexity – different optimization in reality
- **macrobenchmarks** – measure application input/output
  - least performing component affects the whole application

» **understand throughput, elapsed and response time**

- outliers can occur – e.g. GC
- use existing generators than writing own

Approach to Performance Testing

» **understand variability** – changes over time

- internal state
- background effects – load, network
- probabilistic analysis – works with uncertainty

» **test early, test often** – ideally part of development cycle

- ideally some properly repeated mesobenchmarking
- automate tests – scripted
- proper test coverage of functionality and inputs
- test on target system – different code on different systems

B4M36ESW: Efficient software

Lecture 3: Benchmarking

Michal Sojka
michal.sojka@cvut.cz

March 4, 2019

Outline

1 Benchmarking
- Energy
- Memory consumption

2 Measuring execution time
- Timestamping
- Benchmark design
- Summarizing benchmark results
- Repeating iterations
- Repeating executions and compilation
- Multi-level repetition

3 Measuring speedup

Outline

1 Benchmarking
- Energy
- Memory consumption

2 Measuring execution time
- Timestamping
- Benchmark design
- Summarizing benchmark results
- Repeating iterations
- Repeating executions and compilation
- Multi-level repetition

3 Measuring speedup

Benchmark

Wikipedia defines benchmark as:

1. the act of running a computer program, a set of programs, or other operations, in order to **assess the relative performance** of an **object**, normally by running a number of standard tests and trials against it.
2. a benchmarking program itself (i.e. “XY is a free benchmark that tests your computer’s performance.”)

Object examples:

- Hardware
- Compiler
- Algorithm
- ...

Types of benchmarks:

- Micro-benchmarks (synthetic)
- Application benchmarks

Types of benchmark

1. Micro-benchmark
   - Evaluates very little part of an application
   - It is easy to determine source of speed-up/slow down
   - Typically, improvements in micro-benchmark do not imply improvements application performance
2. Application benchmarks
   - Evaluate performance of the whole applications
   - Performance is influenced by many read-world factors
   - For complex applications, it might difficult to determine the source of speed-up/slow-down

How to measure software performance?

- What to measure?
  - Execution time
  - Memory consumption
  - Energy
- How to measure?
  - Not as easy as it sounds
  - See the rest of the lecture

Outline

1 Benchmarking
- Energy
- Memory consumption

2 Measuring execution time
- Timestamping
- Benchmark design
- Summarizing benchmark results
- Repeating iterations
- Repeating executions and compilation
- Multi-level repetition

3 Measuring speedup

Measuring energy

- Connect power meter to your computer/board
- Use hardware-provided interfaces for power/energy measurement/control
  - These are more and more common these days

Example

Intel RAPL (Running Average Power Limit)

- Allows to monitor and/or limit power consumption of individual components
- Package domain, memory domain (DRAM)
- Interface via MSRs
- See Intel Software Developer's Manual: System Programming Guide

Outline

1 Benchmarking
- Energy
- Memory consumption

2 Measuring execution time
- Timestamping
- Benchmark design
- Summarizing benchmark results
- Repeating iterations
- Repeating executions and compilation
- Multi-level repetition

3 Measuring speedup

Measuring memory consumption

- Under modern OSes, measuring memory usage is surprisingly complex
- How programs consume memory?
  1. Program memory
     - Code, static data, heap, stack
     - Stack is allocated for each thread
  2. Operating system kernel memory
     - Allocated by the OS kernel on behalf of the program
     - network buffers, disk and file system caches, system objects (timers, semaphores, ...)
     - Sometimes, it is not possible to account this memory to an individual process – e.g. network receive buffers.
  3. Shared libraries
     - How to account memory consumed by libraries shared by multiple programs?

Basics of Linux Memory Statistics

- Tools like `top` or `htop` report several memory statistics

  VIRT Total amount of virtual memory reserved by the process. Not all this memory needs to be backed by physical memory. It does not include kernel memory.

  - Example: Allocate 1 GiB of virtual memory without allocating physical memory immediately.

```
mmap(NULL, 1ULL << 34, PROT_READ | PROT_WRITE,
     MAP_ANONYMOUS | MAP_SHARED, -1, 0);
```

  RES Currently resident (physical) memory

  SHR Memory shared with other processes (data, .so)

Outline

1 Benchmarking
- Energy
- Memory consumption

2 Measuring execution time
- Timestamping
- Benchmark design
- Summarizing benchmark results
- Repeating iterations
- Repeating executions and compilation
- Multi-level repetition

3 Measuring speedup

Outline

1 Benchmarking
- Energy
- Memory consumption

2 Measuring execution time
- Timestamping
- Benchmark design
- Summarizing benchmark results
- Repeating iterations
- Repeating executions and compilation
- Multi-level repetition

3 Measuring speedup

Measuring execution time

Timestamping

1. Use system calls
   - Linux: gettimeofday, clock_gettime(CLOCK_MONOTONIC)
   - Resolution: depends on available hardware (down to 1 ns), earlier it was a system tick period (1–10 ms)
   - Overhead – hundreds of CPU cycles (but see next slide)
2. Use hardware directly (e.g. timestamp counter)
   - TSC register on x86 (resolution 1 clock cycle, overhead few (≈ 8) clock cycles
   - Similar registers on other architectures
   - Cons: Can be subject to CPU frequency scaling, TSC counters on different CPU cores/sockets may not be synchronized

```
static inline uint64_t rdtsc() {
    uint64_t ret;
    asm volatile ( "rdtsc" : "=A" (ret) );
    return ret;
}
```

3. Combine both: Virtual syscall

Virtual syscall for fast timestamping

- Reading TSC is fast, but HW/frequency/socket dependent
  - Problematic when two timestamps need to be subtracted
  - OS kernel knows everything about HW/frequency/socket but calling kernel has overhead
- Idea: OS kernel publishes enough information for user space to reliably convert TSC value to wall-clock time without calling the kernel
  - time_ns = rdtsc() * tsc_scale + tsc_offset
- Virtual Dynamic Shared Object – VDSO
  - Kernel memory mapped to process address space
  - Looks like shared library
  - Application can call ordinary functions from there
  - `cat /proc/$$/maps|grep vdso`
  - gettimeofday, clock_gettime are functions implemented in VDSO

Outline

1 Benchmarking
- Energy
- Memory consumption

2 Measuring execution time
- Timestamping
- Benchmark design
- Summarizing benchmark results
- Repeating iterations
- Repeating executions and compilation
- Multi-level repetition

3 Measuring speedup

Measuring execution time

- Execution time exhibits variations
- Influenced by many factors:
  - Hardware, input data, compiler, memory layout, measuring overhead, rest of the system, network load, ... you name it
  - Same factors can be controlled, others cannot
- Repeatability of measurements
- How to design benchmark experiments properly?
- How to measure *speedup*?

Example

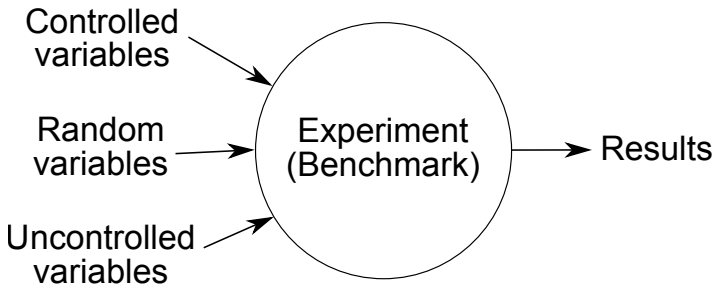
The Challenge of Reasonable Repetition

- Variations
- Measurements must be repeated
- We want to eliminate the influence of random (non-deterministic) factors
- Statistics

Controlled variables (e.g. compiler flags, hardware, algorithm changes) – we are interested how they impact the results

Random variables (e.g. hardware interrupts, OS scheduler) – we are interested in statistical properties of our results in face of these variables

Uncontrolled variables – mostly fixed, but can cause bias of the results



Benchmark goal

- Estimate (a confidence interval for) the mean of execution time of a given benchmark on one or more platforms.
- The mean is the property of the probability distribution of the random execution times
- We can only estimate the mean value from the measurements
- Confidence interval is important
  - CI of 95% $\Rightarrow$ in 95% of cases, the true mean will be within the interval.

Levels of repetition

- Results variance occurs typically at multiple levels, e.g.:
  - (re)compilation
  - execution
  - iteration inside a program
- Sound benchmarking methodology should evaluate all the levels with random variations

Next slides give answer to:

- How many times to repeat the experiment at each level?
  - As little times as possible to not waste time
  - As many times as possible to get reasonable confidence in results
- How to summarize the results?

Outline

1 Benchmarking
- Energy
- Memory consumption

2 Measuring execution time
- Timestamping
- Benchmark design
- Summarizing benchmark results
- Repeating iterations
- Repeating executions and compilation
- Multi-level repetition

3 Measuring speedup

Significance testing

Is it likely that two systems have different performance?

- Statistics can answer this with **Significance testing**
- However, this technique has problems, especially when used with results of computer benchmarks – see Kalibera's paper.
  - It is better to ask what is the speedup.
- Significance testing is implemented in the `ministat` tool (FreeBSD)

From ministat man page

The ministat command was written by Poul-Henning Kamp out of frustration over all the bogus benchmark claims made by people with no understanding of the importance of uncertainty and statistics.

ministat examples

```
+--------------------------------------------------------------------------+
|     +++ + +   x       x    x                                             |
|   +++++ +++   x  x    x    x                                             |
|   +++++ +++   x  xx xxx x  x                                             |
|   +++++ +++   x xxxxxxx x  x x                                           |
|   +++++ +++++ xxxxxxxxx x xx x                          +               x|
|             |________MA_________|                                        |
||_____M_A_______|                                                         |
+--------------------------------------------------------------------------+
    N           Min           Max        Median           Avg        Stddev
x  40         88.92       122.527        92.594      93.34845     5.3399441
+  40        82.313       112.625         84.52     85.447325     4.6810848
Difference at 95.0% confidence
        -7.90112 +/- 2.2355
        -8.46412% +/- 2.39479%
        (Student's t, pooled s = 5.02133)
Difference at 99.5% confidence
        -7.90112 +/- 3.59073
        -8.46412% +/- 3.84658%
```

Too little data with too similar distribution:

```
+------------------------------------------------------------------------+
|                                  +  +                                  |
|+ x       +   +    +  x           +  +    * x x        x          x  xxx +|
|                         |_______________________A______M_______________|  |
|          |____________________A___M________________|                     |
+------------------------------------------------------------------------+
    N           Min           Max        Median           Avg        Stddev
x  10       151.527       155.963       154.936      154.5278     1.4673007
+  10       151.371       156.096       153.618      153.3248     1.3398755
No difference proven at 95.0% confidence
```

Confidence interval

- We want to estimate the mean of a probability distribution
- We only have a limited set of r measurements and know almost nothing about the distribution
- We calculate the average value $\bar{Y}$ from the measurements
- How is the average different from the true mean value?
- $\bar{Y} \pm \frac{S_Y}{\sqrt{r}} q_{t(r-1)}(1 - \frac{\alpha}{2})$, where
  - $q_{t(r-1)}(1 - \frac{\alpha}{2})$ is $(1 - \frac{\alpha}{2})$-quantile of the Student's t-distribution with $r - 1$ degrees of freedom.
  - α is significance level (e.g. 5%)
- We say: Execution time of our benchmark is 25.4 ± 3.2 ms with 95% confidence.
- This means that the true mean is somewhere between 22.2 and 28.6 with probability of 95%.
- `https://stackoverflow.com/questions/15033511/compute-a-confidence-interval-from-sample-data`

Visual tests

- Calculate and visualize confidence intervals.
- Do the two confidence intervals overlap?
- No $\Rightarrow$ different performance is likely
- Yes $\Rightarrow$ more statistics needed
- Hard to estimate speedup and its confidence interval
- Note: `ministat` does not calculate confidence intervals, but standard deviations, i.e. S_Y

Recommendation

Analysis of results should be statistically rigorous and in particular should quantify any variation. Report performance changes with effect size confidence intervals.

Outline

1 Benchmarking
- Energy
- Memory consumption

2 Measuring execution time
- Timestamping
- Benchmark design
- Summarizing benchmark results
- Repeating iterations
- Repeating executions and compilation
- Multi-level repetition

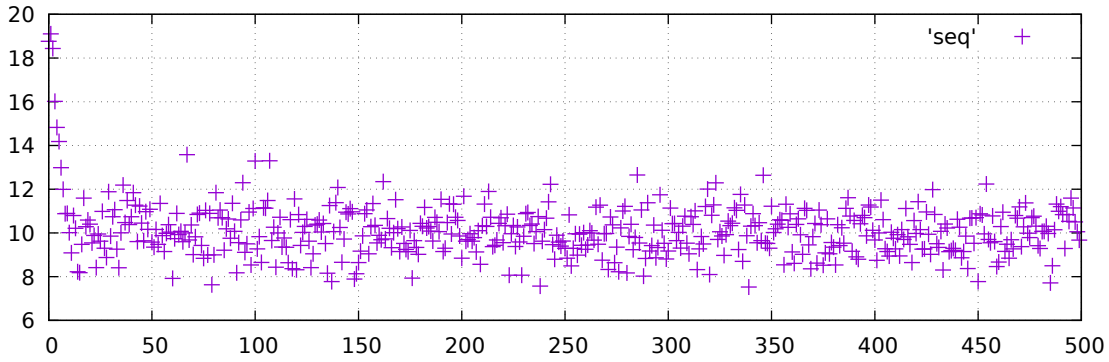
3 Measuring speedup

Repeating iterations

- Iteration = one execution of a loop body
- We are interested in *steady state performance*
- Initialization phase
  - First few iterations typically include the initialization overheads
  - Warming up caches, teaching branch predictor, memory allocations
- Independent state
  - Ideally, measurements should be *independent, identically distributed* (i.i.d.)
  - Independent: measurement does not depend on any a previous measurement
  - Independent $\Rightarrow$ initialized

When a benchmark reaches independent state?

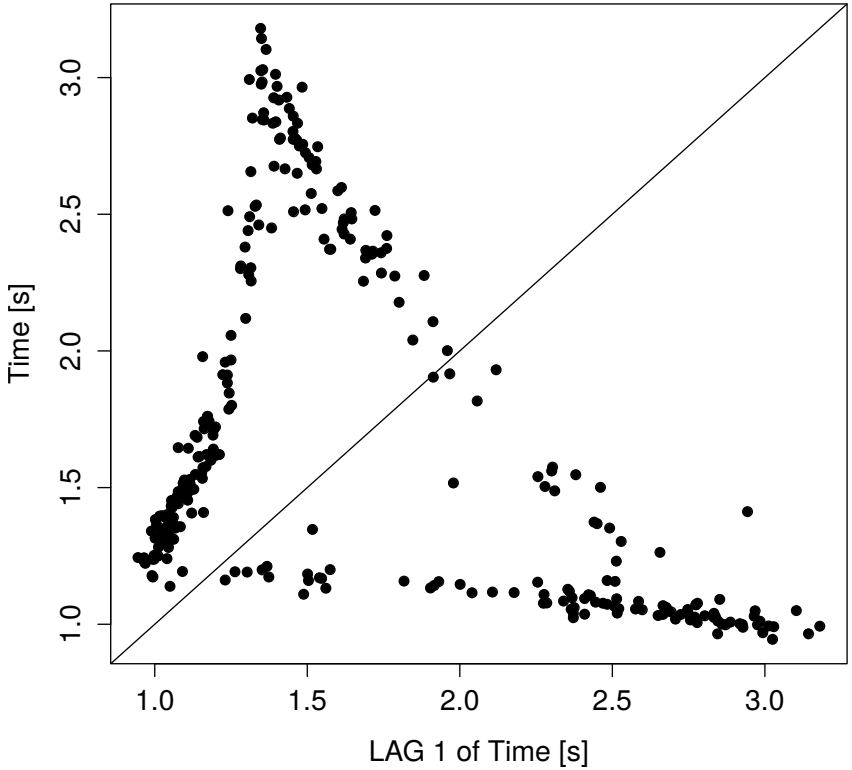
- Manual inspection of graphs from measured data
  1. run-sequence plot ⇒ easy identification of initialization phase ⇒ strip

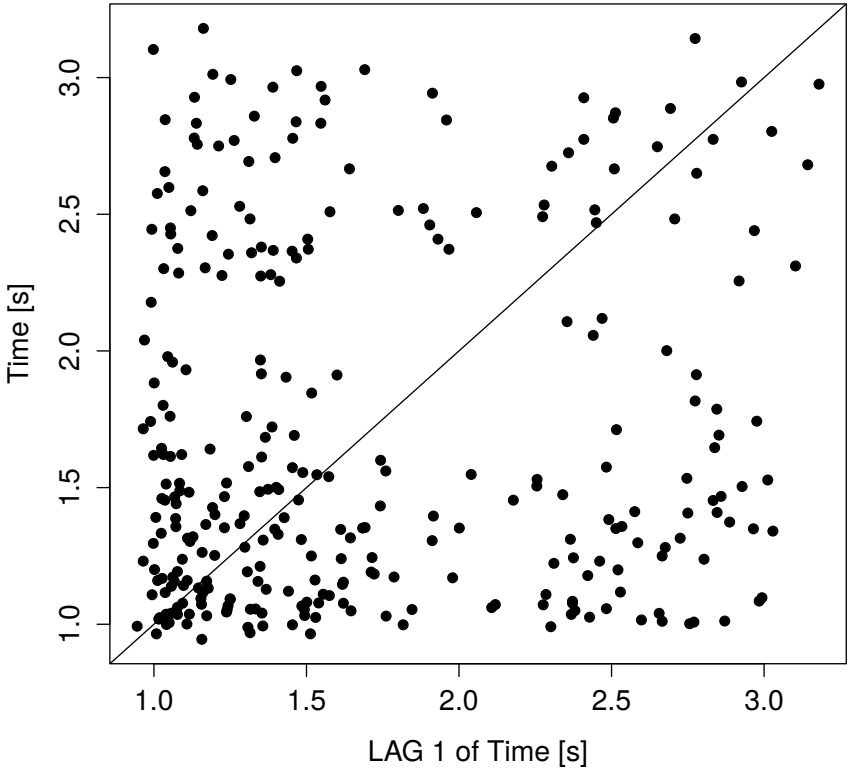


  2. Independence assessment – plot the following plots on original and randomly reordered sequence
     - lag plot (for several lags – e.g. 1–4)
     - auto-correlation function
  3. Any visible pattern suggests the measurements are not independent

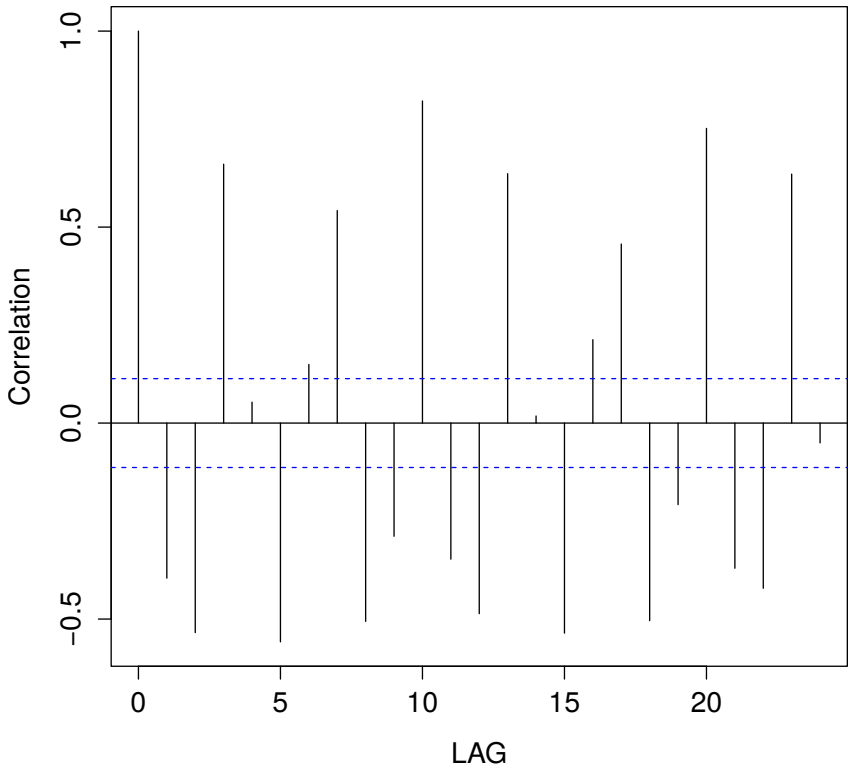
Lag plot

Dependency of a measured values on the previously measured value.

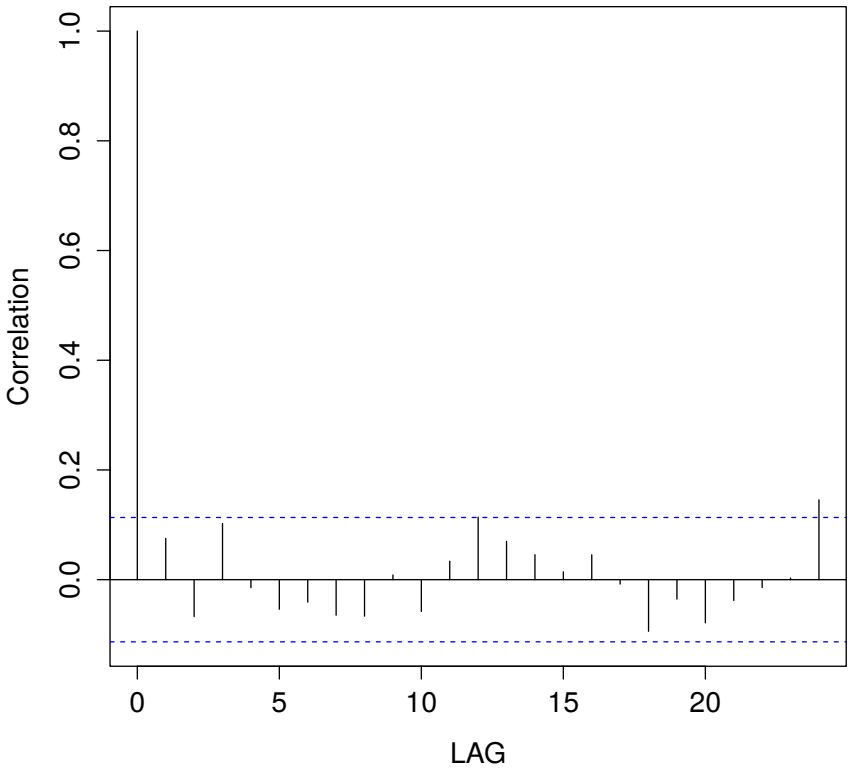




Auto-correlation function



dependent



independent

Recommendations

Use this manual procedure just once to find how many iterations each benchmark, VM and platform combination requires to reach an independent state.

If a benchmark does not reach an independent state in a reasonable time, take the same iteration from each run.

Outline

1 Benchmarking
- Energy
- Memory consumption

2 Measuring execution time
- Timestamping
- Benchmark design
- Summarizing benchmark results
- Repeating iterations
- Repeating executions and compilation
- Multi-level repetition

3 Measuring speedup

Repeating executions

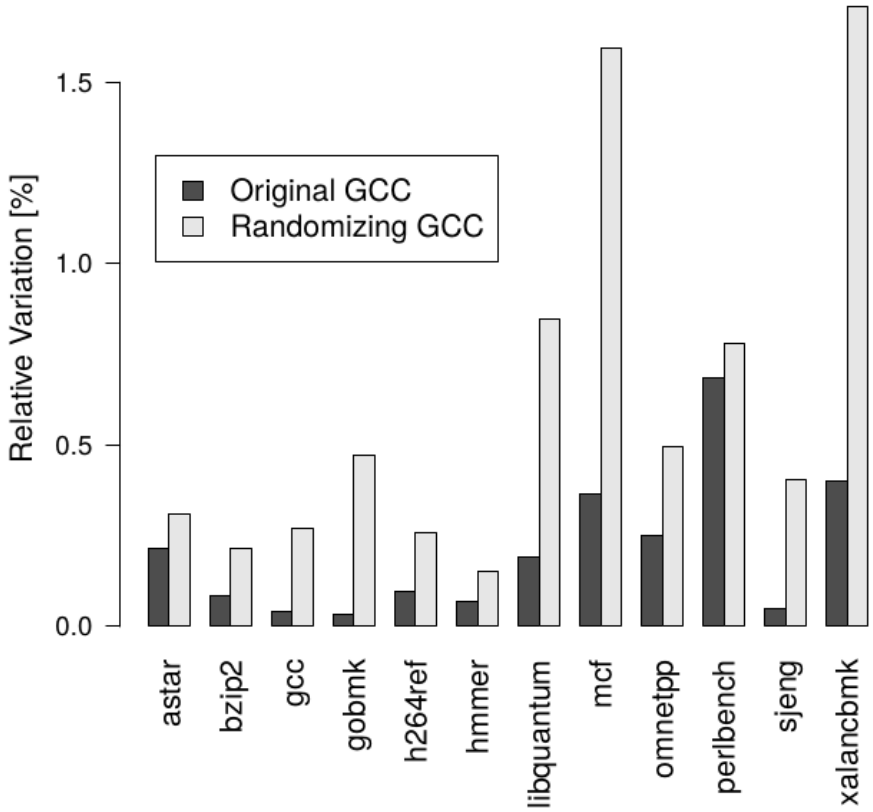
- Running a benchmark program multiple times
  - Effect of JIT compiler etc.
- Example: Variance in % of different benchmarks from DaCapo/OpenJDK benchmark suite

	bloat6	*eclipse9*	*lusearch9*	*tradebeans9*	*tradesoap9*	*xalan6*	*xalan9*
Iteration	14.1	0.8	3.3	1.5	0.8	7.0	3.5
Execution	3.7	0.4	30.3	0.4	0.4	9.1	1.0

- What if different executions exhibit higher variance than iterations? (see lusearch9)
- Determine initialized and independent state for executions as for iterations.

Repeating compilation

- Sometimes even a compiler can influences the benchmark results.
- Experiment: Code layout generated by the compiler: original vs. randomized



- Why code layout makes a difference?
- If you cannot control the factor, make it random!

Outline

1 Benchmarking
- Energy
- Memory consumption

2 Measuring execution time
- Timestamping
- Benchmark design
- Summarizing benchmark results
- Repeating iterations
- Repeating executions and compilation
- Multi-level repetition

3 Measuring speedup

Multi-level repetition

- We have to repeat the experiments to narrow confidence interval
- If the variance occurs at higher levels (execution, compilation), we need to repeat at least at that level.
- Repeating at lower level may be cheaper (no execution overhead, compilation overhead, etc.)
  - Time can be saved by repeating at lower levels.
- How to find required number of repetitions at each level to reach given confidence interval?
  - Can be formulated mathematically.
  - If you repeat too little, you have wide confidence intervals.
  - If you repeat too much, you waste your time with running unnecessary experiments.

Notation

- Levels
  - Lowest level (iteration) = 1
  - Highest level (e.g. compilation) = n
- *Initial experiment*
  - bold letters
  - $\mathbf{r_1}, \mathbf{c_1}$
- *Real experiment*
  - normal letters
  - r_1, c_1

Initial experiment

Goal is to find the required number of iterations at each level.

- Select number of repetitions (exclusive of warm-up) $\mathbf{r_1}, \mathbf{r_2}, \ldots$ to be arbitrary but sufficient value, say 20.
- Gather the cost of repetition at each level (time added exclusively by that level, e.g. compile time)
  - $\mathbf{c_1}$ iteration duration
  - $\mathbf{c_2}$ time execute benchmark up to independent state
  - $\mathbf{c_3}$ compilation time
- Measurement times: $\mathbf{Y_{j_n \cdots j_1}}, \quad j_1 = 1 \ldots r_1, j_2 = 1 \ldots r_2, \ldots$
- Calculate arithmetic means for different levels: $\bar{\mathbf{Y}}_{j_n \bullet \cdots \bullet}$

Variance estimators

- After initial experiments, we will calculate n unbiased variance estimators $\mathbf{T_1^2}, \ldots, \mathbf{T_n^2}$
- They describe how much each level contributes independently to variability in the results
- Start with calculating $\mathbf{S_i^2}$ – biased estimator of the variance at each level $i, 1 \leq i \leq n$:

$$\mathbf{S_i^2} = \frac{1}{\prod_{k=i+1}^{n} \mathbf{r_k}} \frac{1}{\mathbf{r_i} - 1} \sum_{j_n=1}^{r_n} \cdots \sum_{j_i=1}^{r_i} \left(\bar{\mathbf{Y}}_{j_n \ldots j_i \bullet \cdots \bullet} - \bar{\mathbf{Y}}_{j_n \ldots j_{i+1} \bullet \cdots \bullet} \right)^2$$

- Then obtain T_i^2:

$$\begin{aligned} T_1^2 &= \mathbf{S}_1^2 \\ \forall i, 1 < i \leq n, T_i^2 &= \mathbf{S}_i^2 - \frac{\mathbf{S}_{i-1}^2}{\mathbf{r_{i-1}}} \end{aligned}$$

- If $T_i^2 \leq 0$, this level induces little variation and repetitions can be skipped.

Real Experiment: Confidence Interval

- Optimum number of repetitions at different levels $r_1, \ldots, r_{n-1}$ can be calculated as:

$$\forall i, 1 \leq i < n, \quad r_i = \left\lceil \sqrt{\frac{c_{i+1}}{c_i} \frac{T_i^2}{T_{i+1}^2}} \right\rceil$$

- Then recalculate: S_n^2 and $\bar{Y}_{j_n \bullet \ldots \bullet}$ as before but with data from real experiment.
- Asymptotic confidence interval with confidence $(1 - \alpha)$ is:

$$\bar{Y} \pm t_{1-\frac{\alpha}{2},\nu} \sqrt{\frac{S_n^2}{r_n}}$$

where $t_{1-\frac{\alpha}{2},\nu}$ is $(1 - \frac{\alpha}{2})$-quantile of the t-distribution with $\nu = r_n - 1$ degrees of freedom.

Recommendation

For each benchmark/VM/platform, conduct a dimensioning experiment to establish the optimal repetition counts for each but the top level of the real experiment. Re-dimension only if the benchmark/VM/platform changes.

Outline

1 Benchmarking
- Energy
- Memory consumption

2 Measuring execution time
- Timestamping
- Benchmark design
- Summarizing benchmark results
- Repeating iterations
- Repeating executions and compilation
- Multi-level repetition

3 Measuring speedup

Measuring speedup

- Speedup: “With my optimization, the program runs 10% faster.”
- Speedup is a ratio of two execution times (random variables)
- What is the speedup confidence interval?
  E.g. 10%$\pm$2% faster with confidence of 99%
- How many times to repeat the speedup experiments?

Speedup confidence interval

- $\bar{Y}$ – old system execution time (average of measured times)
- $\bar{Y}'$ – new system execution time
- Speedup: $\bar{Y}'/\bar{Y}$
- Speedup confidence interval:

$$\frac{\bar{Y} \cdot \bar{Y}' \pm \sqrt{(\bar{Y} \cdot \bar{Y}')^2 - (\bar{Y}^2 - h^2)(\bar{Y}'^2 - h'^2)}}{\bar{Y}^2 - h^2}$$

$$h = \sqrt{t^2_{\frac{\alpha}{2},\nu} \frac{S_n^2}{r_n}} \quad h' = \sqrt{t^2_{\frac{\alpha}{2},\nu} \frac{S_n'^2}{r_n}}$$

Repetition count

- Relation of confidence interval of the speedup to confidence interval on individual measurements:

$$\mathbf{e_s} \approx \frac{\bar{Y}'}{\bar{Y}} \sqrt{e^2 + e'^2}$$

- $\mathbf{e_s}$, e, e' relative half-width of the speedup resp. old resp. new confidence interval, i.e. $e = h/\bar{Y}$

- Old system: 10±1 s, e=0.1 (10%)
- New system: 9±0.9 s, e'=0.1
- Speedup: ≈0.9±0.13
- Outcome: Speedup can be 1, i.e. no speedup!

Recommendation

Always provide effect size confidence intervals for results. Either for single systems or for speedups.

References

- Kalibera, T. and Jones, R. E. (2013) **Rigorous Benchmarking in Reasonable Time**. In: ACM SIGPLAN International Symposium on Memory Management (ISMM 2013), 20–12 June, 2013, Seattle, Washington, USA. `http://kar.kent.ac.uk/33611/`
- `https://www.cse.unsw.edu.au/~gernot/benchmarking-crimes.html`
- Erik van der Kouwe et al. (2018) **Benchmarking Crimes: An Emerging Threat in Systems Security**, `https://arxiv.org/abs/1801.02381`

PERFORMANCE ENGINEERING OF SOFTWARE SYSTEMS

Basic Performance Engineering

Saman Amarasinghe

Fall 2010

Basic Performance Engineering

Maximum use of the compiler/processor/system — Matrix Multiply Example

Modifying data structures

Modifying code structures

— Today

Using the right algorithm — Most Bithacks

© Saman Amarasinghe 2009

Bentley's Rules

There is no "theory" of performance programming

Performance Programming is:

- Knowledge of all the layers involved
- Experience in knowing when and how performance can be a problem
- Skill in detecting and zooming in on the problems
- A good dose of common sense

A set of rules

- Patterns that occur regularly
- Mistakes many make
- Possibility of substantial performance impact
- Similar to "Design Patterns" you learned in 6.005

© Saman Amarasinghe 2009

Bentley's Rules

A. **Modifying Data**
B. **Modifying Code**

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

1. Space for Time
2. Time for Space
3. Space and Time

B. Modifying Code

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

1. Space for Time
    a. Data Structure Augmentation
    b. Storing Precomputed Results
    c. Caching
    d. Lazy Evaluation
2. Time for Space
3. Space and Time

B. Modifying Code

© Saman Amarasinghe 2009

Data Structure Augmentation

Add some more info to the data structures to make common operations quicker

When is this viable?

- Additional information offers a clear benefit
- Calculating the information is cheap/easy
- Keeping the information current is not too difficult

Examples?

- Faster Navigation
  - Doubly linked list and Delete Operation
- Reduced Computation
  - Reference Counting

© Saman Amarasinghe 2009

Storing Precomputed Results

Store the results of a previous calculation. Reuse the precomputed results than redoing the calculation.

When is this viable?

- Function is expensive
- Function is heavily used
- Argument space is small
- Results only depend on the arguments
- Function has no side effects
- Function is deterministic

Examples:

© Saman Amarasinghe 2009

Precomputing Template Code

```
result precompute[MAXARG];

result func_initialize(int arg)
{
    for(i=0; i < MAXARG; i++)
      precompute[arg] = func(arg);
}

result func_apply(int arg)
{
    return precompute[arg];
}
```

© Saman Amarasinghe 2009

Pascal's Triangle

```
int pascal(int y, int x)
 {
   if(x == 0) return 1;
   if(x == y) return 1;
   return pascal(y-1, x-1) + pascal(y-1, x);
 }
```

Normal

```
 int pt[MAXPT][MAXPT];

 main() {
 ...
  for(i=0; i < PTMAX; i++) {
     pt[i][0] = 1;
     pt[i][i] = 1;
     for(j=1; j < i; j++)
       pt[i][j] = pt[i-1][j-1] + pt[i-1][j];
 }

int pascal(int y, int x) {
     return pt[y][x];
}
```

Precomputation

$$
\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1
\end{array}
$$

© Saman Amarasinghe 2009

Another example of precomputing

```
unsigned long fib(int n)
{
    if(n==1) return 1;
    if(n==2) return 1;
    return fib(n-1) + fib(n-2);
}
```

```
unsigned long fib(int n)
{
    int i;
    unsigned long prev, curr, tmp;
    if(n==1) return 1;
    if(n==2) return 1;
    prev = 1;
    curr = 1;
    for(i=3; i <=n; i++ ) {
        tmp = prev + curr;
        prev = curr;
        curr = tmp;
    }
    return curr;
}
```

© Saman Amarasinghe 2009

Caching

Store some of the heavily used/recently used results so they don't need to be computed

When is this viable?

- Function is expensive
- Function is heavily used
- Argument space is large
- There is temporal locality in accessing the arguments
- A single hash value can be calculated from the arguments
- There exists a "good" hash function
- Results only depend on the arguments
- Function has no side effects
- Coherence
  - Is required:
    - Ability to invalidate the cache when the results change
    - Function is deterministic
  - Or stale data can be tolerated for a little while

© Saman Amarasinghe 2009

Caching Template Code

```
typedef struct cacheval {
        argtype1 arg1;
        …
        argtypen argn;
        resulttype result;
}
struct cacheval cache[MAXHASH];

resulttype func_driver(argtype1 a1, …, argtypen an) {
        resulttype res;
        int bucket;
        bucket = get_hash(a1, a2, …, an);
        if((cache[bucket].arg1 == a1)&&…&&(cache[bucket].argn == an))
                return cache[bucket].result;
        res = func(a1, …, an);
        cache[bucket].arg1 = a1;
        …
        cache[bucket].argn = an;
        cache[bucket].result = res;
        return res;
}
```

© Saman Amarasinghe 2009

Lazy Evaluation

Differ the computation until the results are really needed

When is this viable?

- Only a few results of a large computation is ever used
- Accessing the result can be done by a function call
- The result values can be calculated incrementally
- All the data needed to calculate the results will remain unchanged or can be packaged-up

© Saman Amarasinghe 2009

Lazy Template Code

```
resulttype precompute[MAXARG];

resulttype func_apply(int arg)
{
    resulttype res;
    if(precompute[arg] != EMPTY)
      return precompute[arg];

    res = func(arg);
    precompute[arg] = res;
    return res;
}
```

© Saman Amarasinghe 2009

Pascal's Triangle

```
int pascal(int y, int x)
{
  if(x == 0) return 1
  if(x == y) return 1;
  return pascal(y-1, x-1) + pascal(y-1, x);
}
```

Normal

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
```

```
int pt[MAXPT][MAXPT];

main() {
...
 for(i=0; i < PTMAX; i++) {
    pt[i][0] = 1;
    pt[i][i] = 1;
    for(j=1; j < i; j++)
      pt[i][j] = pt[i-1][j-1] + pt[i-1][j];
}

int pascal(int y, int x) {
    return pt[y][x];
}
```

Precomputation

```
int pt[MAXPT][MAXPT];

int pascal(int y, int x)
{
  if(x == 0) return 1;
  if(x == y) return 1;
  if(pt[y][x] > 0) return pt[y][x];
  val = pascal(y-1, x-1) + pascal(y-1, x);
  pt[y][x] = val;
  return val;
}
```

Lazy Evaluation

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

1. Space for Time
2. Time for Space
    1. Packing/Compression
    2. Interpreters
3. Space and Time

B. Modifying Code

© Saman Amarasinghe 2009

Packing/Compression

Reduce the space of the data by storing them as "processed" which will require additional computation to get the data.

When is it viable?

- Storage is at a premium
  - Old days → most of the time!
  - Now
    - Embedded devices with very little memory/storage
    - Very large data sets
- Ability to drastically reduce the data size in storage
- Extraction process is amenable to the required access pattern
  - Batch – expand it all
  - Steam
  - Random access

© Saman Amarasinghe 2009

Packing / Compression

Packing Level

- Packing in memory
- Packing out of core storage

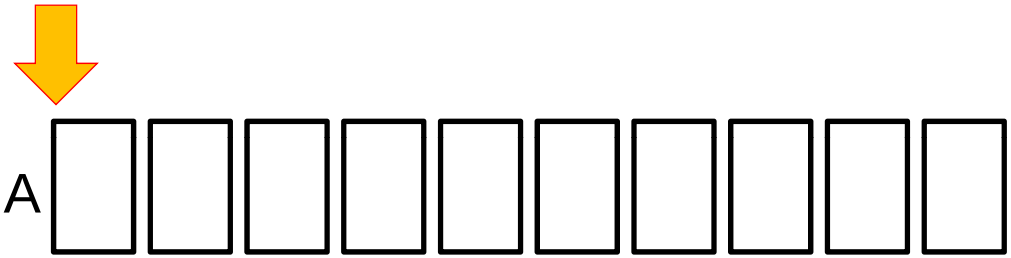
Packing Methods

- Use smaller data size
- Eliminate leading zeros
- Eliminate repetitions (LZ77)
- Heavyweight compression

© Saman Amarasinghe 2009

LZ77 Basics

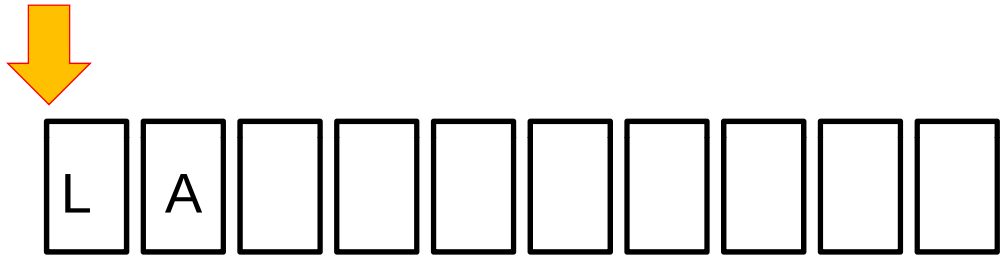
input stream → decompress → output stream



<1, 3> O <2, 4> L A

LZ77 Basics

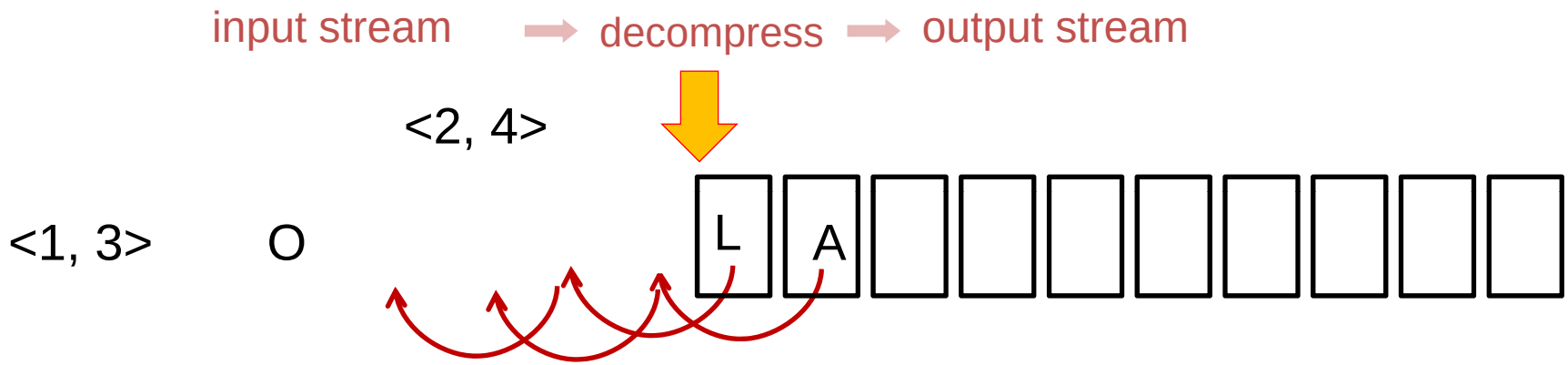
input stream → decompress → output stream



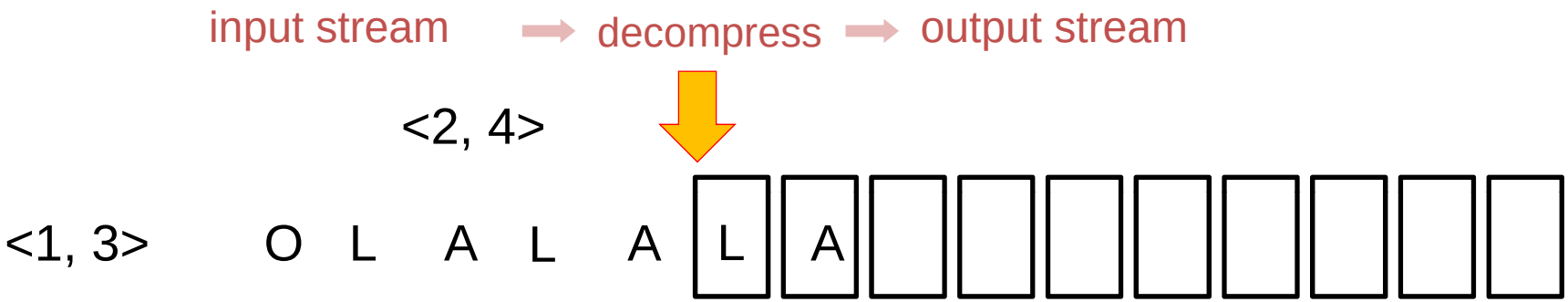
<1, 3> O <2, 4>

L	A								

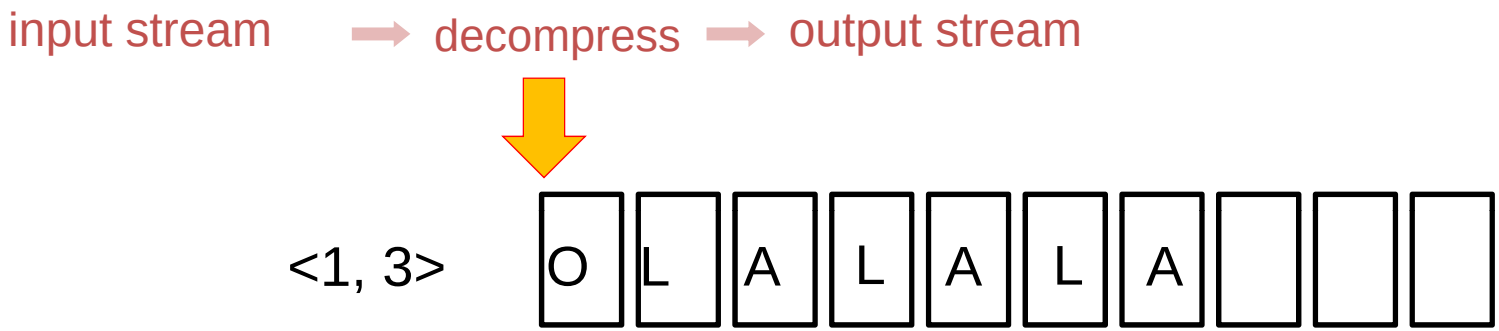
LZ77 Basics



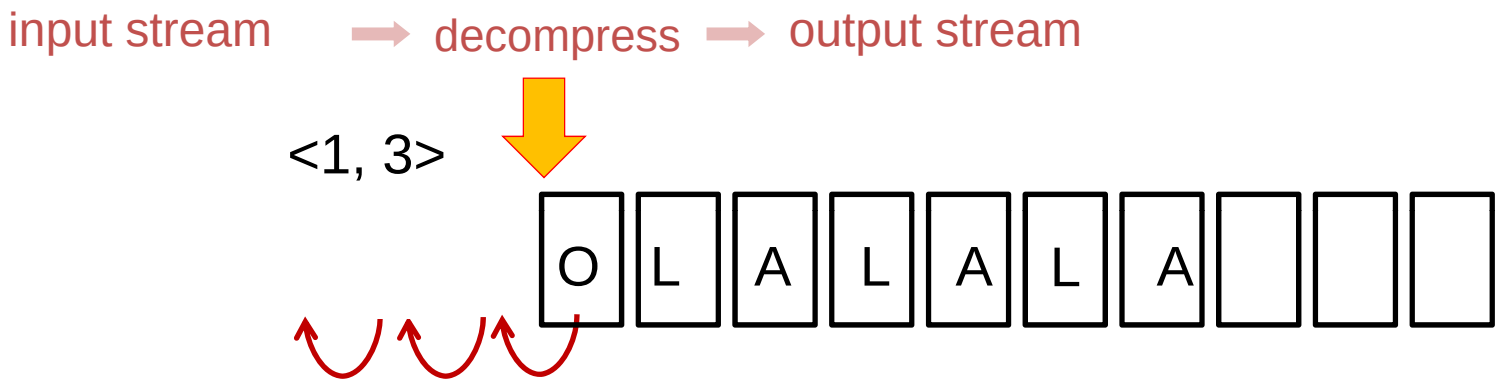
LZ77 Basics



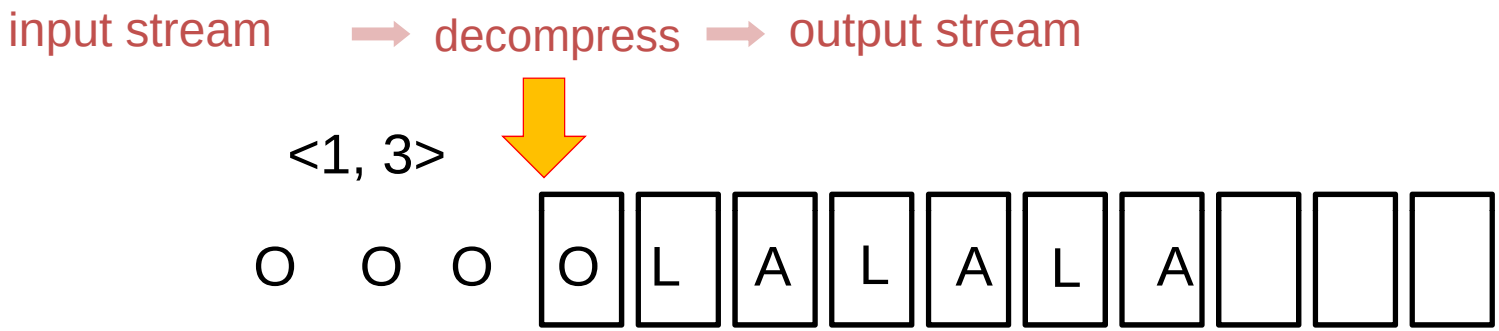
LZ77 Basics



LZ77 Basics



LZ77 Basics



Interpreters

Instead of writing a program to do a computation, use a language to describe the computation at a high level and write an interpreter for that language

Benefits:

- Nice and clean abstraction of the language
- Easy to add/change operations by changing the HLL program
- Much more compact representation

Examples:

- String processing
- Bytecodes

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

1. Space for Time
2. Time for Space
3. Space and Time
    1. SIMD

B. Modifying Code

© Saman Amarasinghe 2009

SIMD

Store short width data packed into the machine word (our Intel machines are 64 bit)

- 64 Booleans (unsigned long long)
- 2 32-bit floats
- 2 32-bit integers
- 4 16-bit integers
- 8 8-bit integers

Intel SSE instructions operate on these multi granularities

Single operation on all the data items

Win-win situation both faster and less storage

When viable?

- If the same operation is performed on all the data items
- Items can be stored contiguous in memory
- Common case don't have to pick or operate on each item separately

© Saman Amarasinghe 2009

Example: overlap in battleship boards

```
#define BSZ      64

int overlap(int board1[BSZ][BSZ],
            int board2[BSZ][BSZ])
{
   int i; j;
   for(i=0; i<BSZ; i++)
      for(j=0; j<BSZ; j++)
         if((board1[i][j] == 1)&&
            (board2[i][j] == 1))
               return 1;
   return 0;
}
```

```
#define BSZ      64

int overlap(
   unit64_t board1[BSZ],
   unit64_t board2[BSZ])
{
  int i;
   for(i=0; i<BSZ; i++)
      if((board1[i] & board2[i]) != 0)
         return 1;
   return 0;
}
```

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

B. Modifying Code

1. Loop Rules
2. Logic Rules
3. Procedure Rules
4. Expression Rules
5. Parallelism Rules

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

B. Modifying Code

1. Loop Rules
    a. Loop Invariant Code Motion
    b. Sentinel Loop Exit Test
    c. Loop Elimination by Unrolling
    d. Partial Loop Unrolling
    e. Loop fusion
    f. Eliminate wasted iterations
2. Logic Rules
3. Procedure Rules
4. Expression Rules
5. Parallelism Rules

© Saman Amarasinghe 2009

Loops

If each program instruction is only executed once

- 3 GHz machine
- Requres 12 Gbytes of instructions a second! (1 CPI, 32 bit instructions)
- A 100 GB disk full of programs done in 8 seconds

Each program instruction has to run millions of times

→ Loops

90% of the program execution time in 10% of the code

- All in inner loops

© Saman Amarasinghe 2009

Loop Invariant Code Motion

Move as much code as possible out of the loops

Compilers do a good job today

- Analyzable code
- Provably results are the same in every iteration
- Provably no side effects

Viability?

- Loop invariant computation that compiler cannot find
- The cost of keeping the value in a register is amortized by the savings

© Saman Amarasinghe 2009

Loop Invariant Code Motion Example

```
for(i=0;  i < N;  i++)
   X[i] = X[i]  * exp(sqrt(PI/2));
```

```
double factor;
factor = exp(sqrt(PI/2));
for(i=0;  i < N;  i++)
   X[i] = X[i]  * factor;
```

© Saman Amarasinghe 2009

Sentinel Loop Exit Test

When we iterate over a data to find a value, we have to check the end of the data as well as for the value.

Add an extra data item at the end that matches the test

Viability?

- Early loop exit condition that can be harnessed as the loop test
- When an extra data item can be added at the end
- Data array is modifiable

© Saman Amarasinghe 2009

Example of Sentinel Loop Exit Test

```
#define DSZ 1024
datatype array[DSZ];

int find(datatype val)
{
    int i;
    for(i=0; i<DSZ; i++)
        if(array[i] == val)
            return i;
    return -1;
```

OR

```
   i = 0;
   while((i<DSZ)&&(array[i] != val))
        i++;
    if(i==DSZ) return -1;
    return i;
}
```

```
#define DSZ 1024
datatype array[DSZ+1];

int find(datatype val)
{
    int i;
    array[DSZ] = val;
    i = 0;
    while(array[i] != val)
        i++;

    if(i == DSZ)
            return -1;
    return i;
}
```

© Saman Amarasinghe 2009

Loop Elimination by Unrolling

Known loop bounds → can fully unroll the loop

Viability?

- Small number of iterations (code blow-up is manageable)
- Small loop body (code blow-up is manageable)
- Little work in the loop body (loop test cost is non trivial)

Can get the compiler to do this.

Example:

```
sum = 0;
for(i=0; i<10; i++)
   sum = sum + A[i];
```

```
sum = A[0] + A[1] + A[2] + A[3] +
      A[4] + A[5] + A[6] + A[7] +
      A[8] + A[9];
```

© Saman Amarasinghe 2009

Partial Loop Unrolling

Make a few copies of the loop body.

Viability?

- Work in the loop body is minimal (viable impact of running the loop test fewer number of times)
- Or the ability to perform optimizations on combine loop bodies

Can get the compiler to do this

Example:

```
sum = 0;
for(i=0; i<n; i++)
   sum = sum + A[i];
```

```
sum = 0;
for(i=0; i<n-3; i += 4)
    sum += A[i] + A[i+1] + A[i+2] + A[i+3];
for(; i<n; i++)
    sum = sum + A[i];
```

© Saman Amarasinghe 2009

Loop Fusion

When multiple loops iterate over the same set of data put the computation in one loop body.

Viability?

- No aggregate from one loop is needed in the next
- Loop bodies are manageable

Example

```
amin = INTMAX;
for(i=0; i<n; i++)
   if(A[i] < amin) amin = A[i];

amax = INTMIN;
for(i=0; i<n; i++)
   if(A[i] > amax) amax = A[i];
```

```
amin = INTMIN;
amax = INTMAX;
for(i=0; i<n; i++) {
    int atmp = A[i];
     if(atmp < amin) amin = atmp;
     if(atmp > amax) amax = atmp;
}
```

© Saman Amarasinghe 2009

Eliminate Wasted Iterations

Change the loop bounds so that it will not iterate over an empty loop body

Viability?

- For a lot of iterations the loop body is empty
- Can change the loop bounds to make the loop tighter
- Or ability to change the data structures around (efficiently and correctly)

Example 1

```
for(i=0; i<n; i++)
  for(j = i;  j < n – i; j++)
     …
```

```
for(i=0; i<n/2; i++)
  for(j = i; j < n – i; j++)
     …
```

© Saman Amarasinghe 2009

Example II

```
int val[N];
int visited[N];

for(i=0; i < N; i++)
   visited[i] = 0;

for(i=0;  i < N;  ++) {
   int minloc;
   int minval = MAXINT;
   for(j=0;  < N;  j++)
      if(!visited[j])
         if(val[j] < minval) {
            minval = val[j];
            minloc = j;
         }
   visited[minloc] = 1;
    // process val[minloc]
}
```

```
int  val[N];

for(i=0;  i < N;  i++) {
   int minloc;
   int minval = MAXINT;
   for(j=0; j < N – i; j++)
      if(val[j] < minval) {
         minval = val[j];
         minloc =  ;
      }
    // process val[minloc]
    val[minloc] = val[N –  i – 1];
}
```

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

B. Modifying Code

1. Loop Rules
2. Logic Rules
    a. Exploit Algebraic Identities
    b. Short Circuit Monotone functions
    c. Reordering tests
    d. Precompute Logic Functions
    e. Boolean Variable Elimination
3. Procedure Rules
4. Expression Rules
5. Parallelism Rules

© Saman Amarasinghe 2009

Exploit Algebraic Identities

If the evaluation of a logical expression is costly, replace algebraically equivalent

Examples

- sqr (x) > 0 →
- sqrt(x*x + y*y) < sqrt(a*a + b*b) →
- ln(A) + ln(B) →
- SIN(X)*SIN(X) + COS(X)*COS(X) →

© Saman Amarasinghe 2009

Short Circuit Monotone Functions

In checking if a monotonically increasing function is over a threshold, don't evaluate beyond the threshold

Example:

```
sum = 0;
for(i=0; i < N; i++)
   sum = sum + X[i];
if(sum > cutoff) return 0;
return 1;
```

```
sum = 0;
i = 0;
while((i < N) && (sum < cutoff))
   sum = sum + X[i++];
if(sum > cutoff) return 0;
return 1;
```

```
sum = 0;
i = 0;
X[N] = cutoff;
while (sum < cutoff)
   sum = sum + X[i++];
if(i == N) return 1;
return 0;
```

© Saman Amarasinghe 2009

Reordering tests

Logical tests should be arranged such that inexpensive and often successful tests precede expensive and rarely successful ones

Add inexpensive and often successful tests before expensive ones

Example:

```
if (sqrt(sqr(x1 – x2) +
          sqr(y1 – y2))  < rad1+rad2)
    return COLLITION;
return OK;
```

```
if(abs(x1-x2) > rad1+rad2)
  return OK;
if(abs(y1-y2) > rad1+rad2)
  return OK;
if (sqrt(sqr(x1 – x2) +
          sqr(y1 – y2))  < rad1+rad2)
    return COLLITION;
return OK;
```

© Saman Amarasinghe 2009

Precompute Logic Functions

A logical function over a small finite domain can be replaced by a lookup in a table that represents the domain.

Example:

```
int palindrome(unsigned char val)
{
  unsigned char l, r;
  int i;
  l = 0x01;
  r = 0x80;
  for(i=0; i <4; i++) {
    if(((val & l) == 0) ^ ((val & r) == 0))
      return 0;
    l = l << 1;
    r = r >> 1;
  }
  return 1;
}
```

```
int isPalindrome[256];

// initialize array

int palindrome(unsigned char val)
{
    return isPalindrome[val];
}
```

© Saman Amarasinghe 2009

Boolean Variable Elimination

Replace the assignment to a Boolean variable by re placing it by an IF-THEN-ELSE

Example:

```
int v;
v = Boolean expression;

S1;
if(v)
   S2;
else
   S3;
S4;
if(v)
   S5;
```

```
if(Boolean expression) {
   S1;
   S2;
   S4;
   S5;
}else {
   S1;
   S3;
   S4;
}
```

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

B. Modifying Code

1. Loop Rules
2. Logic Rules
3. Procedure Rules
    a. Collapse Procedure Hierarchies
    b. Coroutines
    c. Tail Recursion Elimination
4. Expression Rules
5. Parallelism Rules

© Saman Amarasinghe 2009

Collapse Procedure Hierarchies

Inline small functions into the main body

- Eliminates the call overhead
- Provide further opportunities for compiler optimization

```
int getmax() {
    int xmax, i;
    xmax = MININT;
    for(i=0; i < N; i++)
        xmax = max(xmax, X[i]);
    return xmax;
}
```

```
#define max(a, b) (((a) > (b))?(a):(b))

int getmax() {
    int xmax, i;
    xmax = MININT;
    for(i=0; i < N; i++)
        xmax = max(xmax, X[i]);
    return xmax;
}
```

```
inline int max(int a,  int b) {
   if(a > b) return a;
   return b;
}

int getmax() {
    int xmax, i;
    xmax = MININT;
    for(i=0; i < N; i++)
        xmax = max(xmax, X[i]);
    return xmax;
}
```

© Saman Amarasinghe 2009

Coroutines

Multiple passes over data should be combined

- Similar to loop fusion(B.1.f), but at a scale and complexity beyond a single loop

Example pattern

```
Loop {
   Read from I
   ProcessA
   Write to II
}
Loop {
   Read from II
   ProcessB
   Write to III
}
```

```
Loop {
   Loop {
      Read from I
      ProcessA
      Write to buffer
   }
   Loop {
      Read from buffer
      Process B
      Write to III
   }
}
```

© Saman Amarasinghe 2009

Tail Recursion Elimination

In a self recursive function, if the last action is calling itself, eliminate the recursion.

Example pattern

```
int fact(int n,  int res) {
  if(n == 1) return res;
  return fact(n – 1,  res*n);
}
```

```
int fact(int n, int res) {
    while(1) {
        if(n == 1) return res;
        res = res*n;
        n = n  – 1;
    }
}
```

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

B. Modifying Code

1. Loop Rules
2. Logic Rules
3. Procedure Rules
4. Expression Rules
    a. Compile-time Initialization
    b. Common Subexpression Elimination
    c. Pairing Computation
5. Parallelism Rules

© Saman Amarasinghe 2009

Compile-Time Initialization

If a value is a constant, make it a compile-time constant.

- Save the effort of calculation
- Allow value inlining
- More optimization opportunities

Example

```
……..
vol = 2 * pi() * r * r;
```

```
#define PI 3.14159265358979
#define  R  12

{
    ……..
    vol = 2 * PI * R * R;
```

© Saman Amarasinghe 2009

Common Subexpression Elimination

If the same expression is evaluated twice, do it only once

Viability?

- Expression has no side effects
- The expression value does not change between the evaluations
- The cost of keeping a copy is amortized by the complexity of the expression
- Too complicated for the compiler to do it automatically

Example

```
x = sin(a) * sin(a);
```

```
double tmp;
…
tmp = sin(a);
x = tmp * tmp;
```

© Saman Amarasinghe 2009

Pairing Computation

If two similar functions are called with the same arguments close to each other in many occasions, combine them.

- Reduce call overhead
- Possibility of sharing the computation cost
- More optimization possibilities

Example

```
x = r * cos(a);
y = r * sin(a);
```

```
typedef struct twoduble {
    double d1;
    double d2;
}
....
    twodouble dd;
     dd = sincos(a);
     x = r * dd.d1;
     y = r * dd.d2;
```

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

B. Modifying Code

1. Loop Rules
2. Logic Rules
3. Procedure Rules
4. Expression Rules
5. Parallelism Rules
    a. Exploit Implicit Parallelism
    b. Exploit Inner Loop Parallelism
    c. Exploit Coarse Grain Parallelism
    d. Extra computation to create parallelism

© Saman Amarasinghe 2009

Implicit Parallelism

Reduce the loop carried dependences so that "software pipelining" can execute a compact schedule without stalls.

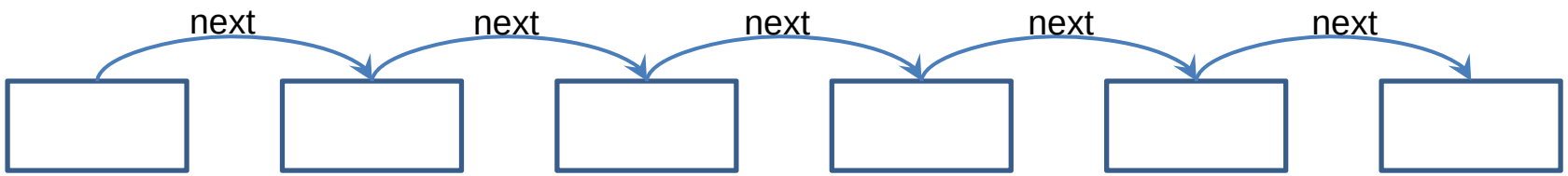
Example:

```
xmax = MININT;
for(i=0; i < N; i++)
   if(X[i] > xmax) xmax = X[i];
```

```
xmax1 = MININT;
xmax2 = MININT;
for(i=0; i < N –1; I += 2) {
   if(X[i] > xmax1) xmax1 = X[i];
   if(X[i+1] > xmax2) xmax2 = X[i+1];
}
if((i < N) &&(X[i] > xmax1)) xmax1 = X[i];
xmax = (xmax1 > xmax2)?xmax1:xmax2;
```

© Saman Amarasinghe 2009

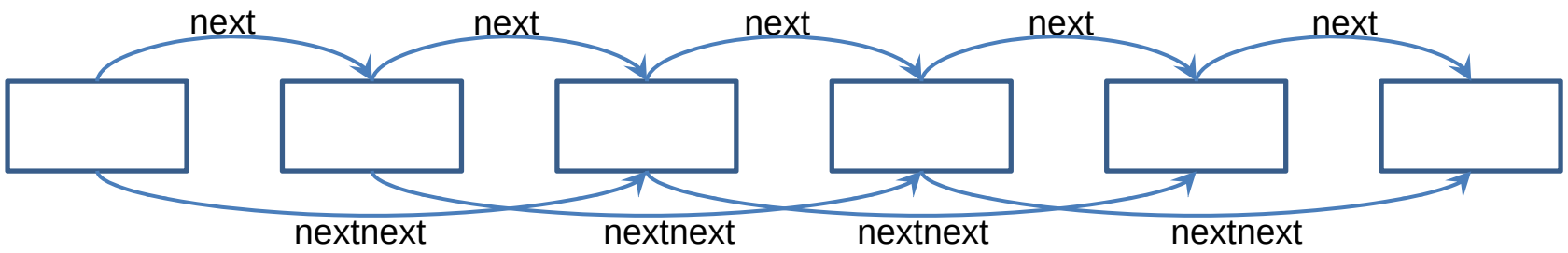
Example 2




```
curr = head;
tot = 0;
while(curr != NULL) {
    tot = tot + curr→val;
    curr = curr→next;
}
return tot;
```

© Saman Amarasinghe 2009

Example 2




```
curr = head;
tot = 0;
while(curr != NULL) {
    tot = tot + curr→val;
    curr = curr→next;
}
return tot;
```

```
curr = head;
if(curr == NULL) return 0;
tot1 = 0;
tot2 = 0;
while(curr→next) {
    tot1 = tot1 + curr→val;
    tot2 = tot2 + curr→next→val;
    curr = curr→nextnext;
}
if(curr)
    tot1 = tot1 + curr→val;
return tot1 + tot2;
```

Also see Rule A.1.a Data Structure Augmentation

© Saman Amarasinghe 2009

Exploit Inner Loop Parallelism

Facilitate inner loop vectorization (for SSE type instructions)

How? → by gingerly guiding the compiler to do so

- Iterative process by looking at why the loop is not vectorized and fixing those issues
- Most of the rules above can be used to simplify the loop so that the compiler can vectorize it

© Saman Amarasinghe 2009

Exploit Coarse Grain Parallelism

Outer loop parallelism (doall and doacross loops)

Task parallelism

Ideal for multicores

You need to do the parallelism yourself → later lectures

© Saman Amarasinghe 2009

Extra Computation to Create Parallelism

In many cases doing a little more work (or a slower algorithm) can make a sequential program a parallel one. Parallel execution may amortize the cost

Example:

```
double tot;
tot = 0;
for(i = 0; i < N; i++)
   for(j = 0;  j < N; j++)
      tot = tot + A[i][j];
```

```
double tot;
double tottmp[N];
for(i = 0; i < N; i++)
   tottmp[i] = 0;
for(i = 0; i < N; i++) {  //parallelizable
   double tmp;
   for(j = 0;  j < N; j++)
      tmp = tmp + A[i][j];
    tottmp[i]= tottmp[i]+ tmp;
}
tot = 0;
for(i = 0; i < N; i++)
   tot = tot + tottmp[i];
```

© Saman Amarasinghe 2009

Bentley's Rules

A Modifying Data

1. Space for Time
    a. Data Structure Augmentation
    b. Storing Precomputed Results
    c. Caching
    d. Lazy Evaluation
2. Time for Space
    a. Packing/Compression
    b. Interpreters
3. Space and Time
    a. SIMD

B Modifying Code

1. Loop Rules
    a. Loop Invariant Code Motion
    b. Sentinel Loop Exit Test
    c. Loop Elimination by Unrolling
    d. Partial Loop Unrolling
    e. Loop fusion
    f. Eliminate wasted iterations
2. Logic Rules
    a. Exploit Algebraic Identities
    b. Short Circuit Monotone functions
    c. Reordering$_{t}$ ests
    d. Precompute Logic Functions
    e. Boolean Variable Elimination
3. Procedure Rules
    a. Collapse Procedure Hierarchies
    b. Coroutines
    c. Tail Recursion Elimination
4. Expression Rules
    a. Compile-time Initialization
    b. Common Subexpression Elimination
    c. Pairing Computation
5. Parallelism Rules
    a. Exploit Implicit Parallelism
    b. Exploit Inner Loop Parallelism
    c. Exploit Coarse Grain Parallelism
    d. Extra computation to create parallelism

© Saman Amarasinghe 2009

Traveling Salesman Problem

Definition

- List of cities
- Location of each city (x,y coordinates on a 2-D map)
- Need to visit all the cities
- What order to visit the cities so that the distance traveled is shortest

Exact Shortest Distance Algorithm → Exponential

A Good Greedy Heuristic

- Start with any city
- Find the closest city that haven't been visited
- Visit that city next
- Iterate until all the cities are visited

© Saman Amarasinghe 2009

TSP Example

Original

```
  void get_path_1(int path[]) {
    int visited[MAXCITIES];
    int i, j;
    int curr, closest;
    double cdist;
    double totdist;

    for(i=0; i <MAXCITIES; i++)
      visited[i] = 0;

  curr = MAXCITIES-1;
    visited[curr] = 1;
    totdist = 0;
    path[0] = curr;
    for(i=1; i < MAXCITIES; i++) {
      cdist = MAXDOUBLE;
      for(j=0; j < MAXCITIES; j++)
        if(visited[j] == 0)
          if(dist(curr, j) < cdist) {
            cdist = dist(curr, j);
            closest = j;
          }

      path[i] = closest;
      visited[closest] = 1;
      totdist += dist(curr, closest);
      curr = closest;
    }
}

double dist(int i, int j) {
  return sqrt((Cities[i].x - Cities[j].x)*(Cities[i].x - Cities[j].x)+
                (Cities[i].y - Cities[j].y)*(Cities[i].y - Cities[j].y));
}
```

© Saman Amarasinghe 2009

TSP Example

Original

```
void get_path_1(int path[]) {
  int visited[MAXCITIES];
  int i, j;
  int curr, closest;
  double cdist;
  double totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0)
        if(dist(curr, j) < cdist) {
          cdist = dist(curr, j);
          closest = j;
        }

    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}

double dist(int i, int j) {
  return sqrt((Cities[i].x - Cities[j].x)*(Cities[i].x - Cities[j].x)+
              (Cities[i].y - Cities[j].y)*(Cities[i].y - Cities[j].y));
}
```

B.2.a Exploit Algebraic Identities

```
void get_path_2(int path[]) {
  int visited[MAXCITIES];
  int i, j;
  int curr, closest;
  double cdist;
  double totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0)
        if(distsq(curr, j) < cdist) {
          cdist = distsq(curr, j);
          closest = j;
        }

    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}

double distsq(int i, int j) {
  return ((Cities[i].x - Cities[j].x)*(Cities[i].x - Cities[j].x)+
          (Cities[i].y - Cities[j].y)*(Cities[i].y - Cities[j].y));
}
```

© Saman Amarasinghe 2009

TSP Example

B.2.a Exploit Algebraic Identities

```
void get_path_2(int path[])
{
  int visited[MAXCITIES];
  int i, j;
  int curr, closest;
  double cdist;
  double totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0)

        if(distsq(curr, j) < cdist) {
          cdist = distsq(curr, j);
          closest = j;
        }


    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}

double distsq(int i, int j) {
  return ((Cities[i].x - Cities[j].x)*(Cities[i].x - Cities[j].x)+
          (Cities[i].y - Cities[j].y)*(Cities[i].y - Cities[j].y));
}
```

B.4.b Common Subexpression Elimination

```
void get_path_3(int path[])
{
  int visited[MAXCITIES];
  int i, j;
  int curr, closest;
  double cdist, tdist;
  double totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0) {
        tdist = distsq(curr, j);
        if(tdist < cdist) {
          cdist = tdist;
          closest = j;
        }
      }

    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}
```

© Saman Amarasinghe 2009

TSP Example

B.4.b Common Subexpression Elimination

```
void get_path_3(int path[])
{
  int visited[MAXCITIES];
  int i, j;
  int curr, closest;
  double cdist, tdist;
  double totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;

    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0) {
        tdist = distsq(curr, j);

        if(tdist < cdist) {
          cdist = tdist;
          closest = j;
        }
      }

    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}
```

B.1.a Loop Invariant Code Motion

```
void get_path_4(int path[])
{
  int visited[MAXCITIES];
  int i, j;
  int curr, closest;
  double cdist, tdist;
  double cx, cy;
  double totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    cx = Cities[curr].x;
    cy = Cities[curr].y;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0) {
        tdist = (Cities[j].x - cx)*(Cities[j].x - cx) +
                (Cities[j].y - cy)*(Cities[j].y - cy);
        if(tdist < cdist) {
          cdist = tdist;
          closest = j;
        }
      }

    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}
```

© Saman Amarasinghe 2009

TSP Example

B.1.a Loop Invariant Code Motion

```
void get_path_4(int path[])
{
  int visited[MAXCITIES];
  int i, j, curr, closest;
  double cdist, tdist, cx, cy, totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    cx = Cities[curr].x;
    cy = Cities[curr].y;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0) {
        tdist = (Cities[j].x - cx)*(Cities[j].x - cx) +
                (Cities[j].y - cy)*(Cities[j].y - cy);

        if(tdist < cdist) {
          cdist = tdist;
          closest = j;
        }
      }

    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}
```

B.1.c Reordering Tests

```
void get_path_5(int path[])
{
  int visited[MAXCITIES];
  int i, j, curr, closest;
  double cdist, tdist, cx, cy, totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    cx = Cities[curr].x;
    cy = Cities[curr].y;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0) {
        tdist = (Cities[j].x - cx)*(Cities[j].x - cx);
        if(tdist < cdist) {
          tdist += (Cities[j].y - cy)*(Cities[j].y - cy);
          if(tdist < cdist) {
            cdist = tdist;
            closest = j;
          }
        }
      }
    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}
```

© Saman Amarasinghe 2009

TSP Example

B.1.c Reordering Tests

```
void get_path_5(int path[])
{
  int visited[MAXCITIES];
  int i, j, curr, closest;
  double cdist, tdist, cx, cy, totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    cx = Cities[curr].x;
    cy = Cities[curr].y;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0) {
        tdist = (Cities[j].x - cx)*(Cities[j].x - cx);

        if(tdist < cdist) {
          tdist += (Cities[j].y - cy)*(Cities[j].y - cy);

          if(tdist < cdist) {
            cdist = tdist;
            closest = j;
          }
        }
      }
    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}
```

B.4.b Common Subexpression Elimination

```
void get_path_6(int path[])
{
  int visited[MAXCITIES];
  int i, j, curr, closest;
  double cdist, tdist, cx, cy, totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    cx = Cities[curr].x;
    cy = Cities[curr].y;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0) {
        double tx =(Cities[j].x - cx);
        tdist = tx*tx;
        if(tdist < cdist) {
          double ty = (Cities[j].y - cy);
          tdist += ty*ty;
          if(tdist < cdist) {
            cdist = tdist;
            closest = j;
          }
        }
      }
    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}
```

© Saman Amarasinghe 2009

TSP Example

B.4.b Common Subexpression Elimination

```
void get_path_6(int path[])
{
  int visited[MAXCITIES];
  int i, j, curr, closest;
  double cdist, tdist, cx, cy, totdist;
  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    cx = Cities[curr].x;
    cy = Cities[curr].y;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0) {

        double tx =(Cities[j].x - cx);
        tdist = tx*tx;
        if(tdist < cdist) {
          double ty = (Cities[j].y - cy);
          tdist += ty*ty;
          if(tdist < cdist) {
            cdist = tdist;
            closest = j;

          }
        }
      }
    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}
```

B.1.f Eliminate Wasted Iterations

```
void get_path_7(int path[])
{
  int unvisited[MAXCITIES];
  int i, j, k, curr, closek, closej;
  double cdist, tdist, cx, cy, totdist;
  for(i=0; i <MAXCITIES; i++)
    unvisited[i] = i;

  curr = MAXCITIES-1;

  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    cx = Cities[curr].x;
    cy = Cities[curr].y;
    for(j=0; j < MAXCITIES-i; j++) {
      double tx;
      k = unvisited[j];
      tx =(Cities[k].x - cx);
      tdist = tx*tx;
      if(tdist < cdist) {
        double ty = (Cities[k].y - cy);
        tdist += ty*ty;
        if(tdist < cdist) {
          cdist = tdist;
          closek = k;
          closej = j;
        }
      }
    }
    path[i] = closek;
    unvisited[closej] = unvisited[MAXCITIES - i - 1];
    totdist += dist(curr, closek);
    curr = closek;
  }
}
```

© Saman Amarasinghe 2009

TSP Example

B.4.b Common Subexpression Elimination

```
void get_path_6(int path[])
{
  int visited[MAXCITIES];
  int i, j, curr, closest;
  double cdist, tdist, cx, cy, totdist;
  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;
  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    cx = Cities[curr].x;
    cy = Cities[curr].y;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0) {
        double tx =(Cities[j].x - cx);
        tdist = tx*tx;
        if(tdist < cdist) {
          double ty = (Cities[j].y - cy);
          tdist += ty*ty;
          if(tdist < cdist) {
            cdist = tdist;
            closest = j;
          }
        }
      }
    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);

    curr = closest;
  }
}
```

B.1.f Eliminate Wasted Iterations

```
void get_path_8(int path[])
{

  int i, j, curr, closest;
  double cdist, tdist, cx, cy, totdist;



  curr = MAXCITIES-1;

  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    cx = Cities[curr].x;
    cy = Cities[curr].y;
    for(j=0; j < MAXCITIES-i; j++)
      if(curr != j) {
        double tx =(Cities[j].x - cx);
        tdist = tx*tx;
        if(tdist < cdist) {
          double ty = (Cities[j].y - cy);
          tdist += ty*ty;
          if(tdist < cdist) {
            cdist = tdist;
            closest = j;
          }
        }
      }
    path[i] = closest;

    totdist += dist(curr, closest);
    Cities[curr] = Cities[MAXCITIES - i];
    curr = closest;
  }
}
```

© Saman Amarasinghe 2009

Performance

			Result		Time				
1		Original	96.344605		519.0697	ms	1.52		
2	B.2.a	Exploit Algebric Identities	96.344605		341.0185	ms		1.08	1.52
3	B.4.b	Common Subexpression Elimination	96.344605		314.5534	ms	1.02		1.65
4	B.1.a	Loop Invariant code Motion	96.344605		308.9298	ms		1.02	1.68
5	B.1.c	Reordering Tests	96.344605		302.6724	ms	0.91		1.71
6	B.4.b	Common Subexpression Elimination	96.344605		331.6027	ms		2.7	1.57
7	B.1.f	Eliminate Wated Iterations	97.738769		123.7693	ms	1.23		4.19
8	B.1.f	Eliminate Wated Iterations	99.003897		100.6096	ms			5.16

© Saman Amarasinghe 2009

MIT OpenCourseWare
http://ocw.mit.edu

6.172 Performance Engineering of Software Systems

Fall 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.

B4M36ESW: Efficient software

Lecture 4: C program compilation and execution
Bentley's rules

Michal Sojka
michal.sojka@cvut.cz

April 1, 2019

Outline

1 Motivating example

2 C/C++ compiler
- Frontend
- Optimization passes
  - High-level optimizations
  - High-level optimizations – Example
  - Low-level optimizations
  - Low-level optimizations – Example
- Miscellaneous

3 Linker

4 Execution

Outline

1 Motivating example

2 C/C++ compiler
- Frontend
- Optimization passes
  - High-level optimizations
  - High-level optimizations – Example
  - Low-level optimizations
  - Low-level optimizations – Example
- Miscellaneous

3 Linker

4 Execution

Example

```c
// vecadd.c
#define MM 100000000
unsigned a[MM], b[MM], c[MM];

void main()
{
  clock_t start,end;
  for (size_t i = 0; i < MM; ++i)
    a[i] = b[i] = c[i] = i;
  start = clock();
  vecadd(a, b, c, MM);
  end = clock();
  printf("time = %lf\n", (end - start)/
        (double)CLOCKS_PER_SEC);
}
```

```c
// veclib.c
void vecadd(int *a, int *b, int *c, size_t n)
{
  for (size_t i = 0; i < n; ++i) {
    a[i] += c[i];
    b[i] += c[i];
  }
}
```

```
gcc -Wall -g -O0 -march=core2 -o vecadd *.c
./vecadd
# time = 0.37
gcc -Wall -g -O2 -march=core2 -o vecadd *.c
./vecadd
# time = 0.12 ~ 300% speedup
```

```
gcc --g -O2 -march=core2 -o veclib.o veclib.c
objdump -d veclib.o
```

```
vecadd:
    test   %rcx,%rcx
    je     29 <vecadd+0x29> ---------.
    xor    %eax,%eax                 |
    nopw   0x0(%rax,%rax,1)          |
                                     |
    mov    (%rdx,%rax,4),%r8d ←-.    |
    add    %r8d,(%rdi,%rax,4)   |    |
    mov    (%rdx,%rax,4),%r8d   |    |
    add    %r8d,(%rsi,%rax,4)   |    |
    add    $0x1,%rax            |    |
    cmp    %rax,%rcx            |    |
    jne    10 <vecadd+0x10> ----'    |
    retq   ←-------------------------'
```

Why is the red mov in the code twice?

Pointer aliasing

- Because `c` may **alias** with `a`!
- vecadd() must work correctly even when called as
  `vecadd(a, a, a, MM)`
- Pointer aliasing = multiple pointers of the same type can point to the same memory
  - This prevents certain optimizations
- `restrict` qualifier = promise that pointer parameters of the same type can never alias

```
void vecadd(int * restrict a, int * b, int * c, size_t n)
{ ... }
```

```
./vecadd
# time = 0.10, speedup 10%!
```

- With `restrict`, the second `mov` disappears.

Compile Explorer

Play with the example at: `https://godbolt.org/z/opLwvN`

COMPILER EXPLORER

Add... More Share Other Policies

C source #1

A Save/Load + Add new... C

```c
#include <string.h>

void vecadd(int * restrict a,  int *b,  in
{
  for (size_t i = 0; i < n; ++i) {
    a[i] += c[i];
    b[i] += c[i];
  }
}
```

x86-64 gcc 8.3 (Editor #1, Compiler #1) C

x86-64 gcc 8.3 -O2 -std=c11

A 11010 .LX0: lib.f: .text // \s+ Intel Demangle

Libraries + Add new... Add tool...

```asm
vecadd:
        test    rcx, rcx
        je      .L1
        xor     eax, eax
.L3:
        mov     r8d, DWORD PTR [rdx+rax*4]
        add     DWORD PTR [rdi+rax*4], r8d
        add     DWORD PTR [rsi+rax*4], r8d
        add     rax, 1
        cmp     rcx, rax
        jne     .L3
.L1:
        ret
```

Output (0/0) x86-64 gcc 8.3 - 454ms (4970B)

Outline

1 Motivating example

2 C/C++ compiler
- Frontend
- Optimization passes
  - High-level optimizations
  - High-level optimizations – Example
  - Low-level optimizations
  - Low-level optimizations – Example
- Miscellaneous

3 Linker

4 Execution

C/C++ compilation

C/C++ compiler typically contains the following parts:

1. Compiler **frontend** – converts source code into intermediate representation (IR)
   - Preprocessor
   - Parser
2. **Optimization** passes
   - High-level optimizations
   - Low-level optimizations
3. A target-dependent **backend**
   - Generates assembly code or machine code
4. **Linker** – can be, and usually is, independent of the compiler

Open-source compilers:

- GCC
- LLVM/clang

LLVM has easier to understand code base. GCC improves code readability as well.

Outline

1 Motivating example

2 C/C++ compiler
- Frontend
- Optimization passes
  - High-level optimizations
  - High-level optimizations – Example
  - Low-level optimizations
  - Low-level optimizations – Example
- Miscellaneous

3 Linker

4 Execution

Abstract Syntax Tree (AST)

Parser produces AST

example.c:

```c
unsigned square(unsigned x)
{
  unsigned sum = 0, tmp;
  for (unsigned i = 1; i < x; i++) {
    tmp = x;
    sum += x;
  }
  return sum + tmp;
}
```

clang -Xclang -ast-dump -fsyntax-only example.c

```
TranslationUnitDecl <<invalid sloc>> <invalid sloc>
`-FunctionDecl <example.c:1:1, line:9:1> line:1:10 square 'unsigned int (unsigned int)'
  |-ParmVarDecl <col:17, col:26> col:26 used x 'unsigned int'
  `-CompoundStmt <line:2:1, line:9:1>
    |-DeclStmt <line:3:3, col:24>
    | |-VarDecl <col:3, col:18> col:12 used sum 'unsigned int' cinit
    | | `-ImplicitCastExpr <col:18> 'unsigned int' <IntegralCast>
    | |   `-IntegerLiteral <col:18> 'int' 0
    | `-VarDecl <col:3, col:21> col:21 used tmp 'unsigned int'
    |-ForStmt <line:4:3, line:7:3>
    | |-DeclStmt <line:4:8, col:22>
    | | `-VarDecl <col:8, col:21> col:17 used i 'unsigned int' cinit
    | |   `-ImplicitCastExpr <col:21> 'unsigned int' <IntegralCast>
    | |     `-IntegerLiteral <col:21> 'int' 1
    | |-<<<NULL>>>
    | |-BinaryOperator <col:24, col:28> 'int' '<'
    | | |-ImplicitCastExpr <col:24> 'unsigned int' <LValueToRValue>
    | | | `-DeclRefExpr <col:24> 'unsigned int' lvalue Var 'i' 'unsigned int'
    | | `-ImplicitCastExpr <col:28> 'unsigned int' <LValueToRValue>
    | |   `-DeclRefExpr <col:28> 'unsigned int' lvalue ParmVar 'x' 'unsigned int'
    | |-UnaryOperator <col:31, col:32> 'unsigned int' postfix '++'
    | | `-DeclRefExpr <col:31> 'unsigned int' lvalue Var 'i' 'unsigned int'
    | `-CompoundStmt <col:36, line:7:3>
    |   |-BinaryOperator <line:5:5, col:11> 'unsigned int' '='
    |   | |-DeclRefExpr <col:5> 'unsigned int' lvalue Var 'tmp' 'unsigned int'
    |   | `-ImplicitCastExpr <col:11> 'unsigned int' <LValueToRValue>
    |   |   `-DeclRefExpr <col:11> 'unsigned int' lvalue ParmVar 'x' 'unsigned int'
    |   `-CompoundAssignOperator <line:6:5, col:12> 'unsigned int' '+=' ComputeLHSTy='unsigned int' Co
    |     |-DeclRefExpr <col:5> 'unsigned int' lvalue Var 'sum' 'unsigned int'
    |     `-ImplicitCastExpr <col:12> 'unsigned int' <LValueToRValue>
    |       `-DeclRefExpr <col:12> 'unsigned int' lvalue ParmVar 'x' 'unsigned int'
    `-ReturnStmt <line:8:3, col:16>
      `-BinaryOperator <col:10, col:16> 'unsigned int' '+'
        |-ImplicitCastExpr <col:10> 'unsigned int' <LValueToRValue>
        | `-DeclRefExpr <col:10> 'unsigned int' lvalue Var 'sum' 'unsigned int'
        `-ImplicitCastExpr <col:16> 'unsigned int' <LValueToRValue>
          `-DeclRefExpr <col:16> 'unsigned int' lvalue Var 'tmp' 'unsigned int'
```

Intermediate representation (IR)

- AST is converted to IR
- This usually involves “dumb” expansion of templates

example.c:

```
unsigned square(unsigned x)
{
  return x*x;
}
```

LLVM intermediate representation
$ clang -S -emit-llvm example.c

```
define dso_local i32 @square(i32) #0 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = load i32, i32* %2, align 4
  %5 = mul i32 %3, %4
  ret i32 %5
}
```

clang -Xclang -ast-dump -fsyntax-only example.c

```
TranslationUnitDecl <<invalid sloc>> <invalid sloc>
`-FunctionDecl <example.c:1:1, line:4:1> line:1:10 square 'unsigned int (unsig
  |-ParmVarDecl <col:17, col:26> col:26 used x 'unsigned int'
  `-CompoundStmt <line:2:1, line:4:1>
    `-ReturnStmt <line:3:3, col:12>
      `-BinaryOperator <col:10, col:12> 'unsigned int' '*'
        |-ImplicitCastExpr <col:10> 'unsigned int' <LValueToRValue>
        | `-DeclRefExpr <col:10> 'unsigned int' lvalue ParmVar 'x' 'unsigned i
        `-ImplicitCastExpr <col:12> 'unsigned int' <LValueToRValue>
          `-DeclRefExpr <col:12> 'unsigned int' lvalue ParmVar 'x' 'unsigned i
```

Conversion of for loops to IR

- C code:

```
for (initializer; condition; modifier) {
  body
}
```

- IR “template”:

```
initializer
  goto COND
COND:
  if (condition)
    goto BODY
  else
    goto EXIT
BODY:
  body
  modifier
  goto COND
EXIT:
```

Intermediate representation vs. assembler

example.c:

```
unsigned square(unsigned x)
{
  return x*x;
}
```

$ clang -S -emit-llvm example.c

```
; ModuleID = 'example.c'
source_filename = "example.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @square(i32) #0 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = load i32, i32* %2, align 4
  %5 = mul i32 %3, %4
  ret i32 %5
}

attributes #0 = { noinline nounwind optnone uwtable "correct...

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 7.0.1-8 (tags/RELEASE_701/final)"}
```

IR is machine independent

$ llc -O0 -march=x86-64 example.ll

```
square:                                 ...
# %bb.0:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %edi
        imull   -4(%rbp), %edi
        movl    %edi, %eax
        popq    %rbp
        retq
.Lfunc_end0:
                                        ...
```

$ llc -O0 -march=arm example.ll

```
square:
@ %bb.0:
        sub     sp, sp, #8
        mov     r1, r0
        str     r0, [sp, #4]
        ldr     r0, [sp, #4]
        mul     r2, r0, r0
        mov     r0, r2
        str     r1, [sp]                ...
        add     sp, sp, #8
        mov     pc, lr
.Lfunc_end0:
                                        ...
```

Assembler generation from IR is detailed later.

Outline

1 Motivating example

2 C/C++ compiler
- Frontend
- Optimization passes
  - High-level optimizations
  - High-level optimizations – Example
  - Low-level optimizations
  - Low-level optimizations – Example
- Miscellaneous

3 Linker

4 Execution

Optimizations in general

- Many, many options
- `https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/Optimize-Options.html`
- `gcc -Q --help=optimizers -O2`
- `https://llvm.org/docs/Passes.html`

High-level optimizations (clang/LLVM)

Analysis passes – add information for use in other passes

- Exhaustive Alias Analysis Precision Evaluator (-aa-eval)
- Basic Alias Analysis (stateless AA impl) (-basicaa)
- Basic CallGraph Construction (-basiccg)
- Count Alias Analysis Query Responses (-count-aa)
- Dependence Analysis (-da)
- AA use debugger (-debug-aa)
- Dominance Frontier Construction (-domfrontier)
- Dominator Tree Construction (-domtree)
- Simple mod/ref analysis for globals (-globalsmodref-aa)
- Counts the various types of Instructions (-instcount)
- Interval Partition Construction (-intervals)
- Induction Variable Users (-iv-users)
- Lazy Value Information Analysis (-lazy-value-info)
- LibCall Alias Analysis (-libcall-aa)
- Statically lint-checks LLVM IR (-lint)
- Natural Loop Information (-loops)
- Memory Dependence Analysis (-memdep)
- Decodes module-level debug info (-module-debuginfo)
- Post-Dominance Frontier Construction (-postdomfrontier)
- Post-Dominator Tree Construction (-postdomtree)
- Detect single entry single exit regions (-regions)
- Scalar Evolution Analysis (-scalar-evolution)
- ScalarEvolution-based Alias Analysis (-scev-aa)
- Target Data Layout (-targetdata)

High-level optimizations (clang/LLVM)

Transform passes

- Aggressive Dead Code Elimination (-adce)
- Inliner for always_inline functions (-always-inline)
- Promote 'by reference' arguments to scalars (-argpromotion)
- Basic-Block Vectorization (-bb-vectorize)
- Profile Guided Basic Block Placement (-block-placement)
- Break critical edges in CFG (-break-crit-edges)
- Optimize for code generation (-codegenprepare)
- Merge Duplicate Global Constants (-constmerge)
- Simple constant propagation (-constprop)
- Dead Code Elimination (-dce)
- Dead Argument Elimination (-deadargelim)
- Dead Type Elimination (-deadtypeelim)
- Dead Instruction Elimination (-die)
- Dead Store Elimination (-dse)
- Deduce function attributes (-functionattrs)
- Dead Global Elimination (-globaldce)
- Global Variable Optimizer (-globalopt)
- Global Value Numbering (-gvn)
- Canonicalize Induction Variables (-indvars)
- Function Integration/Inlining (-inline)
- Combine redundant instructions (-instcombine)
- Internalize Global Symbols (-internalize)
- Interprocedural constant propagation (-ipconstprop)
- Interprocedural Sparse Conditional Constant Propagation (-ipsccp)
- Jump Threading (-jump-threading)
- Loop-Closed SSA Form Pass (-lcssa)
- Loop Invariant Code Motion (-licm)
- Delete dead loops (-loop-deletion)
- Extract loops into new functions (-loop-extract)
- Extract at most one loop into a new function (-loop-extract-single)
- Loop Strength Reduction (-loop-reduce)
- Rotate Loops (-loop-rotate)
- Canonicalize natural loops (-loop-simplify)
- Unroll loops (-loop-unroll)
- Unswitch loops (-loop-unswitch)
- Lower atomic intrinsics to non-atomic form (-loweratomic)
- Lower invokes to calls, for unwindless code generators (-lowerinvoke)
- Lower SwitchInsts to branches (-lowerswitch)
- Promote Memory to Register (-mem2reg)
- MemCpy Optimization (-memcpyopt)
- Merge Functions (-mergefunc)
- Unify function exit nodes (-mergereturn)
- Partial Inliner (-partial-inliner)
- Remove unused exception handling info (-prune-eh)
- Reassociate expressions (-reassociate)
- Demote all values to stack slots (-reg2mem)
- Scalar Replacement of Aggregates (-sroa)
- Sparse Conditional Constant Propagation (-sccp)
- Simplify the CFG (-simplifycfg)
- Code sinking (-sink)
- Strip all symbols from a module (-strip)
- Strip debug info for unused symbols (-strip-dead-debug-info)
- Strip Unused Function Prototypes (-strip-dead-prototypes)
- Strip all llvm.dbg.declare intrinsics (-strip-debug-declare)
- Strip all symbols, except dbg symbols, from a module (-strip-nondebug)
- Tail Call Elimination (-tailcallelim)

Common optimization passes together (-O2)

example.c:

```c
unsigned square(unsigned x)
{
  unsigned sum = 0, tmp;
  for (unsigned i = 1; i < x; i++) {
    tmp = x;
    sum += x;
  }
  return sum + tmp;
}
```

$ opt -S example.ll

```llvm
define dso_local i32 @square(i32) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  store i32 0, i32* %3, align 4
  store i32 1, i32* %5, align 4
  br label %6

; <label>:6:
  %7 = load i32, i32* %5, align 4
  %8 = load i32, i32* %2, align 4
  %9 = icmp ult i32 %7, %8
  br i1 %9, label %10, label %18

; <label>:10:
  %11 = load i32, i32* %2, align 4
  store i32 %11, i32* %4, align 4
  %12 = load i32, i32* %2, align 4
  %13 = load i32, i32* %3, align 4
  %14 = add i32 %13, %12
  store i32 %14, i32* %3, align 4
  br label %15

; <label>:15:
  %16 = load i32, i32* %5, align 4
  %17 = add i32 %16, 1
  store i32 %17, i32* %5, align 4
  br label %6

; <label>:18:
  %19 = load i32, i32* %3, align 4
  %20 = load i32, i32* %4, align 4
  %21 = add i32 %19, %20
  ret i32 %21
}
```

$ opt -S -O2 example.ll

```llvm
define dso_local i32 @square(i32) local_unnamed_addr
  %2 = icmp ugt i32 %0, 1
  %umax = select i1 %2, i32 %0, i32 1
  %3 = mul i32 %umax, %0
  ret i32 %3
}
```

Dead store elimination pass

example.c:

```
int fun()
{
  int a = 1;
  a = 2;
  return a;
}
```

$ opt -S example.ll

```
define dso_local i32 @fun() #0 {
  %1 = alloca i32, align 4
  store i32 1, i32* %1, align 4
  store i32 2, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  ret i32 %2
}
```

$ opt -S -dse example.ll

```
define dso_local i32 @fun() #0 {
  %1 = alloca i32, align 4
  store i32 2, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  ret i32 %2
}
```

Optimization passes – one by one

Source code

example.c:

```
unsigned square(unsigned x)
{
  unsigned sum = 0, tmp;
  for (unsigned i = 1; i < x; i++) {
    tmp = x;
    sum += x;
  }
  return sum + tmp;
}
```

Optimization passes – one by one

Simplify the CFG

```
; Function Attrs: noinline nounwind uwtable
define dso_local i32 @square(i32) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  store i32 0, i32* %3, align 4
  store i32 1, i32* %5, align 4
  br label %6

; <label>:6:                                        ; preds = %10, %1
  %7 = load i32, i32* %5, align 4
  %8 = load i32, i32* %2, align 4
  %9 = icmp ult i32 %7, %8
  br i1 %9, label %10, label %17

; <label>:10:                                       ; preds = %6
  %11 = load i32, i32* %2, align 4
  store i32 %11, i32* %4, align 4
  %12 = load i32, i32* %2, align 4
  %13 = load i32, i32* %3, align 4
  %14 = add i32 %13, %12
  store i32 %14, i32* %3, align 4
  %15 = load i32, i32* %5, align 4
  %16 = add i32 %15, 1
  store i32 %16, i32* %5, align 4
  br label %6

; <label>:17:                                       ; preds = %6
  %18 = load i32, i32* %3, align 4
  %19 = load i32, i32* %4, align 4
  %20 = add i32 %18, %19
  ret i32 %20
}
```

Optimization passes – one by one

SROA

```
; Function Attrs: noinline nounwind uwtable
define dso_local i32 @square(i32) #0 {
  br label %2

; <label>:2:                                        ; preds = %4, %1
  %.09 = phi i32 [ 0, %1 ], [ %5, %4 ]
  %.0 = phi i32 [ 1, %1 ], [ %6, %4 ]
  %3 = icmp ult i32 %.0, %0
  br i1 %3, label %4, label %7

; <label>:4:                                        ; preds = %2
  %5 = add i32 %.09, %0
  %6 = add i32 %.0, 1
  br label %2

; <label>:7:                                        ; preds = %2
  %8 = add i32 %.09, %0
  ret i32 %8
}
```

Optimization passes – one by one

Global Variable Optimizer

```
source_filename = "example.c"
  target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
  target triple = "x86_64-pc-linux-gnu"

  ; Function Attrs: noinline nounwind uwtable
  define dso_local i32 @square(i32) local_unnamed_addr #0 {
    br label %2

  ; <label>:2:                                      ; preds = %4, %1
    %.09 = phi i32 [ 0, %1 ], [ %5, %4 ]
    %.0 = phi i32 [ 1, %1 ], [ %6, %4 ]
    %3 = icmp ult i32 %.0, %0
    br i1 %3, label %4, label %7

  ; <label>:4:                                      ; preds = %2
    %5 = add i32 %.09, %0
    %6 = add i32 %.0, 1
    br label %2

  ; <label>:7:                                      ; preds = %2
    %8 = add i32 %.09, %0
    ret i32 %8
  }

  attributes #0 = { noinline nounwind uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="fal

  !llvm.module.flags = !{!0}
  !llvm.ident = !{!1}

  !0 = !{i32 1, !"wchar_size", i32 4}
  !1 = !{!"clang version 7.0.1-8 (tags/RELEASE_701/final)"}
```

Optimization passes – one by one

Simplify the CFG

```
; Function Attrs: noinline nounwind uwtable
define dso_local i32 @square(i32) local_unnamed_addr #0 {
  br label %2

; <label>:2:                                        ; preds = %5, %1
  %.09 = phi i32 [ 0, %1 ], [ %4, %5 ]
  %.0 = phi i32 [ 1, %1 ], [ %6, %5 ]
  %3 = icmp ult i32 %.0, %0
  %4 = add i32 %.09, %0
  br i1 %3, label %5, label %7

; <label>:5:                                        ; preds = %2
  %6 = add i32 %.0, 1
  br label %2

; <label>:7:                                        ; preds = %2
  ret i32 %4
}
```

Optimization passes – one by one

Deduce function attributes

```
; Function Attrs: noinline norecurse nounwind readnone uwtable
define dso_local i32 @square(i32) local_unnamed_addr #0 {
  br label %2

; <label>:2:                                        ; preds = %5, %1
  %.09 = phi i32 [ 0, %1 ], [ %4, %5 ]
  %.0 = phi i32 [ 1, %1 ], [ %6, %5 ]
  %3 = icmp ult i32 %.0, %0
  %4 = add i32 %.09, %0
  br i1 %3, label %5, label %7

; <label>:5:                                        ; preds = %2
  %6 = add i32 %.0, 1
  br label %2

; <label>:7:                                        ; preds = %2
  ret i32 %4
}
```

Optimization passes – one by one

Loop-Closed SSA Form Pass

```
; Function Attrs: noinline norecurse nounwind readnone uwtable
define dso_local i32 @square(i32) local_unnamed_addr #0 {
  br label %2

; <label>:2:                                        ; preds = %5, %1
  %.09 = phi i32 [ 0, %1 ], [ %4, %5 ]
  %.0 = phi i32 [ 1, %1 ], [ %6, %5 ]
  %3 = icmp ult i32 %.0, %0
  %4 = add i32 %.09, %0
  br i1 %3, label %5, label %7

; <label>:5:                                        ; preds = %2
  %6 = add i32 %.0, 1
  br label %2

; <label>:7:                                        ; preds = %2
  %.lcssa = phi i32 [ %4, %2 ]
  ret i32 %.lcssa
}
```

Optimization passes – one by one

Rotate Loops

```
; Preheader:
    br label %2

  ; Loop:
  ; <label>:2:                                        ; preds = %2, %1
    %.09 = phi i32 [ 0, %1 ], [ %4, %2 ]
    %.0 = phi i32 [ 1, %1 ], [ %5, %2 ]
    %3 = icmp ult i32 %.0, %0
    %4 = add i32 %.09, %0
    %5 = add i32 %.0, 1
    br i1 %3, label %2, label %6

  ; Exit blocks
  ; <label>:6:                                        ; preds = %2
    %.lcssa = phi i32 [ %4, %2 ]
    ret i32 %.lcssa
```

Optimization passes – one by one

Combine redundant instructions

```
; Function Attrs: noinline norecurse nounwind readnone uwtable
define dso_local i32 @square(i32) local_unnamed_addr #0 {
  br label %2

; <label>:2:                                        ; preds = %2, %1
  %.09 = phi i32 [ 0, %1 ], [ %4, %2 ]
  %.0 = phi i32 [ 1, %1 ], [ %5, %2 ]
  %3 = icmp ult i32 %.0, %0
  %4 = add i32 %.09, %0
  %5 = add i32 %.0, 1
  br i1 %3, label %2, label %6

; <label>:6:                                        ; preds = %2
  ret i32 %4
}
```

Optimization passes – one by one

Loop-Closed SSA Form Pass

```
; Function Attrs: noinline norecurse nounwind readnone uwtable
define dso_local i32 @square(i32) local_unnamed_addr #0 {
  br label %2

; <label>:2:                                        ; preds = %2, %1
  %.09 = phi i32 [ 0, %1 ], [ %4, %2 ]
  %.0 = phi i32 [ 1, %1 ], [ %5, %2 ]
  %3 = icmp ult i32 %.0, %0
  %4 = add i32 %.09, %0
  %5 = add i32 %.0, 1
  br i1 %3, label %2, label %6

; <label>:6:                                        ; preds = %2
  %.lcssa = phi i32 [ %4, %2 ]
  ret i32 %.lcssa
}
```

Optimization passes – one by one

Induction Variable Simplification

```
; Preheader:
    %2 = icmp ugt i32 %0, 1
    %umax = select i1 %2, i32 %0, i32 1
    br label %3

  ; Loop:
  ; <label>:3:                                        ; preds = %3, %1
    %.0 = phi i32 [ 1, %1 ], [ %5, %3 ]
    %4 = icmp ult i32 %.0, %0
    %5 = add i32 %.0, 1
    br i1 %4, label %3, label %6

  ; Exit blocks
  ; <label>:6:                                        ; preds = %3
    %7 = mul i32 %0, %umax
    ret i32 %7
```

Optimization passes – one by one

Global Value Numbering

```
; Function Attrs: noinline norecurse nounwind readnone uwtable
define dso_local i32 @square(i32) local_unnamed_addr #0 {
  %2 = icmp ugt i32 %0, 1
  %umax = select i1 %2, i32 %0, i32 1
  %3 = mul i32 %0, %umax
  ret i32 %3
}
```

Optimization passes – one by one

Combine redundant instructions

```
; Function Attrs: noinline norecurse nounwind readnone uwtable
define dso_local i32 @square(i32) local_unnamed_addr #0 {
  %2 = icmp ugt i32 %0, 1
  %umax = select i1 %2, i32 %0, i32 1
  %3 = mul i32 %umax, %0
  ret i32 %3
}
```

Low-level optimizations

Related to a particular hardware

- Instruction Selection
- Expand ISel Pseudo-instructions
- Tail Duplication
- Optimize machine instruction PHIs
- Merge disjoint stack slots
- Local Stack Slot Allocation
- Remove dead machine instructions
- Early If-Conversion
- Machine InstCombiner
- Machine Loop Invariant Code Motion
- Machine Common Subexpression Elimination
- Machine code sinking
- Peephole Optimizations
- Remove dead machine instructions
- X86 LEA Optimize
- X86 Optimize Call Frame
- Process Implicit Definitions
- Live Variable Analysis
- Machine Natural Loop Construction
- Eliminate PHI nodes for register allocation
- Two-Address instruction pass
- Simple Register Coalescing
- Machine Instruction Scheduler
- Greedy Register Allocator
- Virtual Register Rewriter
- Stack Slot Coloring
- Machine Loop Invariant Code Motion
- X86 FP Stackifier
- Shrink Wrapping analysis
- Prologue/Epilogue Insertion & Frame Finalization
- Control Flow Optimizer
- Tail Duplication
- Machine Copy Propagation Pass
- Post-RA pseudo instruction expansion pass
- X86 pseudo instruction expansion pass
- Post RA top-down list latency scheduler
- Analyze Machine Code For Garbage Collection
- Branch Probability Basic Block Placement
- Execution dependency fix
- X86 vzeroupper inserter
- X86 Atom pad short functions
- X86 LEA Fixup
- Contiguously Lay Out Funclets
- StackMap Liveness Analysis
- Live DEBUG_VALUE analysis

Low-level optimization passes

Source code

example.c:

```
unsigned square(unsigned x)
{
  return x*x;
}
```

Low-level optimization passes

After Instruction Selection:

```
Frame Objects:
  fi#0: size=4, align=4, at location [SP+8]
Function Live Ins: $edi in %0

bb.0 (%ir-block.1):
  liveins: $edi
  %0:gr32 = COPY $edi
  %1:gr32 = COPY killed %0:gr32
  MOV32mr %stack.0, 1, $noreg, 0, $noreg, %1:gr32 :: (store 4 into %ir.2)
  %6:gr32 = MOV32rm %stack.0, 1, $noreg, 0, $noreg :: (load 4 from %ir.2)
  %5:gr32 = IMUL32rm killed %6:gr32, %stack.0, 1, $noreg, 0, $noreg, implicit-def $eflags :: (load 4 from %ir.2
  $eax = COPY %5:gr32
  RETQ implicit $eax
```

Low-level optimization passes

After Live Variable Analysis:

```
Frame Objects:
  fi#0: size=4, align=4, at location [SP+8]
Function Live Ins: $edi in %0

bb.0 (%ir-block.1):
  liveins: $edi
  %0:gr32 = COPY killed $edi
  %1:gr32 = COPY killed %0:gr32
  MOV32mr %stack.0, 1, $noreg, 0, $noreg, killed %1:gr32 :: (store 4 into %ir.2)
  %6:gr32 = MOV32rm %stack.0, 1, $noreg, 0, $noreg :: (load 4 from %ir.2)
  %5:gr32 = IMUL32rm killed %6:gr32, %stack.0, 1, $noreg, 0, $noreg, implicit-def dead $eflags :: (load 4 from
  $eax = COPY killed %5:gr32
  RETQ implicit killed $eax
```

Low-level optimization passes

After Two-Address instruction pass:

```
Frame Objects:
  fi#0: size=4, align=4, at location [SP+8]
Function Live Ins: $edi in %0

bb.0 (%ir-block.1):
  liveins: $edi
  %0:gr32 = COPY killed $edi
  %1:gr32 = COPY killed %0:gr32
  MOV32mr %stack.0, 1, $noreg, 0, $noreg, killed %1:gr32 :: (store 4 into %ir.2)
  %6:gr32 = MOV32rm %stack.0, 1, $noreg, 0, $noreg :: (load 4 from %ir.2)
  %5:gr32 = COPY killed %6:gr32
  %5:gr32 = IMUL32rm %5:gr32, %stack.0, 1, $noreg, 0, $noreg, implicit-def dead $eflags :: (load 4 from %ir.2)
  $eax = COPY killed %5:gr32
  RETQ implicit killed $eax
```

Low-level optimization passes

After Simple Register Coalescing:

```
Frame Objects:
  fi#0: size=4, align=4, at location [SP+8]
Function Live Ins: $edi in %0

0B        bb.0 (%ir-block.1):
          liveins: $edi
16B          %1:gr32 = COPY $edi
48B          MOV32mr %stack.0, 1, $noreg, 0, $noreg, %1:gr32 :: (store 4 into %ir.2)
64B          %5:gr32 = MOV32rm %stack.0, 1, $noreg, 0, $noreg :: (load 4 from %ir.2)
96B          %5:gr32 = IMUL32rm %5:gr32, %stack.0, 1, $noreg, 0, $noreg, implicit-def dead $eflags :: (load 4 f
112B          $eax = COPY %5:gr32
128B          RETQ implicit killed $eax
```

Low-level optimization passes

After Greedy Register Allocator:

```
Frame Objects:
  fi#0: size=4, align=4, at location [SP+8]
Function Live Ins: $edi in %0

0B        bb.0 (%ir-block.1):
          liveins: $edi
16B          %1:gr32 = COPY $edi
48B          MOV32mr %stack.0, 1, $noreg, 0, $noreg, %1:gr32 :: (store 4 into %ir.2)
64B          %5:gr32 = MOV32rm %stack.0, 1, $noreg, 0, $noreg :: (load 4 from %ir.2)
96B          %5:gr32 = IMUL32rm %5:gr32, %stack.0, 1, $noreg, 0, $noreg, implicit-def dead $eflags :: (load 4 f
112B          $eax = COPY %5:gr32
128B          RETQ implicit $eax
```

Low-level optimization passes

After Virtual Register Rewriter:

```
Frame Objects:
  fi#0: size=4, align=4, at location [SP+8]
Function Live Ins: $edi

0B        bb.0 (%ir-block.1):
          liveins: $edi
48B          MOV32mr %stack.0, 1, $noreg, 0, $noreg, killed renamable $edi :: (store 4 into %ir.2)
64B          renamable $eax = MOV32rm %stack.0, 1, $noreg, 0, $noreg :: (load 4 from %ir.2)
96B          renamable $eax = IMUL32rm killed renamable $eax, %stack.0, 1, $noreg, 0, $noreg, implicit-def dead
128B          RETQ implicit $eax
```

Low-level optimization passes

After Stack Slot Coloring:

```
Frame Objects:
  fi#0: size=4, align=4, at location [SP+8]
Function Live Ins: $edi

bb.0 (%ir-block.1):
  liveins: $edi
  MOV32mr %stack.0, 1, $noreg, 0, $noreg, killed renamable $edi :: (store 4 into %ir.2)
  renamable $eax = MOV32rm %stack.0, 1, $noreg, 0, $noreg :: (load 4 from %ir.2)
  renamable $eax = IMUL32rm killed renamable $eax, %stack.0, 1, $noreg, 0, $noreg, implicit-def dead $eflags ::
  RETQ implicit $eax
```

Low-level optimization passes

After Prologue/Epilogue Insertion & Frame Finalization:

```
Frame Objects:
  fi#-1: size=8, align=16, fixed, at location [SP-8]
  fi#0: size=4, align=4, at location [SP-12]
Function Live Ins: $edi

bb.0 (%ir-block.1):
  liveins: $edi
  frame-setup PUSH64r killed $rbp, implicit-def $rsp, implicit $rsp
  CFI_INSTRUCTION def_cfa_offset 16
  CFI_INSTRUCTION offset $rbp, -16
  $rbp = frame-setup MOV64rr $rsp
  CFI_INSTRUCTION def_cfa_register $rbp
  MOV32mr $rbp, 1, $noreg, -4, $noreg, killed renamable $edi :: (store 4 into %ir.2)
  renamable $eax = MOV32rm $rbp, 1, $noreg, -4, $noreg :: (load 4 from %ir.2)
  renamable $eax = IMUL32rm killed renamable $eax, $rbp, 1, $noreg, -4, $noreg, implicit-def dead $eflags :: (l
  $rbp = frame-destroy POP64r implicit-def $rsp, implicit $rsp
  CFI_INSTRUCTION def_cfa $rsp, 8
  RETQ implicit $eax
```

Outline

1 Motivating example

2 C/C++ compiler
- Frontend
- Optimization passes
  - High-level optimizations
  - High-level optimizations – Example
  - Low-level optimizations
  - Low-level optimizations – Example
- Miscellaneous

3 Linker

4 Execution

Profile-guided optimization

1. Compile your application with `-fprofile-generate`
2. Run tests of your application, gather profiling data
3. Recompile with `-fprofile-use`

Volatile keyword in C

```
volatile int x;
```

- It tells the compiler not to optimize the access to the variable.
  - When the variable appears in the source code, load or store instruction appears in the machine code.
- In C, volatile is much weaker than in Java, where it generates barrier and results in non-cached access.

Outline

1 Motivating example

2 C/C++ compiler
- Frontend
- Optimization passes
  - High-level optimizations
  - High-level optimizations – Example
  - Low-level optimizations
  - Low-level optimizations – Example
- Miscellaneous

3 Linker

4 Execution

Linker

- Combines multiple modules (object files) together
- Resolves references to symbols from other modules
- Can also perform some optimizations

Basics of working with libraries

```
$ gcc -o file1.o file1.c
$ gcc -o file2.o file2.c
$ ar rvs libmyfiles.a file1.o file2.o # create static library

$ gcc -o myprog.o myprog.c
$ ld -o myprog myprog.o -lmyfiles

$ gcc -o myprog myprog.c -lmyfiles # shortcut
```

Resolving references

```c
extern int var; // variable in another .c file
int func();     // function in another .c file
// The above is usually contained in a header file
int foo()
{
  return func() + var;
}
```

- Linker works by reading relocation records stored in the object files
  - Location within the binary section
  - Format (type) of the value
  - Value of what
- Example below:
  - Put the address of `func` in PLT32 format at address 0xA in extern.o.
  - Put the address `var` in PC32 format (relative to program counter) at address 0x12 in extern.o.

$ objdump -r extern.o

```
extern.o:     file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
000000000000000a R_X86_64_PLT32    func-0x0000000000000004
0000000000000012 R_X86_64_PC32     var-0x0000000000000004
```

Linker-related optimizations

- Linker's work is driven by a "linker script"
  - By modifying the linker script, you can, for example, reorder functions, e.g. put hot functions together to avoid cache self eviction
  - Default linker scripts already contain this:

    ```
    int hot_function(...) __attribute__((hot));
    ```
- Can perform "Link-time optimization"
  - Unused function removal:

    ```
    gcc -ffunction-sections ...
    ld --gc-sections ...
    ```
  - Function inlining
  - Interprocedural constant propagation
  - ...

Outline

1 Motivating example

2 C/C++ compiler
- Frontend
- Optimization passes
  - High-level optimizations
  - High-level optimizations – Example
  - Low-level optimizations
  - Low-level optimizations – Example
- Miscellaneous

3 Linker

4 Execution

Starting of a binary program (Linux)

1. OS kernel loads binary header(s)
2. For statically linked binaries:
   - sets virtual memory data structures up and jumps to the program entry point
3. For dynamically linked binaries (those who require shared libraries):
   - Reads the name of program interpreter (e.g. /lib64/ld-linux-x86-64.so.2)
   - Loads the interpreter binary
   - Execute the interpreter with binary name as a parameter
     - This allows things like transparently running ARM binaries on x86 via Qemu emulator

Binary interpreter and dynamic linking

- Interpreter's task is to perform dynamic linking
- Similar to static linking (it uses relocation table), but at runtime
- Linking big libraries with huge amount of symbols (e.g. Qt) is slow
  - Lazy linking
  - Not good for real-time applications

Program execution and memory management

Summary: things are done lazily if possible

- Executed binary is not loaded into memory at the beginning
  - Loading is done lazily as a response to page faults
  - Only those parts of the binary, that are actually “touched” are loaded
  - Other things (e.g. debug information, unused data and code) stay on disk
- Memory allocation is also lazy
  - When an app asks OS for memory, only VM data is set up
  - Only when the memory is touched, it is actually allocated and mapped to the proper place
  - Allows you to allocate more memory than you physically have
- Memory allocations
  - Two levels: OS level and application level
  - Application asks OS for chunks of memory (via brk() or mmap())
  - Application manages this memory as heap (malloc(), new())

References

- John Regehr: How Clang Compiles a Function
  `https://blog.regehr.org/archives/1605`
- John Regehr: How LLVM Optimizes a Function
  `https://blog.regehr.org/archives/1603`

B4M36ESW: Efficient software

Lecture 5: Scalable synchronization

Michal Sojka
michal.sojka@cvut.cz

March 19, 2019

Synchronization

- Multi-core CPUs are today's norm, many-core CPUs will come tomorrow
- To take advantage of such a hardware, parallel (multi-threaded) programs must be run on them
- It is not useful when the threads are completely independent, i.e. threads have to communicate (synchronize)
- Basic forms of synchronization:
  - Mutual exclusion (e.g. access to shared data)
  - Producer-consumer (e.g. database waits for requests)
  - ...

Outline

1 Naive synchronization
- Problems
  - Non-atomic data manipulation
  - Unwanted compiler optimizations
  - Reordering at hardware level
  - Deadlock

2 Cost of atomic operations and barriers

3 Kernel semaphores

4 Futex

5 Read-mostly workload

6 Read-Copy-Update (RCU)
- RCU implementations

Naive synchronization

Mutual exclusion

Data should be modified at most by one thread at a time:

```
bool locked;

void func() {
  while (locked == true)
    /* busy wait */;
  locked = true;
  data++;
  locked = false;
}
```

Terminology: code in the "locked" region is called *critical section*

Problems:

1. Checking and setting the lock is not atomic
2. Compiler can optimize out all accesses to *locked*
3. Compiler can move access to data out of critical section
4. Hardware can reorder memory accesses even if compiler does not
5. Can easily deadlock
6. Busy waiting wastes energy

Outline

1 Naive synchronization
- Problems
  - Non-atomic data manipulation
  - Unwanted compiler optimizations
  - Reordering at hardware level
  - Deadlock

2 Cost of atomic operations and barriers

3 Kernel semaphores

4 Futex

5 Read-mostly workload

6 Read-Copy-Update (RCU)
- RCU implementations

Atomic operations

- Example of non-atomic increment:
  - C expression: `data++;`
  - Assembler (x86): `inc ($data)` – atomic (uninterruptible) on a single CPU
  - Hardware (as seen from other CPUs): memory bus read, ALU, memory bus write

CPU0	CPU1	data
bus read		0
ALU	bus read	0
bus write	ALU	1
	bus write	1

- Atomic operations ensure that the operation (typically read-modify-write) is atomic (uninterruptible) even at the hardware (bus) level.
  - compare-and-swap/CAS instruction (x86: cmpxchg)
  - In C: `TYPE __atomic_exchange_n(TYPE *ptr, TYPE val, int memorder)` Atomically set `*ptr` to `val` and return old value of `*ptr`.

```
void lock() {                          void lock() {
  while (locked == true)                 while (__atomic_exchange_n(&locked, true,
    /* busy wait */;          ⇒                                   ...) == true)
    locked = true;                         /* busy wait */;
}                                      }
```

Atomic operations in C and C++

- For long time, atomic operations were not standardized in C/C++
  - Solution: Incompatible compiler extensions, inline assembler
- C11, C++11 introduced thread-aware memory model and defined platform independent atomic operations
- C11: stdatomic.h, atomic_* functions
- C++11
  - `std::atomic` template
  - `std::atomic<int> x;`
    `x++; // atomic increment`

Compiler optimizations

```
bool locked;

while (locked)
{}
locked = true;
data++;
locked = false;
```

⇒

```
#define barrier() \
    asm volatile("" : : : "memory"
volatile bool locked;

while (locked)
{}
locked = true;
barrier();
data++;
barrier();
locked = false;
```

- Compiler expects the memory is only modified by the program being compiled
- Locked seems to be useless ⇒ optimize out
- Compiler is free to reorder operations as long as the result of the computation is the same

Compiler optimizations cont.

- Defining the variable volatile makes all accesses “volatile” i.e. slow.
- Sometimes, we need only certain accesses to have volatile semantics and the rest can be optimized:

```
#define ACCESS_ONCE(x) (*(volatile typeof(x) *)&(x))
#define LOAD_SHARED(p) ACCESS_ONCE(p)
#define STORE_SHARED(x, v) ({ ACCESS_ONCE(x) = (v); })

#define barrier() asm volatile("" : : : "memory")
```

- The macro `barrier` is only a **compiler barrier**, not hardware barrier, i.e., the compiler will not reorder generated instructions.

Hardware reordering

- Different CPU architectures implement different memory consistency models
- Some operations can be reordered with respect to other operations

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	z/Architecture
Loads → loads	Y	Y	Y	Y	Y	N	N	N	Y	N	Y	N
Loads → stores	Y	Y	Y	Y	Y	N	N	N	Y	N	Y	N
Stores → stores	Y	Y	Y	Y	Y	Y	N	N	Y	N	Y	N
Stores → loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic → loads	Y	Y	N	Y	Y	N	N	N	N	N	Y	N
Atomic → stores	Y	Y	N	Y	Y	Y	N	N	N	N	Y	N
Dependent loads	Y	N	N	N	N	N	N	N	N	N	N	N
Incoherent inst. cache pipeline	Y	Y	N	Y	Y	Y	Y	Y	Y	N	Y	

Source: Wikipedia

- x86 can reorder stores after loads, i.e. `data` can be read before other CPUs see `locked` set to true!
- Why? Stores may have to wait for cache-line ownership. Not waiting with subsequent reads improves **performance**.
- **Solution:** Insert memory barrier instructions.
  - e.g. mfence, lfence on x86

Specifying memory ordering requirements in C/C++

```
std::atomic<int> x;
x.load(order);
w.store(0, order);
```

- `order` specifies how regular, non-atomic memory accesses are to be ordered around an atomic operation
  - relaxed: no overhead, no order guarantee
  - consume
  - acquire
  - release
  - acq_rel,
  - seq_cst: high overhead, sequential consistency
- Depending on the CPU architecture, different `order`s cause the compiler to generate barrier instructions (e.g., lfence on x86)

Deadlock

- Example:
  - Single-core system
  - Two threads low- and high-priority

```
LP_thread     HP_thread
~~~~~~~~~     ~~~~~~~~~
lock();
data++;
   →  preemption →
                 deadlock();
```

- **Solution:** When the lock is not available, ask the OS scheduler to put your thread to sleep and wake you up after the lock is available
  - Problem: atomicity of checking the lock and going to sleep
  - Requires implementation in the OS kernel

Outline

1 Naive synchronization
- Problems
  - Non-atomic data manipulation
  - Unwanted compiler optimizations
  - Reordering at hardware level
  - Deadlock

2 Cost of atomic operations and barriers

3 Kernel semaphores

4 Futex

5 Read-mostly workload

6 Read-Copy-Update (RCU)
- RCU implementations

Cost of atomic operations & barriers

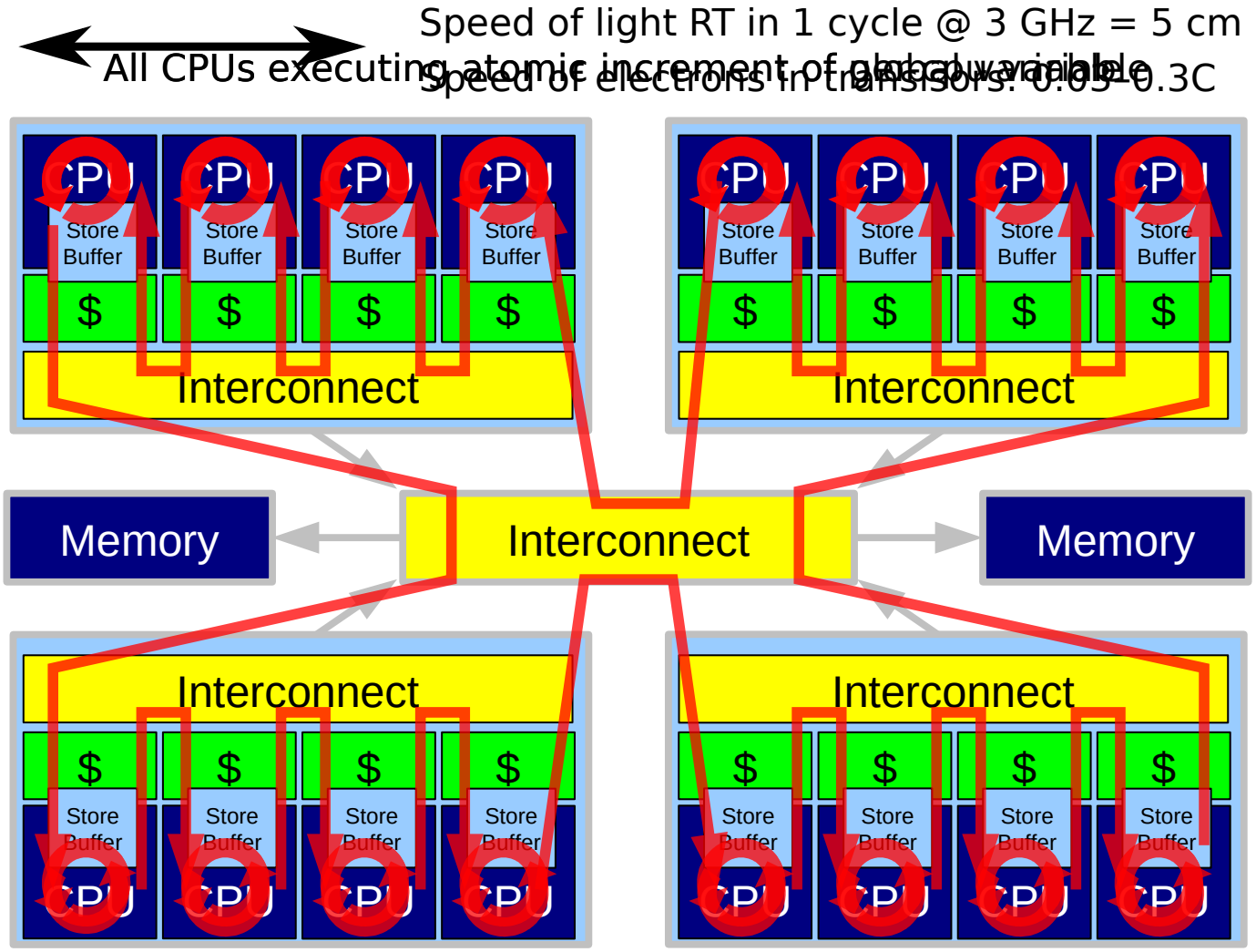
16-CPU 2.8GHz Intel X5550 (Nehalem) System

Operation	Cost (ns)	Ratio
Clock period	0.4	1.0
“Best-case” CAS	12.2	33.8
Best-case lock	25.6	71.2
Single cache miss	12.9	35.8
CAS cache miss	7.0	19.4
Single cache miss (off-core)	31.2	86.6
CAS cache miss (off-core)	31.2	86.5
Single cache miss (off-socket)	92.4	256.7
CAS cache miss (off-socket)	95.9	266.4

Source: Paul E. McKenney, IBM

- Atomic operations are costly (here 19–266 times slower than non-atomic operations)
- Barriers are typically cheaper (weak barriers more that full barriers)

Cost of atomic operations & laws of physics



Every CPU experiences a cache miss, because other CPUs access the variable as well No cache miss ⇒ much faster

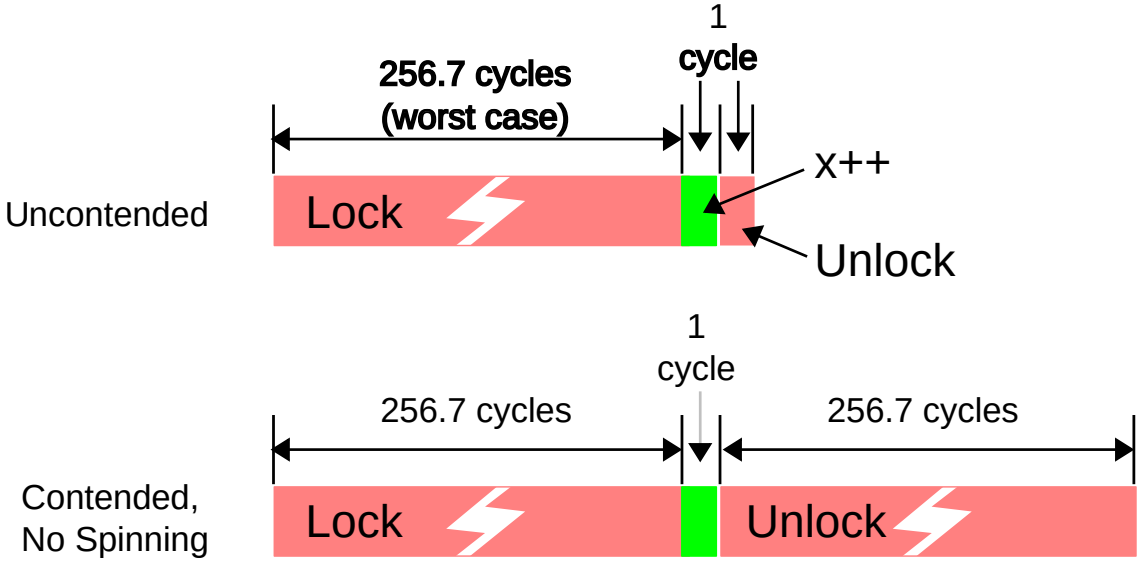
Locking overhead

- Classical locks are typically implemented with atomic instructions and ensure that lock manipulation is not reordered with critical section content.

```
pthread_mutex_lock(mutex);
x++;
pthread_mutex_unlock(mutex);
```

- Uncontended case: during lock(), mutex is not in the cache, during unlock() it is
- Contended case: mutex is not in the cache even during unlock, because there is (probably) another CPU trying to lock the mutex and thus “stealing” the lock from mutex-owner’s cache

Single-instruction critical sections protected by a lock



Outline

1 Naive synchronization
- Problems
  - Non-atomic data manipulation
  - Unwanted compiler optimizations
  - Reordering at hardware level
  - Deadlock

2 Cost of atomic operations and barriers

3 Kernel semaphores

4 Futex

5 Read-mostly workload

6 Read-Copy-Update (RCU)
- RCU implementations

Kernel semaphores

- Each system call adds overhead ($\approx$ 100 cycles on modern HW)
- It is preferable to use “fine-grain” locking, i.e. locks protect as little data as possible to prevent lock contention.
- If fine-grain locking is effective the lock is not contended and threads rarely have to sleep, but always pay the syscall overhead!
- That’s not efficient – the solution in Linux is called **futex**.

Outline

1 Naive synchronization
- Problems
  - Non-atomic data manipulation
  - Unwanted compiler optimizations
  - Reordering at hardware level
  - Deadlock

2 Cost of atomic operations and barriers

3 Kernel semaphores

4 Futex

5 Read-mostly workload

6 Read-Copy-Update (RCU)
- RCU implementations

Futex

Fast Userspace Mutex

- Uncontended mutex never goes to kernel
- It uses atomic instruction `cmpxchg(val, expct, new) → prev`
- `futex_wait()` and `futex_wake()` are system calls that are called only in case of contention

```
class mutex {
public:
  mutex() : val (0) { }

  void lock() {
    int c;
    if ((c = cmpxchg (val, 0, 1)) != 0) {
      if (c != 2)
        c = xchg (val, 2);
      while (c != 0) {
        futex_wait (&val, 2);
        c = xchg (val, 2);
      }
    }
  }
  void unlock() {
    if (atomic_dec (val) != 1) {
      val = 0;
      futex_wake (&val, 1);
    }
  }
private:
  int val;
};
```

U. Drepper, *Futexes Are Tricky*, 2011,
Online: `https://www.akkadia.org/drepper/futex.pdf`

Futex uses

- Futex primitive can be used to implement the following higher-level synchronization mechanisms:
  - Mutexes
  - Semaphores
  - Conditional variables
  - Thread barriers
  - Read-write locks
- Today, `pthread_mutex_t` in Linux is implemented via futex.
- JVM's synchronization via Mark word and thin/fat locking is conceptually the same as futex (see next lecture)

Outline

1 Naive synchronization
- Problems
  - Non-atomic data manipulation
  - Unwanted compiler optimizations
  - Reordering at hardware level
  - Deadlock

2 Cost of atomic operations and barriers

3 Kernel semaphores

4 Futex

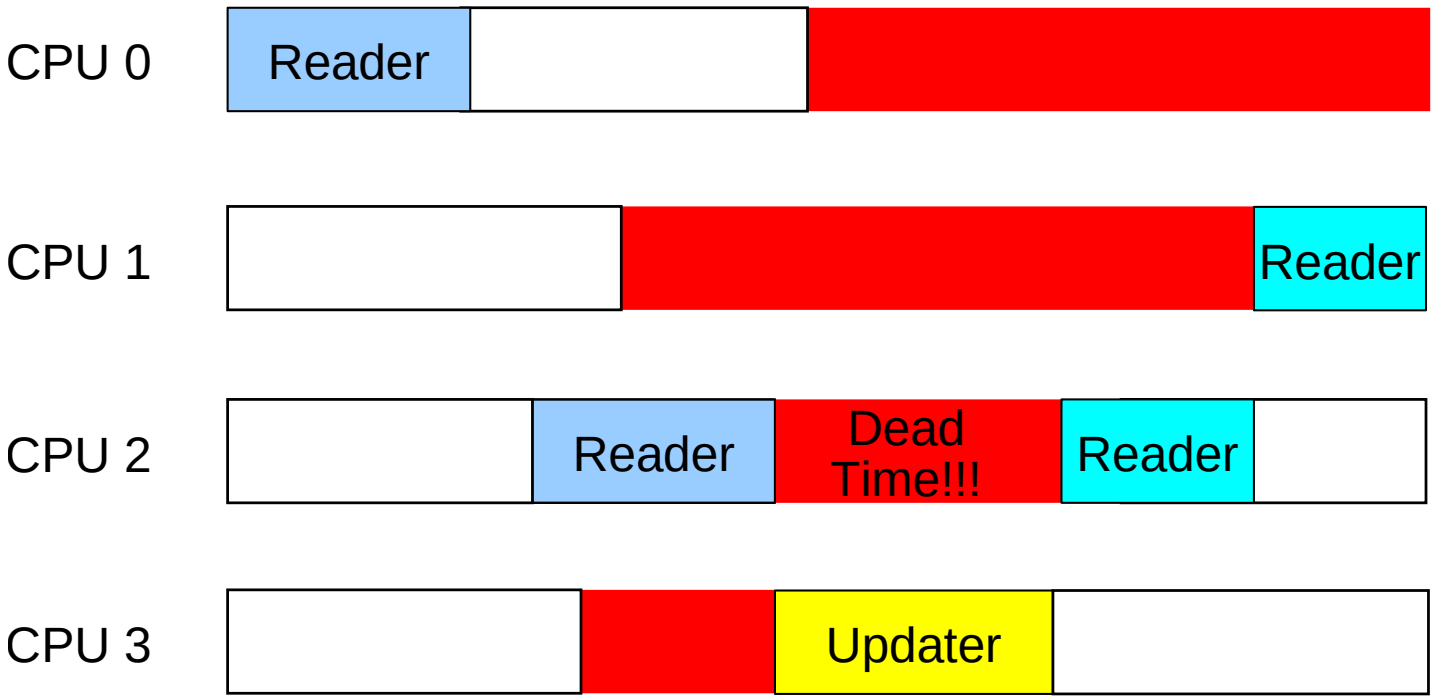
5 Read-mostly workload

6 Read-Copy-Update (RCU)
- RCU implementations

The problem of mutex

Mutual exclusion in massively parallel **read-mostly workload**

1. Lock/unlock overhead
2. Dead time during updates

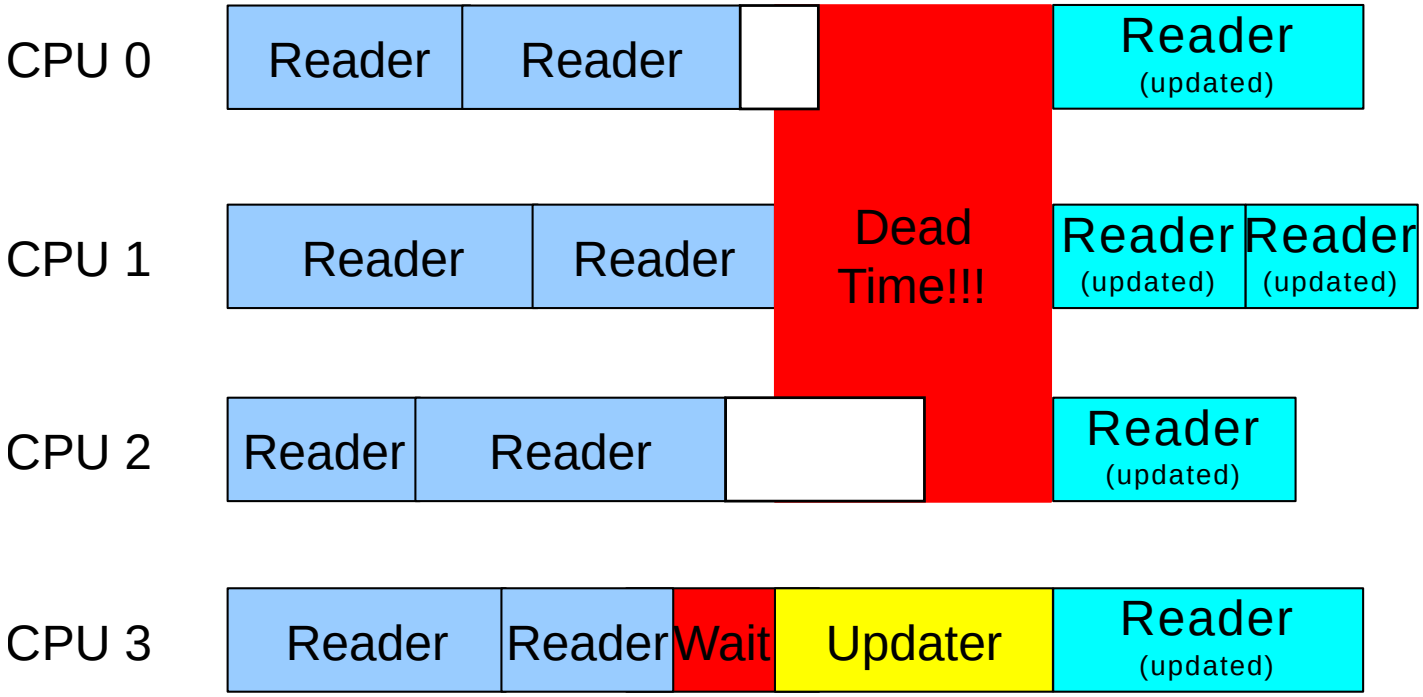


Read-mostly workload

- Many real-world workloads are “read-mostly”
- Caching – Most users only read the data. The cache is updated only from time to time (new article is published, etc.)
  - Web site cache (template rendering)
  - Content delivery network (CDN)
  - File system access
  - Domain Name System (DNS)
  - ...
- Maintaining consistency while updating – synchronization between readers and updaters (writers)

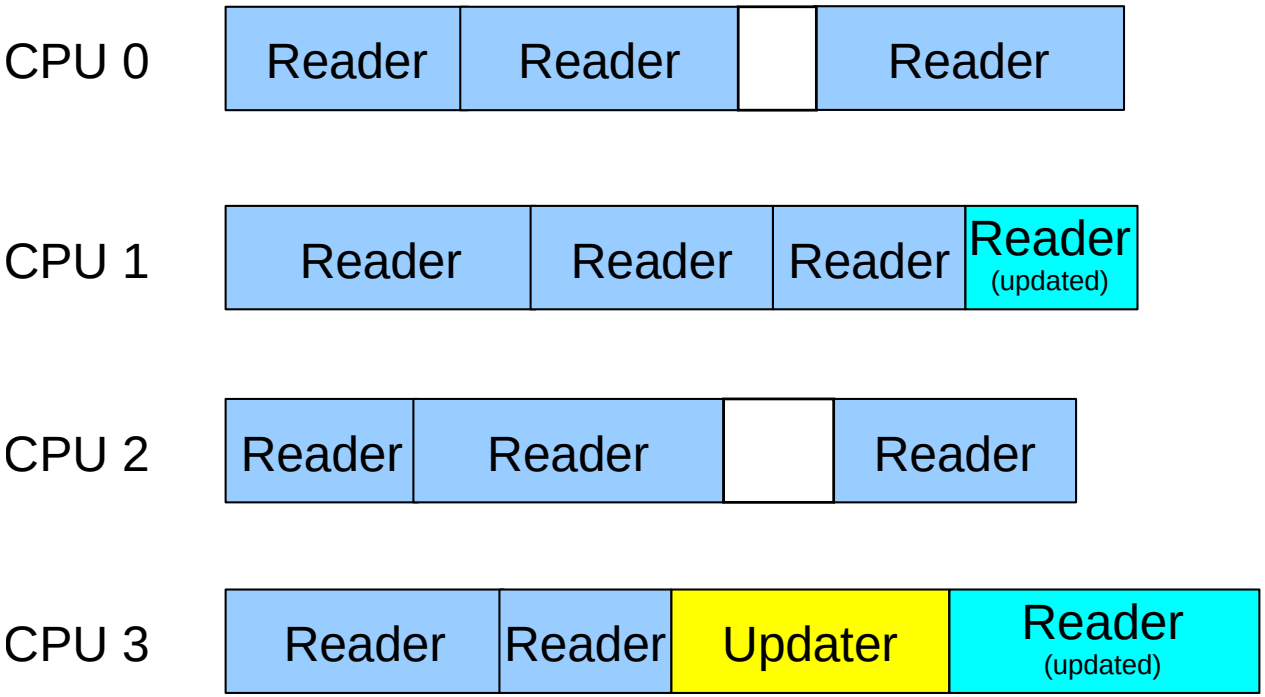
Read-Write lock

- Classical solution – read-write lock
- Multiple readers can read simultaneously
- Update blocks all readers



- Can be implemented on top of mutex(es)
- Scales badly

We want this



- Updater does not block readers
- Is that possible? Yes
- For what cost? See next slides

Outline

1 Naive synchronization
- Problems
  - Non-atomic data manipulation
  - Unwanted compiler optimizations
  - Reordering at hardware level
  - Deadlock

2 Cost of atomic operations and barriers

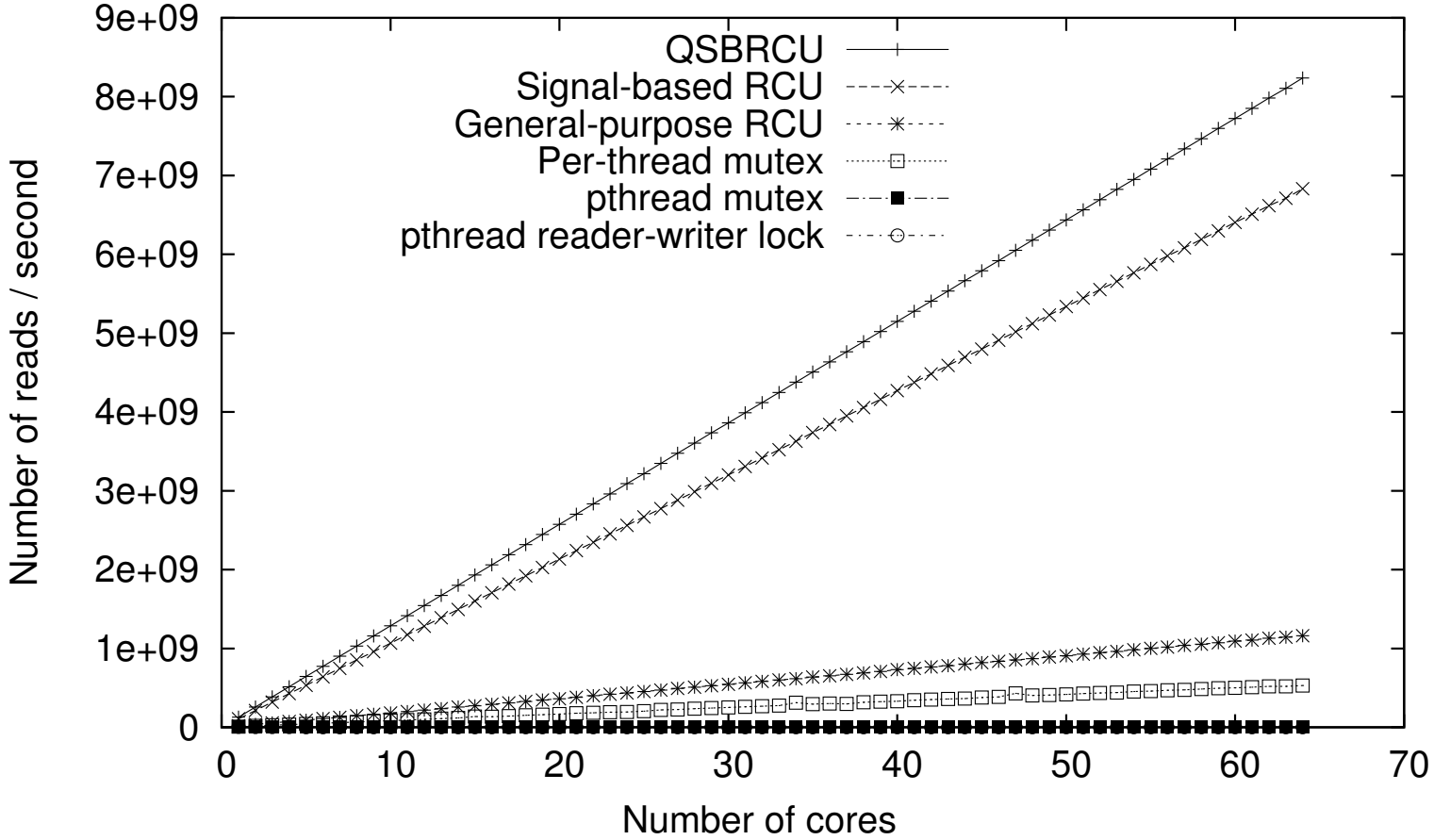
3 Kernel semaphores

4 Futex

5 Read-mostly workload

6 Read-Copy-Update (RCU)
- RCU implementations

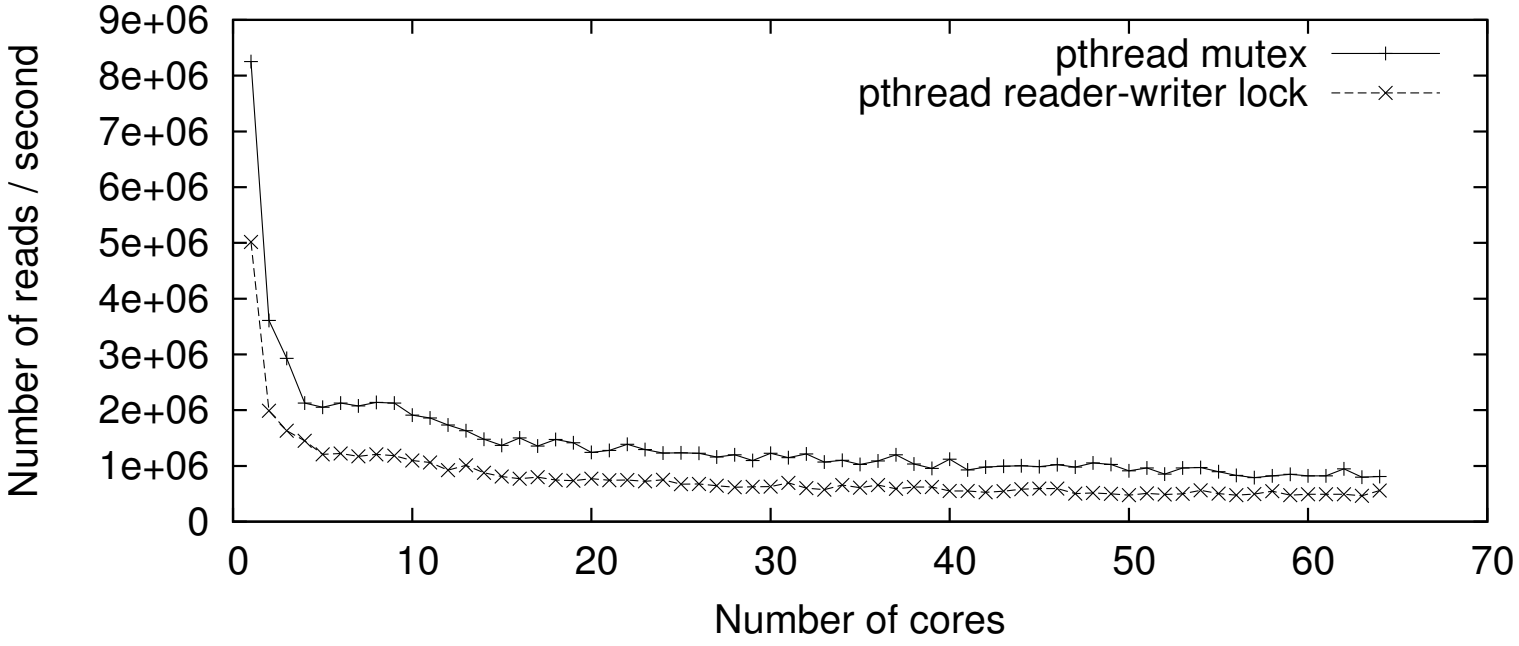
Read-side scalability of synchronization primitives



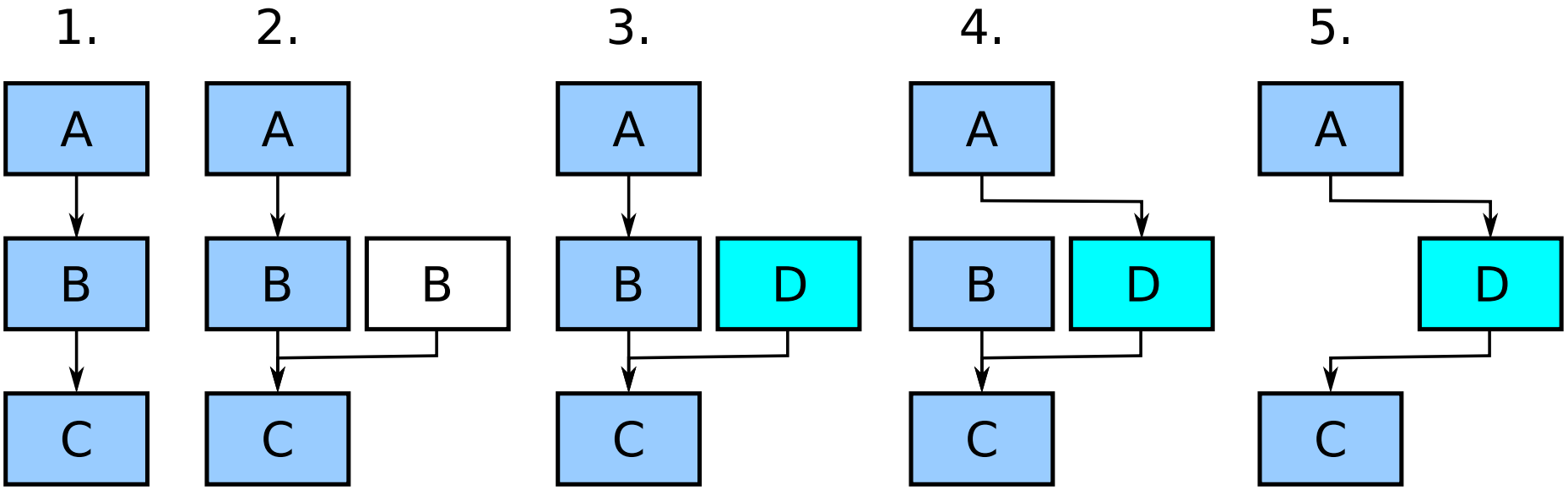
- RCU is scalable – typically it scales almost linearly up to hundreds or thousands of CPUs
- Locking does not scale (see next slide)

Locking scalability

- Zoomed in version of the previous graph



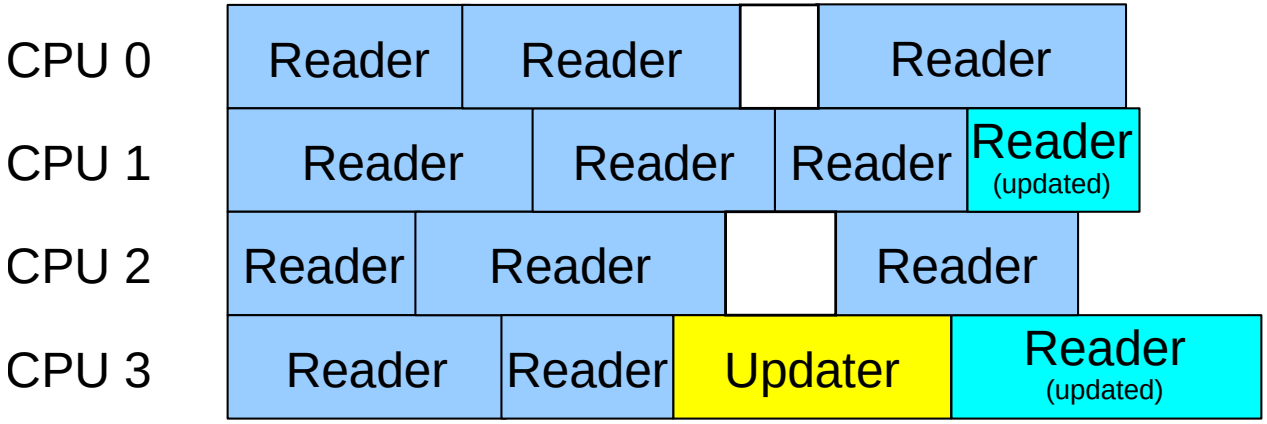
Updating an RCU-protected list



1. Original list
2. Copy B
3. Update B to D
4. Make the updated element visible to readers
5. Wait after all readers stop accessing B and free it

Steps 1–4 are trivial, making step 5 efficient is why RCU is needed!

How does RCU know when no reader access old data?



- **No explicit (and expensive) tracking** of each reader (e.g. no reference counting, no locking in readers)
- RCU uses **indirect** way of determining the end of all read-side sections
- In certain implementations (QSBR) read-side has zero overhead

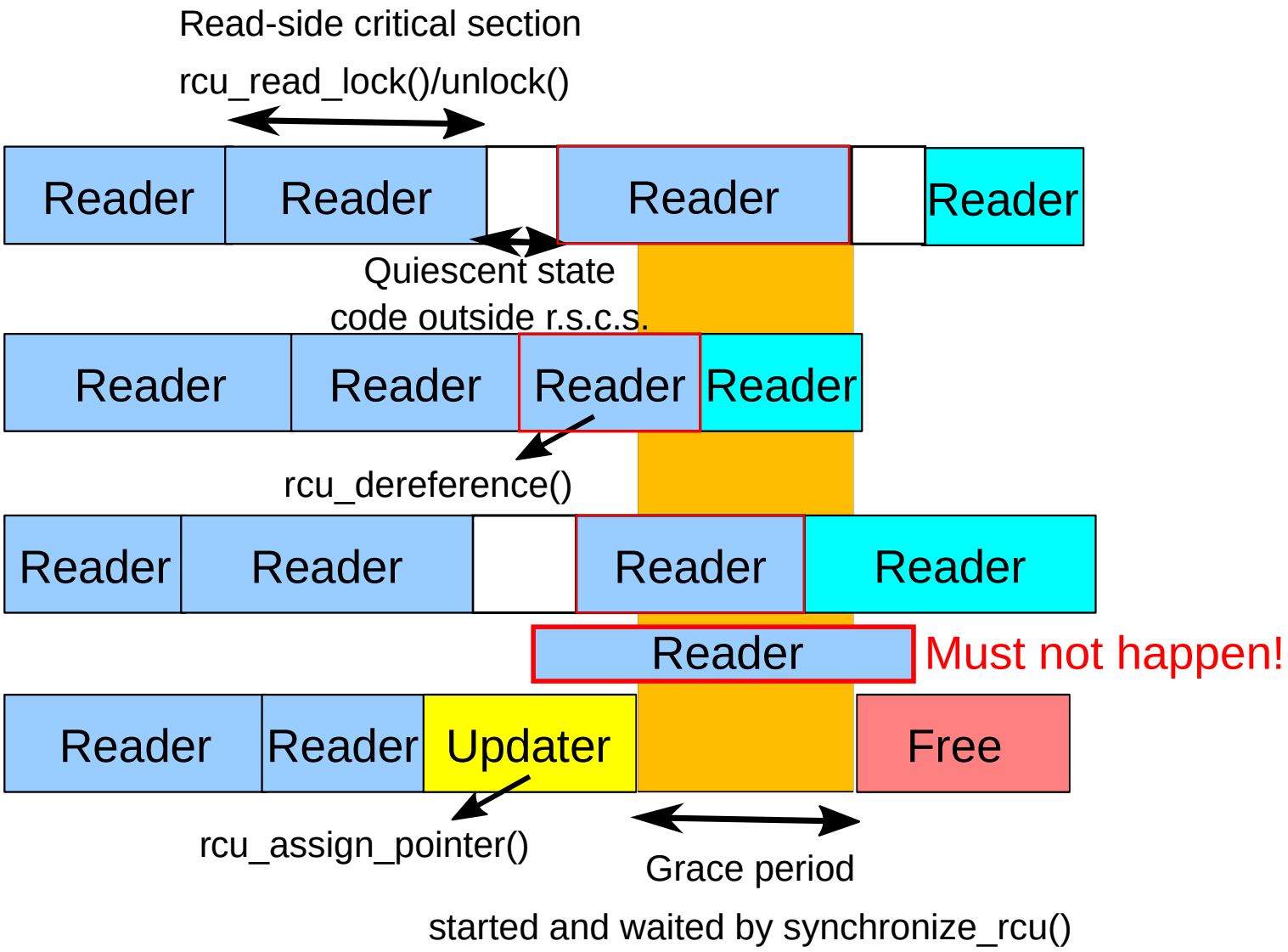
Main mechanisms of RCU

1. Publishing of updates (3→4)
   - Ensure that updated data reach memory before the updated pointer
   - Compiler and memory barrier ⇒ `rcu_assign_pointer()`
2. Accessing new versions of data (how readers traverse the list)
   - Ensure that we see all the updates made before publishing
   - Compiler and memory barrier ⇒ `rcu_dereference()`
3. Waiting for all readers to finish
   - The tricky part! ⇒ `synchronize_rcu()`

Note: RCU and Java

- Does it makes sense to use RCU in Java?
- No, because JVM decides when to free objects by using a tracing garbage collector. Garbage collectors have much higher overhead than RCU (stop-the-world, marking overhead, ...).

RCU concepts and API



RCU read-side critical section

```
rcu_read_lock(); /* Start critical section. */
p = rcu_dereference(cptr);
/* *p guaranteed to exist. */
do_something_with(p);
rcu_read_unlock(); /* End critical section. */
/* *p might be freed!!! */
```

- `rcu_read_lock()`/`unlock()` and `rcu_dereference()` are cheap, sometimes *nop*.
- Updaters are more heavy-weight.

RCU updater

```
pthread_mutex_lock(&updater_lock);
old_p = cptr;
*new_p = *old_p;                    // copy if needed
new_p->... = ...;
rcu_assign_pointer(cptr, new_p); // update
pthread_mutex_unlock(&updater_lock);
synchronize_rcu();                  // wait for grace period end
free(old_p);
```

- Updater can use locks, because we deal with read-mostly workload, where updates are infrequent.
- Locks are not needed if there is just one updater

Outline

1 Naive synchronization
- Problems
  - Non-atomic data manipulation
  - Unwanted compiler optimizations
  - Reordering at hardware level
  - Deadlock

2 Cost of atomic operations and barriers

3 Kernel semaphores

4 Futex

5 Read-mostly workload

6 Read-Copy-Update (RCU)
- RCU implementations

How does it work?

- Many implementations possible
- Trade-off between read-side overhead and constraints of application structure
- We will look at the following implementations:
  - Quiescent-state based reclamation (QSBR)
  - General-purpose
- See `https://www.efficios.com/pub/rcu/urcu-supp.pdf`, Appendix D for more implementations and details.

Quiescent-state based reclamation (QSBR)

```c
// Protects registry from concurrent accesses
pthread_mutex_t rcu_gp_lock =
    PTHREAD_MUTEX_INITIALIZER;

LIST_HEAD(registry);

struct rcu_reader {
  unsigned long ctr;
  char need_mb;
  struct list_head node;
  pthread_t tid;
};

// per-thread variable
struct rcu_reader __thread rcu_reader;

void rcu_register_thread(void) {
  rcu_reader.tid = pthread_self();
  mutex_lock(&rcu_gp_lock);
  list_add(&rcu_reader.node, &registry);
  mutex_unlock(&rcu_gp_lock);
  rcu_thread_online();
}
void rcu_unregister_thread(void) {
  rcu_thread_offline();
  mutex_lock(&rcu_gp_lock);
  list_del(&rcu_reader.node);
  mutex_unlock(&rcu_gp_lock);
}
```

```c
#define RCU_GP_ONLINE 0x1
#define RCU_GP_CTR    0x2

// global counter - note: not accessed with atomic instr.
unsigned long rcu_gp_ctr = RCU_GP_ONLINE;

static inline void rcu_read_lock(void) {}
static inline void rcu_read_unlock(void) {}

// Every thread must call this function periodically
// outside of read-side critical section.
// Note 1: There is no atomic instruction and barriers are
//         cheaper than atomic instructions.
// Note 2: This is not called per read-side critical section
//         only "from time to time" e.g. in the main program
//         loop.
static inline void rcu_quiescent_state(void) {
  smp_mb();
  STORE_SHARED(rcu_reader.ctr, LOAD_SHARED(rcu_gp_ctr));
  smp_mb();
}
// call before a blocking system call
static inline void rcu_thread_offline(void) {
  smp_mb();
  STORE_SHARED(rcu_reader.ctr, 0);
}
// call after return from a blocking system call
static inline void rcu_thread_online(void) {
  STORE_SHARED(rcu_reader.ctr, LOAD_SHARED(rcu_gp_ctr));
  smp_mb();
}
```

Quiescent-state based reclamation (QSBR), cont.

```
// Updater will call these
void synchronize_rcu(void) {
  unsigned long was_online;
  was_online = rcu_reader.ctr;
  smp_mb();
  if (was_online)
    STORE_SHARED(rcu_reader.ctr, 0);
  mutex_lock(&rcu_gp_lock);
  update_counter_and_wait();
  mutex_unlock(&rcu_gp_lock);
  if (was_online)
    STORE_SHARED(rcu_reader.ctr, LOAD_SHARED(rcu_gp_ctr));
  smp_mb();
}
static void update_counter_and_wait(void) {
  struct rcu_reader *index;
  STORE_SHARED(rcu_gp_ctr, rcu_gp_ctr + RCU_GP_CTR);
  barrier();
  list_for_each_entry(index, &registry, node) {
    while (rcu_gp_ongoing(&index->ctr))
      msleep(10);
  }
}
static inline int rcu_gp_ongoing(unsigned long *ctr)
{
  unsigned long v;
  v = LOAD_SHARED(*ctr);
  return v && (v != rcu_gp_ctr);
}
```

Properties:

- Grace periods are not shared
- Long waiting ⇒ higher memory consumption
- Works only on 64-bit architectures – the counter must not overflow

QSBR example

	Startup	rcu_register_thread	rcu_quiescent_state @CPU0	rcu_quiescent_state @CPU1	synchronize_rcu start @CPU3	rcu_quiescent_state @CPU0	rcu_quiescent_state @CPU2	rcu_quiescent_state @CPU1	synchronize_rcu end @CPU3		synchronize_rcu start @CPU1	rcu_quiescent_state @CPU3	rcu_quiescent_state @CPU0	rcu_quiescent_state @CPU2	synchronize_rcu end @CPU1
rcp_gp_ctr	1				3						5				
CPU0 rcu_reader.ctr		1				3							5		
CPU1 rcu_reader.ctr		1						3			0				5
CPU2 rcu_reader.ctr		1					3							5	
CPU3 rcu_reader.ctr		1			0				3			5			

General-purpose RCU

```
#define RCU_GP_CTR_PHASE 0x10000
#define RCU_NEST_MASK 0x0ffff
#define RCU_NEST_COUNT 0x1

unsigned long rcu_gp_ctr = RCU_NEST_COUNT;

static inline void rcu_read_lock(void) {
  unsigned long tmp;
  tmp = rcu_reader.ctr;
  if (!(tmp & RCU_NEST_MASK)) {
    STORE_SHARED(rcu_reader.ctr, LOAD_SHARED(rcu_gp_ctr));
    smp_mb();
  } else {
    STORE_SHARED(rcu_reader.ctr, tmp + RCU_NEST_COUNT);
  }
}
static inline void rcu_read_unlock(void)
{
  smp_mb();
  STORE_SHARED(rcu_reader.ctr, rcu_reader.ctr - RCU_NEST_COUNT);
}
```

Properties:

- Does not restrict application structure
  - No need to call rcu_quiescent_state
  - No need to call rcu_thread_(on|off)line around blocking syscalls
- No counter-overflow problem (different mechanism with only 1-bit counters)
- Higher read-side overhead: memory barrier (still less than typical locks).

Genereal-purpose RCU, cont.

```
void synchronize_rcu(void)
{
  smp_mb();
  mutex_lock(&rcu_gp_lock);
  update_counter_and_wait();
  barrier();
  update_counter_and_wait();
  mutex_unlock(&rcu_gp_lock);
  smp_mb();
}
static void update_counter_and_wait(void)
{
  struct rcu_reader *index;
  STORE_SHARED(rcu_gp_ctr, rcu_gp_ctr ^ RCU_GP_CTR_PHASE);
  barrier();
  list_for_each_entry(index, &registry, node) {
    while (rcu_gp_ongoing(&index->ctr))
      msleep(10);
  }
}
static inline int rcu_gp_ongoing(unsigned long *ctr)
{
  unsigned long v;
  v = LOAD_SHARED(*ctr);
  return (v & RCU_NEST_MASK) && ((v ^ rcu_gp_ctr) & RCU_GP_CTR_PHASE);
}
```

Update benchmarks

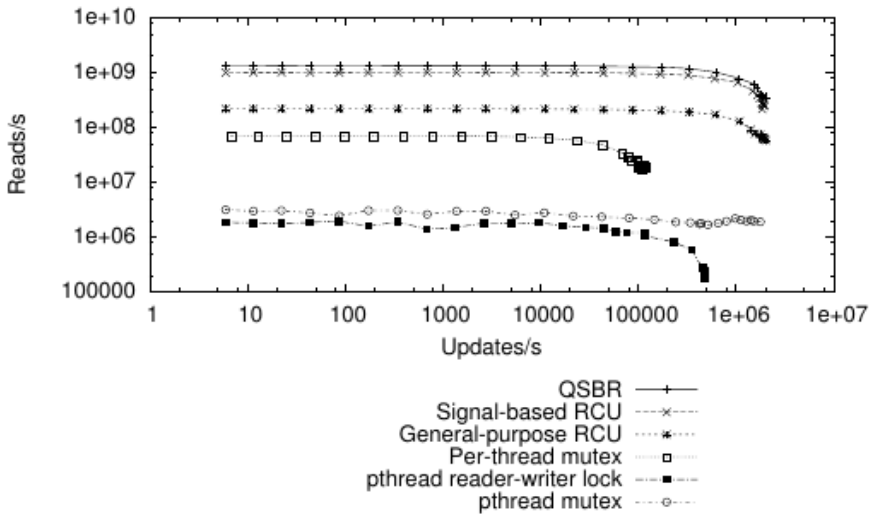


Fig. 9. Update Overhead, 8-core Intel Xeon, Logarithmic Scale

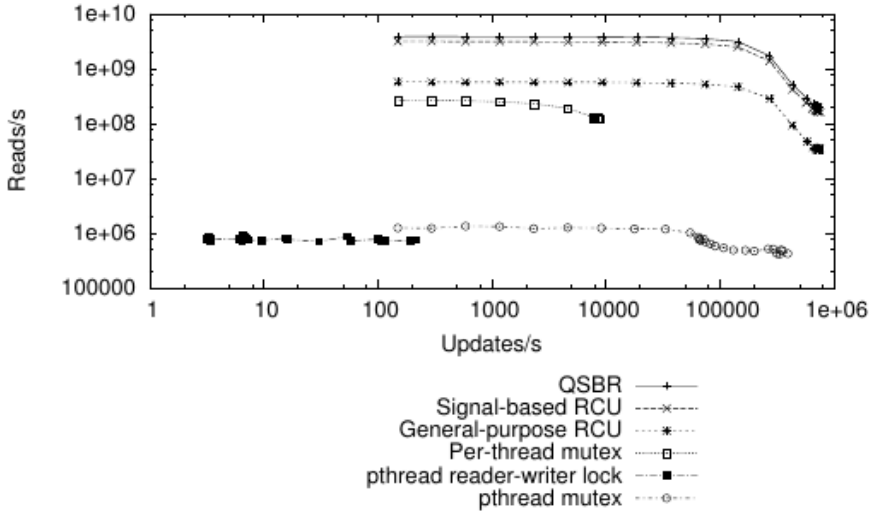


Fig. 10. Update Overhead, 64-core POWER5+, Logarithmic Scale

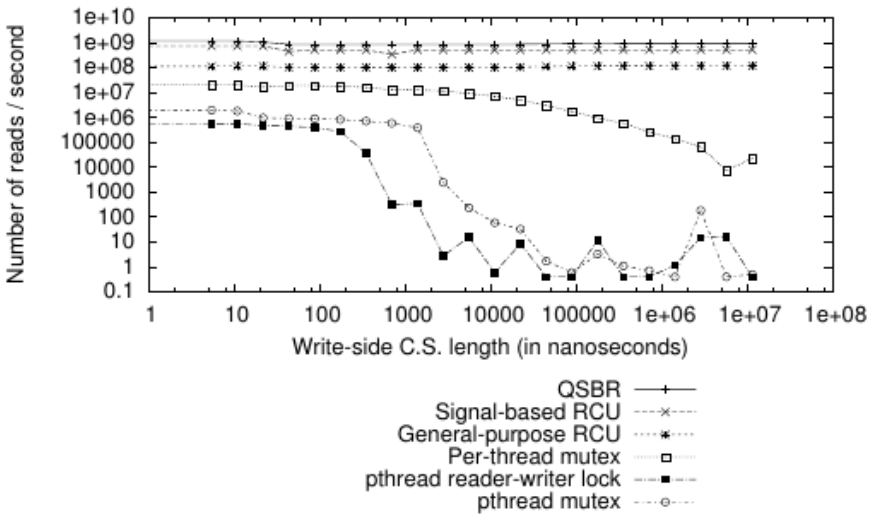


Fig. 11. Impact of Update-Side Critical Section Length on Read-Side, 8-core Intel Xeon, Logarithmic Scale

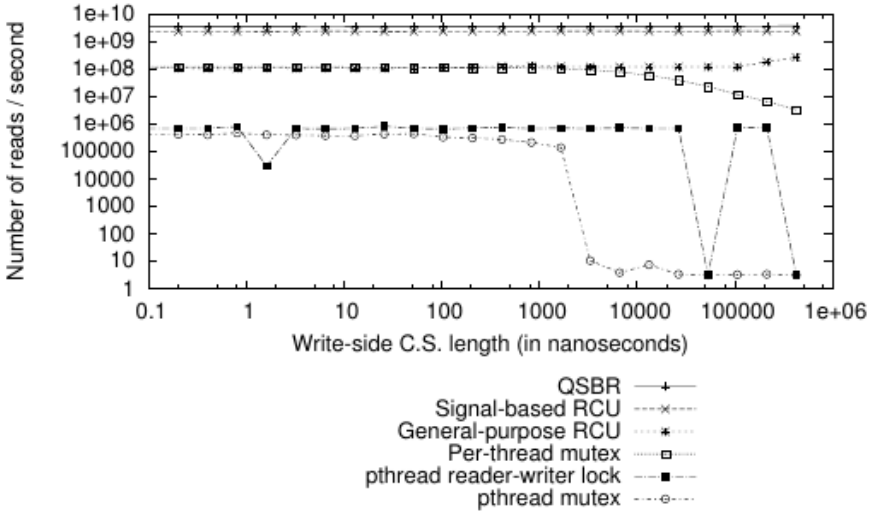


Fig. 12. Impact of Update-Side Critical Section Length on Read-Side, 64-core POWER5+, Logarithmic Scale

Conclusion

- RCU is a scalable synchronization mechanism for hundreds/thousands of CPUs and read-mostly workload
- We have seen an RCU-based implementation of single-linked list, but many other common data structures can be implemented in an RCU-compatible way

References

- Paul E. McKenney et al.: Is Parallel Programming Hard, And, If So, What Can You Do About It?

  `https://mirrors.edge.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html`

- Desnoyers, Mathieu, McKenney, Paul. E., Stern, Alan S., Dagenais, Michel R. and Walpole, Jonathan, User-Level Implementations of Read-Copy Update. IEEE Transaction on Parallel and Distributed Systems, 23 (2): 375-382 (2012).

  `https://www.efficios.com/publications`

- Paul E. McKenney, What Is RCU? Guest Lecture for Technische Universität Dresden

  `http://www2.rdrop.com/users/paulmck/RCU/RCU.2014.05.18a.TU-Dresden.pdf`

Appendix

- RCU is being proposed for inclusion into C++ standard:

  `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1122r2.pdf`

- RCU is patented by IBM, which freely licenses the patent to copy-left software. Userspace RCU library (copyleft – LGPL) can be used in proprietary code. The patents should expire soon.

  `https://lwn.net/Articles/777519/`

Effective Software

Lecture 6: Data races, synchronization, atomic operations, non-blocking algorithms

David Šišlák
david.sislak@fel.cvut.cz

[1] Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Elsevier, 2008.

[2] Fog, A.: The microarchitecture of Intel, AMD and VIA CPU, 2016.

[3] Russell, K., Detlefs, D.: Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing in OOPSLA'06. ACM, USA 2006.

[4] Oaks, S.: Java Performance: The Definitive Guide. O'Reilly, USA 2014.

Data Races – Multi-threaded Environments

```
public int A = 0;
public int B = 0;
public int C = 0;
public int D = 0;
```

Thread 1	Thread 2

`public void method1() {`
`  int r2 = A;`
`  B = 1;`
`  D = r2;`
`}`
	`public void method2() {`
	`  int r1 = B;`
	`  A = 2;`
	`  C = r1;`
	`}`

» what can be the results for C and D?

Data Races – Multi-threaded Environments

```
public int A = 0;
public int B = 0;
public int C = 0;
public int D = 0;
```

Thread 1	Thread 2

`public void method1() {`
`  int r2 = A;`
`  B = 1;`
`  D = r2;`
`}`
	`public void method2() {`
	`  int r1 = B;`
	`  A = 2;`
	`  C = r1;`
	`}`

» what can be the results for C and D?

- C=0, D=0
- C=1, D=0
- C=0, D=2
- anything else?

Data Races – Disassembled Method and Assembly Code

```
public void method1() {
    int r2 = A;
    B = 1;
    D = r2;
}
```

```
 0: aload_0
 1: getfield      #2 // Field A:I
 4: istore_1
 5: aload_0
 6: iconst_1
 7: putfield      #3 // Field B:I
10: aload_0
11: iload_1
12: putfield      #5 // Field D:I
15: return
```

Heap object structure:

8B - mark word
4B / 8B – Klass ref.
… object data

Klass – internal JVM representation of class Metadata

4B – 32bit, or 64bit <32GB heap
8B – 64bit no compressedOOP

instructions **reordered in C2 compiler**:

RSI is **this**

```
0x000000010639924c: movl    $0x1,0x10(%rsi)     ;*putfield B
                                                ; - datarace.DataRace::method1@7 (line 11)

0x0000000106399253: mov     0xc(%rsi),%r11d
0x0000000106399257: mov     %r11d,0x18(%rsi)    ;*putfield D
                                                ; - datarace.DataRace::method1@12 (line 12)
```

note: all machine code examples are from JVM 8 64-bit <32GB, Intel Haswell CPU in AT&T syntax

» **the same reordering happens in method2 resulting into fourth output**

- **C=1, D=2**

Data Races – CPU Execution Pipelining

» simplified non-parallel instruction pipelining in **each core**

IF: Instruction fetch
EX : Execution
WB : Write back

ID : Instruction decode
MEM: Memory access

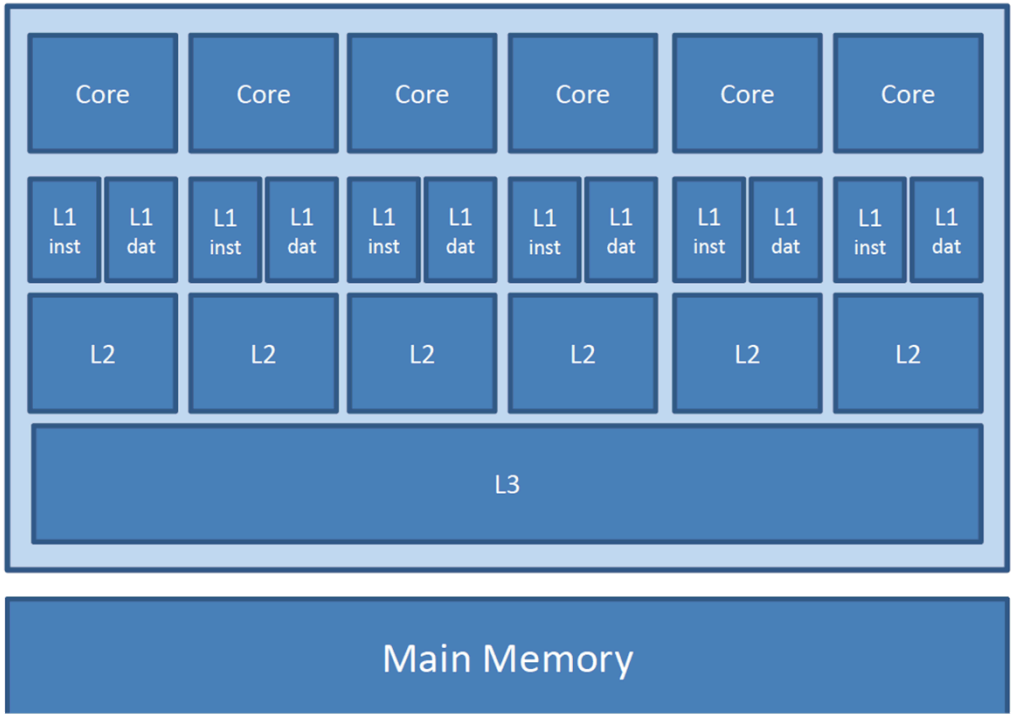
Cycles

Instruction #	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

» each step is parallelized as well, e.g. Haswell does 4 instructions in single cycle (execution depends on type and independency of instructions)

Data Races – CPU Memory Model

» CPU vs. core vs. thread



L1 Data Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	4 ns	8-way
L1 Instruction Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	4 ns	4-way
L2 Cache			
Size	Line Size	Latency	Associativty
256 KB	64 bytes	10 ns	8-way
L3 Cache			
Size	Line Size	Latency	Associativty
12 MB	64 bytes	50 ns	16-way
Main Memory			
Size	Line Size	Latency	Associativty
	64 bytes	75 ns	

» all writes to main memory are done in **write-back** cache mode

- standard writes requires data to be cached (expensive cache miss)
- non-temporal writes (especially useful for large block writes)
  - content directly queued to memory without caching at all
- prefetch instructions available

Data Races – CPU Execution Pipelining – Superscalar Execution

» modern CPUs have multiple execution units **in each core** (8 in Intel Haswell)

- units have various capabilities (4x integer ALU, 2x FPU mul, 2x mem read, …)
- multiple **µops** with various latency executed **in parallel** during each per cycle

» independent instructions can be **executed out-of-order** or in parallel

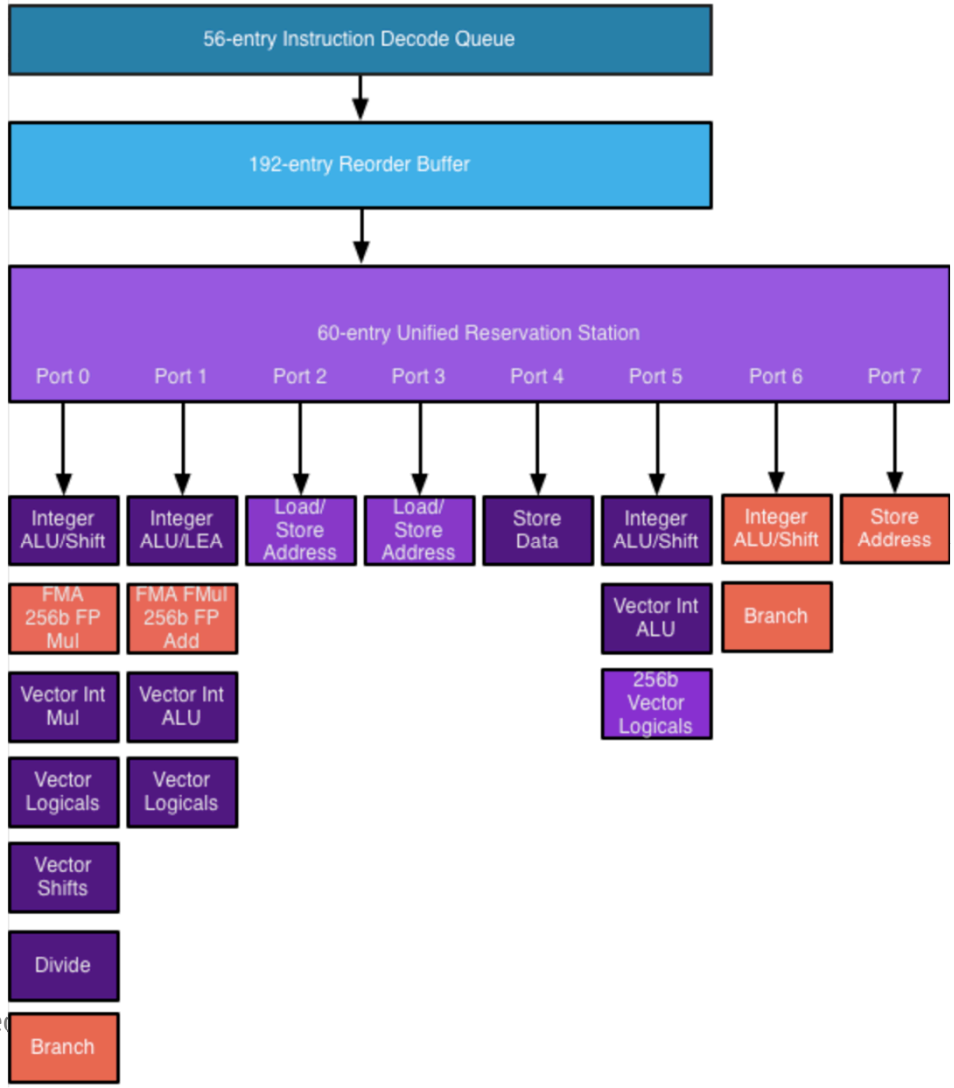
- not using the same register or address

» **memory reads are never reordered**

- parallel independent reads

» **later (independent) reads can be reordered and executed before writes**

- serialized writes only



Volatile Variable – Memory Barrier

making A and B volatile:

```
public volatile int A = 0;
public volatile int B = 0;
public int C = 0;
public int D = 0;

public void method1() {
    int r2 = A;
    B = 1;
    D = r2;
}
```

results into assembly code:

```
0x000000010710e08c: mov    0xc(%rsi),%r11d
0x000000010710e090: movl   $0x1,0x10(%rsi)
0x000000010710e097: lock addl $0x0,(%rsp)
0x000000010710e09c: mov    %r11d,0x18(%rsi)
```

8B - mark word
4B / 8B – Klass ref.
… object data

» memory operations over store in volatile are **not reordered** in C1/C2 compiler

» no need for read barriers – not reordered during execution in CPU

» instruction **lock prefix** **forbids all reordering around** and **synchronize previous writes to be visible by all others CPUs**

» *lock addl $0x0,(%rsp)* is fastest memory barrier – no operation inside CPU

Volatile Variable

- **never cached thread-locally** – all access directly to main memory
- guarantees **atomic read and write** operations (defines memory barrier)
- can be used for both primitives and references to objects
- don't block thread execution
- BUT:
  - volatile writes are much slower due to cache flush (~100x)
  - volatile reads (if there are writes) are slower (~25x, #CPU/cores)
    - due to invalidated cache
  - still faster than synchronization/locks

- <u>not necessary for:</u>
  - immutable objects
  - variable accessed by only one thread (context switch properly flushes cache already)
  - where variable is within complex synchronized operation

Counter Example - Volatile

```java
public class VolatileCounter {
    private volatile int cnt=0;

    public int get() {
        return cnt;
    }

    public void increment() {
        cnt++;
    }
}
```

» will it work as expected in multi-threaded environment?

Counter Example - Volatile

```java
public class VolatileCounter {
    private volatile int cnt=0;

    public int get() {
        return cnt;
    }

    public void increment() {
        cnt++;
    }
}
```

increment assembly code:

RSI is **this**

```
0x000000010911544c: mov    0xc(%rsi),%edi

0x000000010911544f: inc    %edi
0x0000000109115451: mov    %edi,0xc(%rsi)
0x0000000109115454: lock addl $0x0,(%rsp)
```

8B - mark word
4B / 8B – Klass ref.
… object data

» will it work as expected in multi-threaded environment?
NO

» **volatile**

- **not suitable for read-update-write operations**
- **useful for one-thread write** (e.g. termination flag)
  - must be used if flag is set by different thread otherwise C2 compiler could create **infinite loop** without testing

Volatile Arrays

```java
public class VolatileIntArray {
    private volatile int[] array;

    public VolatileIntArray(int capacity) {
        array = new int[capacity];
    }

    public int get(int index) {
        return array[index];
    }

    public void put(int index, int value) {
        array[index] = value;
    }
}
```

» Is put operation to array member volatile?

Volatile Arrays

```java
public class VolatileIntArray {
    private volatile int[] array;

    public VolatileIntArray(int capacity) {
        array = new int[capacity];
    }

    public int get(int index) {
        return array[index];
    }

    public void put(int index, int value) {
        array[index] = value;
    }
}
```

8B - mark word
4B / 8B – Klass ref.
… object data

8B - mark word
4B / 8B – Klass ref.
4B – array length
sequence of values

» Is put operation to array member volatile?

NO – see assembly code, there is no cache synchronization with lock

```
# this:     rsi:rsi   = 'datarace/VolatileIntArray'
# parm0:    rdx       = int
# parm1:    rcx       = int

0x000000011170bbcc: mov    0xc(%rsi),%esi
0x000000011170bbcf: shl    $0x3,%rsi          ;*getfield array
                                              ; - datarace.VolatileIntArray::put@1 (line 15)

0x000000011170bbd3: movslq %edx,%rdi
0x000000011170bbd6: cmp    0xc(%rsi),%edx     ; implicit exception: dispatches to 0x000000011170bbef
0x000000011170bbd9: jae    0x000000011170bbf9 —— ArrayOutOfBoundsException
0x000000011170bbdf: mov    %ecx,0x10(%rsi,%rdi,4)  ;*iastore
                                              ; - datarace.VolatileIntArray::put@6 (line 15)
```

Volatile Arrays - Solution

```
private volatile int[] array;

public void put(int index, int value) {
    array[index] = value;
    array = array;
}
```

8B - mark word
4B / 8B – Klass ref.
… object data

» just array reference is volatile

» added unnecessary **array** reference update adds assembly code

```
0x000000010db21a67: mov    %r8d,0xc(%rsi)
0x000000010db21a80: lock addl $0x0,(%rsp)     ;*putfield array
                                              ; - datarace.VolatileIntArray::put@12 (line 16)
```

» instruction **lock prefix** forbids all reordering around and synchronize previous writes to be visible by all others CPUs

» **not suitable for read-update-write operations**

Counter Example – Synchronized and ReentrantLock

```java
public class SynchronizedCounter {
    private int cnt=0;

    public int get() {
        return cnt;
    }

    public synchronized void increment() {
        cnt++;
    }
}
```

```java
public class ReentrantCounter {
    private int cnt=0;
    private ReentrantLock lock = new ReentrantLock();

    public int get() {
        return cnt;
    }

    public void increment() {
        lock.lock();
        try {
            cnt++;
        } finally {
            lock.unlock();
        }
    }
}
```

!

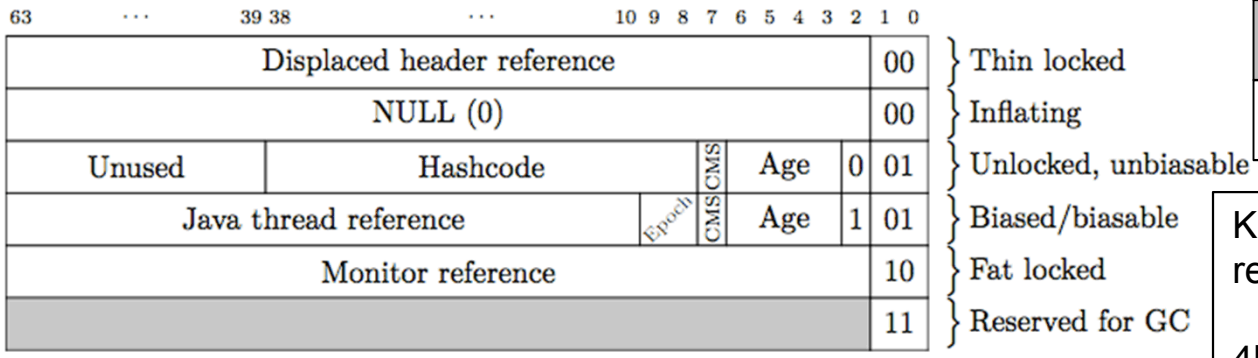
» no issue with read-update-write operations

» synchronized

- method vs. block
- object instance vs. class instance (static methods)

JVM - Synchronize Implementation

Mark word (64-bit JVM):



8B - mark word
4B / 8B – Klass ref.
… object data

Klass – internal JVM representation of class Metadata

4B – 32bit, or 64bit <32GB heap
8B – 64bit no compressedOOP

» *prototype mark word* in Klass

» **lock records** in stack (on pre-compiled locations for compiled code)

- 8B - displacement of original object mark word – recursive lock has 0
- 4B / 8B - compressedOOP/OPP to locked object

» **thin lock** is using CAS instruction on lock/unlock to modify mark word

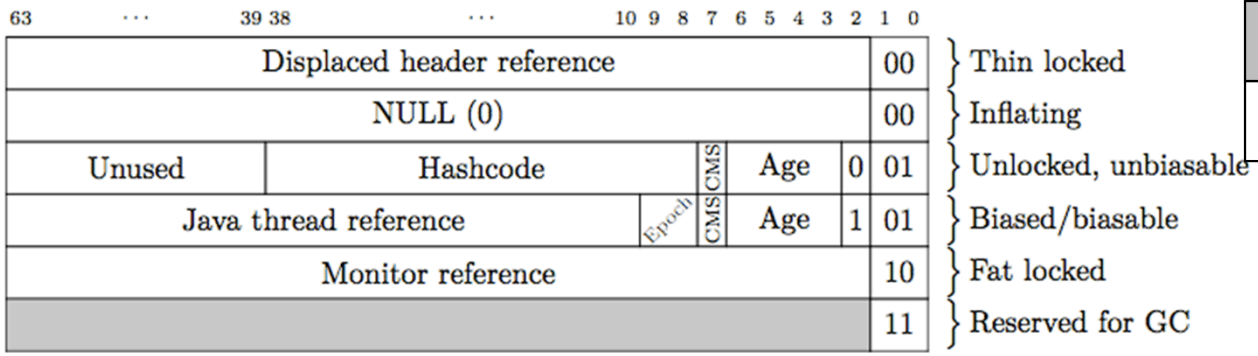
- use spin-locking (10 cycles with volatile read + NOPs) before fat locking

» **fat lock** is using monitor object on heap (inflating creates, deflating destroys)

- contended lock or call of **wait/notify**
- monitor: original mark word, OS lock, conditions, set of threads; support parking

JVM - Synchronize Implementation

Mark word (64-bit JVM):



8B - mark word
4B / 8B – Klass ref.
… object data

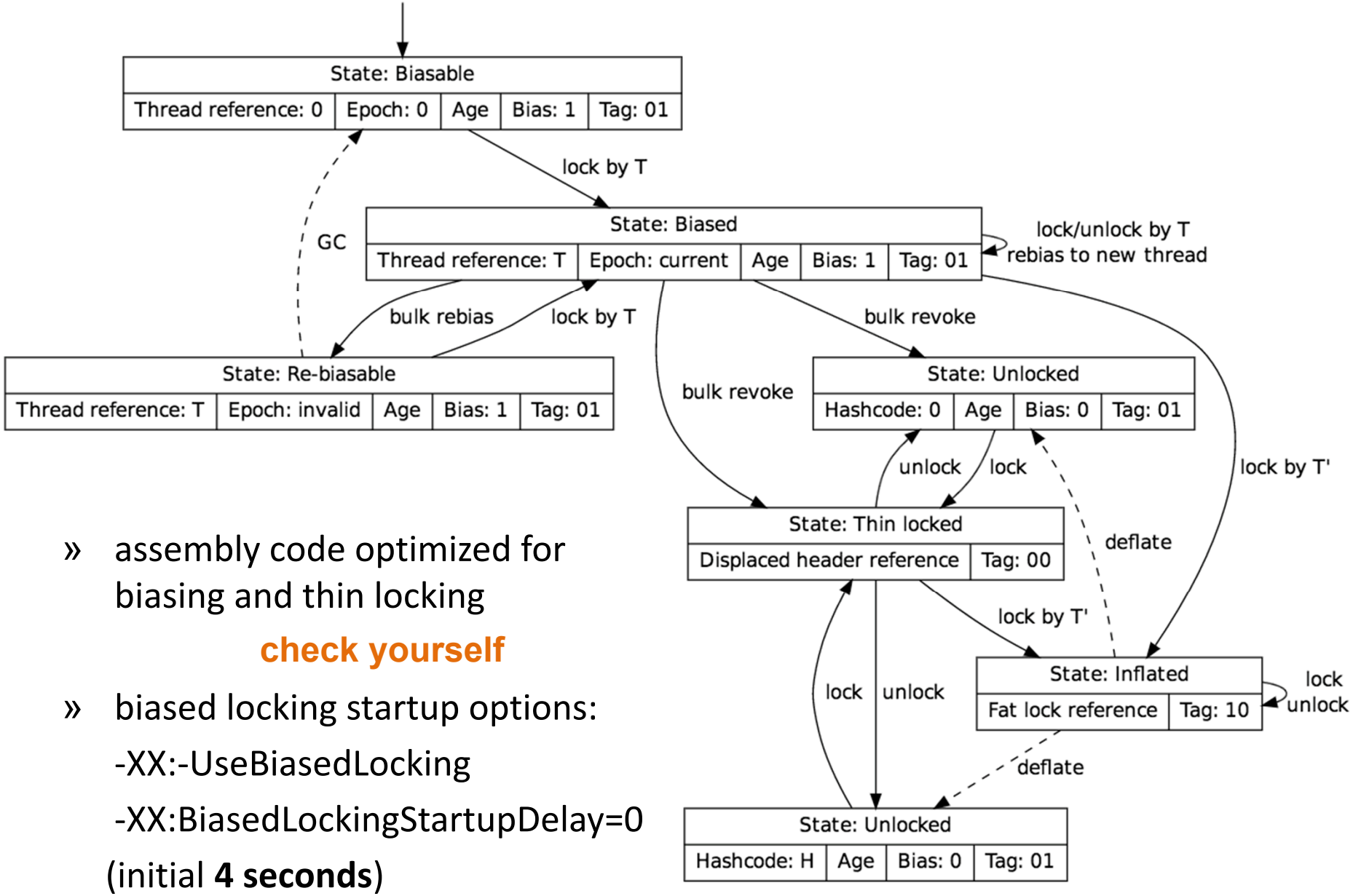
» **biased lock** is fast locking/unlocking by single thread without any CAS instruction

- *biasable* state – enabled **4 seconds** after JVM start (startup-up, learning)
- different thread and valid epoch -> instance **re-biasing** OR **thin/fat locking**
- global **safe pointing** needed for setting *biasable*, *re-biasing*, *bias revocation*
- bulk operations amortizing cost for safe pointing (all instance types)

  >20 re-biasing -> **bulk re-biasing** (increment epoch in prototype, scan locks)

  >40 re-biasing -> **bulk revocation** (change in prototype)
- mark word **normalization** during GC – preserve *hashed*, *locked*, *un-biasable*
- **identity hash** (Object.toString) or **fat lock** disable instance biased locking

JVM - Synchronize Implementation



» assembly code optimized for biasing and thin locking

check yourself

» biased locking startup options:

-XX:-UseBiasedLocking

-XX:BiasedLockingStartupDelay=0

(initial **4 seconds**)

Reentrant Locks

» bias-based locking with extended operations in comparison to **synchronized**

- lock(), unlock()
- lockInterruptibly() throws InterruptedException
- boolean tryLock()
- boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException

» **fairness**

- blocked threads are **ordered** for fair locking
- **new** ReentrantLock(boolean fair), by default unfair
- **synchronized** is unfair
- unfair ReentrantLocks are slightly faster than synchronized
  - but another instance in HEAP
- fair locks are slower (~100x)

Counter Example – AtomicInteger

```
public class AtomicCounter {
    private AtomicInteger cnt = new AtomicInteger( initialValue: 0);

    public int get() {
        return cnt.get();
    }

    public void increment() {
        cnt.incrementAndGet();
    }
}
```

AtomicInteger implementation

```
private static final long valueOffset;

static {
    try {
        valueOffset = unsafe.objectFieldOffset
            (AtomicInteger.class.getDeclaredField( name: "value"));
    } catch (Exception ex) { throw new Error(ex); }
}

private volatile int value;

public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5: var5 + var4));

    return var5;
}

public final int getAndIncrement() {
    return unsafe.getAndAddInt( o: this, valueOffset, i: 1);
}
```

non-blocking pattern

Counter Example – AtomicInteger – Assembly Code

C2 compiler assembly code for AtomicCounter::increment

RSI is **this**

```
0x000000010b108d4c: mov    0xc(%rsi),%r11d    ;*getfield cnt
                                              ; - datarace.AtomicCounter::increment@1 (line 13)

0x000000010b108d50: test   %r11d,%r11d
0x000000010b108d53: je     0x000000010b108d68
0x000000010b108d55: lock addl $0x1,0xc(%r12,%r11,8)  ;*invokevirtual getAndAddInt
                                              ; - java.util.concurrent.atomic.AtomicInteger::incrementAndGet@8 (line 186)
                                              ; - datarace.AtomicCounter::increment@4 (line 13)
```

null pointer check with **exception**

- » while cycle optimized and replaced with **single instruction**
- » instruction **lock prefix** forbids all reordering around and synchronize previous writes to be visible by all others CPUs
- » instruction **lock prefix** ensures that core has exclusive ownership of the appropriate cache line for the duration of the operation
  - cache coherency using MESIF (Haswell) with fall-back to mem bus lock
- » **AtomicInteger-based counter is fastest of all for multi-threaded**

Atomic Operations

» 32-bit CPUs support 64-bit CAS operations
  - **cmpxchg** src_operand, dst_operand – implicit instruction lock prefix

» 64-bit CPUs support 128-bit CAS operations
  - **cmpxchg16b** – works with RDX:RAX and RCX:RBX register pairs

» JAVA uses only 64-bit CAS operations in java.util.concurrent.atomic
  - AtomicBoolean
  - AtomicInteger
  - AtomicLong
  - AtomicReference
  - AtomicIntegerArray
  - AtomicLongArray
  - AtomicReferenceArray

Atomic Field Updaters

» suitable with large number of objects of the given type – it saves memory

- don't require single instance to have an extra object embedded

» refer volatile variable directly without getter and setters

```
public class ObjectWithAtomic {
    private final AtomicInteger value =
        new AtomicInteger(0);
    // ...

    public void method1() {
        // ...
        if (value.compareAndSet(1, 2)) {
            // ...
        }
    }
}
```

```
public class ObjectWithAtomic {
    private static AtomicIntegerFieldUpdater<ObjectWithAtomic>
        valueUpdater = AtomicIntegerFieldUpdater.newUpdater(ObjectWithAtomic.class, "value");
    private volatile int value = 0;
    // ...

    public void method1() {
        // ...
        if (valueUpdater.compareAndSet(this, 1, 2)) {
            // ...
        }
    }
}
```

Atomic Field Updaters

» but beware of **less efficient** operations for atomic field updaters

» AtomicIntegerFieldUpdater implementation

```
private void fullCheck(T obj) {
    if (!tclass.isInstance(obj))
        throw new ClassCastException();
    if (cclass != null)
        ensureProtectedAccess(obj);
}

public boolean compareAndSet(T obj, int expect, int update) {
    if (obj == null || obj.getClass() != tclass || cclass != null) fullCheck(obj);
    return unsafe.compareAndSwapInt(obj, offset, expect, update);
}
```

» existing field updaters

- AtomicIntegerFieldUpdater
- AtomicLongFieldUpdater
- AtomicReferenceFieldUpdater

» no array field updaters

Atomic Complex Types

» **AtomicMarkableReference**

- **object reference** along with a **mark bit**

» **AtomicStampedReference**

- **object reference** along with an **integer "stamp"**

» <u>notes</u>:

- useful for ABA problem
  - A -> B and B -> A, how can I know that A has been changed since the last observation?
- doesn't use double-wide CAS (CAS2, CASX) -> much slower than simple atomic types due to **object allocation**

Atomic Complex Types – Larger Than 64-bits

» **AtomicMarkableReference**

- **object reference** along with a **mark bit**

» **AtomicStampedReference**

- **object reference** along with an **integer "stamp"**

```
public class AtomicStampedReference<V> {

    private static class Pair<T> {
        final T reference;
        final int stamp;
        private Pair(T reference, int stamp) {
            this.reference = reference;
            this.stamp = stamp;
        }
        static <T> Pair<T> of(T reference, int stamp) {
            return new Pair<T>(reference, stamp);
        }
    }

    private volatile Pair<V> pair;

    public boolean compareAndSet(V   expectedReference,
                                 V   newReference,
                                 int expectedStamp,
                                 int newStamp) {
        Pair<V> current = pair;
        return
            expectedReference == current.reference &&
            expectedStamp == current.stamp &&
            ((newReference == current.reference &&
              newStamp == current.stamp) ||
             casPair(current, Pair.of(newReference, newStamp)));
    }
```

Non-blocking Algorithms

» **lock-free**, block-less but not usually **wait-free** (because of unbounded loops)
  - based on CMPXCHG and LOCK prefixed instructions

» shared resources secured by locks have drawbacks
  - high-priority thread can be blocked (e.g. interrupt handler)
  - parallelism reduced by coarse-grained locking (unfair locks)
  - fine-grained locking and fair locks increases overhead
  - can lead to **deadlocks**, **priority inversion** (low-priority thread holds a shared resource which is required by high-priority thread)

» **non-blocking algorithms properties**:
  - outperform blocking algorithms because most of CMPXCHG succeeds on the first try
  - removes cost for synchronization, thread suspension, context switching

» note: **real-time systems require wait-free algorithms** (finite number of steps)

Non-blocking stack (LIFO)

» Treiber's algorithm (1986)

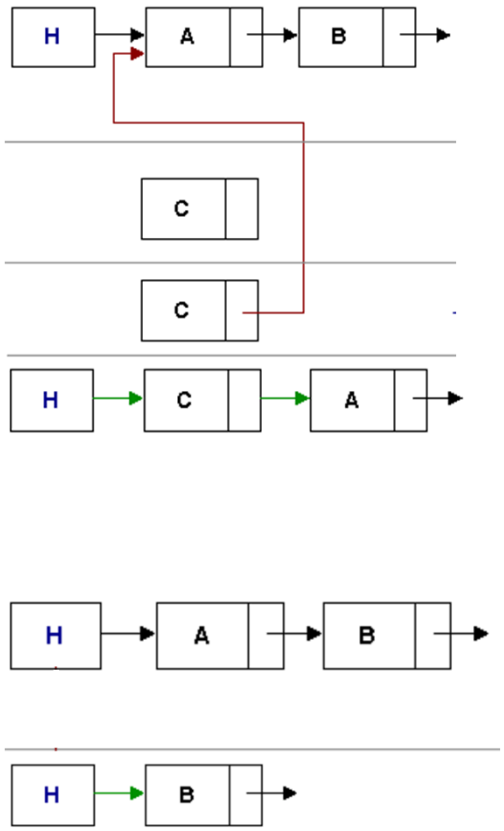
```
static class Node<E> {
    final E item;
    Node<E> next;

    public Node(E item) { this.item = item; }
}

AtomicReference<Node<E>> head = new AtomicReference<Node<E>>();

public void push(E item) {
    Node<E> newHead = new Node<E>(item);
    Node<E> oldHead;
    do {
        oldHead = head.get();
        newHead.next = oldHead;
    } while (!head.compareAndSet(oldHead, newHead));
}

public E pop() {
    Node<E> oldHead;
    Node<E> newHead;
    do {
        oldHead = head.get();
        if (oldHead == null)
            return null;
        newHead = oldHead.next;
    } while (!head.compareAndSet(oldHead,newHead));
    return oldHead.item;
}
```



push after pop can cause ABA problem if address is reused !

Thread-safe collections and maps

» blocking collections and maps
- static<T> Collection<T> Collections.synchronizedCollection(Collection<T> c)
- static<T> List<T> Collections.synchronizedList(List<T> list)
- static<K,V> Map<K,V> Collections.synchronizedMap(Map<K,V> m)
- static<T> Set<T> Collections.synchronizedSet(Set<T> s)
- also for SortedSet and SortedMap

» non-blocking collections and maps
- ConcurrentLinkedQueue (interface Collection, Queue):
  - E peek(), E poll(), add(E)
- ConcurrentHashMap (interface Map):
  - putIfAbsent(K key, V value), remove(Object key, Object value)
  - replace(K key, V oldValue, V newValue)
- ConcurrentSkipListMap (interface SortedMap), ConcurrentSkipListSet (interface SortedSet)

» non-blocking collections and maps are slower for single-threaded access
- due to usage of CASE instructions in comparison to biased locking

ConcurrentHashMap

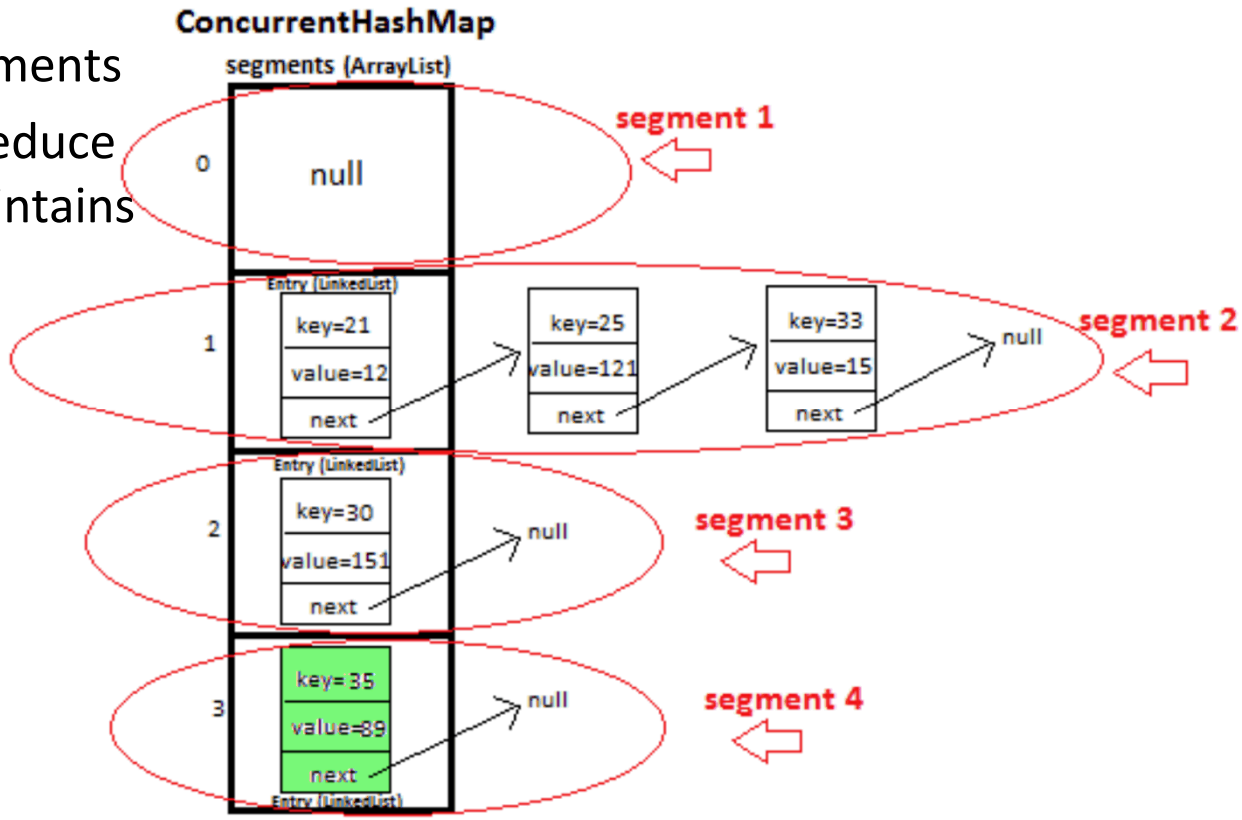
» concurrent reads – get, iterator

» minimize update contention

- initial concurrency level 16 (can be changed) - # updating threads
  - initial insertion into empty segment uses CMPXCHG operation
  - later modifications are based on segment-based locks

» segment contention

- use lists for <8 elements
- balanced tree to reduce search times – maintains next for iteration



ConcurrentHashMap

» table resizing (occupancy exceed load factor 0.75)

- power of two expansions
  - same index or power of two index
- reusing internal Node if next is not changed – majority of cases
- any thread can help resizing instead of block
- Forward nodes to notify users about moved

Effective Software

Lecture 7: Non-blocking I/O, C10K, efficient networking

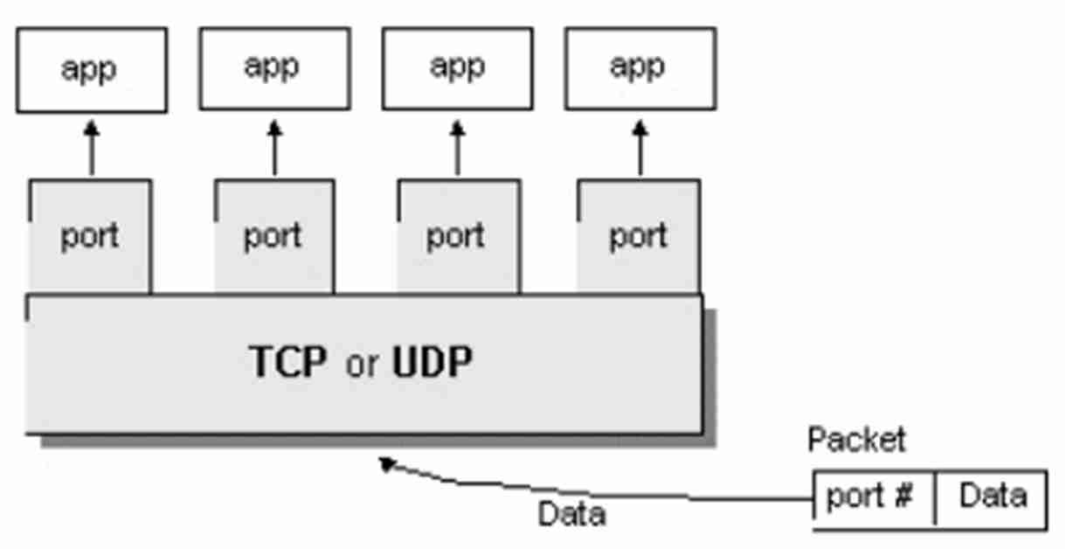
David Šišlák
david.sislak@fel.cvut.cz

[1] Tanenbaum, A. S., Wetherall, D. J.: Computer Networks. Pearson, 2011.
[2] Kegel, D.: The C10K problem. http://www.kegel.com/c10k.html
[3] Hitchens, R.: Java NIO. O'Reilly, 2002.

Network Communication – OSI Model

7 – Application Interface to end user. Interaction directly with software application.		**Software App Layer** Directory services, email, network management, file transfer, web pages, database access.	FTP, HTTP, WWW, SMTP, TELNET, DNS, TFTP, NFS
6 – Presentation Formats data to be "presented" between application-layer entities.		**Syntax/Semantics Layer** Data translation, compression, encryption/decryption, formatting.	ASCII, JPEG, MPEG, GIF, MIDI
5 – Session Manages connections between local and remote application.		**Application Session Management** Session establishment/teardown, file transfer checkpoints, interactive login.	SQL, RPC, NFS
4 – Transport Ensures integrity of data transmission.	Segment	**End-to-End Transport Services** Data segmentation, reliability, multiplexing, connection-oriented, flow control, sequencing, error checking.	TCP, UDP, SPX, AppleTalk
3 – Network Determines how data gets from one host to another.	Packet	**Routing** Packets, subnetting, logical IP addressing, path determination, connectionless.	IP, IPX, ICMP, ARP, PING, Traceroute
2 – Data Link Defines format of data on the network.	Frame	**Switching** Frame traffic control, CRC error checking, encapsulates packets, MAC addresses.	Switches, Bridges, Frames, PPP/SLIP, Ethernet
1 – Physical Transmits raw bit stream over physical medium.	Bits	**Cabling/Network Interface** Manages physical connections, interpretation of bit stream into electrical signals	Binary transmission, bit rates, voltage levels, Hubs

Network Communication – Introduction



TCP		UDP	
FTP	20,21	DNS	53
SSH	22	BooTPS/DHCP	67
Telnet	23	TFTP	69
SMTP	25	SNMP	161
DNS	53		
HTTP	80		
POP3	110		
NTP	123		
IMAP4	143		
HTTPS	443		

» **ports** – 16-bit number

» **IPv4** – 32-bit address

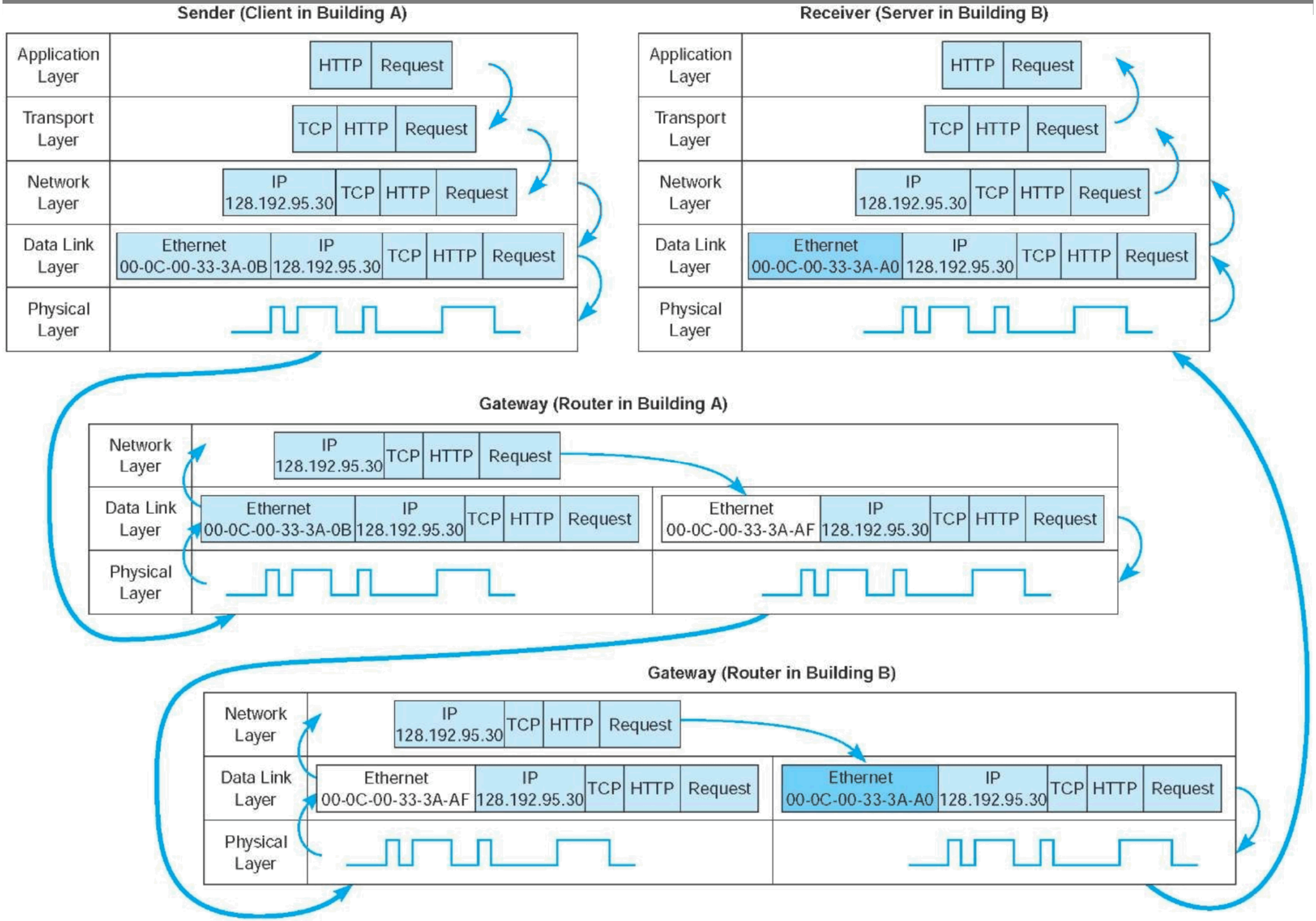
» **IPv6** – 128-bit address

- 48-bit or more routing prefix, 16-bit or less subnet id, 64-bit interface

http://[1fff:0:a88:85a3::ac1f]:8080/index.html

» TCP/UDP connection identification – **quad** – src IP, src port, dst IP, dst port

Network Communication – HTTP Example



C10k Problem

- handling a large number of clients (10 000s) at the same time (late 90s)
  - **concurrent connections at one server** requiring efficient scheduling
  - not related to requests per second
- sometime known as C1M or C10M problem (nowadays)
- approach
  - don't use **threading servers** like *Apache*
    - each connection handled by own thread/process (pooled but limited)
    - connection operations usually use **blocking** operations
    - thread scheduling doesn't scale (+cost for thread context switching)
    - thread scheduling used as packet scheduling
  - use **event-driven I/O servers** like *Nginx*
    - do packet scheduling yourself – single/multi-threaded event loop
    - using **non-blocking** (asynchronous) operations with **event interceptors**
    - **multi-core scalability** with controlled number of worker threads
    - reuse thread-based data structures, avoid locks (atomics, non-blocking)

Threads

» **processes** vs. **threads**

- both support concurrent execution
- one process has one or multiple threads
- threads share the same address space (data and code)
- context switching between threads is usually less expensive
- thread inter-communication is relatively efficient using shared memory

» **JVM**

- a thread executes sequence of code with own stack with frames

  t.getStackTrace()

- own local variables
- own method parameters

» thread creation by

- subclass of **java.lang.Thread**
- implementation of **java.lang.Runnable**

JAVA Thread Pool - ExecutorService

- concept of **thread pooling**
- suitable for execution of large number of asynchronous tasks
  - e.g. processing of requests in server
- reduce overhead with Thread creation for each task, context switching
- interface - java.util.concurrent.ExecutorService
  - shutdown(), shutdownNow(), awaitTermination
  - **execute**(Runnable r)
  - Future<?> **submit**(Runnable r), Future<T> **submit**(Callable<T> c)
- java.util.concurrent.Future<T>
  - boolean cancel(boolean mayInterruptIfRunning)
  - isCancelled(), isDone()
  - V get(), V get(long timeout, TimeUnit unit)
- java.util.concurrent.Executors (optionally with ThreadFactory)
  - **newSingleThreadExecutor**()
  - **newFixedThreadPool**(nThreads)
  - **newCachedThreadPool**() – default 60 seconds keep-alive

Non-Blocking I/O Approach

- **polling**
  - looping to regularly check status (readiness for I/O)
  - wastes CPU cycles
- **signals**
  - OS generated signals on I/O readiness
  - might leave state inconsistent in the process inconsistent
- **callbacks**
  - pointer to handler function
  - stack deepening issue (callback issuing I/O)
- **interrupts**
  - hardware interrupts in kernel mode
- **event-based**
  - select
  - poll
  - epoll

Event-Based I/O - select

» **select**

- defined in POSIX (Portable Operating System Interface)
- originally used for blocking I/O
- passed lists of descriptors cannot be reused in subsequent calls as they are modified by the system call
- not scalable – limited descriptors + iterate over to find the event

```
int
select(int nfds, fd_set *restrict readfds, fd_set *restrict writefds, fd_set *restrict errorfds,
struct timeval *restrict timeout);

void
FD_CLR(fd, fd_set *fdset);

void
FD_COPY(fd_set *fdset_orig, fd_set *fdset_copy);

int
FD_ISSET(fd, fd_set *fdset);

void
FD_SET(fd, fd_set *fdset);

void
FD_ZERO(fd_set *fdset);
```

Event-Based I/O - poll

» **poll**

- polled descriptors not limited
- descriptors can be reused
- better but you still need iterate over descriptors to find events

```
int
poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

```
struct pollfd {
    int    fd;       /* file descriptor */
    short  events;   /* events to look for */
    short  revents;  /* events returned */
};
```

Event-Based I/O - epoll

- **epoll**
  - Linux only (e.g. Windows has IOCP – IO Completion Ports)
  - scalable
  - monitored events can be modified while polling (via syscall)
  - returns triggered events directly
- **API**
  - epoll_create & epoll_create1 – initialize epoll instance (kerne structure)
  - epoll_ctl – add/modify/remove descriptors to epoll instance
  - epoll_wait – wait for events up to timeout
- **modes**
  - **level triggered** – wait always returns if event is available
  - **event triggered** (EPOLLET) – readiness returned upon incoming event only
    (you have to process all pending events before next wait !)
- **events**
  - EPOLLIN, EPOLLOUT, EPOLLPRI
  - EPOLLRDHUP, EPOLLHUP
  - EPOLLERR

Epoll Usage

epoll structure:

```
typedef union epoll_data
{
  void        *ptr;
  int          fd;
  __uint32_t   u32;
  __uint64_t   u64;
} epoll_data_t;

struct epoll_event
{
  __uint32_t   events; /* Epoll events */
  epoll_data_t data;   /* User data variable */
};
```

initialization:

```
int epfd = epoll_create1(0);
...
struct epoll_event ev;
int client_sock;
...
ev.events = EPOLLIN | EPOLLPRI | EPOLLERR | EPOLLHUP;
ev.data.fd = client_sock;
int res = epoll_ctl(epfd, EPOLL_CTL_ADD, client_sock, &ev);
```

Epoll Event Loop

```
while (1) {
    // wait for something to do...
    int nfds = epoll_wait(epfd, events,
                                MAX_EPOLL_EVENTS_PER_RUN,
                                EPOLL_RUN_TIMEOUT);
    if (nfds < 0) die("Error in epoll_wait!");

    // for each ready socket
    for(int i = 0; i < nfds; i++) {
      int fd = events[i].data.fd;
      handle_io_on_socket(fd);
    }
}
```

JAVA Networking – TCP Client

» **Socket**

- client end-point of network TCP/IP connection
- is bound to particular destination IP and port
- each TCP/IP connection is uniquely identified by its two end-points
- provides input/output streams

```
try (
    Socket echoSocket = new Socket( host: "localhost",  port: 7);
    PrintWriter out = new PrintWriter(echoSocket.getOutputStream(),  autoFlush: true);
    BufferedReader in = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));
    BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in))
) {

    String userInput;

    while ((userInput = stdIn.readLine()) != null) {
        out.println(userInput);
        System.out.println("echo: " + in.readLine());
    }
}
```

JAVA Networking – TCP Server

» **ServerSocket**

- special socket representing listening TCP/IP end-point
- within constructor you specify the port, and optionally IP where it has to be bound
- wait for establishing connection using method Socket **accept**()

JAVA Networking – TCP Server - Example

threading server example – each handler runs in own thread with blocking I/O

```
ExecutorService clientRunner = Executors.newCachedThreadPool();
try (
        ServerSocket serverSocket = new ServerSocket( port: 7)
) {
    while (true) {
        final Socket s = serverSocket.accept();
        clientRunner.execute(() -> {
            try (
                    BufferedReader in = new BufferedReader(new InputStreamReader(s.getInputStream()));
                    PrintWriter out = new PrintWriter(s.getOutputStream(), autoFlush: true)
            ) {
                String line;
                while (s.isConnected()) {
                    if ((line = in.readLine()) != null) {
                        out.println(line);
                    }
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        });
    }
} catch (Exception e) {
    e.printStackTrace();
} finally {
    clientRunner.shutdownNow();
}
```

JAVA Networking - UDP

» **DatagramPacket**

- independent, self-contained message sent over the network
- like packet
  - InetAddress address, int port – destination
  - byte data[], int length, int offset
  - SocketAddress sa – sender

» **DatagramSocket**

- sending or receiving point for a packet delivery service
- can be bound to any available port (using default constructor)
- connect(InetAddress,int) – can sent or receive packets only specified host, if not set in DatagramPacket automatically fill
- send(DatagramPacket p), receive(DatagramPacket p) – blocking IO

» **MulticastSocket**

- additional capabilities for joining/leaving multicast groups, loopback
- multicast IP (IGMP – Internet Group Management Protocol)

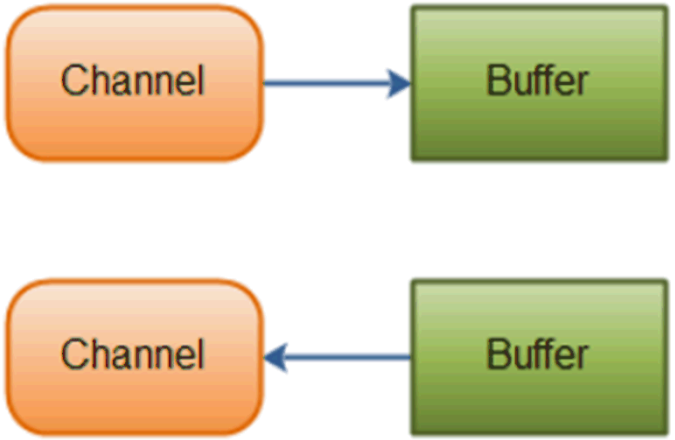
  224.0.0.0 – 239.255.255.255

JAVA Networking - NIO

» **scalable I/O** – asynchronous I/O requests and polling

» high-speed **block-oriented** binary and character I/O working – including mapping files to the memory, using channels and selectors

» Channel is like a block device working with Buffers

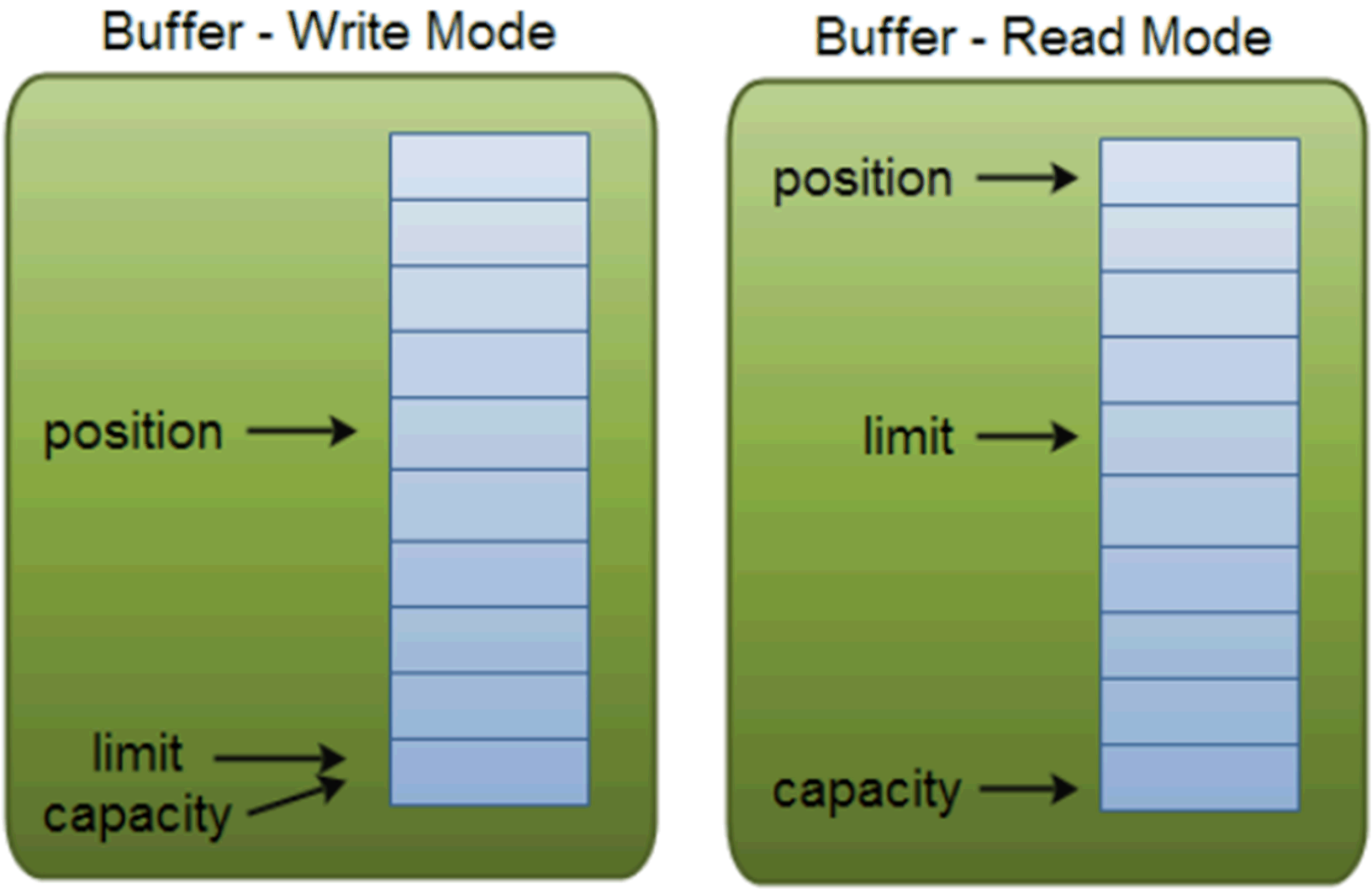


JAVA – NIO - Buffer

» java.nio.Buffer

- **linear, finite sequence of elements** of a specific primitive type
  - ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, ShortBuffer, MappedByteBuffer {FileChannel.map(…)}
- not thread safe, **multi mode** for the same buffer (both read & write)
- key properties – 0 <= mark <= position <= limit <= capacity
  - capacity – numbers of elements, never changing !
  - limit – index of the first element that should not be read or written
  - position – index of the next element to be read or written
  - mark – index to which its position is set after reset()
- clear() – position=0, limit=capacity => ready for channel read (put)
- flip() – limit=position, position=0 => ready for channel write (get)
- rewind() – limit unchanged, position=0 => ready for re-reading
- mark() – mark = position
- reset() – position=mark

JAVA – NIO - Buffer



» write mode – channel.read(buf); buf.put(…);

» read mode – channel.write(buf); … buf.get();

JAVA – NIO - Buffer

» java.nio.Buffer

- isReadOnly() – can be read-only
- hasArray() – is backed by an accessible array (array())
- equals(), compareTo() – compare remainder sequence

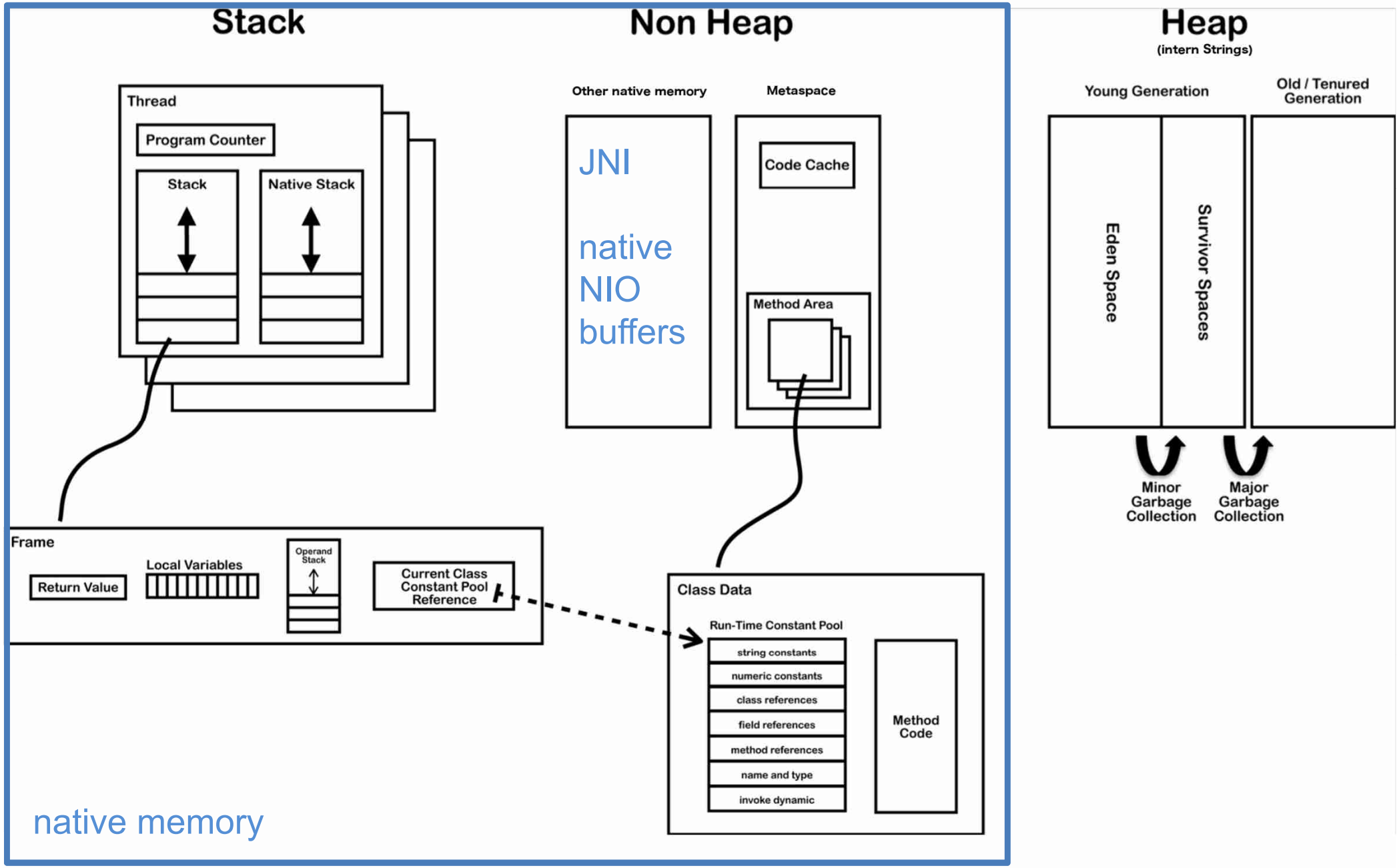
- can be **allocated to native memory** (see next slide)

- **typical usage**

1. Write data into the Buffer
2. Call `buffer.flip()`
3. Read data out of the Buffer
4. Call `buffer.clear()` or `buffer.compact()`

Note: compact() – bytes between position and limit are copied to the beginning of the buffer and prepare for writing again

JVM – Memory Layout – Native Memory

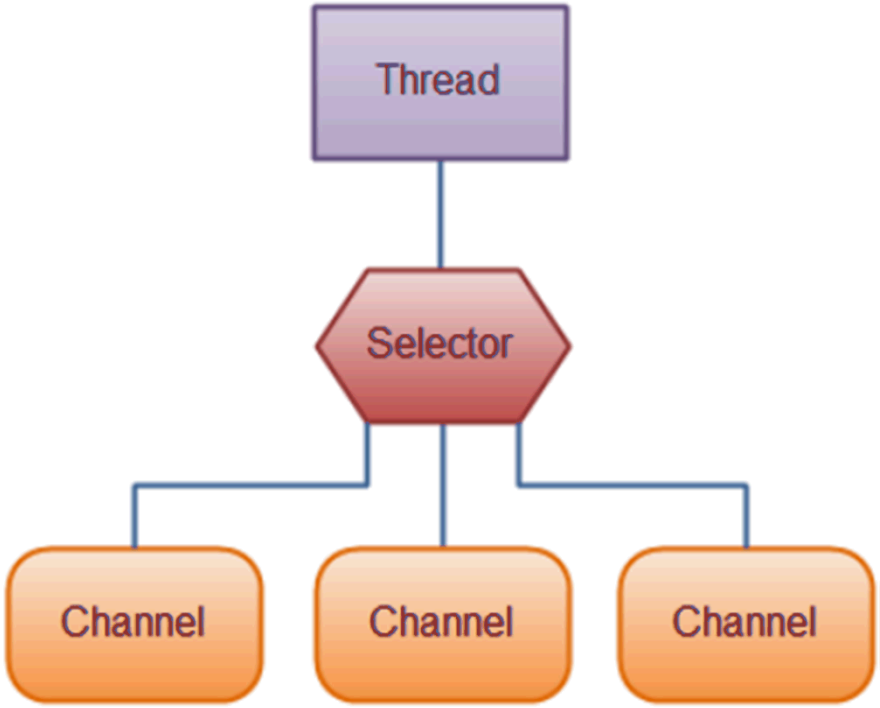


JVM – NIO - Direct Buffers

- **ByteBuffer.allocateDirect**(…)
- stored out of JAVA heap in **native memory**
- allow native code and Java code to **share data without copying**
  - useful for file and socket
    - the same memory is passed to kernel during calls
- multiple buffers can share native memory
  - slice()/duplicate() – independent position, limit, mark, shared content
  - asReadOnlyBuffer() – read only view of shared content
- tuning/tracking
  - -XX:MaxDirectMemorySize=N (default unlimited)
  - -XX:NativeMemoryTracking=off|summary|detail
  - -XX:+PrintNMTStatistics

<u>Note</u>: usage of heap buffers implies content copy out/in Java heap space due to possible relocations by GC

JAVA Networking - NIO – Channel, Selector



» **one thread** works with **multiple channels at the same time**
  - **epoll-based** if OS support epoll

» **Channel** – cover UDP+TCP network IO, file IO
  - FileChannel – from Input/OutputStream or RandomAccessFile
  - DatagramChannel
  - MulticastChannel
  - SocketChannel
  - ServerSocketChannel

JAVA – NIO – Channel

» **Channel**

- read/write at the same time (streams are only one-way)
- always read/write from/to a **buffer**

» **FileChannel**

- only **blocking**
- support – direct buffers, mapped files, locking
- bulk transfers between channels
  - no copy at all, direct transfer e.g. to socket
  - **transferFrom**(sourceChannel, int pos, int count)
  - **transferTo**(int pos, int count, dstChannel)

JAVA – NIO – Channel

» **SocketChannel** – client end-point of TCP/IP

- can be configured as non-blocking before connecting
- SocketChannel socket.getChannel();
- SocketChannel SocketChannel.open();
- sch.connect(…)

- write(…) and read(…) may return without having written/read anything for non-blocking channel

» **ServerSocketChannel** – server end-point of TCP/IP

- can be configured as non-blocking
- can be created directly using open() or from ServerSocket
- accept() – returns SocketChannel in the same mode

JAVA – NIO – Selector

» **Selector**

- Selector Selector.open();
- only channels in **non-blocking** mode can be registered

  channel.configureBlocking(false);

  SelectionKey channel.register(selector, SelectionKey.OP_READ);
- FileChannel doesn't support non-blocking mode

» **SelectionKey** – events you can listen for (can be combined together)

- OP_CONNECT
- OP_ACCEPT
- OP_READ
- OP_WRITE

» events are filled by channel which is ready with operation

JAVA – NIO – Selector

» **SelectionKey** – returned from register method

- interest set – your configured ops
- ready set – which ops are ready, sk.isReadable(), sk.isWritable(), …
- channel
- selector
- optional attached object – sk.attach(Object obj);
  Object sk.attachment()

SelectionKey channel.register(selector, ops, attachmentObj);

JAVA – NIO – Selector

» Selector with registered one or more channels

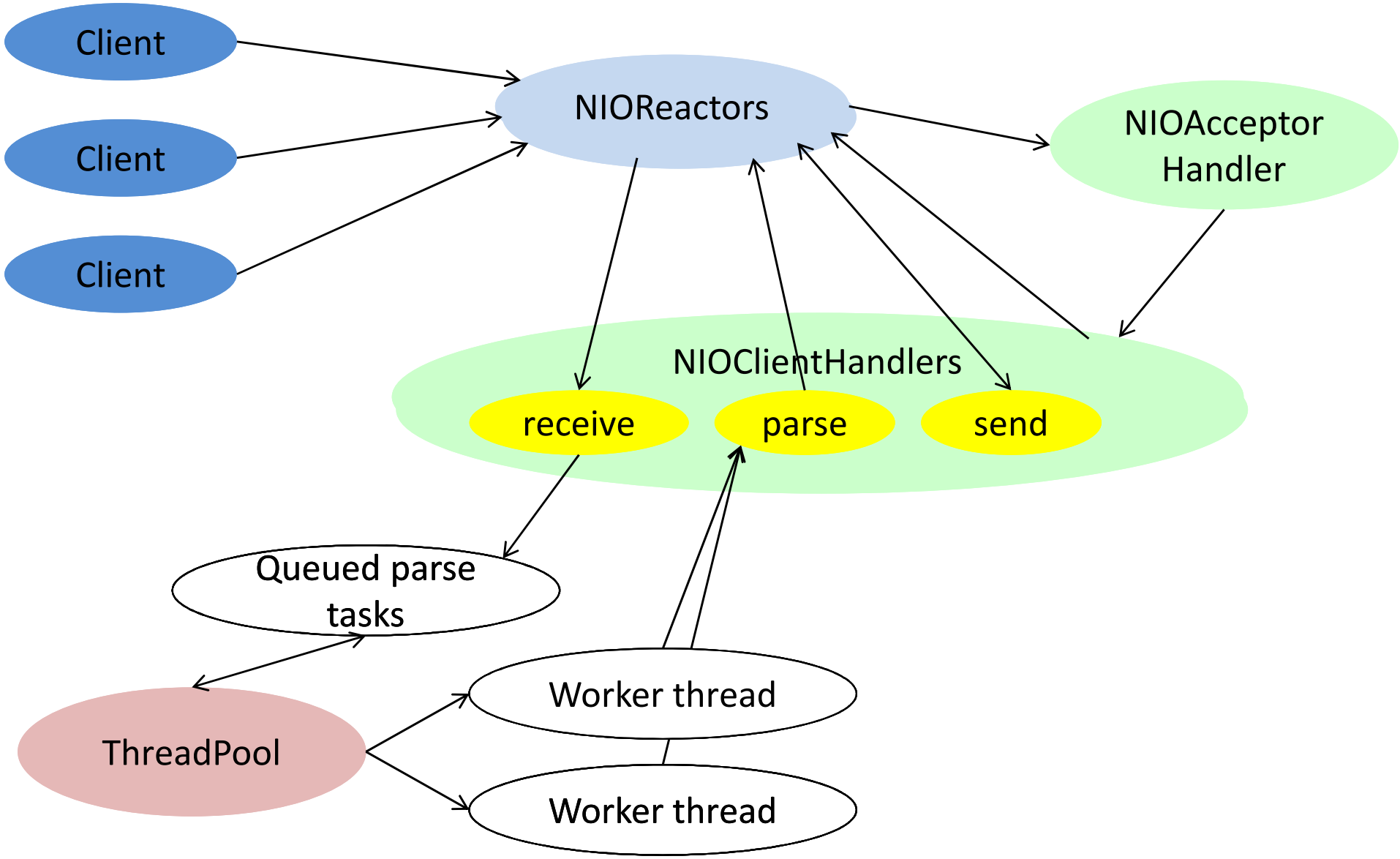
- int **select**() – blocks until at least one channel is ready
- int select(long timeout) – with timeout milliseconds
- int selectNow() – doesn't block at all, returns immediately

return the number of channels which are ready from the last call

Set<SelectionKey> selector.selectedKeys();

- **wakeUp**() – different thread can "wake up" thread blocked in select()
- **close**() – invalidates selector, channels are not closed

JAVA – NIO Server – Using Multiple Reactors



JAVA – NIOServer Example

```java
public class NIOServer {
    final static int MSG_SIZE = 1_000_000;
    private final static NIOReactor[] reactors;
    static ExecutorService workers = Executors.newWorkStealingPool();
    static BufferPool bufferPool = new BufferPool();

    static {
        reactors = new NIOReactor[4];
        try {
            for (int i=0; i<reactors.length; i++) {
                    reactors[i] = new NIOReactor();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void startNIOServer(int port) throws IOException {
        reactors[0].register(NIOAcceptorHandler.getNIOAcceptorHandler(reactors, port));
    }

    static class BufferPool {...}
}
```

JAVA – NIOReactor Example

```java
class NIOReactor implements Runnable {
    private final Selector s = Selector.open();
    private final ConcurrentLinkedQueue<NIOHandler> toRegister = new ConcurrentLinkedQueue<>();

    NIOReactor() throws IOException {
        Thread t = new Thread( target: this);
        t.setDaemon(true);
        t.start();
    }

    void register(NIOHandler target) {
        toRegister.add(target);
        s.wakeup();
    }

    @Override
    public void run() {
        try {
            while (true) {
                s.select();
                for (SelectionKey key : s.selectedKeys()) {
                    if (key.attachment() != null) ((NIOHandler) key.attachment()).run();
                }
                s.selectedKeys().clear();
                NIOHandler t;
                while ((t=toRegister.poll()) != null) {
                    t.setSelectionKey(t.getSelectableChannel().register(s, t.getInitialSelectableOps(), t));
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

JAVA – NIOHandler Example

```java
abstract class NIOHandler implements Runnable {
    private final SelectableChannel selectableChannel;
    private final int initialSelectableOps;
    private SelectionKey selectionKey = null;

    NIOHandler(SelectableChannel selectableChannel, int initialSelectableOps) {
        this.selectableChannel = selectableChannel;
        this.initialSelectableOps = initialSelectableOps;
    }

    SelectableChannel getSelectableChannel() {
        return selectableChannel;
    }

    int getInitialSelectableOps() {
        return initialSelectableOps;
    }

    SelectionKey getSelectionKey() {
        return selectionKey;
    }

    void setSelectionKey(SelectionKey selectionKey) {
        this.selectionKey = selectionKey;
    }
}
```

JAVA – NIOAcceptorHandler Example

```java
class NIOAcceptorHandler extends NIOHandler {
    private final NIOReactor[] reactors;
    private final ServerSocketChannel ssch;
    private int roundRobin = 0;

    static NIOAcceptorHandler getNIOAcceptorHandler(NIOReactor[] reactors, int port) throws IOException {
        ServerSocketChannel ssch = ServerSocketChannel.open();
        ssch.socket().bind(new InetSocketAddress(port));
        ssch.configureBlocking( block: false);
        return new NIOAcceptorHandler(reactors, ssch, SelectionKey.OP_ACCEPT);
    }

    private NIOAcceptorHandler(NIOReactor[] reactors, ServerSocketChannel ssch, int selectableOps) {
        super(ssch, selectableOps);
        this.reactors = reactors;
        this.ssch = ssch;
    }

    @Override
    public void run() {
        try {
            SocketChannel sch = ssch.accept();
            if (sch != null) {
                reactors[roundRobin].register(new NIOClientHandler(sch));
                roundRobin = (roundRobin+1)%reactors.length;
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }

    }
}
```

JAVA – NIOClientHandler Example

```
class NIOClientHandler extends NIOHandler {
    private final SocketChannel socketChannel;
    private ByteBuffer readBuf;
    private ByteBuffer writeBuf = null;

    NIOClientHandler(SocketChannel socketChannel) throws IOException {
        super(socketChannel, SelectionKey.OP_READ);
        this.socketChannel = socketChannel;
        readBuf = NIOServer.bufferPool.getBuffer();
        socketChannel.configureBlocking( block: false);
    }

    @Override
    public void run() {
        try {
            if (getSelectionKey().isReadable()) read();
            else if (getSelectionKey().isWritable()) write( setWriteInterest: false);
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    private void read() throws IOException {...}

    private void process() {...}

    private void write(boolean setWriteInterest) throws IOException {...}
}
```

JAVA – NIOClientHandler Example

```java
private void read() throws IOException {
    if (socketChannel.read(readBuf) == -1) {
        getSelectionKey().cancel();
        socketChannel.close();
    } else if (readBuf.remaining() == 0) {
        getSelectionKey().interestOps(0);
        getSelectionKey().selector().wakeup();
        NIOServer.workers.execute(this::process);
    }
}

private void process() {
    try {
        readBuf.flip();
        writeBuf = NIOServer.bufferPool.getBuffer();

        // DO processing and prepare data in writeBuf
        writeBuf.put(readBuf);
        writeBuf.flip();

        NIOServer.bufferPool.releaseBuffer(readBuf);
        readBuf = null;
        write( setWriteInterest: true);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

JAVA - NIOClientHandler Example

```java
private void write(boolean setWriteInterest) throws IOException {
    if (socketChannel.write(writeBuf) == -1) {
        getSelectionKey().cancel();
        socketChannel.close();
    } else if (writeBuf.remaining() > 0) {
        if (setWriteInterest) {
            getSelectionKey().interestOps(SelectionKey.OP_WRITE);
        }
    } else {
        readBuf = writeBuf;
        readBuf.clear();
        writeBuf = null;
        getSelectionKey().interestOps(SelectionKey.OP_READ);
    }
    if (setWriteInterest) {
        getSelectionKey().selector().wakeup();
    }
}
```

JAVA - NIOClientHandler Example

```java
private void write(boolean setWriteInterest) throws IOException {
    if (socketChannel.write(writeBuf) == -1) {
        getSelectionKey().cancel();
        socketChannel.close();
    } else if (writeBuf.remaining() > 0) {
        if (setWriteInterest) {
            getSelectionKey().interestOps(SelectionKey.OP_WRITE);
        }
    } else {
        readBuf = writeBuf;
        readBuf.clear();
        writeBuf = null;
        getSelectionKey().interestOps(SelectionKey.OP_READ);
    }
    if (setWriteInterest) {
        getSelectionKey().selector().wakeup();
    }
}
```

B4M36ESW: Efficient software

Lecture 8: Data structure serialization, Remote Procedure Calls

Michal Sojka
michal.sojka@cvut.cz

April 8, 2019

Outline

1 Introduction

2 Less efficient data serialization
- XML
- JSON

3 Faster alternative (C/C++)

4 Data serialization “frameworks”
- CORBA
- Protobufs
- Cap’n’proto
- Apache Avro

Outline

1 Introduction

2 Less efficient data serialization
- XML
- JSON

3 Faster alternative (C/C++)

4 Data serialization “frameworks”
- CORBA
- Protobufs
- Cap’n’proto
- Apache Avro

Communication between programs

- Over network
  - Communication protocol (e.g. over TCP)
  - Structured data serialization (JSON, protobufs, ...)
  - Remote Procedure Call (RPC)
    1. Serialize procedure name and arguments
    2. Send request and wait for response
    3. Deserialize reponse
  - Remote Method Invocation (RMI)
    - Almost the same as RPC
- On local host
  - Single address space (threads)
    - Data structures in memory
    - Language type system helps you to avoid mistakes!
  - Different address spaces (processes)
    - Same as "over network"
    - Ideally zero-copy via shared memory
      In Linux: `shm_open(...)` and `mmap()`

Outline

1 Introduction

2 Less efficient data serialization
- XML
- JSON

3 Faster alternative (C/C++)

4 Data serialization “frameworks”
- CORBA
- Protobufs
- Cap’n’proto
- Apache Avro

Outline

1 Introduction

2 Less efficient data serialization
- XML
- JSON

3 Faster alternative (C/C++)

4 Data serialization “frameworks”
- CORBA
- Protobufs
- Cap’n’proto
- Apache Avro

XML

- eXtensible Markup Language

```
<employees>
    <employee>
        <firstName>John</firstName> <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName> <lastName>Smith</lastName>
    </employee>
    <employee>
        <firstName>Peter</firstName> <lastName>Jones</lastName>
    </employee>
</employees>
```

- Very high overhead (both size and computation)
- Complex parser

Outline

1 Introduction

2 Less efficient data serialization
- XML
- JSON

3 Faster alternative (C/C++)

4 Data serialization “frameworks”
- CORBA
- Protobufs
- Cap’n’proto
- Apache Avro

JSON

- JavaScript Object Notation

```
{"employees":[
    { "firstName":"John", "lastName":"Doe" },
    { "firstName":"Anna", "lastName":"Smith" },
    { "firstName":"Peter", "lastName":"Jones" }
]}
```

- lower overhead, simpler parser

json-c parser

https://github.com/json-c/json-c

```c
#include <json.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  struct json_tokener *tok = json_tokener_new();
  char buf[1024*1024];
  struct json_object *jobj;

  FILE *f = fopen("test.json", "r");

  do {
    size_t len = fread(buf, 1, sizeof(buf), f);
    jobj = json_tokener_parse_ex(tok, buf, len);
  } while (json_tokener_get_error(tok) == json_tokener_continue);
  fclose(f);
  return 0;
}
```

Profiling json-c

47 MB JSON file

- `perf stat ./bench-json-c`

```
 Performance counter stats for './bench-json-c':

      3001.802390      task-clock (msec)         #    0.974 CPUs utilized
              412      context-switches          #    0.137 K/sec
                5      cpu-migrations            #    0.002 K/sec
          478,891      page-faults               #    0.160 M/sec
    9,368,533,705      cycles                    #    3.121 GHz
    3,377,028,216      stalled-cycles-frontend   #   36.05% frontend cycles idle
   14,910,459,852      instructions              #    1.59  insn per cycle
                                                 #    0.23  stalled cycles per insn
    3,144,829,442      branches                  # 1047.647 M/sec
       31,808,151      branch-misses             #    1.01% of all branches

      3.082290868 seconds time elapsed
```

- `perf record --freq 10000 -e cycles ./bench-json-c`

```
 21.28%  bench-json-c  bench-json-c       [.] json_tokener_parse_ex
 10.67%  bench-json-c  bench-json-c       [.] _int_malloc
  9.28%  bench-json-c  bench-json-c       [.] _IO_vfscanf_internal
  4.30%  bench-json-c  bench-json-c       [.] __libc_calloc
  3.37%  bench-json-c  bench-json-c       [.] ____strtod_l_internal
  3.30%  bench-json-c  bench-json-c       [.] __memset_sse2_unaligned_erms
  3.05%  bench-json-c  [kernel.kallsyms]  [k] clear_page_c_e
  2.60%  bench-json-c  [kernel.kallsyms]  [k] page_fault
```

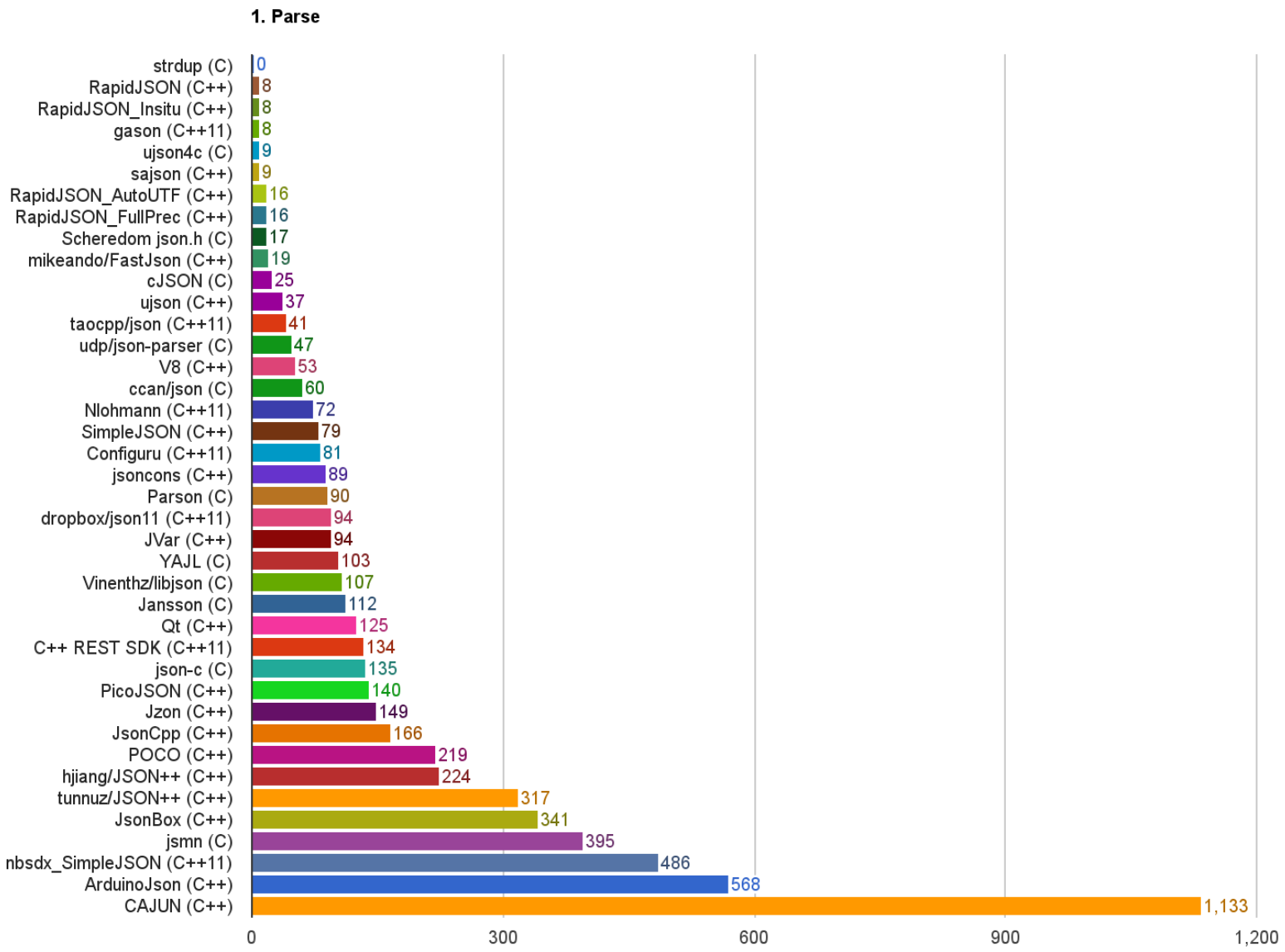
Where is time spent in `json_tokener_parse_ex`?

- Run `perf report` and let it *annotate* the function
- It shows C code intermixed with assembly and the percentage of profiling samples for each instruction

```
     |            while (isspace((int)c)) {
0.21 |        movsbq %dl,%rax
0.07 |        testb  $0x20,0x1(%rcx,%rax,2)
7.08 |      ↓ je     361
0.29 |        xchg   %ax,%ax
     |              if ((!ADVANCE_CHAR(str, tok)) || (!PEEK_CHAR(c, tok)))
0.02 | 330:   mov    0x20(%rbx),%eax
```

JSON benchmark

https://github.com/miloyip/nativejson-benchmark



Trying RapidJSON

- It's C++ ⇒ nicer to read syntax
- bench-rapidjson.cpp

```
#include <rapidjson/document.h>
#include <rapidjson/filereadstream.h>
using namespace rapidjson;

int main(int argc, char *argv[]) {
  FILE* fp = fopen("test.json", "r");
  char readBuffer[1024*1024];
  FileReadStream is(fp, readBuffer, sizeof(readBuffer));
  Document d;
  d.ParseStream(is);
  fclose(fp);
  return 0;
}
```

- perf stat bench-rapidjson

```
 Performance counter stats for './bench-rapidjson':

        389.890403      task-clock (msec)         #    0.998 CPUs utilized
                12      context-switches          #    0.031 K/sec
                 0      cpu-migrations            #    0.000 K/sec
            43,392      page-faults               #    0.111 M/sec
     1,106,686,422      cycles                    #    2.838 GHz
       206,781,432      stalled-cycles-frontend   #   18.68% frontend cycles idle
     2,467,762,722      instructions              #    2.23  insn per cycle
                                                  #    0.08  stalled cycles per insn
       593,437,567      branches                  # 1522.063 M/sec
            61,403      branch-misses             #    0.01% of all branches

       0.390790908 seconds time elapsed
```

What about spaces?

- perf record/report

```
 23.66%  bench-rapidjson    [.] rapidjson::GenericReader<...>::ParseString<0u, ra...
 22.43%  bench-rapidjson    [.] rapidjson::GenericReader<...>::ParseValue<0u, rap...
 18.94%  bench-rapidjson    [.] rapidjson::GenericReader<...>::ParseNumber<0u, ra...
 11.66%  bench-rapidjson    [.] rapidjson::SkipWhitespace<rapidjson::FileReadStream>
  5.70%  libc-2.24.so       [.] __memmove_sse2_unaligned_erms
  2.75%  bench-rapidjson    [.] rapidjson::GenericDocument<rapidjson::UTF8<char>, rapidjson::MemoryPool
  1.96%  [kernel.kallsyms]  [k] page_fault
  1.68%  [kernel.kallsyms]  [k] clear_page_c_e
```

- perf annotate rapidjson::GenericReader<...>::ParseString...

```
       |                 Ch c = is.Peek();
       |                 if (RAPIDJSON_UNLIKELY(c == '\\')) {    // Escape
 12.22 |  96:   cmp    $0x5c,%r14b
       |      ↓ je     178
       |                         TEncoding::Encode(os, codepoint);
       |                     }
       |                     else
       |                         RAPIDJSON_PARSE_ERROR(kParseErrorStringEscapeInvalid, escapeOffset);
       |                 }
       |                 else if (RAPIDJSON_UNLIKELY(c == '"')) {    // Closing double quote
  6.01 |        cmp    $0x22,%r14b
       |      ↓ je     200
       |                     is.Take();
       |                     os.Put('\0');   // null-terminate the string
       |                     return;
       |                 }
```

- What is RAPIDJSON_UNLIKELY?
  Branch predition hint (see `__builtin_expect()` in gcc manual)

Outline

1 Introduction

2 Less efficient data serialization
- XML
- JSON

3 Faster alternative (C/C++)

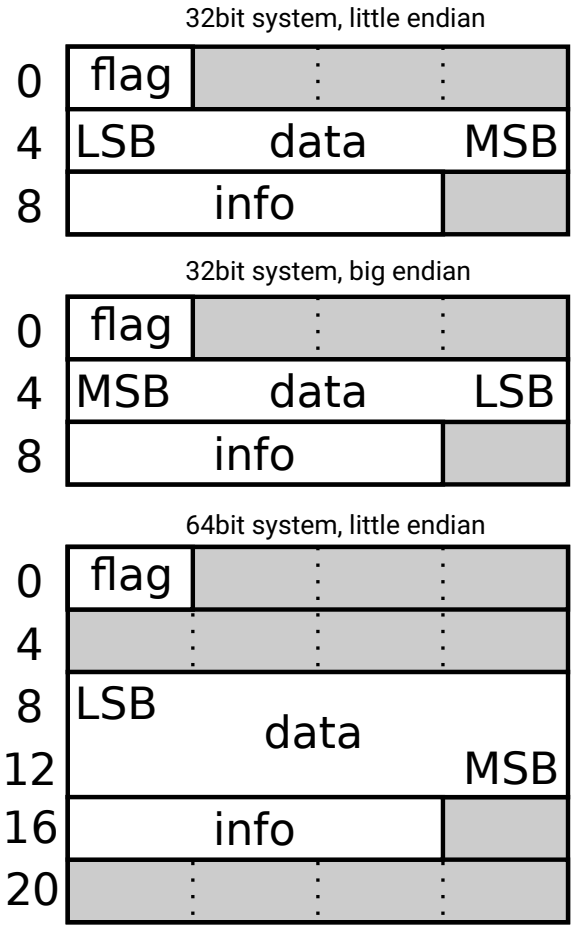
4 Data serialization “frameworks”
- CORBA
- Protobufs
- Cap’n’proto
- Apache Avro

Raw memory

- Sending/receiving directly the content of memory:

```
struct data {
    char flag;
    long int data;
    char info[3];
};

void sendData(struct data &d) {
    send(sock, &d, sizeof(d));
}
void recvData(struct data &d) {
    recv(sock, &d, sizeof(d));
}
```



Raw memory

Problems & solutions

- type size ⇒ `#include <stdint.h>` ⇒ `int32_t`
- endianing ⇒ `#include <endian.h>` ⇒ `htole32()` etc.
  (host to little-endian 32 bits)
- padding ⇒ `__attribute__((__packed__))`

-
```
struct __attribute__ ((__packed__)) data {
        char flag;
        int32_t data;
        char info[3];
};

void recvData(struct data &d) {
        struct data dd;
        recv(sock, &dd, sizeof(dd));
        d = dd;
        d.data = htole32(dd.data);
}
```

0	flag	LSB	data
4	MSB	info	

Raw memory

Properties

- Blazingly fast, but inflexible
- Receive side must know the format of data
  - What if sender has newer version than receiver?
    - e.g. field added/removed, type changed

Versioning of the protocol

Whenever you design a communication protocol, always include a version number (or feature flags) to allow older and newer versions to coexist. It saves you some pain when upgrading your software.

Outline

1 Introduction

2 Less efficient data serialization
- XML
- JSON

3 Faster alternative (C/C++)

4 Data serialization “frameworks”
- CORBA
- Protobufs
- Cap’n’proto
- Apache Avro

Outline

1 Introduction

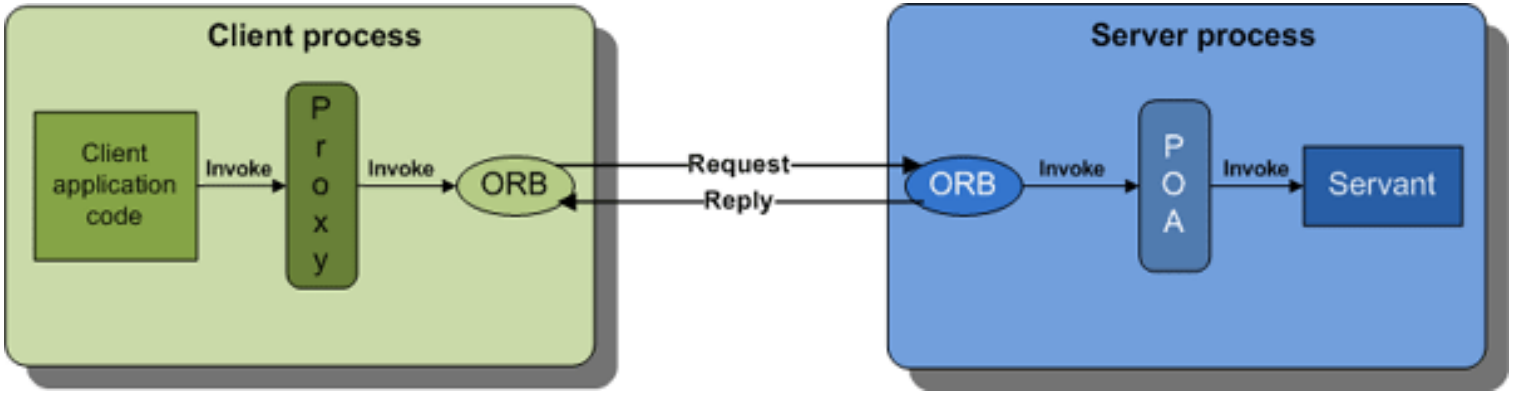
2 Less efficient data serialization
- XML
- JSON

3 Faster alternative (C/C++)

4 Data serialization “frameworks”
- CORBA
- Protobufs
- Cap’n’proto
- Apache Avro

Common Object Request Broker Architecture (CORBA)

- Language independent “RPC framework” from ’90
- Interface Description Language (IDL)
- Automatic generation of (de)serialization code (IDL compiler)
- Description of data structure is not normally sent with the data
- CORBA is not very popular today, perhaps because of its complexity and difficulty of using parts of it (such as CDR) independently



Interface Description Language (IDL)

- In different frameworks called “schema”
- Defines only data types and interfaces
- IDL compiler generates corresponding definitions in target language as well as conversion code to/from the CDR form.
- Example:

```
module Finance {
  typedef sequence<string> StringSeq;
  struct AccountDetails {
    string     name;
    StringSeq  address;
    long       account_number;
    double     current_balance;
  };
  exception insufficientFunds { };
  interface Account {
    void deposit(in double amount);
    void withdraw(in double amount) raises(insufficientFunds);
    readonly attribute AccountDetails details;
  };
};
```

Common Data Representation (CDR)

- Endian
  - Data is sent in sender’s endian
  - Message header specifies, which endian it is ⇒ no expensive endian conversion between similar hosts
- Data padding as in memory – efficient (de)serialization
- TypeCodes – CDR representation of any IDL data type
  - Allows to send Any data type (TypeCode + actual data) and receiver can reconstruct it

Outline

1 Introduction

2 Less efficient data serialization
- XML
- JSON

3 Faster alternative (C/C++)

4 Data serialization “frameworks”
- CORBA
- Protobufs
- Cap’n’proto
- Apache Avro

Google Protocol Buffers (protobufs)

https://developers.google.com/protocol-buffers/

- Data description – conceptually similar to IDL
- Automatic code generation
- Partial description of data sent with the data
  - Less porblems with protocol versioning
- Easy to use API
- Supports multiple languages: Java, Python, C++, C#, ...

```
syntax = "proto3";

message SearchRequest {
  string query = 1;
  int32  page_number = 2;
  int32  result_per_page = 3;
}
```

- Numbered “tags” uniquely identify fields
- Tags also help with maintaining backward compatibility

Wire encoding

- Key-value pairs
- Key = the tag + type information
- Unknown key-values can always be skipped
- Key: `(field_number << 3) | wire_type` (stored as varint)

Type	Meaning	Used For
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64-bit	fixed64, sfixed64, double
2	Length-delimited	string, bytes, embedded messages, packed repeated fields
3	Start group	groups (deprecated)
4	End group	groups (deprecated)
5	32-bit	fixed32, sfixed32, float

Wire encoding – Varint

- Encoded in variable number of bytes, small numbers take only one byte
- 7th bit is 1 in all but last byte.
- Bits 0–6 store the value.
- 9 = 0000 1001b → 0000 1001b
- 300 = 1 0010 1100b → 1010 1100 0000 0010
- Signed integers (sint) use ZigZag encoding:
  - `(n << 1) ^ (n >> 31)`

$$
\begin{array}{rcl}
0 & \rightarrow & 0 \\
-1 & \rightarrow & 1 \\
1 & \rightarrow & 2 \\
-2 & \rightarrow & 3 \\
 & \ldots &
\end{array}
$$

Wire encoding – String and Message

- Varint-encoded length + bytes of string/message
- ```
  message Test2 {
    required string b = 6;
  }
  ```
- b = “testing”
  - 32 07 74 65 73 74 69 6e 67
  - 32h = `(6 << 3) | 2  // 2 = length delimited`
  - 07h = length
  ```

Wire encoding – repeated fields

- ```
  message Test4 {
    repeated int32 d = 4 [packed=true];
  }
  ```
- ```
  0x22              // tag (field number 4, wire type 2)
  0x06              // payload size (6 bytes)
  0x03              // first element (varint 3)
  0x8E 0x02         // second element (varint 270)
  0x9E 0xA7 0x05    // third element (varint 86942)
  ```

Message streaming

- Parsing code does not know where a message begins and ends
- Solution: Put the length of the message before it

Protobuf example – OpenStreetMap

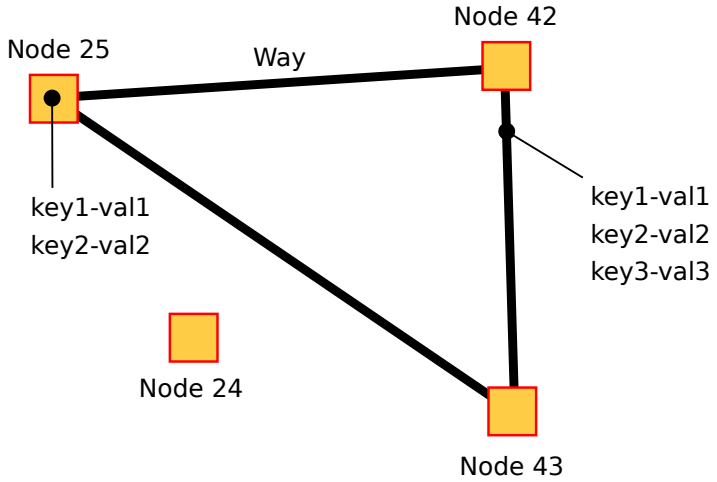
https://wiki.openstreetmap.org/wiki/PBF_Format

```
message Node {
  required sint64 id = 1;
  // Parallel arrays.
  repeated uint32 keys = 2 [packed = true]; // String IDs.
  repeated uint32 vals = 3 [packed = true]; // String IDs.
  optional Info info = 4; // May be omitted in omitmeta
  required sint64 lat = 8;
  required sint64 lon = 9;
}

message Way {
   required int64 id = 1;
   // Parallel arrays.
   repeated uint32 keys = 2 [packed = true];
   repeated uint32 vals = 3 [packed = true];

   optional Info info = 4;

   repeated sint64 refs = 8 [packed = true];  // DELTA coded
}
```



Czech republic: PBF – 670 MB, XML – 16 GB

From .proto to C++

addressbook.proto

```
package tutorial;

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

mycode.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include "addressbook.pb.h" // generated from .proto
using namespace std;

// Iterates though all people in the AddressBook and prints info about them.
void ListPeople(const tutorial::AddressBook& address_book) {
  for (int i = 0; i < address_book.person_size(); i++) {
    const tutorial::Person& person = address_book.person(i);

    cout << "Person ID: " << person.id() << endl;
    cout << "  Name: " << person.name() << endl;
    if (person.has_email()) {
      cout << "  E-mail address: " << person.email() << endl;
    }

    for (int j = 0; j < person.phone_size(); j++) {
      const tutorial::Person::PhoneNumber& phone_number = person.phones(j);

      switch (phone_number.type()) {
        case tutorial::Person::MOBILE:
          cout << "  Mobile phone #: ";
          break;
        case tutorial::Person::HOME:
          cout << "  Home phone #: ";
          break;
        case tutorial::Person::WORK:
          cout << "  Work phone #: ";
          break;
      }
      cout << phone_number.number() << endl;
    }
  }
}
```

Outline

1 Introduction

2 Less efficient data serialization
- XML
- JSON

3 Faster alternative (C/C++)

4 Data serialization “frameworks”
- CORBA
- Protobufs
- Cap’n’proto
- Apache Avro

Cap’n’proto

https://capnproto.org/

- Developed by the original author of protobufs
- Some years later – lessons learnt from protobufs
- Very efficient for communication via shared memory (e.g. between different languages)
- Still usable over network
- No de/encoding needed – serialized form is usable as a native form (if packing is not in use)

Cap’n’proto encoding

- Bool: 1 bit
- Integers: Little endian, native size, aligned to multiple of their size (padding)
- Default values: always encoded as zero, i.e. `enc = val ^ default`
- Optional packing = getting rid of zero bytes
  - Set bits in the first byte indicate which of the following 8 bytes are non-zero. The nonzero bytes follow.
  - `unpacked (hex): 08 00 00 00 03 00 02 00 19 00 00 00 aa 01 00 00`
    `packed (hex): 51 08 03 02 31 19 aa 01`
- Structures: Pointer (= index) to data and sub-structures

Message + structure encoding

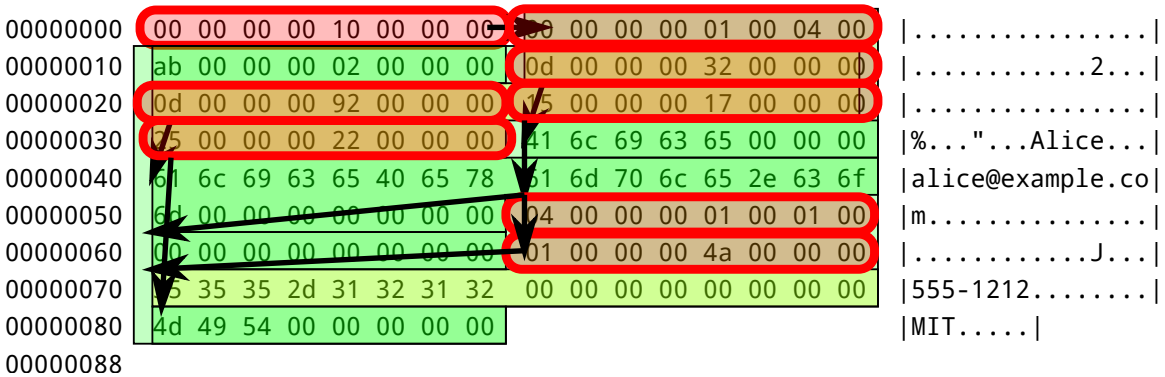
https://capnproto.org/encoding.html

```
struct Person {
  id @0 :UInt32;  # 0xab
  name @1 :Text;  # Alice
  email @2 :Text; # alice@example.com
  phones @3 :List(PhoneNumber);

  struct PhoneNumber {
    number @0 :Text; # "555-1212"
    type @1 :Type;   # mobile

    enum Type {
      mobile @0;
      home @1;
      work @2;
    }
  }

  employment :union {
    unemployed @4 :Void;
    employer @5 :Text;
    school @6 :Text; # MIT
    selfEmployed @7 :Void;
  }
}
```



- Tree-like data structure. Allows skipping of unknown or unwanted data.
- Packing allows getting rid of all zero bytes above and adds 17 more bytes.

From .capnp to C++

addressbook.capnp

```
struct Person {
  id @0 :UInt32;
  name @1 :Text;
  email @2 :Text;
  phones @3 :List(PhoneNumber);

  struct PhoneNumber {
    number @0 :Text;
    type @1 :Type;

    enum Type {
      mobile @0;
      home @1;
      work @2;
    }
  }

  employment :union {
    unemployed @4 :Void;
    employer @5 :Text;
    school @6 :Text;
    selfEmployed @7 :Void;
    # We assume that a person is only one of these.
  }
}

struct AddressBook {
  people @0 :List(Person);
}
```

mycode.cpp

```
#include "addressbook.capnp.h"
#include <capnp/message.h>
#include <capnp/serialize-packed.h>
#include <iostream>

void printAddressBook(int fd) {
  ::capnp::PackedFdMessageReader message(fd);

  AddressBook::Reader addressBook = message.getRoot<AddressBook>();

  for (Person::Reader person : addressBook.getPeople()) {
    std::cout << person.getName().cStr() << ": "
              << person.getEmail().cStr() << std::endl;
    for (Person::PhoneNumber::Reader phone: person.getPhones()) {
      const char* typeName = "UNKNOWN";
      switch (phone.getType()) {
        case Person::PhoneNumber::Type::MOBILE: typeName = "mobile"; break;
        case Person::PhoneNumber::Type::HOME: typeName = "home"; break;
        case Person::PhoneNumber::Type::WORK: typeName = "work"; break;
      }
      std::cout << "  " << typeName << " phone: "
                << phone.getNumber().cStr() << std::endl;
    }
    Person::Employment::Reader employment = person.getEmployment();
    switch (employment.which()) {
      case Person::Employment::UNEMPLOYED:
        std::cout << "  unemployed" << std::endl;
        break;
      case Person::Employment::EMPLOYER:
        std::cout << "  employer: "
                  << employment.getEmployer().cStr() << std::endl;
        break;
      case Person::Employment::SCHOOL:
        std::cout << "  student at: "
                  << employment.getSchool().cStr() << std::endl;
        break;
      case Person::Employment::SELF_EMPLOYED:
```

Outline

1 Introduction

2 Less efficient data serialization
- XML
- JSON

3 Faster alternative (C/C++)

4 Data serialization “frameworks”
- CORBA
- Protobufs
- Cap’n’proto
- Apache Avro

Apache Avro

- Schema in JSON
- Schema handshake after connection establishment
- No tags in data, because schema is known to all parties
- File storage
  - Compression
  - Blocks allowing skip through the data without deserialization

B4M36ESW: Efficient software

Lecture 9: Memory, caches, algorithms

Michal Sojka
michal.sojka@cvut.cz

April 15, 2019

Outline

1 Why is DRAM slow?

2 Caches
- Architecture
- Memory performance characteristics
- Data structures and dynamic memory allocations
- Matrix multiplications

3 Caches & memory in multi-processor systems
- True and false sharing
- NUMA

4 Conclusion

Outline

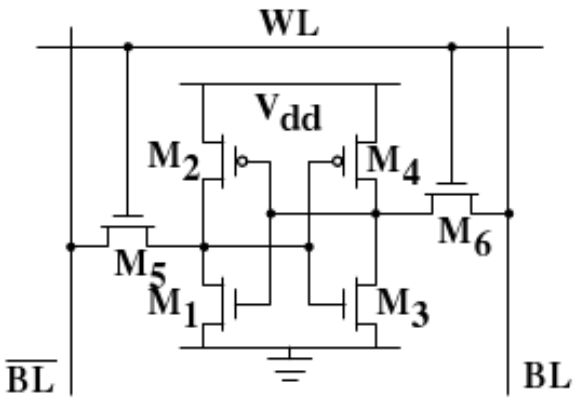
1 Why is DRAM slow?

2 Caches
- Architecture
- Memory performance characteristics
- Data structures and dynamic memory allocations
- Matrix multiplications

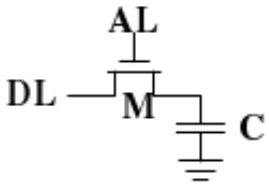
3 Caches & memory in multi-processor systems
- True and false sharing
- NUMA

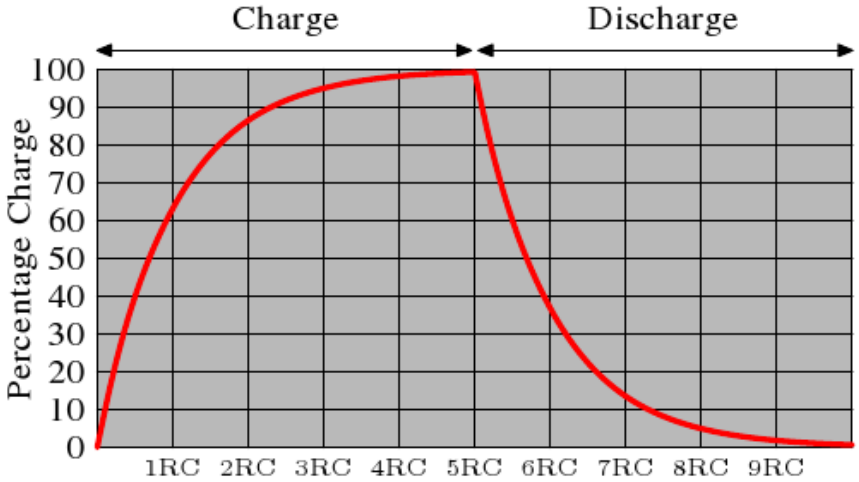
4 Conclusion

Types or RAM

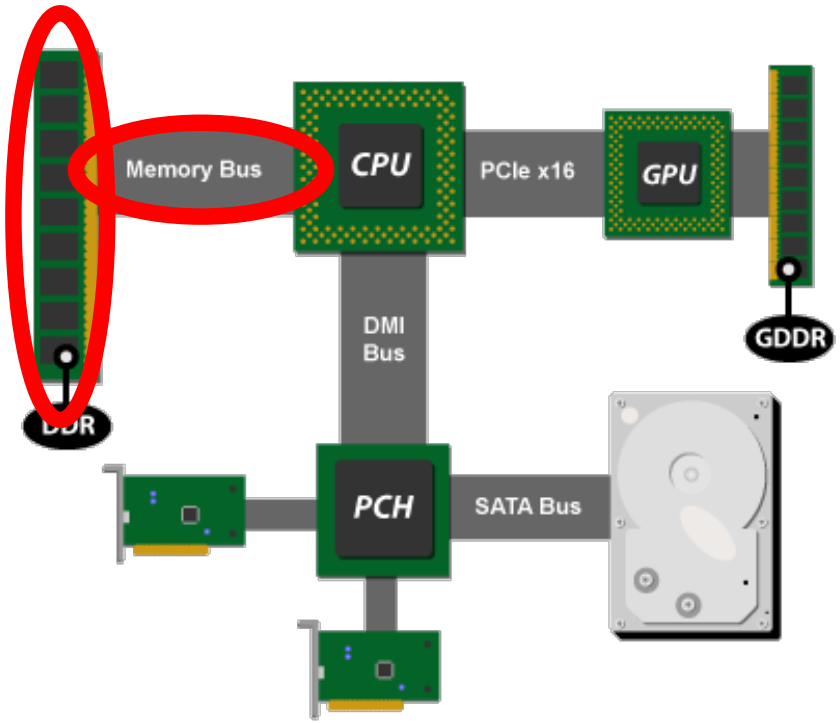


- Static RAM (SRAM)
  - Fast but expensive
  - 6 transistors per bit
- Dynamic RAM (DRAM)
  - Capacitor – (Dis)Charging is not instantaneous
  - Reading discharges capacitor (write after read)
  - Compromise: capacity/size/power consumption





DRAM in the computer

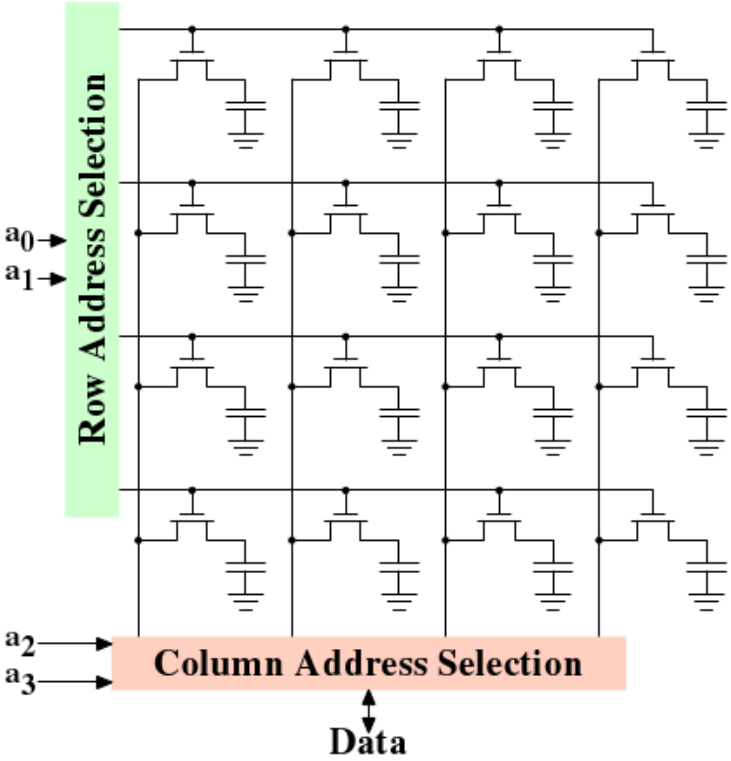


Intel's P55 platform
Source: ArsTechnica

- CPU contains a memory controller (MC)
- MC talks to DRAM chips via "Memory Bus", using a protocol
- Details on next slides

How DRAM chips work?

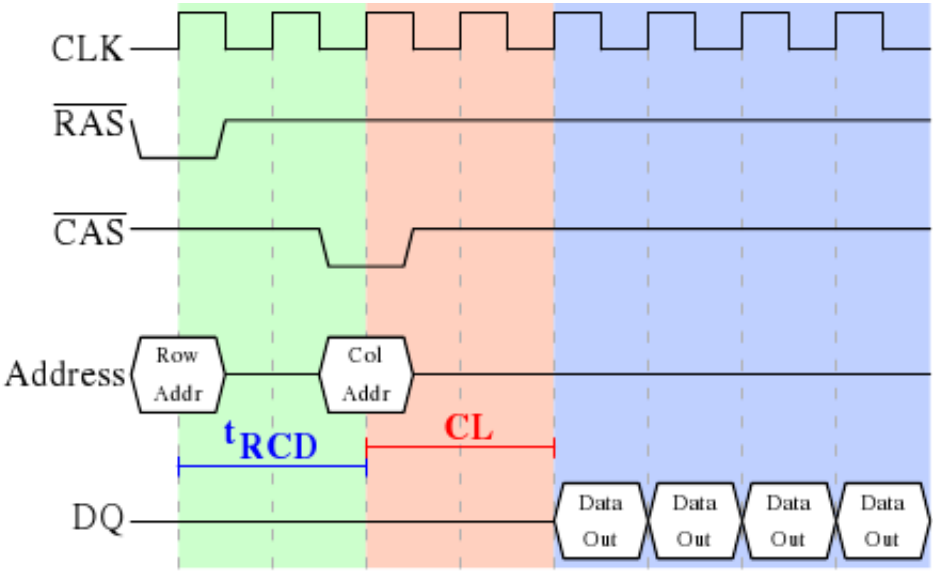
- Addressing individual cells is impractical (many wires)
- Chip is organized in rows and columns (and banks), address is multiplexed
- In the chip, row and column multiplexers (green and pink rectangles) select the lines according to address bits
- R/W operations happen in many chips in parallel to work with the whole data word (64 bits)
- Writing: New value is put on Data signal after row and column address were selected (see next slide)
  - It takes some time to charge the capacitors



(one bit)

SDRAM communication protocol

- Access protocol is synchronous – there is a clock signal
- SDRAM (Synchronous DRAM)
- CLK provided by memory controller (FSB frequency – typ. 800–1600 MHz)
  - Double/Quad-pumped



- Max. speed: 64 bit × 8 × 200 MHz = 12.8 GB/s
  - Not reachable in reality
- DRAM technology requires t_{RCD} and CL delays (they cannot be shortened)
- Data sent in bursts
  - Size of the burst corresponds to cache-line size
  - Sending just one word would be very inefficient due to t_{RCD} and CL delays

Timing parameters of standard DDR4 modules

Standard name	Memory clock (MHz)	I/O bus clock (MHz)	Data rate (MT/s)	Module name	Peak transfer rate (MB/s)	Timings, CL-tRCD-tRP	CAS latency (ns)
DDR4-1600J* DDR4-1600K DDR4-1600L	200	800	1600	PC4-12800	12800	10-10-10 11-11-11 12-12-12	12.5 13.75 15
DDR4-1866L* DDR4-1866M DDR4-1866N	233.33	933.33	1866.67	PC4-14900	14933.33	12-12-12 13-13-13 14-14-14	12.857 13.929 15
DDR4-2133N* DDR4-2133P DDR4-2133R	266.67	1066.67	2133.33	PC4-17000	17066.67	14-14-14 15-15-15 16-16-16	13.125 14.063 15
DDR4-2400P* DDR4-2400R DDR4-2400U	300	1200	2400	PC4-19200	19200	15-15-15 16-16-16 18-18-18	12.5 13.33 15

Source: Wikipedia

Outline

1 Why is DRAM slow?

2 Caches
- Architecture
- Memory performance characteristics
- Data structures and dynamic memory allocations
- Matrix multiplications

3 Caches & memory in multi-processor systems
- True and false sharing
- NUMA

4 Conclusion

Cache terminology

- Spatial locality: accessed memory objects are close to each other
  - Code: inner loops
  - Data: structures (reading of one field is often followed by reads of other fields)
- Temporal locality: The same data will be used multiple times in a short period of time
  - Code: loops
  - Data: e.g. digital filter coefficients are accessed every sampling period
- Cache hit: memory request is serviced from the cache, without going to higher level memory
- Cache miss: opposite of cache hit
  - cold miss, capacity miss, conflict miss
  - true sharing miss, false sharing miss
- Cache line eviction: cache line is removed from the cache to make space for new data
- Cache replacement policy: LRU, pseudo LRU, random

Outline

1 Why is DRAM slow?

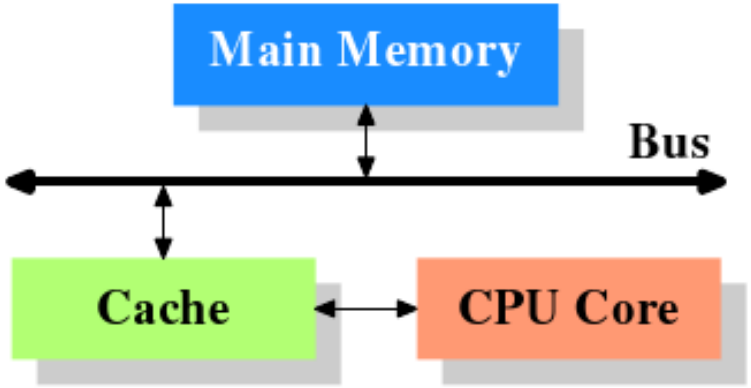
2 Caches
- Architecture
- Memory performance characteristics
- Data structures and dynamic memory allocations
- Matrix multiplications

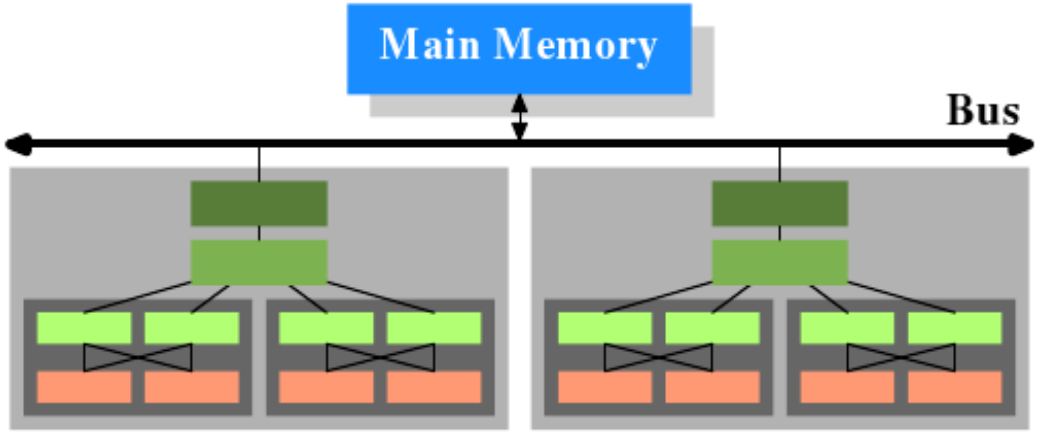
3 Caches & memory in multi-processor systems
- True and false sharing
- NUMA

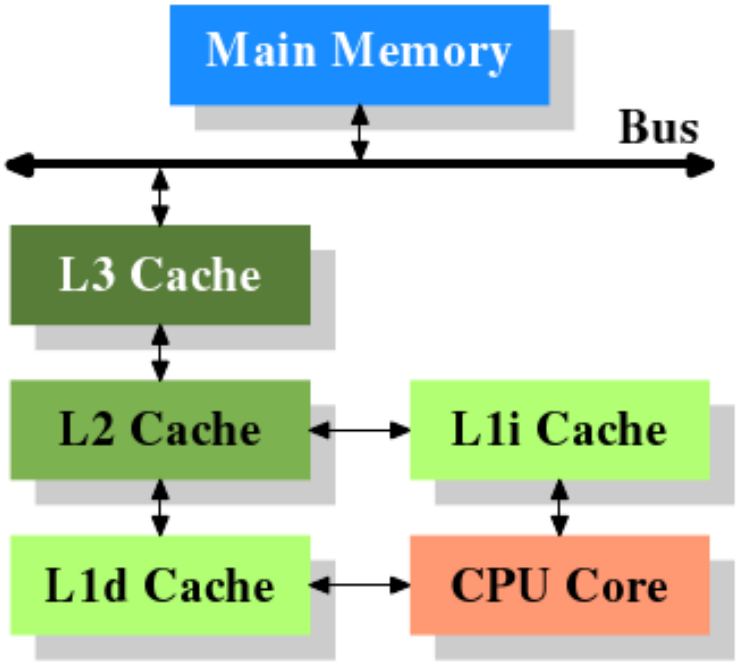
4 Conclusion

CPU caches – big picture

- All loads/stores go through cache
- CPU ↔ Cache: fast connection
- Cache ↔ Main memory: FSB bus
- It is advantage to have separate caches for instructions and data





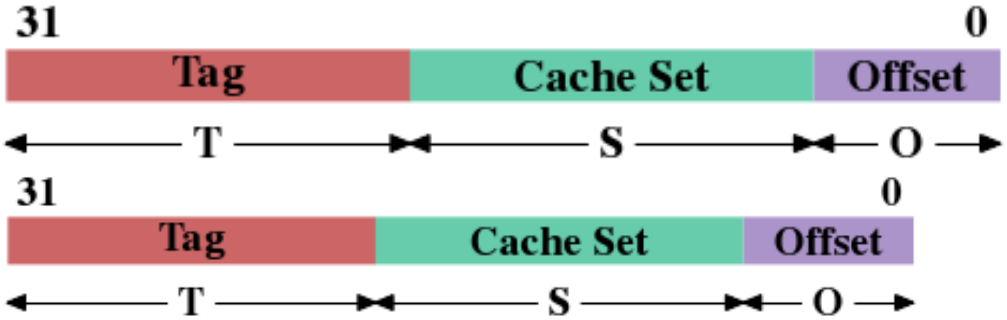


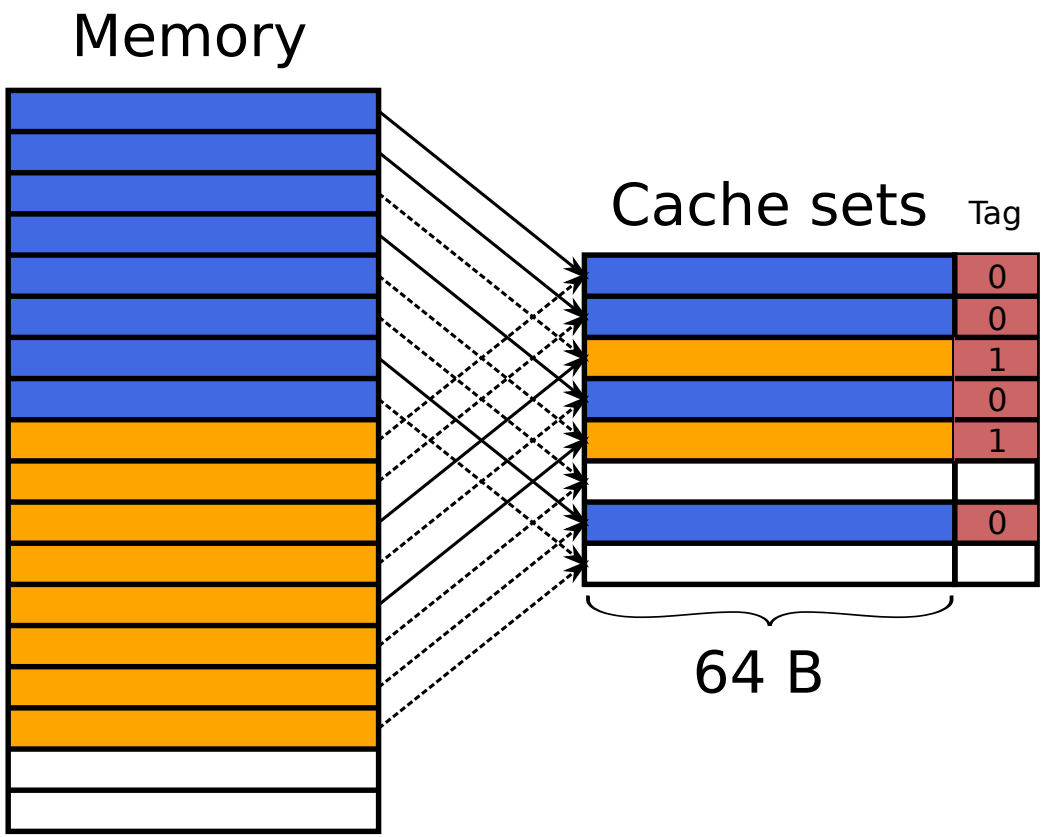
Cache associativity

- Direct-mapped cache
  - simple
- Fully associative cache
  - ideal
- Set associative cache
  - compromise

Direct-mapped cache

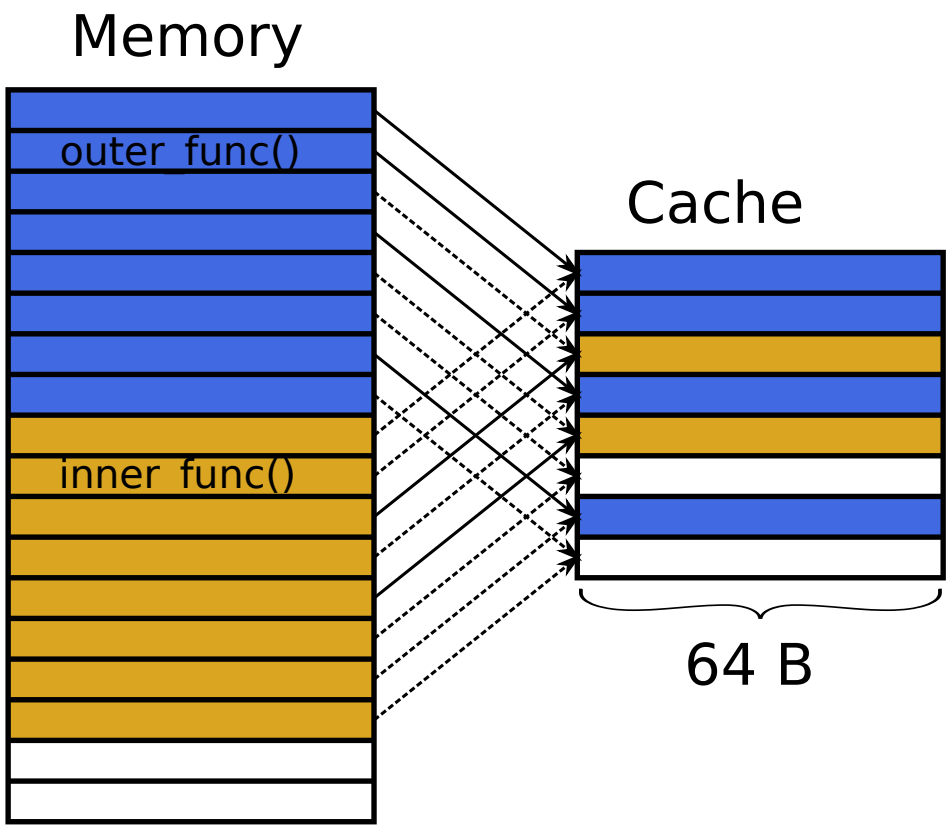
Address:





- Each memory location has just one cache line associated with it
- Memory locations at multiples of cache size always collide!
- Besides the data, cache stores the tag

Problem: Self-evicting of code




```
void outer_func() {
  for (int i = 0; i < 1000; i++)
    inner_func();
}
void inner_func() {
  // do something
}
```

- Two cache misses every iteration (instruction fetches)!
- Solution: Improve code layout by putting related (and hot) functions together.

```
__attribute__((hot)) void outer_func();
__attribute__((hot)) void inner_func();
```

Cache write policies

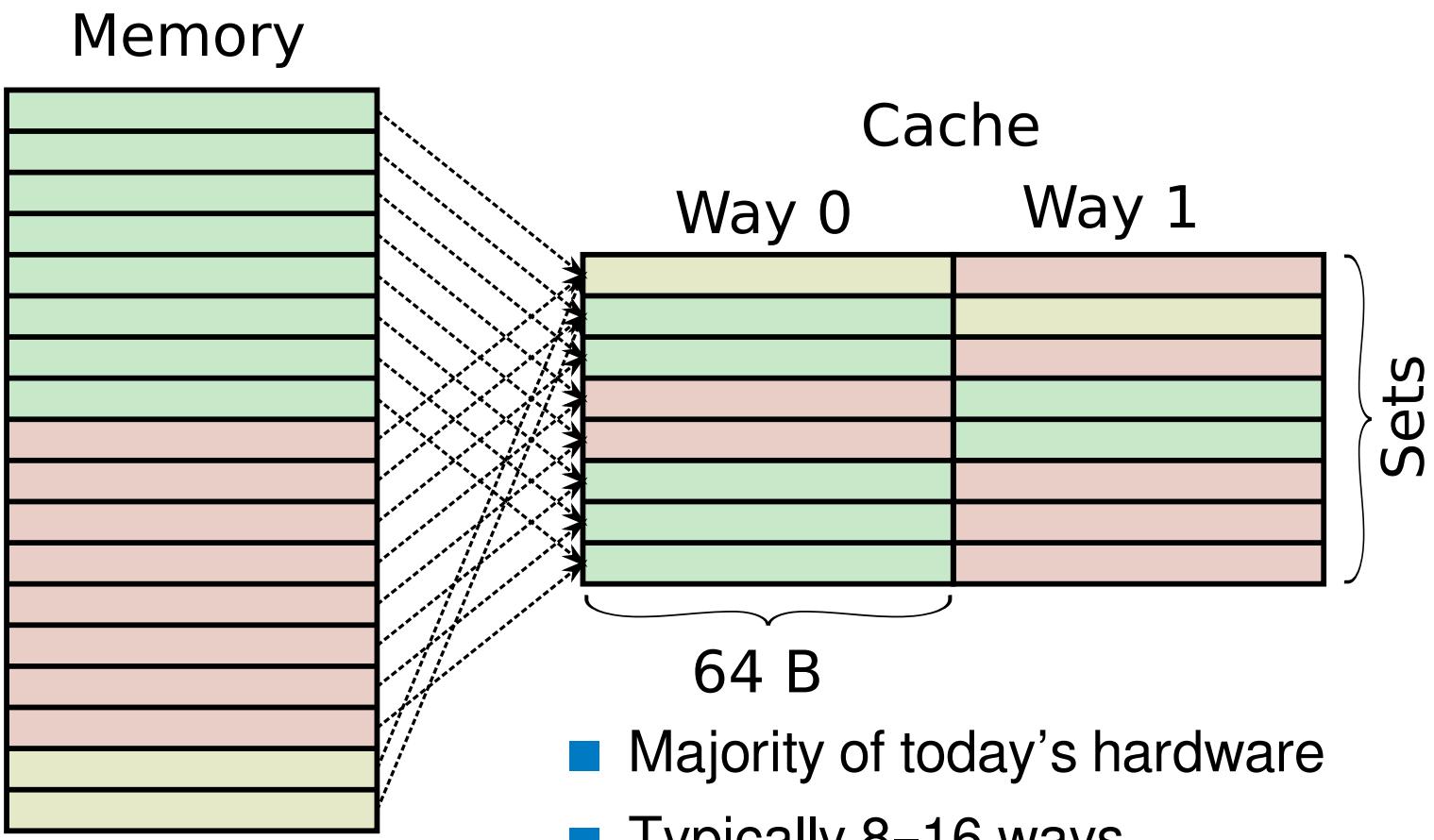
Write-back "Common" case. Written data is cached for later reuse.

Write-through Written data bypass the cache and therefore never evicts other data from the cache. Useful when we know the data will not be needed soon.

```
#include <emmintrin.h>
void _mm_stream_si32(int *p, int a);
void _mm_stream_si128(int *p, __m128i a);
void _mm_stream_pd(double *p, __m128d a);
#include <xmmintrin.h>
void _mm_stream_pi(__m64 *p, __m64 a);
void _mm_stream_ps(float *p, __m128 a);
#include <ammintrin.h>
void _mm_stream_sd(double *p, __m128d a);
void _mm_stream_ss(float *p, __m128 a);
```

Write-Combining All writes to the cache line are combined together and written at once. This avoids one memory read, because when the cache line is fully overwritten, there is no point in reading the old value. Write combining is often used for frame buffer memory (e.g. filling the screen with a color).

Set associative caches



- Majority of today's hardware
- Typically 8–16 ways
- Cache replacement policy – determines which way is evicted
  - Examples: LRU, random, …

Self-evicting code and set associative caches

- Does the problem of self evicting code exist with set associative caches?
- Yes, but it is less likely to occur.
- Why?

Outline

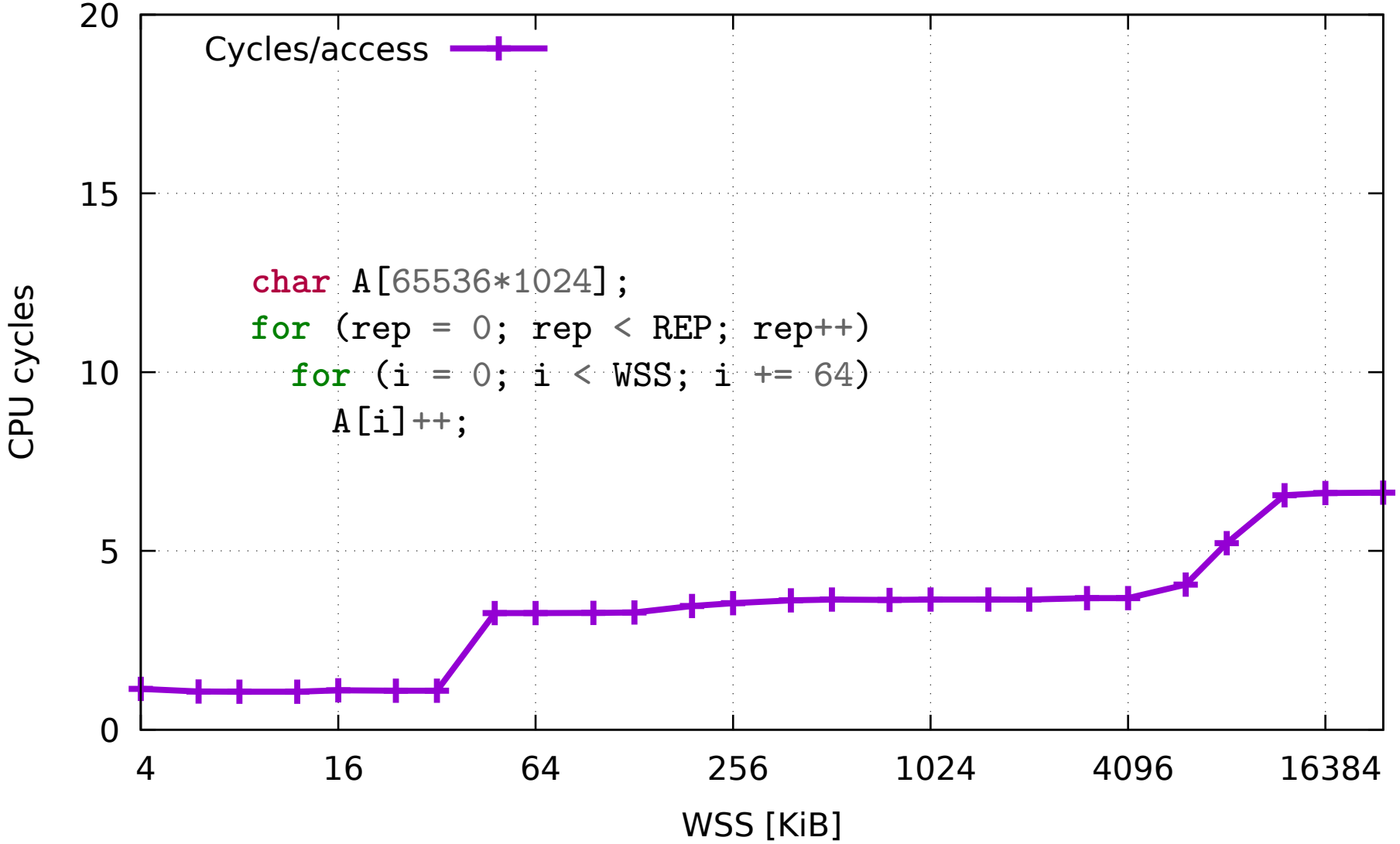
1 Why is DRAM slow?

2 Caches
- Architecture
- Memory performance characteristics
- Data structures and dynamic memory allocations
- Matrix multiplications

3 Caches & memory in multi-processor systems
- True and false sharing
- NUMA

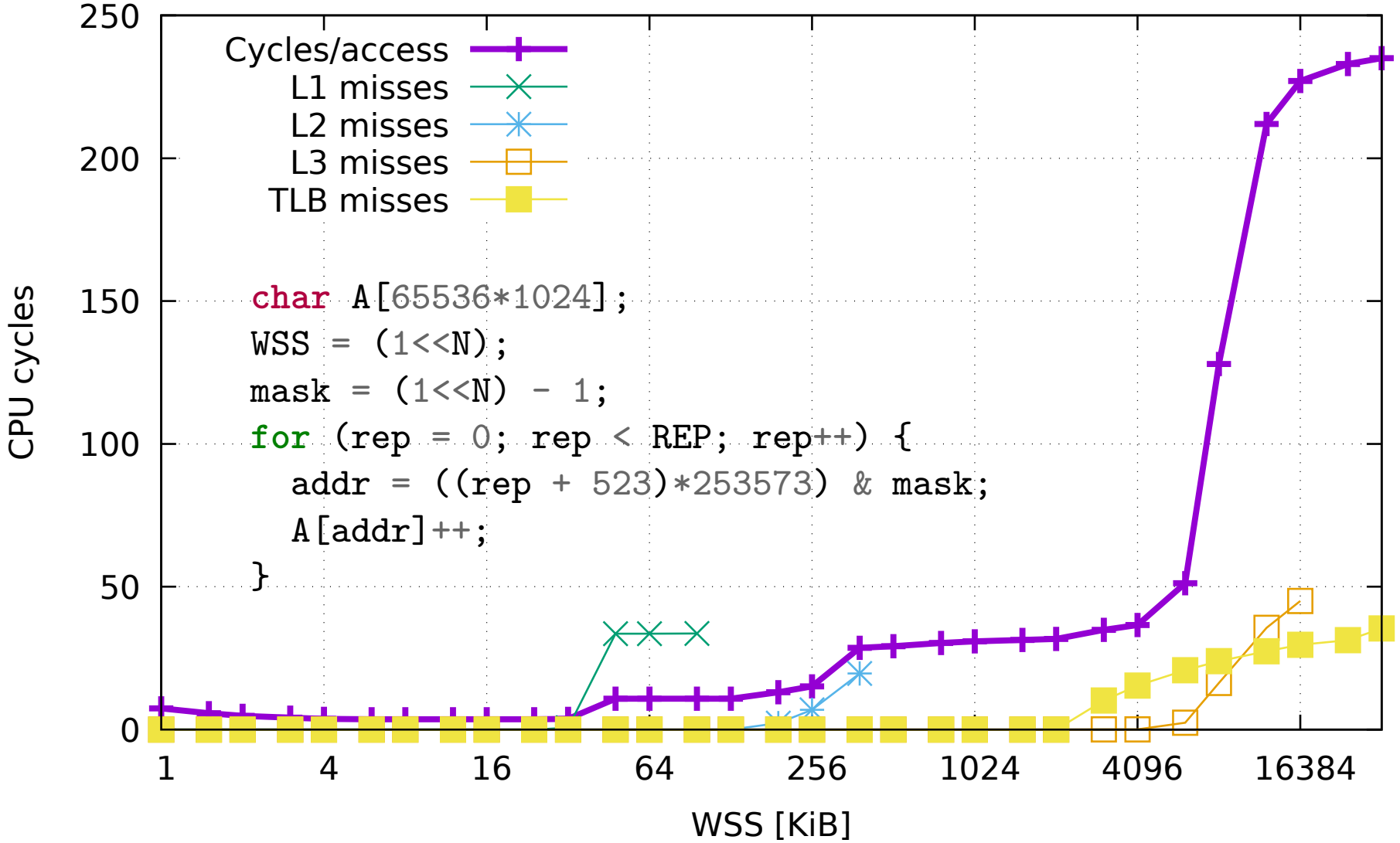
4 Conclusion

Sequential access



Intel Core i7-2600

Random access



Intel Core i7-2600, (perf counters not in scale)

Translation Lookaside Buffer (TLB)

- Caches translation of virtual to physical address
- On TLB miss, page walk has to be performed (2 to 5 levels)
- Intel i7-2600 has 512 L2 TLBs ⇒ 512×4 kB = 2 MB
- Improvement: use so called huge pages (1 page = 2 MB, PS=1)
  - Linux: in some cases automatically or explicitly via `hugetlbfs`

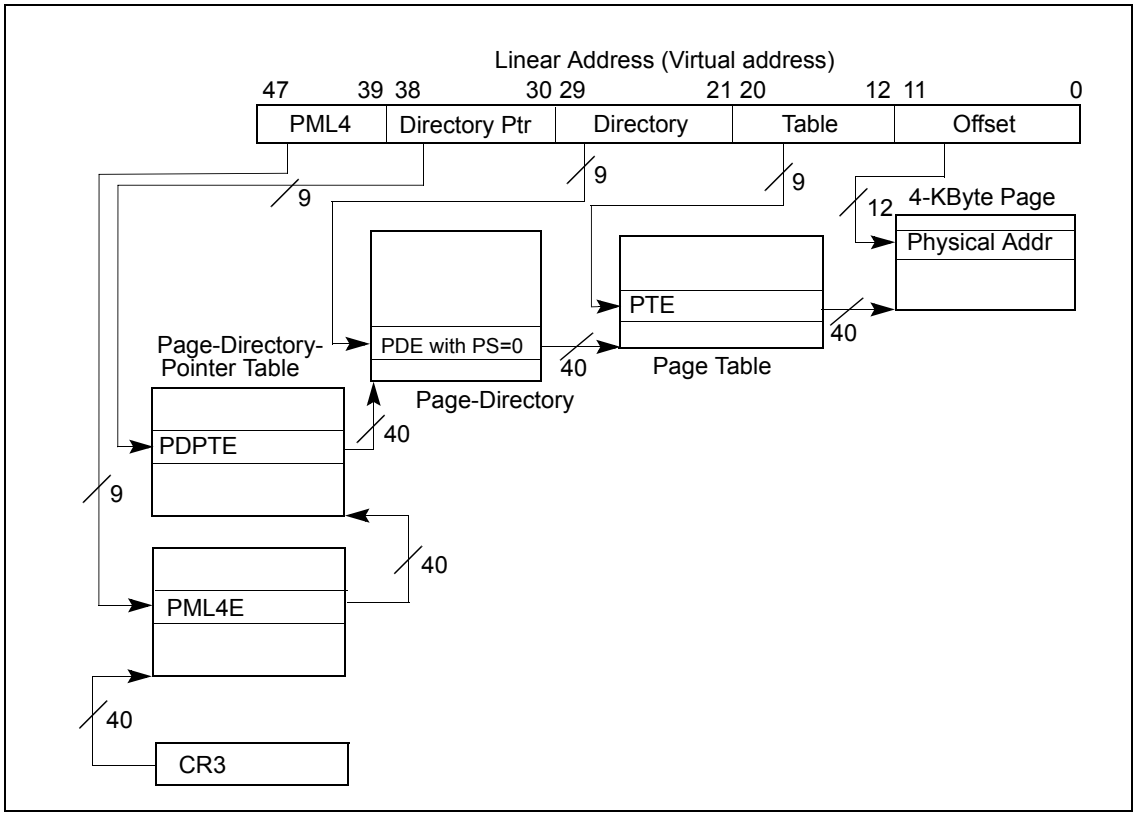


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

Cache-related preemption delay

- When a thread is preempted by another thread, the preempting thread likely evicts some data from the cache.
- After preemption ends, the preempted thread continues executing and experiences a lot of cache misses!
- Certain (older) architectures has to flush TLBs when switching address spaces (processes).
  - Modern architectures allow tagging TLBs with address space identifier (ASID, PCID, ...)
- High-performance software tries to limit preemptions.
  - Beware – limiting preemption increases response time!

Outline

1 Why is DRAM slow?

2 Caches
- Architecture
- Memory performance characteristics
- Data structures and dynamic memory allocations
- Matrix multiplications

3 Caches & memory in multi-processor systems
- True and false sharing
- NUMA

4 Conclusion

Data structures and cache friendliness

- Arrays + sequential access – nice
- Dynamically allocated linked lists – depends on memory allocator (probably like random access – bad)
- Search trees – random access
- For most data structures/algorithms, there exist cache-optimized variants.
- These are more tricky than textbook examples.

Dynamic memory allocator (malloc(), new)

- Memory allocators try to maintain spacial and temporal locality
  - Spatial locality is hard to achieve when heap is fragmented
    - after many new/delete operations
  - Temporal locality – when memory is freed/deleted, subsequent allocation tries to use that memory because it is cache-hot.

Outline

1 Why is DRAM slow?

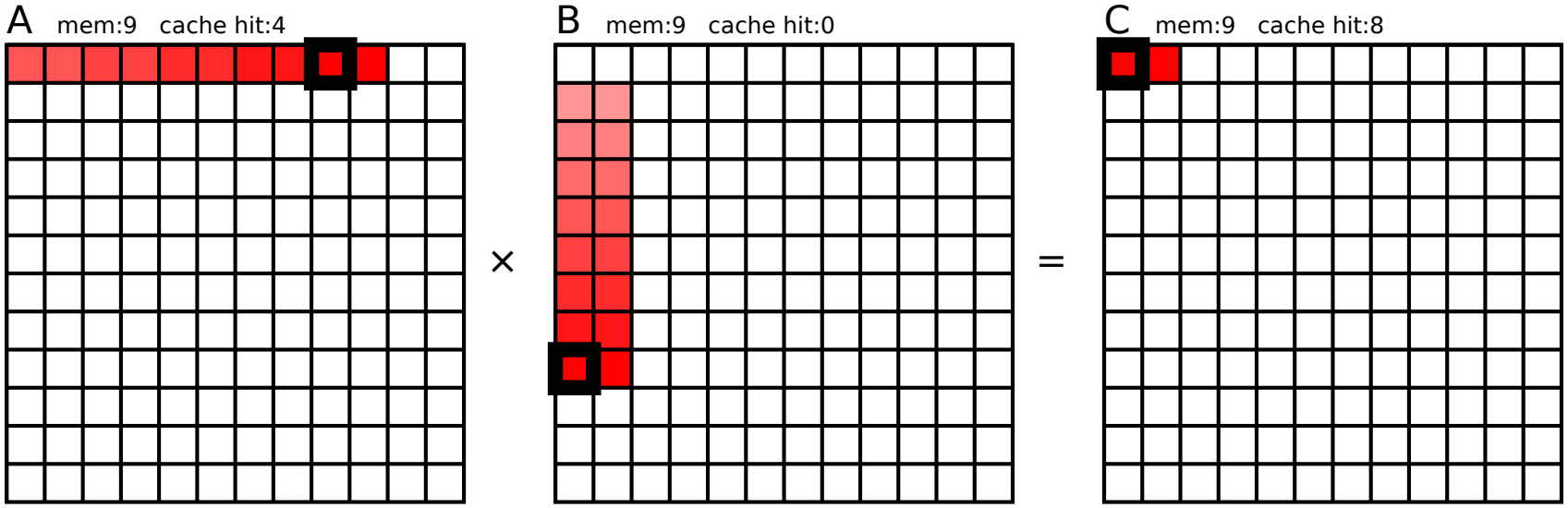
2 Caches
- Architecture
- Memory performance characteristics
- Data structures and dynamic memory allocations
- Matrix multiplications

3 Caches & memory in multi-processor systems
- True and false sharing
- NUMA

4 Conclusion

Matrix multiplication – naive implementation

Matrix multiplication: Naive



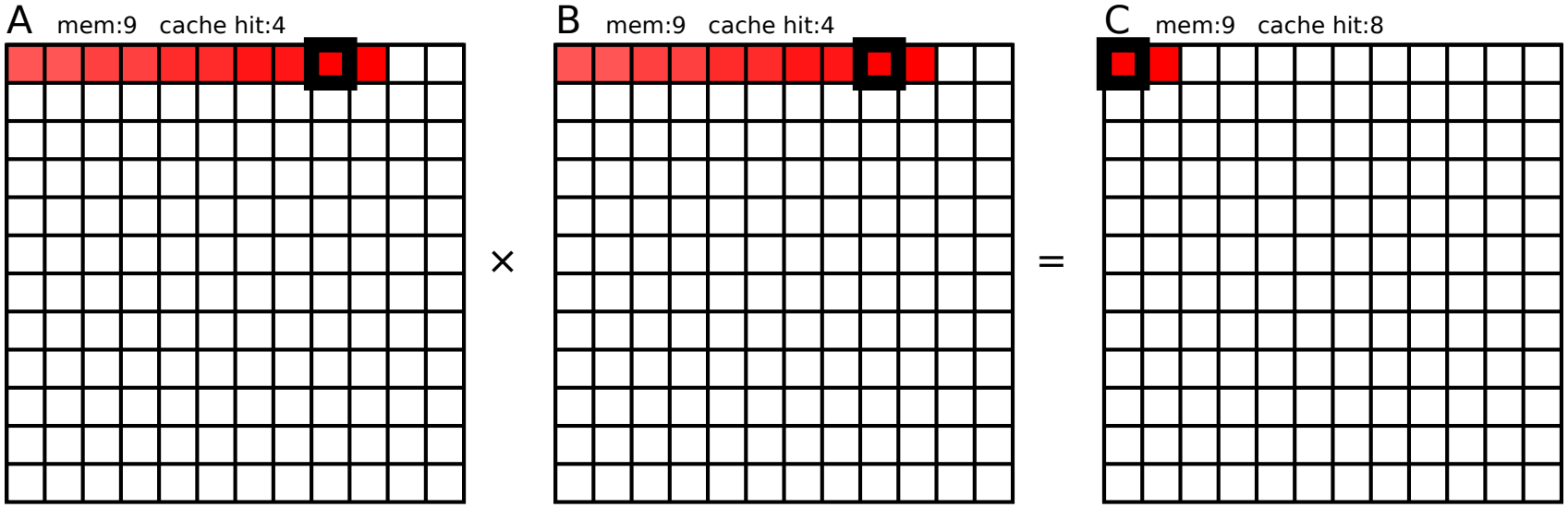
Totals: mem:27 cache hits:12 ≅44%

```
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    for (k = 0; k < N; ++k)
      C[i][j] = A[i][k] * B[k][j];
```

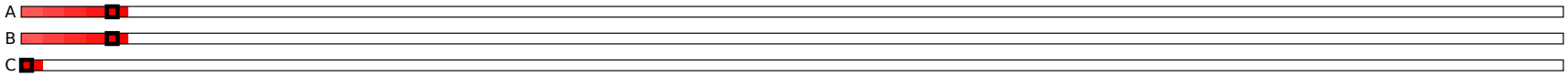
- One matrix element: double (8 B)
- Cache line size: 16 B
- Fully associative caches
- L2 cache: 128 B, L1 cache: 32 B

Implementation with transposition

Matrix multiplication: B transposed



Totals: mem:27 cache hits:16 ≅59%



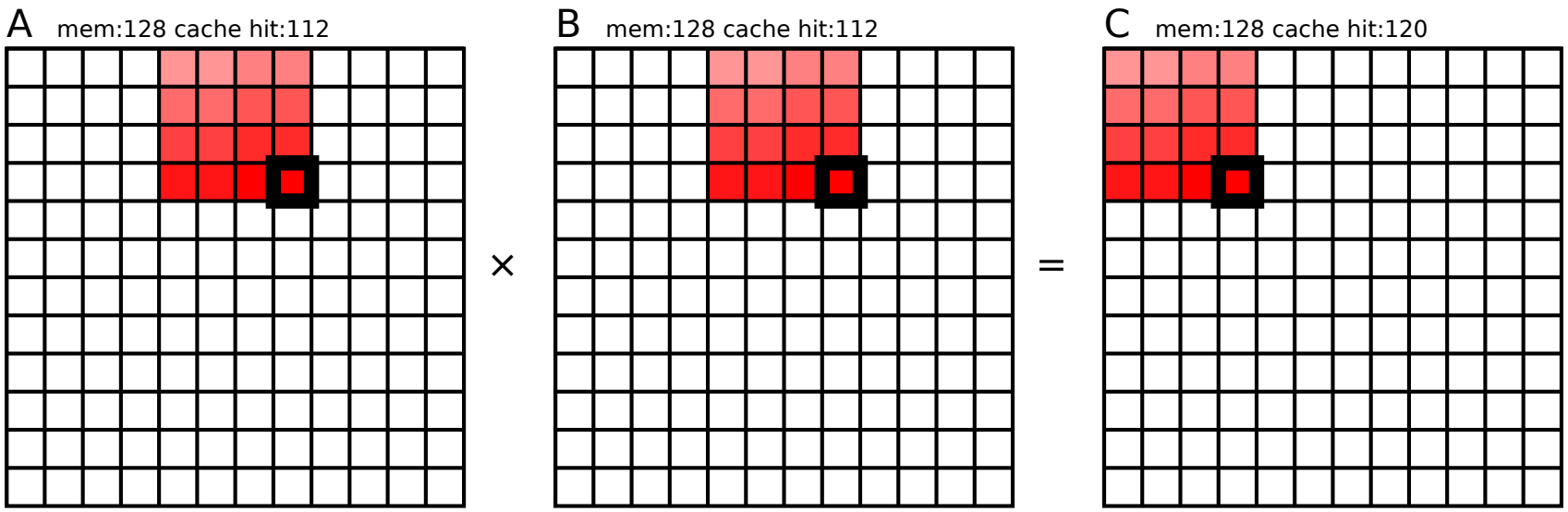

```
double B[N][N];
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    B[i][j] = Bsrc[j][i];
```

```
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    for (k = 0; k < N; ++k)
      C[i][j] = A[i][k] * B[k][j];
```

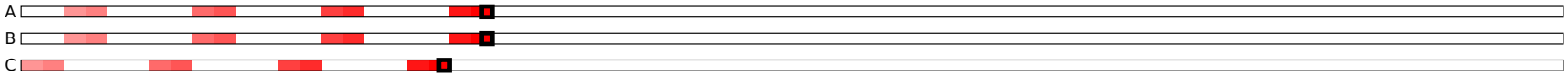
Performance (execution time): naive: 100%, transposed: 23,4%

Tiled implementation

Matrix multiplication: Tiled, B transposed



Totals: mem:384 cache hits:344 ≅89%

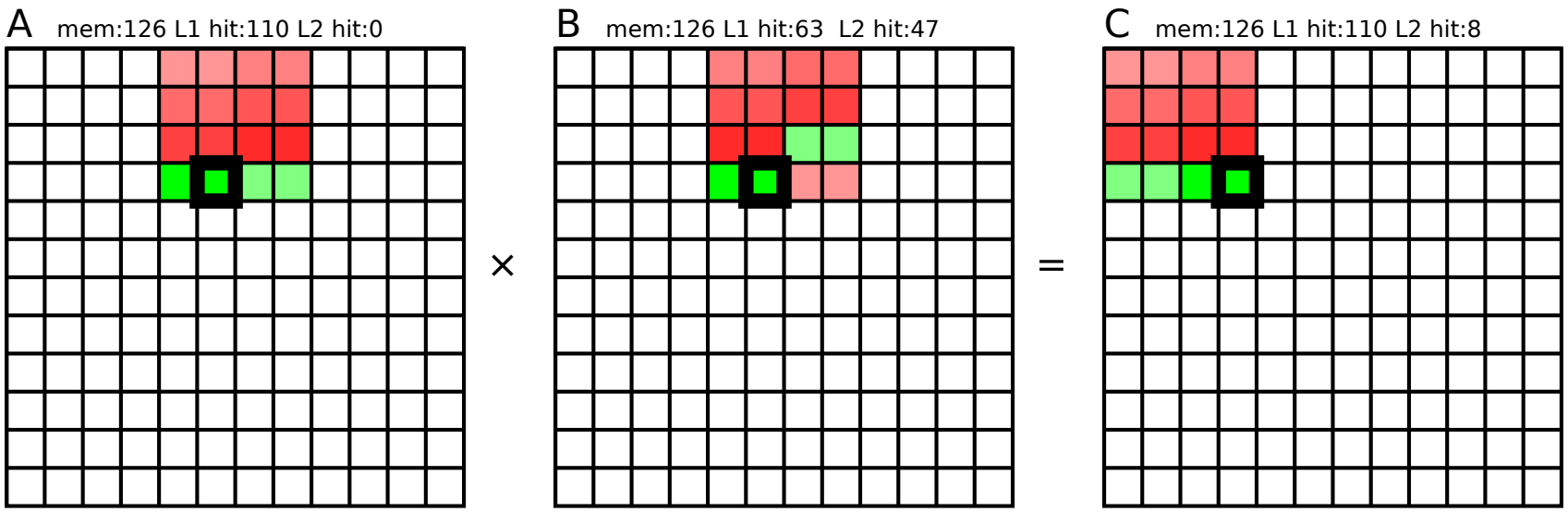



```
for (k1 = 0; k1 < N; k += tile)
  for (j1 = 0; j1 < N; j += tile)
    for (i1 = 0; i1 < N; i += tile)
      for (i = i1; i < i1 + tile; ++i)
        for (j = j1; j < j1 + tile; ++j)
          for (k = k1; k < k1 + tile; ++k)
            C[i][j] += A[i][k] * B[k][j];
```

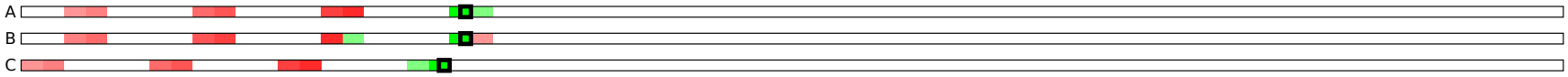
- Each “tile” fits into the cache
- Performance: 17.3% of naive implementation (9.5% with vectorized (SIMD) operations)

Tiled implementation and L1 cache

Matrix multiplication: Tiled, B transposed



Totals: mem:378 L1 hits:283 ≅74% L2 hits:55 ≅14% cache hits:338 ≅89%

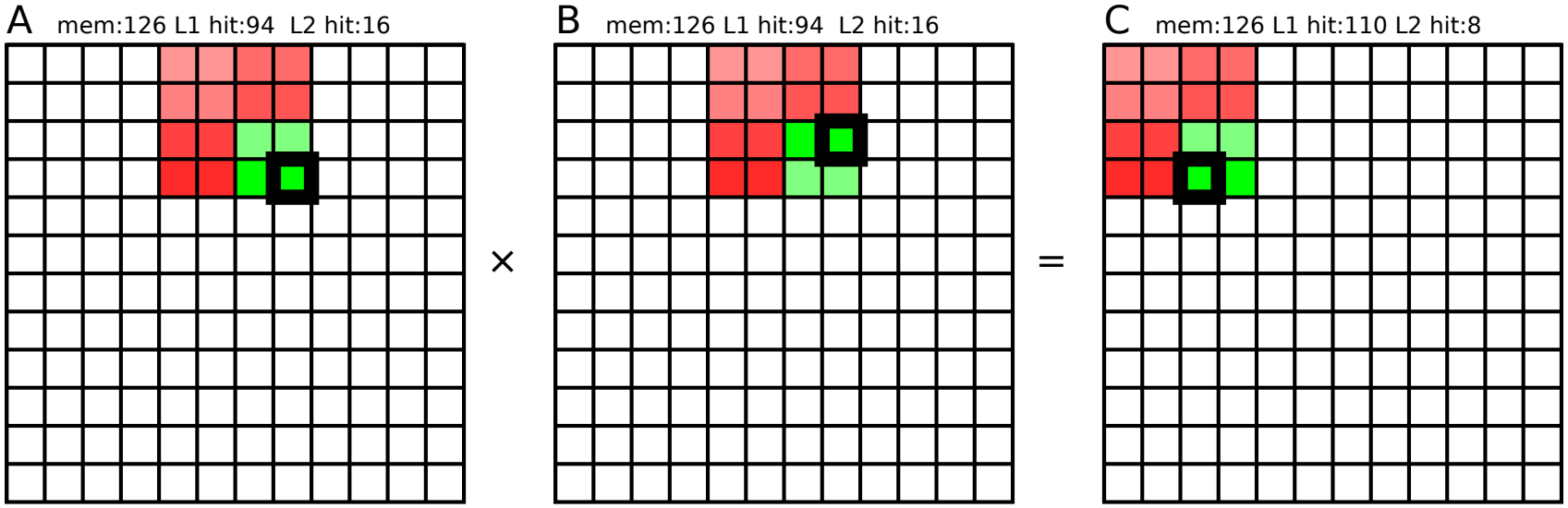



```
for (k1 = 0; k1 < N; k += tile)
  for (j1 = 0; j1 < N; j += tile)
    for (i1 = 0; i1 < N; i += tile)
      for (i = i1; i < i1 + tile; ++i)
        for (j = j1; j < j1 + tile; ++j)
          for (k = k1; k < k1 + tile; ++k)
            C[i][j] += A[i][k] * B[k][j];
```

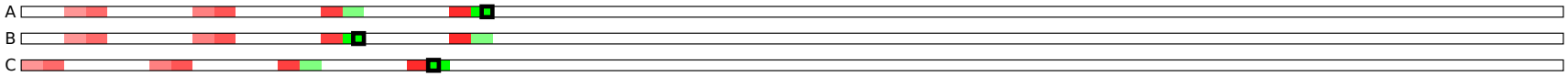
- No temporal L1 cache hit in B
- 75% L1 hits (in total)

Two-level tiled implementation

Matrix multiplication: 2-level tiled, B transposed



Totals: mem:378 L1 hits:298 ≅78% L2 hits:40 ≅10% cache hits:338 ≅89%




```
for (k2 = 0; k2 < N; k2 += tile2)
  for (j2 = 0; j2 < N; j2 += tile2)
    for (i2 = 0; i2 < N; i2 += tile2)
      for (k1 = k2; k1 < k2 + tile2; k += tile1)
        for (j1 = j2; j1 < j2 + tile2; j += tile1)
          for (i1 = i2; i1 < i2 + tile2; i += tile1)
            for (i = i1; i < i1 + tile1; ++i)
              for (j = j1; j < j1 + tile1; ++j)
                for (k = k1; k < k1 + tile1; ++k)
                  C[i][j] += A[i][k] * B[k][j];
```

- 79% L2 hits

Recursive matrix multiplication

- Generalization to arbitrary number of cache levels
- $N \times N$ multiplication = 8 multiply-add of $(N/2) \times (N/2)$ multiplications

$$\left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right] = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \times \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right] =$$

$$\left[\begin{array}{c|c} A_{11}B_{11} & A_{11}B_{12} \\ \hline A_{21}B_{11} & A_{21}B_{12} \end{array}\right] \times \left[\begin{array}{c|c} A_{12}B_{21} & A_{12}B_{22} \\ \hline A_{22}B_{21} & A_{22}B_{22} \end{array}\right]$$

Outline

1 Why is DRAM slow?

2 Caches
- Architecture
- Memory performance characteristics
- Data structures and dynamic memory allocations
- Matrix multiplications

3 Caches & memory in multi-processor systems
- True and false sharing
- NUMA

4 Conclusion

Cache coherency

In symmetric multi-processor (SMP) systems, caches of the CPUs cannot work independently from each other.

- Maintaining of uniform view of memory for all processor is called "cache coherency"
- If some processor writes to a cache line, other processors have to clean the corresponding cache line from their caches.
  - Remember: inter-core (inter-socket) communication is "slow"
- Cache synchronization protocol: MESI(F)
  - A dirty cache line is not present in any other processor's cache.
  - Clean copies of the same cache line can reside in arbitrarily many caches.

Outline

1 Why is DRAM slow?

2 Caches
- Architecture
- Memory performance characteristics
- Data structures and dynamic memory allocations
- Matrix multiplications

3 Caches & memory in multi-processor systems
- True and false sharing
- NUMA

4 Conclusion

True sharing

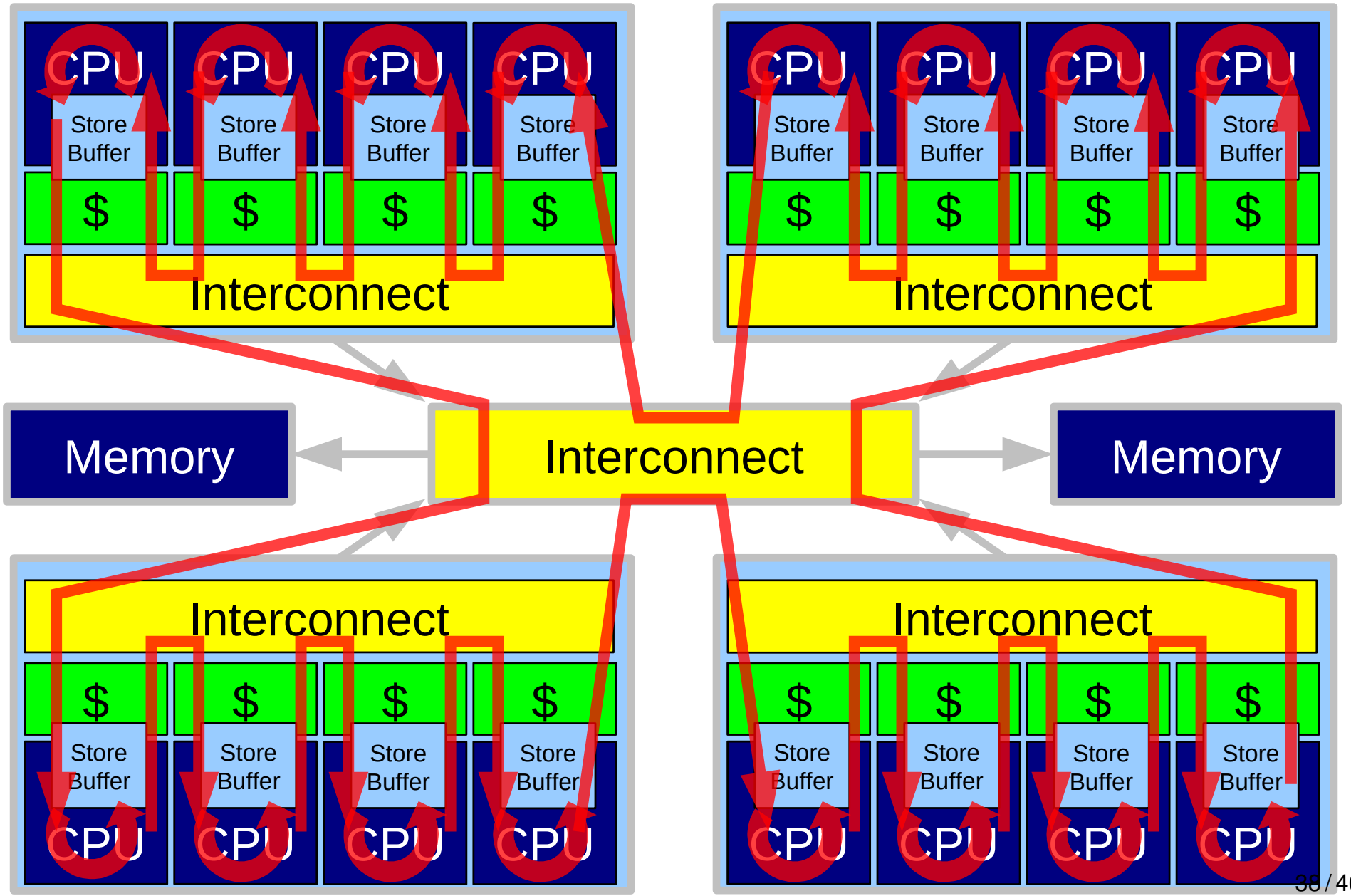
- Program is slow because cache lines with shared data travel from one core to another.
- Typical example of true sharing: each mutex is shared between CPUs.
- When that is a problem (too much contention):
  - make locking more fine-grained,
  - or change your data structure (e.g. per-CPU data),
  - and/or algorithms to be more cache friendly.

```
std::atomic_int32_t counter;

void thread_cpu0() {
  while (true)
    counter++;
}

void thread_cpu1() {
  while (true)
    counter++;
}
```

All CPUs executing atomic increment of global variable



False sharing

- Data accessed from different CPUs is not shared but happen to be stored in a single cache line.

```
// Per-CPU counters
std::atomic_int32_t counter_cpu0;
std::atomic_int32_t counter_cpu1;
```

```
void thread_cpu0() {
  while (true)
    counter_cpu0++;
}
```

```
void thread_cpu1() {
  while (true)
    counter_cpu1++;
}
```

False sharing – solution

- Data accessed from different CPUs is not shared but happen to be stored in a single cache line.

```
// Per-CPU counters, each aligned to cache line boundary
std::atomic_int32_t counter_cpu0 __attribute__((align(64)));
std::atomic_int32_t counter_cpu1 __attribute__((align(64)));
```

```
void thread_cpu0() {
  while (true)
    counter_cpu0++;
}
```

```
void thread_cpu1() {
  while (true)
    counter_cpu1++;
}
```

- How to determine cache size?
  - at run time: `sysconf(_SC_LEVEL1_DCACHE_LINESIZE);`
  - at compile time:

    ```
    gcc -DLEVEL1_DCACHE_LINESIZE=$(getconf LEVEL1_DCACHE_LINESIZE) ...
    ```

Outline

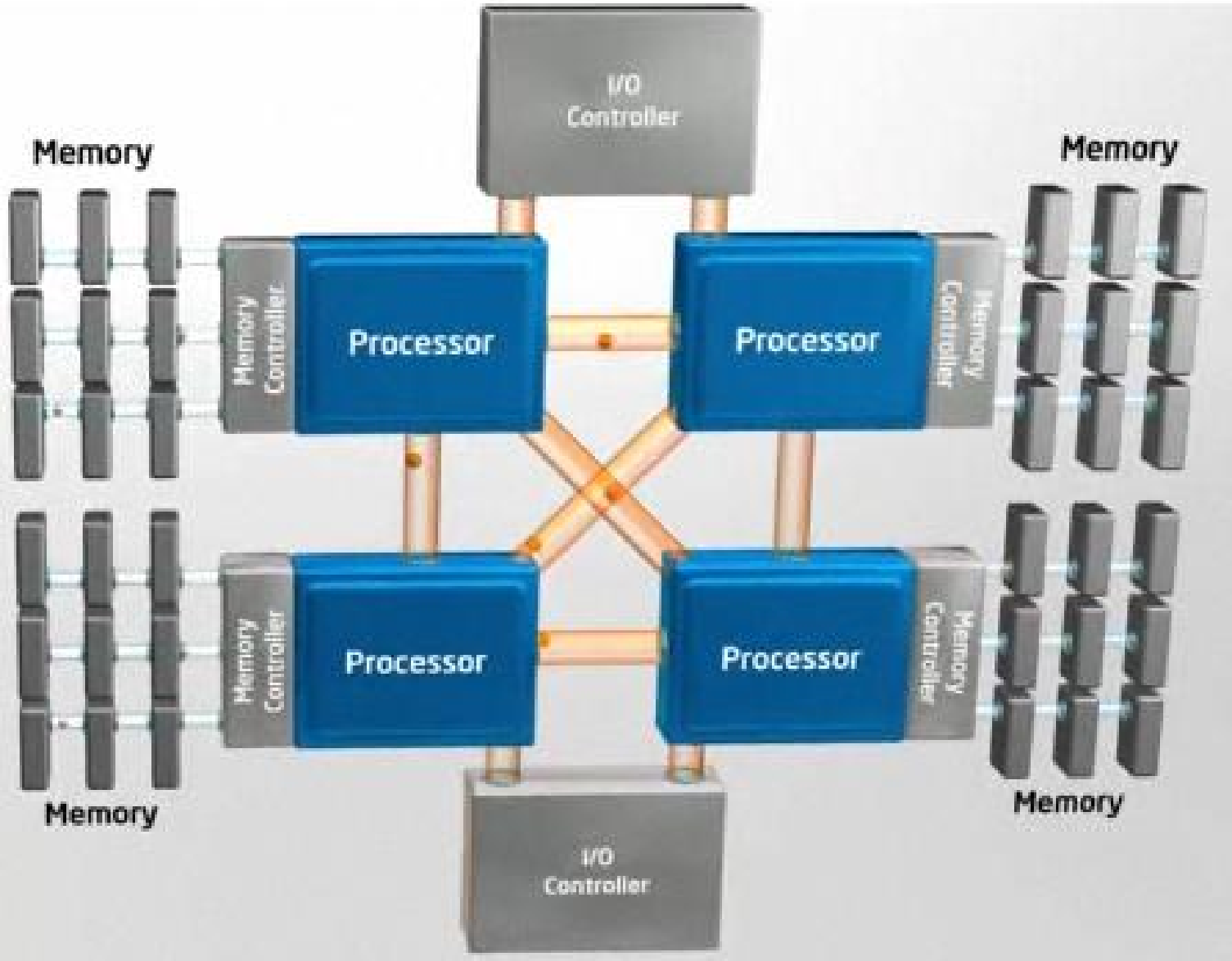
1 Why is DRAM slow?

2 Caches
- Architecture
- Memory performance characteristics
- Data structures and dynamic memory allocations
- Matrix multiplications

3 Caches & memory in multi-processor systems
- True and false sharing
- NUMA

4 Conclusion

Non-Uniform Memory Access (NUMA)



Thread migrations between cores

- OSes tend to do load balancing
  - By default threads are automatically migrated from overloaded to underloaded cores
  - Migrated threads loose their cache footprint (cache-related migration delay)
  - Migrated threads loose their NUMA locality
- If you do your own load balancing in the application, pin the threads to CPUs (set their CPU affinity):

```
cpu_set_t cpuset;
pthread_t thread;
thread = pthread_self();
/* Set affinity mask to include only CPU 3 */
CPU_ZERO(&cpuset);
CPU_SET(1 << 3, &cpuset);
s = pthread_setaffinity_np(thread, sizeof(cpu_set_t), &cpuset);
```

libnuma (Linux)

```
#include <numa.h>

int numa_available(void);

int numa_max_possible_node(void);
int numa_num_possible_nodes();

int numa_max_node(void);
//...
int numa_preferred(void);
void numa_set_preferred(int node);
void numa_set_interleave_mask(struct bitmask *nodemask);
//...
void numa_bind(struct bitmask *nodemask);
void numa_set_localalloc(void);
void numa_set_membind(struct bitmask *nodemask);
```

Outline

1 Why is DRAM slow?

2 Caches
- Architecture
- Memory performance characteristics
- Data structures and dynamic memory allocations
- Matrix multiplications

3 Caches & memory in multi-processor systems
- True and false sharing
- NUMA

4 Conclusion

Size matters

- Even though we have terabytes of memory, size and layout of the data structures still matters.
- Only few kilobytes of memory is fast, the rest is slow!

References

- Ulrich Drepper, “What Every Programmer Should Know About Memory”, 2007/11 [online], `http://people.redhat.com/drepper/cpumemory.pdf`

Effective Software

Lecture 10: JVM - Memory Analysis, Data Structures, Collections for Performance

David Šišlák
david.sislak@fel.cvut.cz

[1] Oaks, S.: Java Performance: The Definitive Guide. O'Reilly, USA 2014.
[2] JVM source code - http://openjdk.java.net
[3] Hylock, R.: Large-Scale Memory Efficient Java Primitive Collections. Journal of Software, March 2016.

JVM Performance Factors and Memory Analysis

» application **performance factors**

- total runtime
  - algorithms (complexity, instructions, synchronization)
  - memory management (garbage collection) overhead
  - data structures (speed of data access, cache efficiency, GC pressure)
- memory consumption
  - data structures (memory usage efficiency)

» **memory analysis**

- static memory analysis
  - analyze memory usage at particular time
  - suitable for data structure efficacy analysis, inspect content
- dynamic memory analysis
  - analyze dynamic changes over time
  - suitable for object allocation analysis and memory leak identification

Static Memory Analysis – Object Histogram

» analyze **histogram of objects** – imply global safepoint (stop the world)

- jmap -histo:live {PID}

```
 num     #instances         #bytes  class name
----------------------------------------------
   1:       3000257       72006168  java.lang.Integer
   2:       1000122       48005856  java.util.HashMap$Node
   3:       1000012       40000480  java.util.LinkedList$Node
   4:       1000000       40000000  gnu.trove.list.linked.TIntLinkedList$TIntLink
   5:             2       33913088  [D
   6:           162       24963896  [I
   7:       1000001       24000024  java.lang.Double
   8:            26       16781936  [Ljava.util.HashMap$Node;
   9:           604        8056600  [Ljava.lang.Object;
  10:            37        2176208  [B
  11:          1549         184864  [C
  12:           635         109344  java.lang.Class
  13:          1519          48608  java.lang.String
  14:           307          14736  java.util.concurrent.ConcurrentHashMap$Node
  15:           108          12960  java.lang.reflect.Field
  16:            85           8840  java.net.URL
  17:           257           6168  java.lang.Byte
  18:           257           6168  java.lang.Long
  19:           257           6168  java.lang.Short
  20:           123           5904  java.util.Hashtable$Entry
  21:           102           5712  java.lang.ref.SoftReference
  22:             5           4984  [Ljava.util.concurrent.ConcurrentHashMap$Node;
  23:           284           4544  java.lang.Object
```

Static Memory Analysis – Heap Dump

» capture **heap dump** – exported during global safepoint (stop the world)

- -XX:+HeapDumpOnOutOfMemoryError
- jmap -dump:live,format=b,file={name}.hprof {PID}
- jvisualvm, yourkit, …

» analyze heap dump – jvisualvm, yourkit, ...

Memory | Threads | Thread tid=1 | Inspections | Summary

/ retained size: 295 MB All the objects are strong reachable Reachability scopes

Class name, string value, thread name or ID (Press "Enter" to apply / ?):

Name	Retained Size	Shallow Size
java.lang.Thread **[Thread]** "main" tid=1 [RUNNABLE]	309,816,648	176
<local variable> MemoryAnalysis [Stack Local]	309,789,752	144
contextClassLoader sun.misc.Launcher$AppClassLoader	95,953	160
<local variable> char[8192] [Stack Local] = {all elements = 0}	16,408	16,408
<local variable> java.io.BufferedInputStream [Stack Local]	8,264	48

Paths from GC Roots | Allocations | Ages | Class Hierarchy | Incoming References | Quick Info

Quick info on object(s) selected in the upper table

Object class:	java.lang.Thread
Object generation:	not available
Object index:	#1011037
Web application:	None
Distance to nearest GC root:	0 (the object is a root itself)
Shallow size:	176
Retained objects (includes the object itself):	7,000,067
Retained size (includes shallow size):	309,816,648

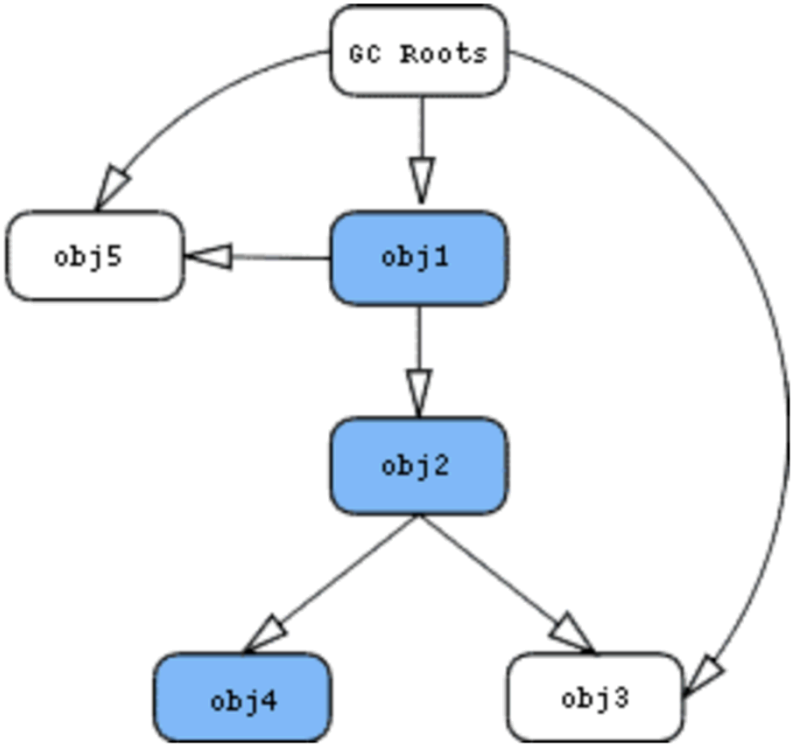
Shallow vs. Retained Size

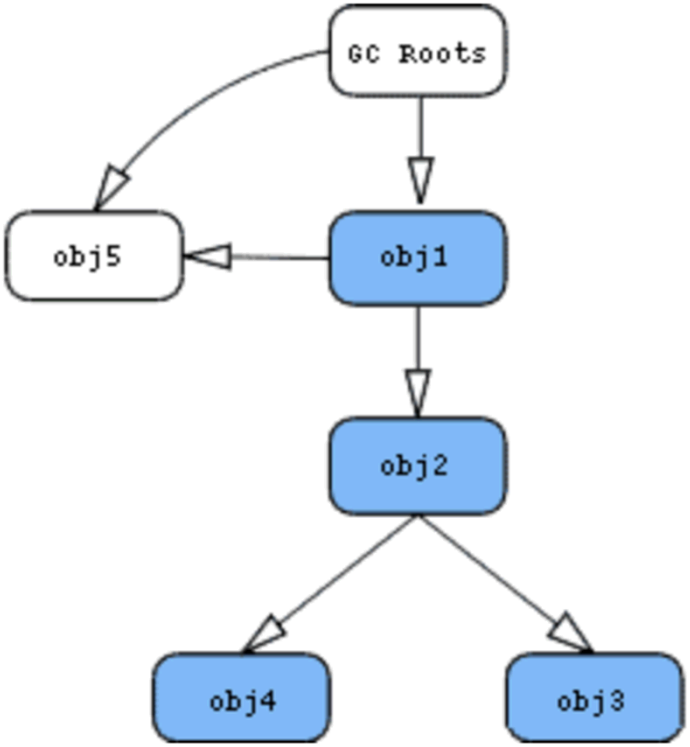
» **shallow size**

- memory allocated to store object itself

» **retained size**

- quantity of memory this object preserves from GC
  - amount of memory freed if the object is GCed
- own shallow size + shallow size of all objects directly or indirectly accessible **ONLY** from this object





Static Analysis Advanced Inspections

» wasting memory – memory doesn't keep any useful content

- **duplicate strings**
  - share string instances via pooling or intern()
- **duplicate objects** – same field contents
  - share them, lazy creation, non-permanent usage
- **zero length arrays**
  - unnecessary load for GC
  - use per-class empty array singleton (e.g. via static field in the class)
- **null fields** - objects having a lot of 'null' fields
  - use subclasses for rarely assigned fields
- **sparse arrays** – big number of 'null', zero or same elements
  - use alternate data structures (e.g. maps or refactor algorithms)
- **inefficient data structure** – large overhead of useless content
  - use different data structures

Static Analysis Advanced Inspections

» memory leak – objects are no longer used but there are still references to them

- **object retained from inner non-static class back reference**
  - implicit back reference from inner class instance (even anonymous), e.g. used for callback objects
  - minimize usage of non-static inner class instances

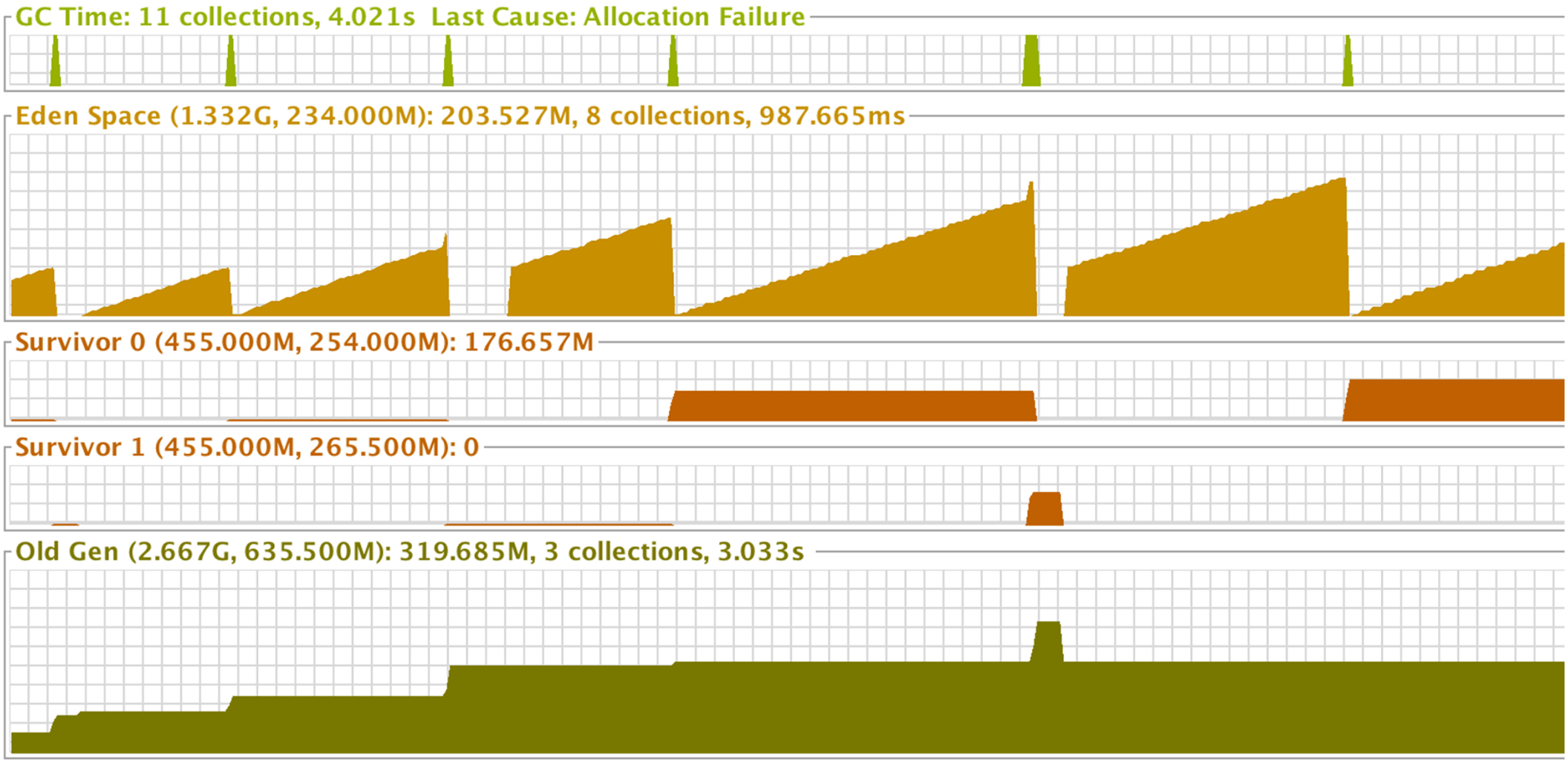
» performance – speed of data read / write

- **hash tables with non-uniformly distributed hash codes**
  - degraded performance due to hash collisions
  - use better hashCode implementation

Dynamic Memory Analysis – GC Telemetry

» analyze **GC telemetry** – e.g. jvisualvm with VisualGC plugin

- usage of eden space in time
- GC collections and their duration
- not affecting performance of monitored application



Dynamic Memory Analysis – Heap Dumps

» compare **heap dumps**

- difference in object count and size in various application state
- dumps with all objects (not just live) can help analyze object allocations if there is no GC run in between
- each heap dump requires global safepoint (time depends on the heap size)

Old snapshot (the baseline): dump1
New snapshot: dump2 ← **this snapshot**

Objects
- Classes
- Classes and packages

Exceptions
- Exceptions

Name	Objects (+/-)		Size (+/-)	
char[]	+5,386	+93 %	+407,040	+6 %
byte[]	+927	+16 %	+292,792	+5 %
jdk.internal.org.objectweb.asm.Item	+3,051	+53 %	+170,856	+3 %
jdk.internal.org.objectweb.asm.Item[]	+174	+3 %	+101,280	+2 %
java.lang.String	+3,219	+55 %	+77,256	+1 %
java.lang.Class[]	+1,275	+22 %	+38,456	+1 %
java.lang.Object[]	+716	+12 %	+34,984	+1 %
jdk.internal.org.objectweb.asm.MethodWriter	+132	+2 %	+29,568	+0 %
java.lang.invoke.MethodType	+729	+13 %	+29,160	+0 %
java.lang.invoke.MethodType$ConcurrentWeakInternSet$WeakEntry	+729	+13 %	+23,328	+0 %
jdk.internal.org.objectweb.asm.Label	+245	+4 %	+15,680	+0 %
java.lang.StringBuilder	+638	+11 %	+15,312	+0 %
jdk.internal.org.objectweb.asm.ClassWriter	+91	+2 %	+15,288	+0 %
java.lang.reflect.Method	+168	+3 %	+14,784	+0 %
jdk.internal.org.objectweb.asm.Type	+439	+8 %	+14,048	+0 %
jdk.internal.org.objectweb.asm.AnnotationWriter	+248	+4 %	+13,888	+0 %
jdk.internal.org.objectweb.asm.ByteVector	+562	+10 %	+13,488	+0 %
java.lang.StringBuffer	+457	+8 %	+10,968	+0 %
jdk.internal.org.objectweb.asm.Frame	+226	+4 %	+10,848	+0 %
java.lang.invoke.MemberName	+326	+6 %	+10,432	+0 %
boolean[]	+34	+1 %	+9,248	+0 %
java.lang.invoke.LambdaForm$Name[]	+130	+2 %	+5,984	+0 %
java.lang.invoke.InvokerBytecodeGenerator	+80	+1 %	+5,120	+0 %
java.lang.invoke.LambdaForm$BasicType[]	+88	+2 %	+4,120	+0 %
java.util.HashMap	+81	+1 %	+3,888	+0 %
jdk.internal.org.objectweb.asm.Type[]	+114	+2 %	+3,880	+0 %
java.util.AbstractList$Itr	+115	+2 %	+3,680	+0 %
java.util.HashMap$ValueIterator	+81	+1 %	+3,240	+0 %

Dynamic Memory Analysis – Allocation Tracking

» allocation tracking - **memory profiler**

- track every n-th object allocation (trade-off between precision and speed)
- affect performance of profiled application, injects **traceObjAlloc** byte code
  - introduce a lot of byte code + consume memory
  - decreases possibility of JIT optimizations

Call Tree	Objects		Size	
<Objects without allocation information>	92,413		12,760,600	
<All threads>	25,264	100 %	1,692,200	100 %
com.intellij.rt.execution.application.**AppMain.main**(String[])	25,264	100 %	1,692,200	100 %
NativeMethodAccessorImpl.java (native) **Lambda.main**(String[])	25,264	100 %	1,692,200	100 %
Lambda.java:44 **Lambda.reversedAlphabeticalOnlyOrder**(String[])	25,258	99 %	1,667,408	99 %
Lambda.java:13 java.lang.invoke.**MethodHandleNatives.linkCallSite**(Object, Object, Object, Object, Object, Object[])	15,709	62 %	1,050,728	62 %
Lambda.java:13 java.lang.invoke.**MethodHandleNatives.linkMethodHandleConstant**(Class, int, Class, String, Object)	5,627	22 %	324,184	19 %
Lambda.java:15 java.util.**Comparator.comparing**(Function)	1,557	6 %	106,040	6 %
Lambda.java:16 java.util.stream.**Collectors.toList**()	1,032	4 %	68,912	4 %
Lambda.java:16 java.util.stream.**ReferencePipeline.collect**(Collector)	869	3 %	71,760	4 %
Lambda.java:14 java.lang.invoke.**MethodHandleNatives.linkCallSite**(Object, Object, Object, Object, Object, Object[])	161	1 %	10,088	1 %
Lambda.java:13 java.lang.invoke.**MethodHandleNatives.findMethodHandleType**(Class, Class[])	98	0 %	5,904	0 %
Lambda.java:13 java.util.**Arrays.stream**(Object[])	87	0 %	17,608	1 %
Lambda.java:16 java.util.stream.**Collectors.<clinit>**()	33	0 %	3,608	0 %
Lambda.java:14 java.lang.invoke.**MethodHandleNatives.linkMethodHandleConstant**(Class, int, Class, String, Object)	24	0 %	872	0 %
Lambda.java:14 java.lang.invoke.**MethodHandleNatives.findMethodHandleType**(Class, Class[])	18	0 %	640	0 %

Classes | Packages | Object Explorer | Generations | Ages | Reachability | Class Loaders | Web Applications | Callees List | Quick Info

Class list for objects selected in the upper table

Class	Objects		Shallow Size		Retained Size	
char[]	5,280	21 %	388,472	23 %	388,472	23 %
byte[]	929	4 %	284,856	17 %	284,856	17 %
jdk.internal.org.objectweb.asm.Item	3,051	12 %	170,856	10 %	170,856	10 %
java.lang.Class	230	1 %	136,328	8 %	165,824	10 %

Dynamic Memory Analysis – Allocation Tracking

» allocation tracking – **flight recording** using jmc – no byte code instrumentation

- identify large object allocations outside TLAB

General | Allocation in New TLAB | Allocation Outside TLABs

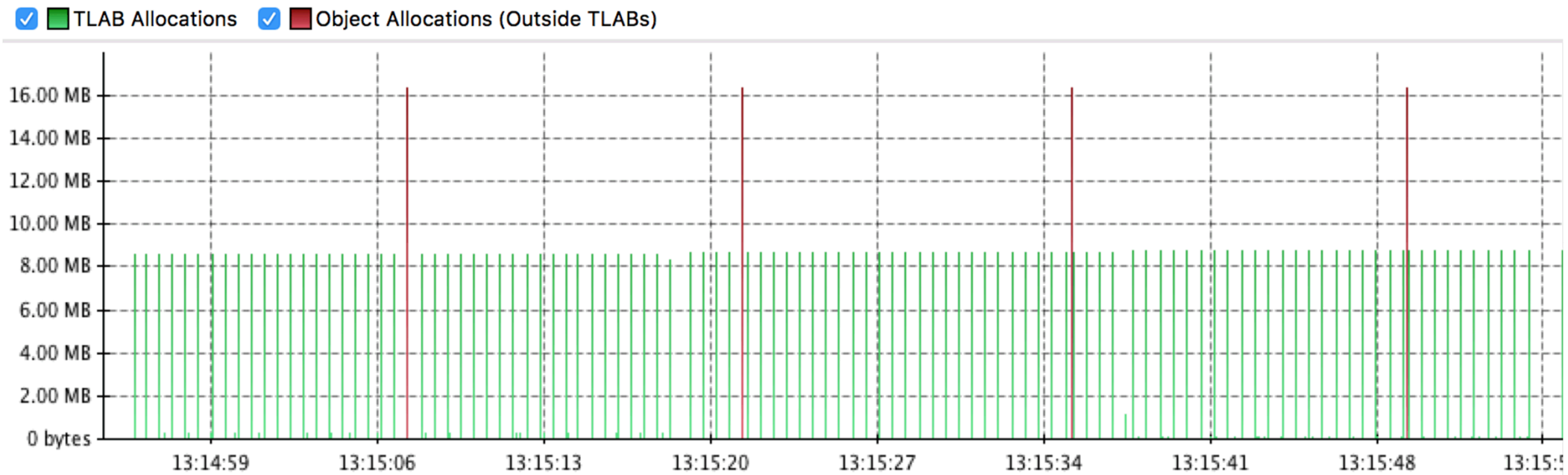
Thread Local Allocation Buffer (TLAB) Statistics

TLAB Count	393
Maximum TLAB Size	16.26 MB
Minimum TLAB Size	2.05 kB
Average TLAB Size	2.45 MB
Total Memory Allocated for TLABs	961.48 MB
Allocation Rate for TLABs	15.65 MB/s

Statistics for Object Allocations (Outside TLABs)

Object Count	48
Maximum Object Size	16.34 MB
Minimum Object Size	24 bytes
Average Object Size	5.67 MB
Total Memory Allocated for Objects	272.12 MB
Allocation Rate for Objects	4.43 MB/s

Allocation



Dynamic Memory Analysis – Allocation Tracking

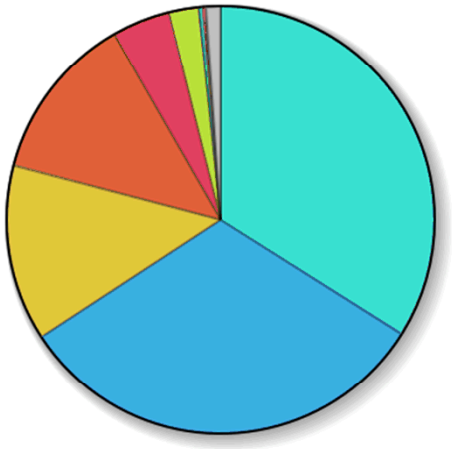
» allocation tracking – **flight recording** using jmc – <u>no byte code instrumentation</u>

- identify large object allocations outside TLAB

General | Allocation in New TLAB | Allocation Outside TLABs

Allocation by Class | Allocation by Thread | Allocation Profile

Allocation Pressure



Class	Average Object Siz	TLABs	Total TLAB Size
java.lang.Integer	24 bytes	38	325.76 MB
java.util.HashMap$Node	48 bytes	39	306.85 MB
java.lang.Double	24 bytes	15	128.52 MB
gnu.trove.list.linked.TIntLinkedList$TIntLink	40 bytes	14	119.89 MB
java.util.LinkedList$Node	40 bytes	5	43.15 MB
java.lang.Object[]	159.71 kB	49	21.43 MB
java.io.ObjectStreamClass$WeakClassKey	56 bytes	35	2.12 MB
char[]	128 bytes	40	1.94 MB
java.lang.Class	160 bytes	1	1.82 MB

Stack Trace

Stack Trace	TLABs	Total TLAB Size	Pressure
java.util.ArrayList.<init>(int)	1	16.26 MB	75.89%
java.util.AbstractCollection.toArray()	1	1.82 MB	8.49%
java.io.ObjectOutputStream.defaultWriteFields(Object, ObjectStreamClass)	17	1.32 MB	6.17%
java.io.ObjectOutputStream$HandleTable.growEntries()	11	1.27 MB	5.94%
sun.management.MemoryUsageCompositeData.getCompositeData()	4	171.98 kB	0.78%
sun.reflect.misc.MethodUtil.invoke(Method, Object, Object[])	3	164.10 kB	0.75%
java.io.ObjectStreamClass.invokeWriteObject(Object, ObjectOutputStream)	2	71.89 kB	0.33%
java.io.ObjectInputStream$HandleTable.<init>(int)	1	64.12 kB	0.29%
java.lang.reflect.Array.newInstance(Class, int)	1	64.05 kB	0.29%
sun.management.MappedMXBeanType$MapMXBeanType.toOpenTypeData(Object)	1	52.48 kB	0.24%

Dynamic Memory Analysis – Allocation Tracking

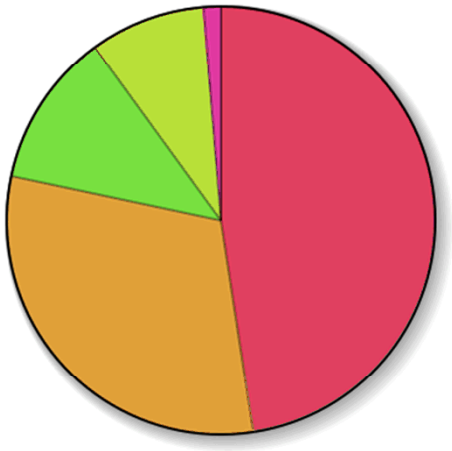
» allocation tracking – **flight recording** using jmc – no byte code instrumentation

- identify large object allocations outside TLAB

General | Allocation in New TLAB | Allocation Outside TLABs

Allocation by Class | Allocation by Thread | Allocation Profile

Allocation Pressure



Class	Objects	Total Size	Pressure
double[]	8	129.37 MB	47.54%
int[]	14	83.76 MB	30.78%
java.util.HashMap$Node[]	2	32.00 MB	11.76%
java.lang.Object[]	10	22.90 MB	8.42%
byte[]	4	4.09 MB	1.50%
java.util.TreeMap$Entry	4	256 bytes	0.00%
java.util.TreeMap	2	160 bytes	0.00%
char[]	1	128 bytes	0.00%
java.util.TreeMap$KeyIterator	1	56 bytes	0.00%
java.lang.Integer	1	24 bytes	0.00%

Stack Trace

Stack Trace	Objects	Total Size	Pressure
gnu.trove.impl.hash.TIntDoubleHash.setUp(int)	4	32.68 MB	39.02%
it.unimi.dsi.fastutil.ints.Int2DoubleOpenHashMap.<init>(int, float)	4	32.00 MB	38.20%
it.unimi.dsi.fastutil.ints.IntArrayList.<init>(int)	4	15.26 MB	18.22%
gnu.trove.list.array.TIntArrayList.<init>(int, int)	1	3.81 MB	4.55%
java.io.ObjectOutputStream$HandleTable.growEntries()	1	1.40 kB	0.00%

Data Structures – Primitives and Objects

» **primitives**: boolean(1), byte(1), char(2), int(4), long(8), float(4), double(8)
- without implicit allocation
- stored in variables or operand stack in frame

» **objects** (object header structure overhead) allocated on the heap
- every object is descendant of Object by default
  - methods – clone(), equals, getClass(), hashCode(), wait(…), notify (…), finalize()
- objects for primitives: Boolean, Byte, Character, Integer, Long, Float, Double; can be **null**
- objects with multiple fields use **type group alignment** and padding in the following order (in the same type group respectinh declaration order):
  - longs and doubles (8B)
  - ints and floats (4B)
  - shorts and chars (2B)
  - bytes and booleans
  - references (4B / 8B)

Object structure (64-bit JVM):
- header 12 or 16 Bytes
- object data super class first

8B - mark word
4B / 8B – Klass ref.
… object data

Data Structures – Object Example 64-bit <32GB Heap

```java
class Structure {
    private boolean boolean1;
    private byte byte1;
    private char char1;
    private short short1;
    private int int1;
    private long long1;
    private float float1;
    private double double1;
    private Object object1;
    private boolean boolean2;
    private byte byte2;
    private char char2;
    private short short2;
    private int int2;
    private long long2;
    private float float2;
    private double double2;
    private Object object2;

    Structure(int value, Object ref

    @Override
    public String toString() {...}
}
```

Object structure (64-bit JVM) **using compressed OOP**:

- object size 80 Bytes

0x00:	mark word					
	Klass ref.		int1			
0x10:	long1					
	double1					
0x20:	long2					
	double2					
0x30:	float1		int2			
	float2		char1		short1	
0x40:	char2	short2	bo1	by1	bo2	by2
	object1 ref.		object2 ref.			

Data Structures – Object Example 64-bit >=32GB Heap

```java
class Structure {
    private boolean boolean1;
    private byte byte1;
    private char char1;
    private short short1;
    private int int1;
    private long long1;
    private float float1;
    private double double1;
    private Object object1;
    private boolean boolean2;
    private byte byte2;
    private char char2;
    private short short2;
    private int int2;
    private long long2;
    private float float2;
    private double double2;
    private Object object2;

    Structure(int value, Object ref

    @Override
    public String toString() {...}
}
```

Object structure (64-bit JVM) **using standard OOP**:

- object size 96 Bytes

<table>
<tr><td>0x00:</td><td colspan="6">mark word</td></tr>
<tr><td></td><td colspan="6">Klass ref.</td></tr>
<tr><td>0x10:</td><td colspan="6">long1</td></tr>
<tr><td></td><td colspan="6">double1</td></tr>
<tr><td>0x20:</td><td colspan="6">long2</td></tr>
<tr><td></td><td colspan="6">double2</td></tr>
<tr><td>0x30:</td><td colspan="4">int1</td><td colspan="2">float1</td></tr>
<tr><td></td><td colspan="4">int2</td><td colspan="2">float2</td></tr>
<tr><td>0x40:</td><td colspan="2">char1</td><td colspan="2">short1</td><td>char2</td><td>short2</td></tr>
<tr><td></td><td>bo1</td><td>by1</td><td>bo2</td><td>by2</td><td colspan="2">empty padding</td></tr>
<tr><td>0x50:</td><td colspan="6">object1 ref.</td></tr>
<tr><td></td><td colspan="6">object2 ref.</td></tr>
</table>

Data Structures – Arrays

» **single-dimension arrays**

- special data structure which store a number of items of the same type in linear order; have the defined limit
- JAVA automatically check limitations
- allocated on the heap
- primitives – keep primitive values directly
- objects – keep references to objects (4B or 8B references)

» **multi-dimensional arrays**

- arrays of arrays - ragged array (non-uniform sub-level lengths)
- slower access due to dereferencing (multiple memory read operations) and multi index bound checks
- consider **flatten array**

Array object structure (64-bit JVM):

- header 16 or 20 Bytes
- sequence of array values

8B - mark word
4B / 8B – Klass ref.
4B – array length
sequence of values

Memory Efficiency – Primitive Objects

» **memory efficiency** – 100% efficiency means zero overhead

$$\frac{useful_content_size}{retained_size} * 100[\%]$$

» correlates with **cache efficacy**

- all others data in cache line (64 B) are read as well

» **data locality** further speed-up processing utilizing already cached data

Object	Useful size	Retained size (Efficiency) <32GB heap	Retained size (Efficiency) >=32GB heap
Boolean	1 bit	16 B (0.78 %)	24 B (0.52 %)
Byte	1 B	16 B (6.25 %)	24 B (4.17 %)
Short, Character	2 B	16 B (12.50 %)	24 B (8.34 %)
Integer, Float	4 B	16 B (25.00 %)	24 B (16.67 %)
Long, Double	8 B	24 B (33.34 %)	24 B (33.34 %)

Primitive Objects

- » **auto boxing** and **un-boxing** during assignment and parameter passing
  - valueOf({primitive}) and {primitve}Value() methods
- » all primitive objects are **immutable** (final values)
- » beware of **inefficiencies** caused by boxing and un-boxing

```
public class PrimitiveObject {
    private static Integer integer = 0;

    public static void main(String[] args) {
        integer++;
    }
}
static {};
  Code:
     0: iconst_0
     1: invokestatic  #4
     4: putstatic     #2
     7: return

public static void main(java.lang.String[]);
  Code:
     0: getstatic     #2
     3: invokevirtual #3
     6: iconst_1
     7: iadd
     8: invokestatic  #4
    11: putstatic     #2
    14: return
```

```
private static Integer integer = 0;

// Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
// Field integer:Ljava/lang/Integer;

integer++;
// Field integer:Ljava/lang/Integer;
// Method java/lang/Integer.intValue:()I

// Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
// Field integer:Ljava/lang/Integer;
```

Conversion Inefficiencies - Example

» count word histogram

```
public static void main(String[] args) {
    Map<String, Integer> m = new TreeMap<String, Integer>();
    for (String word : args) {
        Integer freq = m.get(word);
        m.put(word, (freq == null ? 1 : freq + 1));
    }
    System.out.println(m);
}
```

Conversion Issues - Example

```
int i = 2;
int j = 2;
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(i);
list.add(j);
System.out.printf(Boolean.toString(i==j));
System.out.printf(Boolean.toString(list.get(0)==list.get(1)));
System.out.printf(Boolean.toString(list.get(0).equals(list.get(1))));
```

» what is the output? and what is the output for i=2000 and j=2000 ?

Conversion Issues - Example

```
int i = 2;
int j = 2;
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(i);
list.add(j);
System.out.printf(Boolean.toString(i==j));
System.out.printf(Boolean.toString(list.get(0)==list.get(1)));
System.out.printf(Boolean.toString(list.get(0).equals(list.get(1))));
```

» what is the output? and what is the output for i=2000 and j=2000 ?

true	true
true	false
true	true

<u>Note:</u> after serialization the second is always false

Primitive Object – Identity Semantics

» **identity semantics** using cache for **valueOf({primitive})**

- Short, Integer, Long – caches <-128;+127>
- Byte – caches all values
- Character – caches <0;+127>

» **not working** for objects created by constructor (e.g. new Integer(1))

```
private static class ShortCache {
    private ShortCache(){}

    static final Short cache[] = new Short[-(-128) + 127 + 1];

    static {
        for(int i = 0; i < cache.length; i++)
            cache[i] = new Short((short)(i - 128));
    }
}
public static Short valueOf(short s) {
    final int offset = 128;
    int sAsInt = s;
    if (sAsInt >= -128 && sAsInt <= 127) { // must cache
        return ShortCache.cache[sAsInt + offset];
    }
    return new Short(s);
}

public short shortValue() {
    return value;
}
```

Memory Efficiency – Java Collections

» **LinkedList<E>**
  - uses Node<E> object with bi-directional links

» **ArrayList<E>**
  - backend elementData array with references to objects

» **HashMap<K,V>**
  - backend hash table of Node<K,V> with cached hashCode and linked collisions

Note: Measured for 1 million of elements in Collections and Map

Object	Useful size	Retained size (Efficiency) <32GB heap	Retained size (Efficiency) >=32GB heap
LinkedList<Integer>	4 MiB	34.33 MiB (11.65 %)	47.26 MiB (8.46 %)
ArrayList<Integer>	4 MiB	17.73 MiB (22.56 %)	25.72 MiB (15.55 %)
HashMap<Integer,Double>	12 MiB	70.19 MiB (17.10 %)	87.67 MiB (13.69 %)

Collections for Performance

» **Trove** – Lesser GNU Public License (LGPL)

» **FastUtil** – Apache License 2.0

» **collections for performance**

- type-specific maps, sets, lists and queues
  - remove overheads related auto-boxing and un-boxing
- small memory footprint
  - much better caching
  - sequential access is very fast
- fast access and insertion
- use **open addressing** hashing in Maps instead of chaining approach
- support big collections (>2^{31} elements) in *FastUtil*
- support custom hashing strategies in *Trove*

```
char[] foo, bar;
foo = new char[] {'a','b','c'};
bar = new char[] {'a','b','c'};
System.out.println(foo.hashCode() == bar.hashCode() ? "equal" : "not equal");
System.out.println(foo.equals(bar) ? "equal" : "not equal");
```

Open Addressing Hash Table

» **eliminates** the need for Map.Entry<K,V> **wrapper** supporting chaining
  - typed keys & values arrays
  - state byte array – FREE, FULL, REMOVED (*Trove*, total 3 arrays)
  - special 0/null key tracking + default return value for empty (*FastUtil*, total 2 arrays)

» smaller load factor implies less conflicts (*Trove* 0.5, *FastUtil* 0.75)

» **collision resolution** scheme
  - linear probing (*FastUtil*) – better cache utilization due to data locality
  - double hash probing (Trove) – less conflicts

$$h(i,k) = \left(h_1(k) + i \cdot h_2(k)\right) \bmod |T|$$

    – h_2 cannot be 0

» complex **deletion** to keep conflict searching consistent
  - shift last collision element instead of removed (*FastUtil*)
  - keep removed elements – used by later puts (*Trove*)

» usage of **prime number** size of hash table reduce hashing collisions (*Trove*)

» usage of **power of two** size of hash table leads to fast bit operations (*FastUtil*)

Memory Efficiency – Collections for Performance

Note: 1 million of elements stored

Object	Useful size	Retained size (Efficiency) <32GB heap	Retained size (Efficiency) >=32GB heap
LinkedList<Integer>	4 MiB	34.33 MiB (11.65 %)	47.26 MiB (8.46 %)
TIntLinkedList *(Trove)*	4 MiB	20.60 MiB (19.42 %)	24.54 MiB (13.54 %)
ArrayList<Integer>	4 MiB	17.73 MiB (22.56 %)	25.72 MiB (15.55 %)
TIntArrayList *(Trove)*	4 MiB	~4.00 MiB (~100.00 %)	~4.00 MiB (~100.00%)
IntArrayList *(FastUtil)*	4 MiB	~4.00 MiB (~100.00 %)	~4.00 MiB (~100.00%)
HashMap<Integer,Double>	12 MiB	70.19 MiB (17.10 %)	87.67 MiB (13.69 %)
TIntDoubleHashMap *(Trove)*	12 MiB	27.85 MiB (43.09 %)	27.85 MiB (43.09 %)
Int2DoubleOpenHashMap *(FastUtil)*	12 MiB	25.17 MiB (47.68 %)	25.17 MiB (47.68 %)

Collection Resizing – Default Expected Capacity

- **run-time inefficiencies** caused by collection resizing
  - explicitly specify expected collection capacity
- **ArrayList**
  - shared static default empty backend array
  - backend array default capacity 10 (allocated during first add)
  - grow implies copy of all previous elements - strategy +~50%
  - no automatic shrinking, manual using trimToSize
- **TIntArrayList** (Trove)
  - backend array default capacity 10 (allocated immediately)
  - grow implies copy of all previous elements - strategy *2
  - no automatic shrinking, manual using trimToSize
- **IntArrayList** (FastUtil)
  - backend array default capacity 16 (allocated immediately)
  - grow implies copy of all previous elements - strategy *2
  - no automatic shrinking, manual using trim

Collection Resizing – Default Expected Capacity

- **HashMap**
  - hash table initialized with the first element
  - default hash table size 16 (default load factor 0.75)
    - custom capacity rounded to power of two
  - grow implies re-hashing (iteration + puts) of all previous elements
    - strategy *2
  - hash table shrinking not supported at all
- **TIntDoubleHashMap** (Trove)
  - default hash table size 23 (default load factor 0.5)
    - custom capacity adjusted to nearest bigger prime number
  - grow implies re-hashing (iteration + puts) of all previous elements
    - strategy nearest bigger prime number for size * 2
  - auto compaction after certain number of removals
    - nearest bigger prime number for the currently stored elements
    - can be temporarily disabled if you are planning to do a lot of removals

Collection Resizing – Default Expected Capacity

» **Int2DoubleOpenHashMap**

- backend arrays allocated immediately
- default hash table size 16 (default load factor 0.75)
  - custom capacity rounded to power of two
- grow implies re-hashing (iteration + puts) of all previous elements
  - strategy *2
- auto shrinking after remove if used less than ¼ - strategy :2
  - not shrinking under minimum hash table size 16

» **further optimizations** possible

- use stubs for no/one element collections when your application contains a lot of collections

Effective Software

Lecture 11: JVM - Object Allocation, Bloom Filters, References, Effective Caching

David Šišlák
david.sislak@fel.cvut.cz

[1] Tarkoma, S., Esteve, R.,Lagerspetz, E.: Theory and Practice of Bloom Filters for Distributed Systems. IEEE Communications Surveys and Tutorials, issue 3, vol. 14, 2012.

[2] Oaks, S.: Java Performance: The Definitive Guide. O'Reilly, USA 2014.

[3] JVM source code - http://openjdk.java.net

Fast Object Allocation

- based on **bump-the-pointer** technique
  - track previously allocated object
  - fit new object into remainder of generation end
- **thread-local allocation buffers** (TLABs)
  - each thread has small exclusive area (few % of Eden in total) **aligned NUMA**
  - remove concurrency bottleneck
    - no synchronization among threads (remove slower atomics)
    - remove false sharing (cache line used just by one CPU core)
  - exclusive allocation takes about *few native instructions*
  - infrequent full TLABs implies synchronization (based on *lock inc*)
  - thread-based adaptive resizing of TLAB
    - not working well for thread pools with varying allocation pressure
- tuning options
  - -XX:+UseTLAB ; -XX:AllocatePrefetchStyle=1; -XX:+PrintTLAB
  - -XX:AllocateInstancePrefetchLines=1 ; -XX:AllocatePrefetchLines=3
  - -XX:+ResizeTLAB ; -XX:TLABSize=10k ; -XX:MinTLABSize=2k

Fast Object Allocation - Example

C2 compiler, standard OOP, size 96 Bytes:

```
mov     0x60(%r15),%r11
mov     %r11,%r10
add     $0x60,%r10
cmp     0x70(%r15),%r10
jae     0x0000000107895244
mov     %r10,0x60(%r15)
prefetchnta 0xc0(%r10)
mov     %r11,%rdi
add     $0x10,%rdi
mov     $0xa,%ecx
movabs  $0x220558080,%r10  ;   {metadata('Structure')}
mov     0xa8(%r10),%r8
mov     %r8,(%r11)
mov     %r10,0x8(%r11)
xor     %rax,%rax
shl     $0x3,%rcx
rep rex.W stos %al,%es:(%rdi)  ;*new
                          ; - StructureTest::allocate@4 (line 5)
```

- read **TLAB** allocation pointer
- bump the pointer
- fits into **TLAB** check
- store **TLAB** allocation pointer
- **prefetch** 3 cache lines ahead
- prepare for **object nulling**
- RDI object data; ECX=10 qwords
- fill **object header**
- **null instance**

```
class Structure {
    private boolean boolean1;
    private byte byte1;
    private char char1;
    private short short1;
    private int int1;
    private long long1;
    private float float1;
    private double double1;
    private Object object1;
    private boolean boolean2;
    private byte byte2;
    private char char2;
    private short short2;
    private int int2;
    private long long2;
    private float float2;
    private double double2;
    private Object object2;

    Structure(int value, Object r

    @Override
    public String toString() {...
}
```

8B - mark word
4B / 8B – Klass ref.
… object data

Note: all examples are in JVM 8 64-bit, Intel Haswell CPU, AT&T syntax

Flight Recording to Analyze TLAB

example with million of allocations of Structure, compressed OOP used

General | Allocation in New TLAB | Allocation Outside TLABs

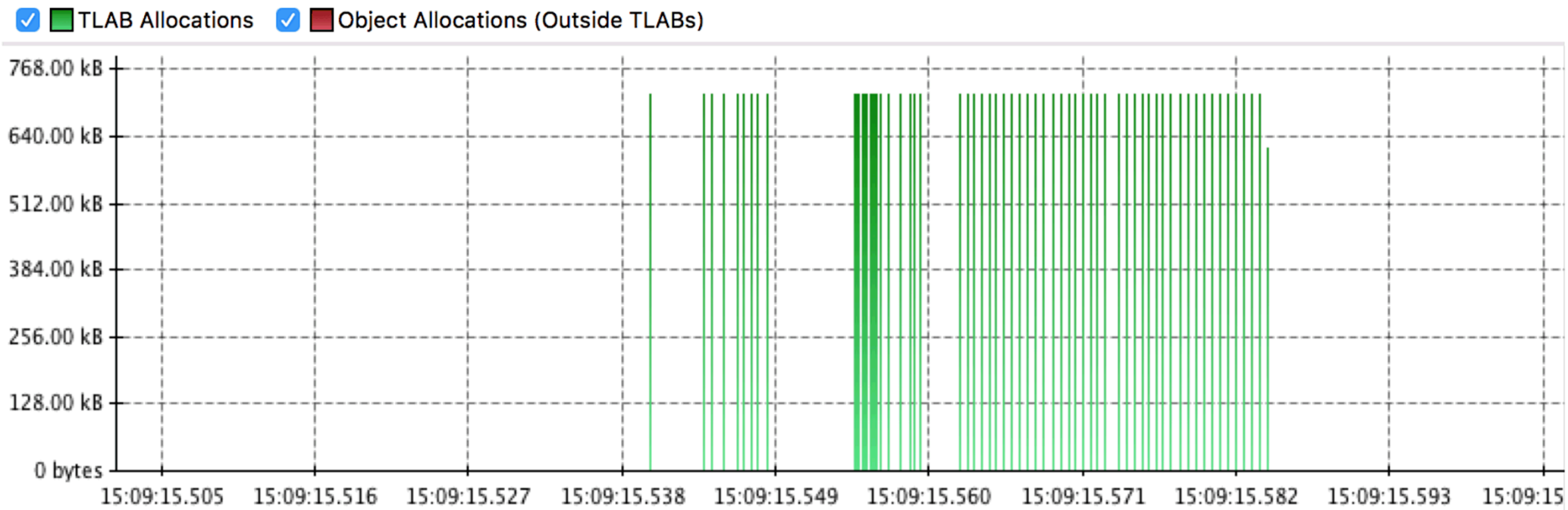
Thread Local Allocation Buffer (TLAB) Statistics

TLAB Count	65
Maximum TLAB Size	720.58 kB
Minimum TLAB Size	615.77 kB
Average TLAB Size	718.96 kB
Total Memory Allocated for TLABs	45.64 MB
Allocation Rate for TLABs	439.59 MB/s

Statistics for Object Allocations (Outside TLABs)

Object Count	0
Maximum Object Size	N/A
Minimum Object Size	N/A
Average Object Size	N/A
Total Memory Allocated for Objects	N/A
Allocation Rate for Objects	N/A

Allocation



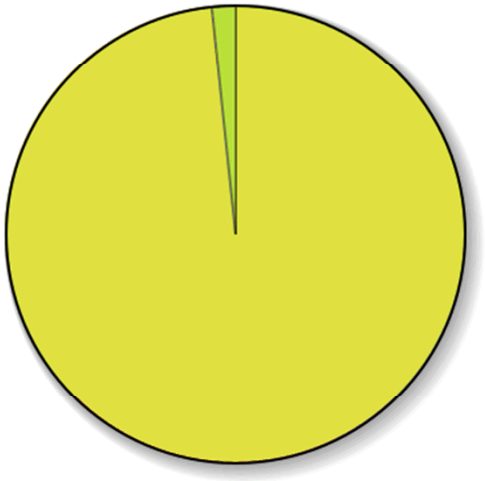
Flight Recording to Analyze TLAB

example with million of allocations of Structure; compressed OOP used

General | Allocation in New TLAB | Allocation Outside TLABs

Allocation by Class | Allocation by Thread | Allocation Profile

Allocation Pressure



Class	Average Object Size	TLABs	Total TLAB Size	Pressure
Structure	80 bytes	64	44.93 MB	98.46%
java.lang.Object[]	88 bytes	1	720.58 kB	1.54%

Stack Trace

Stack Trace	TLABs	Total TLAB Size	Pressure
StructureTest.allocate(int, Object)	64	44.93 MB	100.00%

Example – Dynamic Memory Analysis

```
public static String[] method1(String[] args) {
    return Arrays.stream(args).
            filter(t -> t.matches( regex: "[^0-9]+")).
            sorted(Comparator.<String,String>comparing(String::toLowerCase).reversed()).
            collect(Collectors.toList()).toArray(new String[0]);
}
```

Example – Dynamic Memory Analysis

```
public static String[] method1(String[] args) {
    return Arrays.stream(args).
            filter(t -> t.matches( regex: "[^0-9]+")).
            sorted(Comparator.<String,String>comparing(String::toLowerCase).reversed()).
            collect(Collectors.toList()).toArray(new String[0]);
}
```

allocations when called with 40 elements (27 without digits):

method1(

Method	Objects		Size		Objects (Own)	Size (Own)
Lambda.method1(String[]) Lambda.java	1,486	99 %	101,944	99 %	1	16

Classes | Packages | Object Explorer | Generations | Ages | Reachability | Class Loaders | Web Applications | Callees List | Merged Callees

Class list for objects selected in the upper table

Class	Objects		Shallow Size		Retained Size	
int[]	167	11 %	12,648	12 %	12,648	12 %
byte[]	57	4 %	12,136	12 %	12,136	12 %
char[]	179	12 %	10,928	11 %	10,928	11 %
boolean[]	40	3 %	10,880	11 %	10,880	11 %
jdk.internal.org.objectweb.asm.Item	180	12 %	10,080	10 %	10,080	10 %
jdk.internal.org.objectweb.asm.Item[]	7	0 %	7,280	7 %	7,280	7 %
java.lang.Class	7	0 %	3,968	4 %	3,968	4 %
jdk.internal.org.objectweb.asm.MethodWriter	15	1 %	3,360	3 %	3,360	3 %
java.util.regex.Pattern	40	3 %	2,880	3 %	2,880	3 %
java.lang.String	113	8 %	2,712	3 %	2,712	3 %
java.util.regex.Matcher	40	3 %	2,560	3 %	2,560	3 %
java.util.regex.Pattern$GroupHead[]	40	3 %	2,240	2 %	2,240	2 %
java.util.regex.Pattern$Curly	40	3 %	1,280	1 %	1,280	1 %
java.lang.invoke.MethodType	30	2 %	1,200	1 %	1,200	1 %
java.lang.Object[]	24	2 %	1,200	1 %	1,200	1 %
jdk.internal.org.objectweb.asm.ClassWriter	7	0 %	1,176	1 %	1,176	1 %
java.lang.invoke.MemberName	15	1 %	840	1 %	1,016	1 %
java.lang.invoke.MethodType$ConcurrentWeakInternSet$WeakEntry	30	2 %	960	1 %	960	1 %

Example – Optimized – Dynamic Memory Analysis

```
private static Comparator<String> reverseIgnoreCaseComparator =  String.CASE_INSENSITIVE_ORDER.reversed();

public static String[] reversedAlphabeticalOnlyOrderOptimized(String[] args) {
    String[] arr = new String[args.length];
    int i = 0;
    for (String arg : args) {
        boolean filterOut = false;
        for (int k = 0; k < arg.length(); k++) {
            char c = arg.charAt(k);
            if ((c >= '0') && (c <= '9')) {
                filterOut = true;
                break;
            }
        }
        if (!filterOut) arr[i++] = arg;
    }
    Arrays.sort(arr, fromIndex: 0, i, reverseIgnoreCaseComparator);

    return Arrays.copyOf(arr, i);
}
```

Example – Optimized – Dynamic Memory Analysis

```
private static Comparator<String> reverseIgnoreCaseComparator =  String.CASE_INSENSITIVE_ORDER.reversed();

public static String[] reversedAlphabeticalOnlyOrderOptimized(String[] args) {
    String[] arr = new String[args.length];
    int i = 0;
    for (String arg : args) {
        boolean filterOut = false;
        for (int k = 0; k < arg.length(); k++) {
            char c = arg.charAt(k);
            if ((c >= '0') && (c <= '9')) {
                filterOut = true;
                break;
            }
        }
        if (!filterOut) arr[i++] = arg;
    }
    Arrays.sort(arr, fromIndex: 0, i, reverseIgnoreCaseComparator);

    return Arrays.copyOf(arr, i);
}
```

allocations when called with 40 elements (27 without digits):

rever

Method	Objects		Size		Objects (Own)	Size (Own)
Lambda.reversedAlphabeticalOnlyOrderOptimized(String[]) Lambda.java	2	50 %	304	76 %	1	176

Classes | Packages | Object Explorer | Generations | Ages | Reachability | Class Loaders | Web Applications | Callees List | Merged Callees

Objects selected in the upper table

Class name, string value, thread name or ID (Press "Enter" to apply / ?):

Name	Retained Size	Shallow Size
[Unreachable] java.lang.String[40]	176	176
java.lang.String[27] [Stack Local]	128	128

Know Your Application Behavior

- simple code could be very inefficient – **know what you are using**
- a lot of **small short-lived objects** still slow down your application
  - allocations in TLAB are quite fast but not as fast as no allocation
    - check ***escape analysis*** or change your code
  - objects in TLAB fulfill cache **data locality** and are **NUMA** aligned
  - **no false sharing** between cores (data in cache line are just used by one CPU core)
  - increase pressure on young generation and thus minor GC
    - other objects are promoted earlier to old generation
    - increase number of major GC
- a lot of **long-lived objects** slow your application even more
  - each time all live objects have to be traversed
  - compacting GC have to copy objects
    - breaks original data locality
    - can imply false sharing between cores

Escape Analysis – Not All Objects Are Allocated

» **C2 compiler** perform **escape analysis** of new object *after inline of hot methods*

» each new object allocation is classified into one of the following types:

- ***NoEscape*** – object does not escape method in which it is created
  - all its usages are inlined
  - never assigned to static or object field, just to local variables
  - at any point must be JIT-time determinable and not depending on any unpredictable control flow
  - if the object is an **array**, indexing into it must be JIT-time constant
- ***ArgEscape*** – object is passed as, or referenced from, an argument to a method but does not escape the current thread
- ***GlobalEscape*** – object is accessed by different method and thread

» *NoEscape* objects are **not allocated at all** but JIT does **scalar replacement**

- object deconstructed into its constituent fields (stack allocated)
- disappear automatically after stack frame pop (return from the method)
- no GC impact at all + do not need track references (write comp. barrier)

» *ArgEscape* objects are allocated on the heap but all monitors are eliminated

Escape Analysis Example

```java
public static class Vector {
    private final int a1, a2;

    public Vector(int a1, int a2) {
        this.a1 = a1;
        this.a2 = a2;
    }

    public Vector add(Vector v) {
        return new Vector(a1+v.getA1(),a2+v.getA2());
    }

    public int mul(Vector v) {
        return v.getA1()*a1 +  v.getA2()*a2;
    }

    public int getA1() {
        return a1;
    }

    public int getA2() {
        return a2;
    }
}


public int compute(int val) {
    Vector v = new Vector(val+1, val*2);
    synchronized (v) {
        return v.add(v).mul(v);
    }
}
```

Escape Analysis Example

C2 compilation with inline:

```
74   22   !   4        EscapeExample::compute (37 bytes)
                          @ 10   EscapeExample$Vector::<init> (15 bytes)   inline (hot)
                            @ 1   java.lang.Object::<init> (1 bytes)   inline (hot)
                          @ 20   EscapeExample$Vector::add (26 bytes)   inline (hot)
                            @ 9   EscapeExample$Vector::getA1 (5 bytes)   accessor
                            @ 18   EscapeExample$Vector::getA2 (5 bytes)   accessor
                            @ 22   EscapeExample$Vector::<init> (15 bytes)   inline (hot)
                              @ 1   java.lang.Object::<init> (1 bytes)   inline (hot)
                          @ 24   EscapeExample$Vector::mul (20 bytes)   inline (hot)
                            @ 1   EscapeExample$Vector::getA1 (5 bytes)   accessor
                            @ 10   EscapeExample$Vector::getA2 (5 bytes)   accessor
```

```
sub     $0x18,%rsp
mov     %rbp,0x10(%rsp)
mov     %edx,%r11d
add     %edx,%r11d
mov     %edx,%r10d
shl     %r10d
mov     %r10d,%r8d
add     %r10d,%r8d
imul    %r10d,%r8d
add     $0x2,%r11d
inc     %edx
imul    %r11d,%edx
mov     %edx,%eax
add     %r8d,%eax
add     $0x10,%rsp
pop     %rbp
test    %eax,-0x21742ec(%rip)
retq
```

```
public int compute(int val) {
    Vector v = new Vector(val+1, val*2);
    synchronized (v) {
        return v.add(v).mul(v);
    }
}
```

```
# this:     rsi:rsi    = 'EscapeExample'
# parm0:    rdx        = int
#           [sp+0x20]  (sp of caller)
```

no allocation at all, no synchronization

all done out of stack in registers only

Bloom Filter

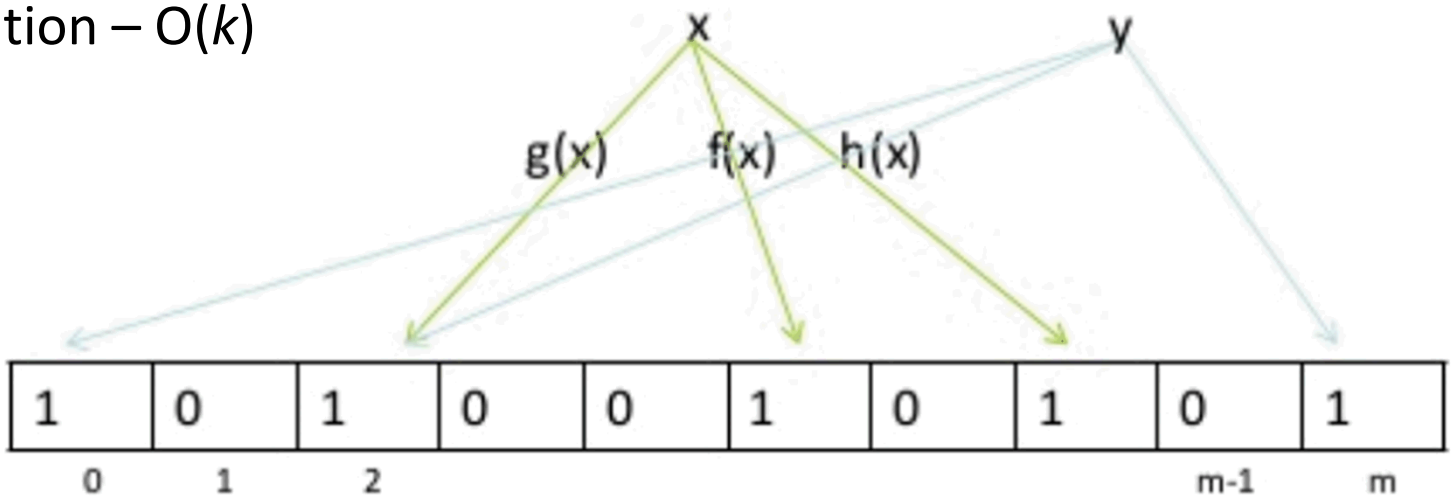
- **bloom filter** operations
  - add a new object to the set
  - test whether a given object is a member of the set
  - **no deletion** is possible
- **strong memory reduction** (few bits per element) compared to other collections
  - compensated by **small false positive** rate (usually 1%)
  - guaranteed **no false negative**
  - not storing object itself (where *all standard collections must store objects*)
- always **constant** add and test/query **complexity** (even for collisions)
- very useful in **big data** processing and other applications
  - used to test that the **object is certainly not present**
  - e.g. reduce a lot of I/O operations reading full collections in a particular file where bloom filters are kept in RAM or read quickly from disk

Bloom Filter

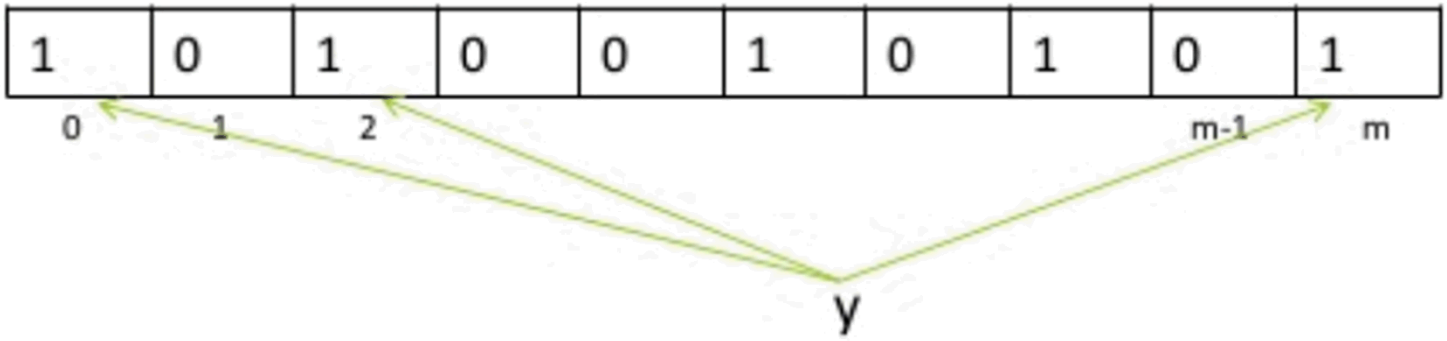
» use bit array with a *m* bits

» use *k* independent hash functions

» **add** operation – O(*k*)



» **query** operation



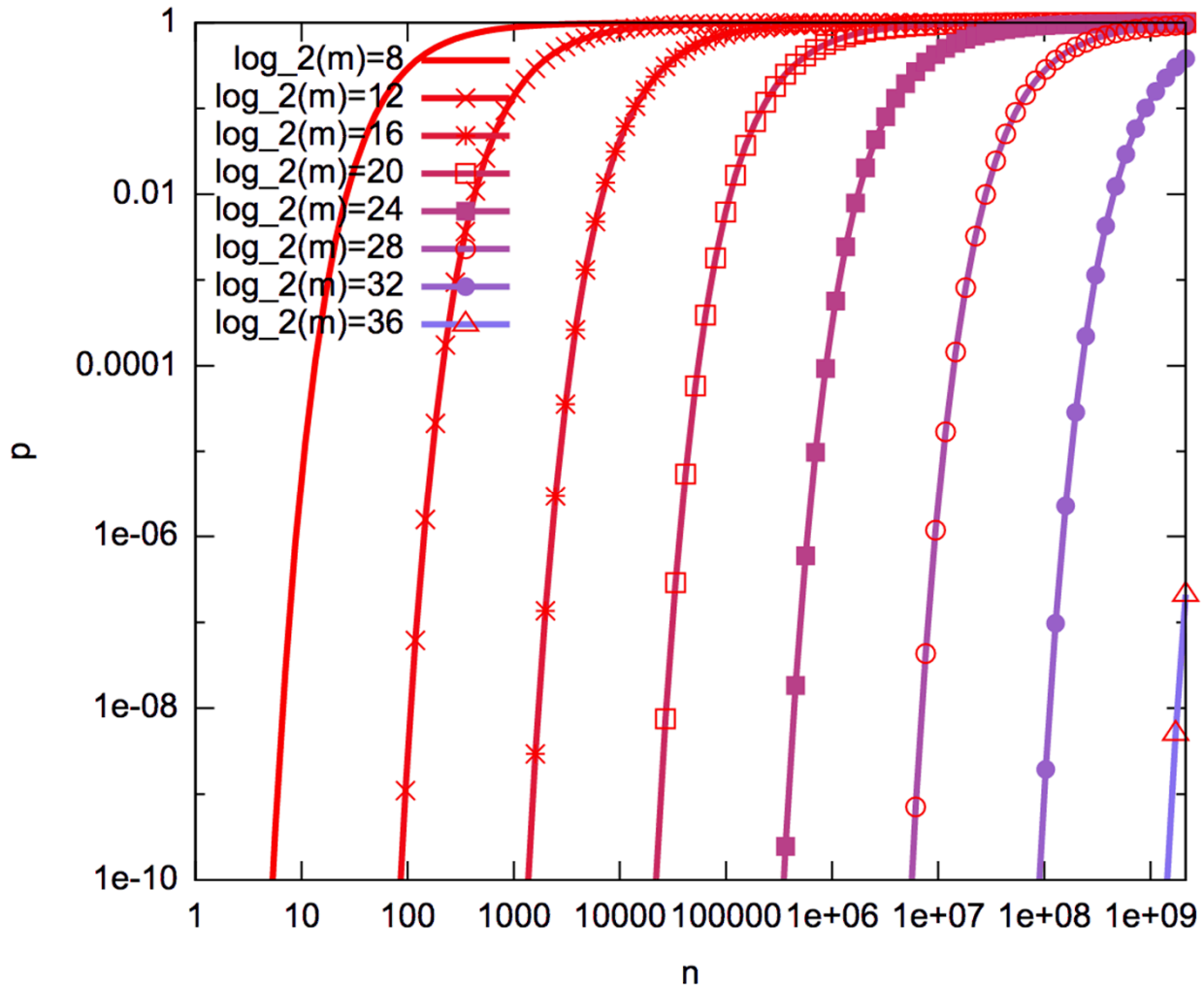
Bloom Filter

» number of bits in the filter

$$ceil\left[\frac{n \cdot \ln(p)}{\ln\left(\frac{1}{2^{\ln(2)}}\right)}\right]$$

» number of hash functions

$$round\left(\frac{\ln(2) \cdot m}{n}\right)$$



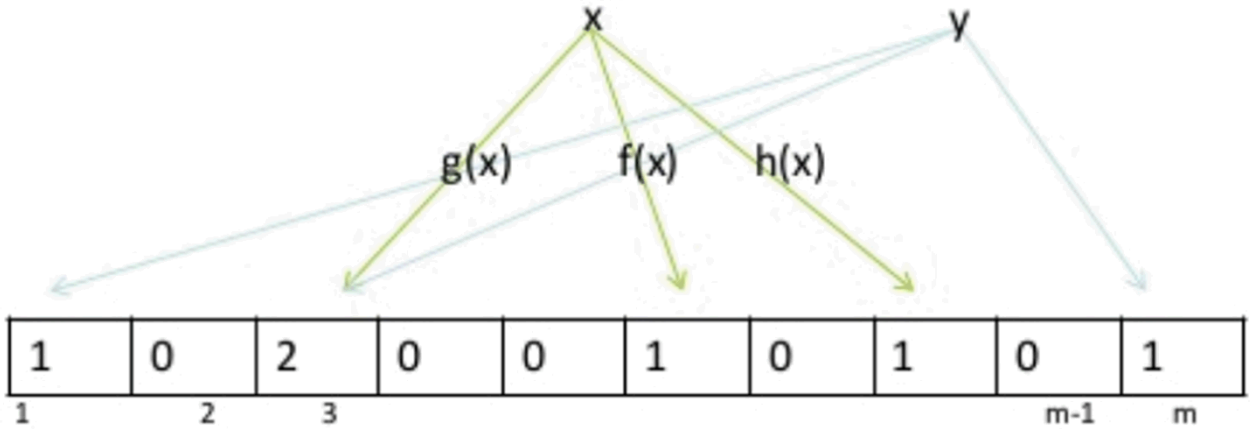
» <u>example</u> – store 1 million of Strings with total size 25 MB

- Set<String> requires >50 MB retained size
- Bloom Filter with FP rate 1% requires 1.13 MB and 7 hash functions
  - more than 44 times smaller and in 99% cases query is TP

Extensions of Bloom Filter

» **counting bloom filter**

- support **delete** and **count estimate** operation



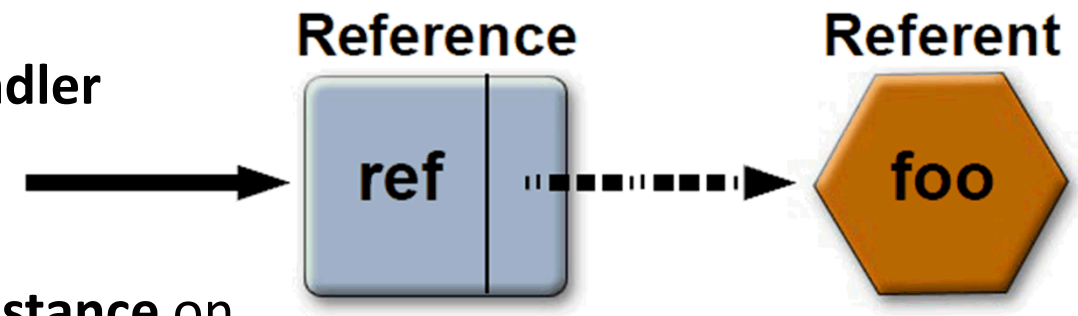
- each position in filter is buckets (e.g. 3 bits) working as counter
  - add – increment
  - delete – decrement; count is min value
  - query – test non-zero
- bucket overflow problem
  - no more increments when there is max counter value
  - increasing FN errors by deletions of elements

» **bitwise bloom filter**

- multiple counting (dynamically added) filters to address issues above

Reference Objects

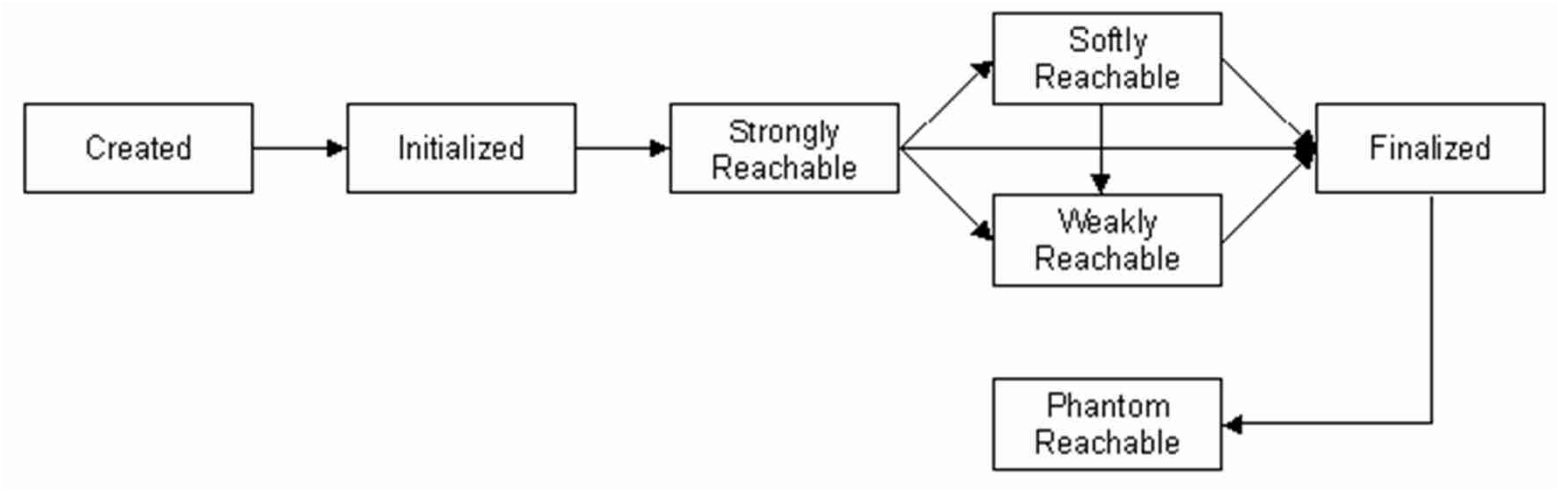
- » mortem hooks more **flexible** than finalization
- » reference types (ordered from strongest one):
  - {strong reference}
  - soft reference – optional reference queue
  - weak reference – optional reference queue
  - {final reference} – mandatory reference queue
  - phantom references – mandatory reference queue
- » can enqueue the reference object on a designated **reference queue** when GC finds its referent to be less reachable, referent is released
- » references are enqueued **only if you have strong reference to REFERENCE**
- » GC has to run to pass them to **Reference Handler** to enqueue them into **reference queue**



- » Reference is **another instance** on the heap – 48 Bytes for standard OOP, 64-bit JVM

```
ref  = new WeakReference(foo, rq);
```

Reachability of Object



- » **strongly reachable** – from GC roots without any Reference object
- » **softly reachable** – not strongly, but can be reached via soft reference
- » **weakly reachable** – not strongly, not softly, but can be reached via weak reference; clear referent link and become eligible for finalization
- » **eligible for finalization** – not strongly, not softly, not weakly and have non-trivial finalize method
- » **phantom reachable** – not strongly, not softly, not weakly, already finalized or no finalize method, but can be reached via phantom reference
- » **unreachable** – none of above; eligible for reclamation

Weak Reference

- pre-finalization processing
- usage:
  - **do not retain this object because of this reference**
  - don't own target, e.g. listeners
  - canonicalizing map – e.g. ObjectOutputStream
  - implement **flexible version of finalization**:
    - prioritize
    - decide when to run finalization
- get() returns
  - referent if not cleared
  - null, otherwise
- **referent is cleared** by GC (cleared when passed to Reference Handler) and **can be reclaimed**
- need **copy referent to strong reference and check that it is not null before using it**
- WeakHashMap<K,V> - uses weak keys; cleanup during all standard operations

Weak Reference – External Resource Clean-up

» **clean-up** approach for ReferenceQueue<T>

- own **dedicated thread**

```
ReferenceQueue<Image3> refQueue =
    NativeImage3.referenceQueue();
while (true) {
    NativeImage3 nativeImg =
        (NativeImage3) refQueue.remove();
    nativeImg.dispose();
}
```

- clean-up **before creation of new** objects
  - limited clean-up processing to mitigate long processing
  - use poll() – non-blocking fetch of first

Custom Finalizer Example

```java
public abstract class CustomFinalizer extends WeakReference<Object> {
    private static final ReferenceQueue<Object> referenceQueue = new ReferenceQueue<>();
    private static final CustomFinalizer circularEnd = new CustomFinalizer() {...};

    private CustomFinalizer next, prev;

    public CustomFinalizer(Object referent) {...}

    private CustomFinalizer() {...}

    private void executeCustomFinalize() {...}

    public abstract void customFinalize();

    static {
        Thread cleanupThread = new Thread(() -> {
            for (;;) {
                try {
                    CustomFinalizer toCleanup = (CustomFinalizer) referenceQueue.remove();
                    toCleanup.executeCustomFinalize();
                } catch (InterruptedException e) {
                }
            }
        }, name: "Custom finalizer");
        cleanupThread.setDaemon(true);
        cleanupThread.start();
    }
}
```

Custom Finalizer Example

```
public CustomFinalizer(Object referent) {
    super(referent, referenceQueue);
    synchronized (circularEnd) {
        next = circularEnd.next;
        circularEnd.next.prev = this;
        prev = circularEnd;
        circularEnd.next = this;
    }
}

private void executeCustomFinalize() {
    if (next == null) return;
    synchronized (circularEnd) {
        prev.next = next;
        next.prev = prev;
    }
    next = prev = null;
    customFinalize();
}
```

» usage example, beware **of implicit this strong reference** in instance context

```
new CustomFinalizer(monitoredObjectForFinalization) {
    @Override
    public void customFinalize() {
        // custom finalization
    }
};
```

Soft Reference

» pre-finalization processing

» usage:

- **would like to keep referent, but can loose it**
- suitable for **caches** – create strong reference to data to keep them
  - objects with long initialization
  - frequently used information
- reclaim only if there is "**memory pressure**" based on heap usage

now – timestamp > (SoftRefLRUPolicyMSPerMB * amountOfFreeMemoryInMB)

-XX:SoftRefLRUPolicyMSPerMB=N (default 1000)

- all are cleared before OutOfMemoryError

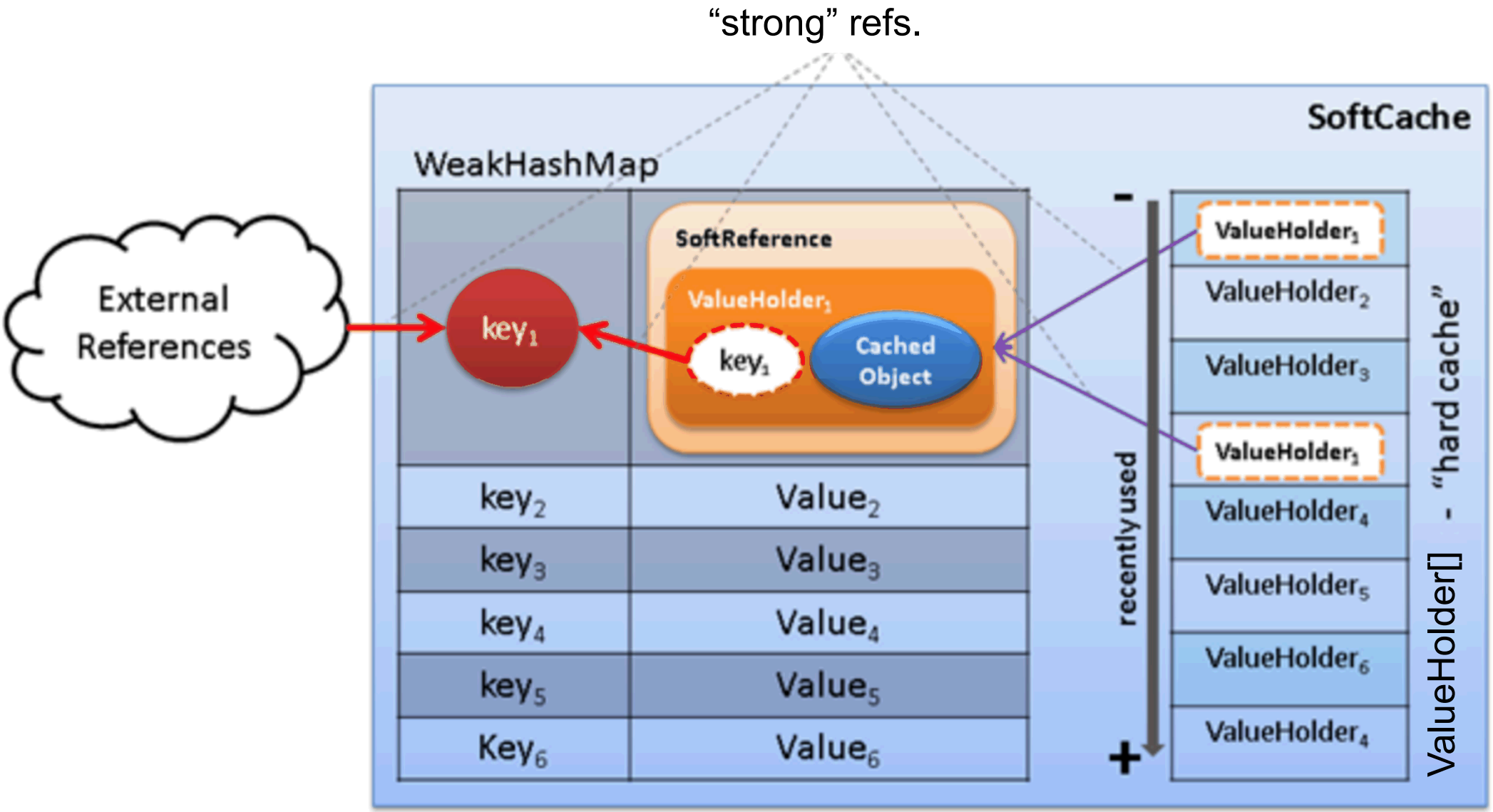
» get() returns:

- referent if not cleared; null, otherwise
- **updates timestamp** of usage (can keep recently used longer)

» **referent is cleared** by GC (cleared when passed to Reference Handler) and **can be reclaimed**

Efficient Cache Example

efficient **LRU** tracking in combination with memory pressure for older



Final Reference – Object with Non-Trivial Finalize

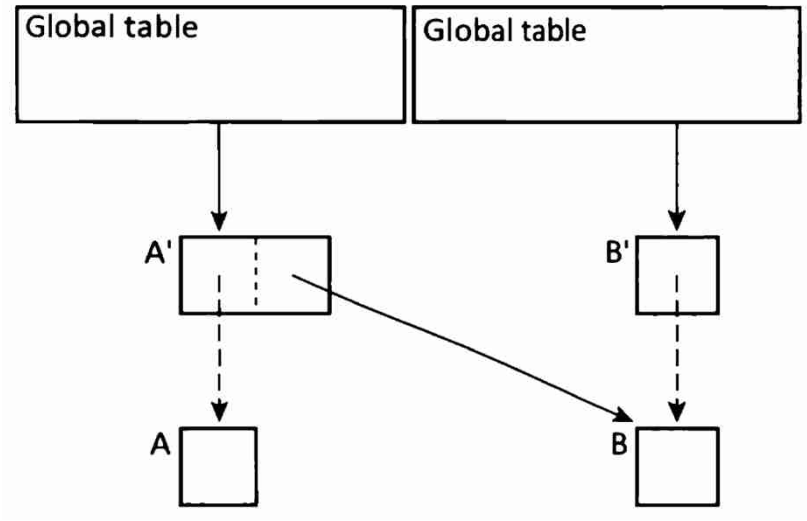
- finalize hook cannot be used directly (package limited)
- instance allocation of object with **non-trivial finalize method**
  - slower allocation than standard objects
  - run time call of **Finalizer.register** with possible global safe point
    - not inlined, all references saved in stack with OopMap
  - allocates **FinalReference** instance and do synchronized tracking
- **referent is not cleared and reclaimed** before finalization
  - **all referenced objects** cannot be reclaimed as well
- only one **Finalizer** thread for all Final references of all types
  - call **finalize** method and **clear** referent
    - **issue** when finalize creates **strong reference again**
    - no priority control between multiple finalize methods
    - long running finalize delays all other finalization
  - daemon thread and JVM can terminate before finalization of all
- finalized objects can be reclaimed during **subsequent GC cycle**

Phantom Reference

» post-finalization processing, pre-mortem hook

» usage:

- **notifies that the object is not used** – before its reclamation
- used to guarantee given order of finalization of objects (not possible with Weak references)
  - A, B – finalizable objects (e.g. Weakly)
  - A’, B’ - PhantomReferences



» get() returns:

- null always
- referent can be read using reflection
  - avoid making strong reference again

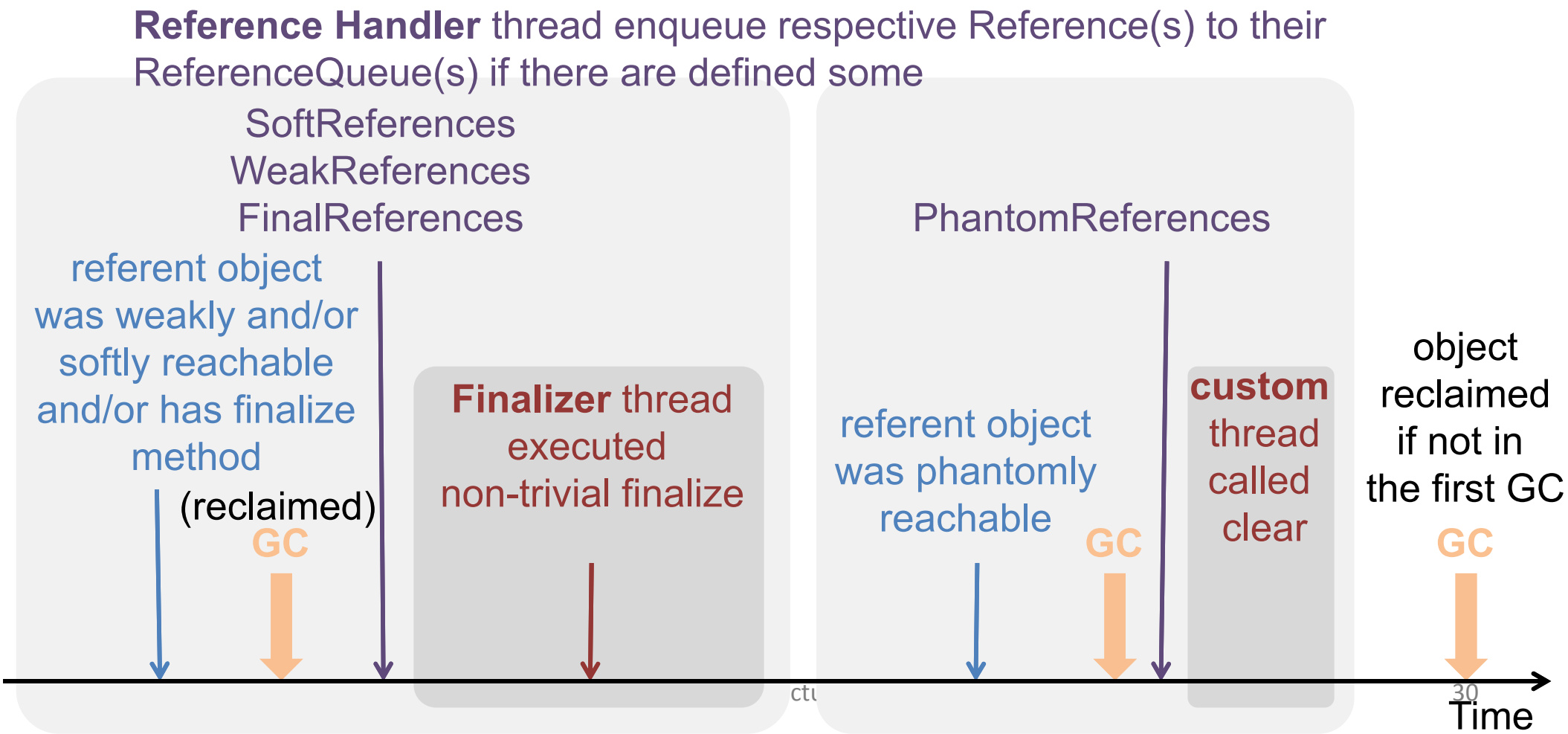
» **have to** specify reference queue for constructor (can be cleared)

» **referent is not cleared and reclaimed** until all phantom references are not become unreachable or manually cleared using method clear()

  » **all referenced objects** cannot be reclaimed as well

Reference Object

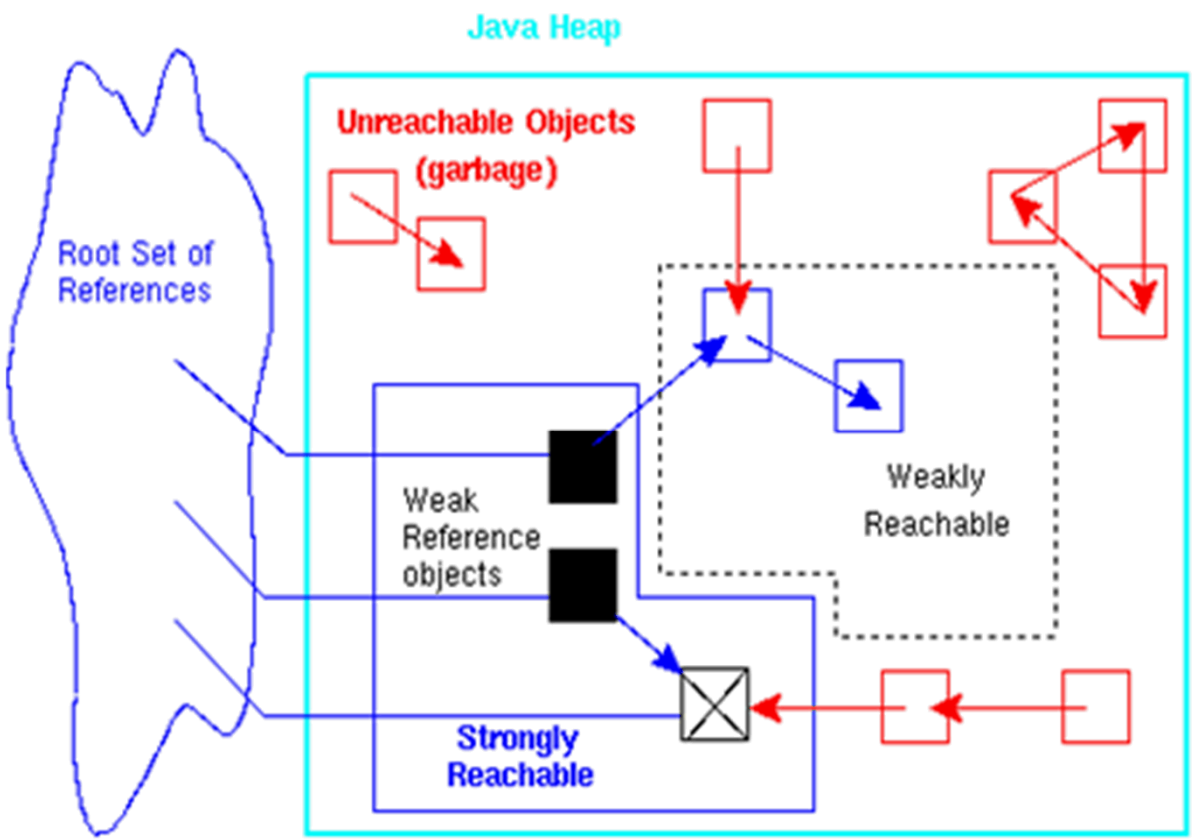
- » only **one GC cycle** needed to reclaim *referent object* if there is only soft references or weak references to the same object
- » **multiple GC cycles** needed for *referent objects* with multiple reference types or have at least one final or phantom reference



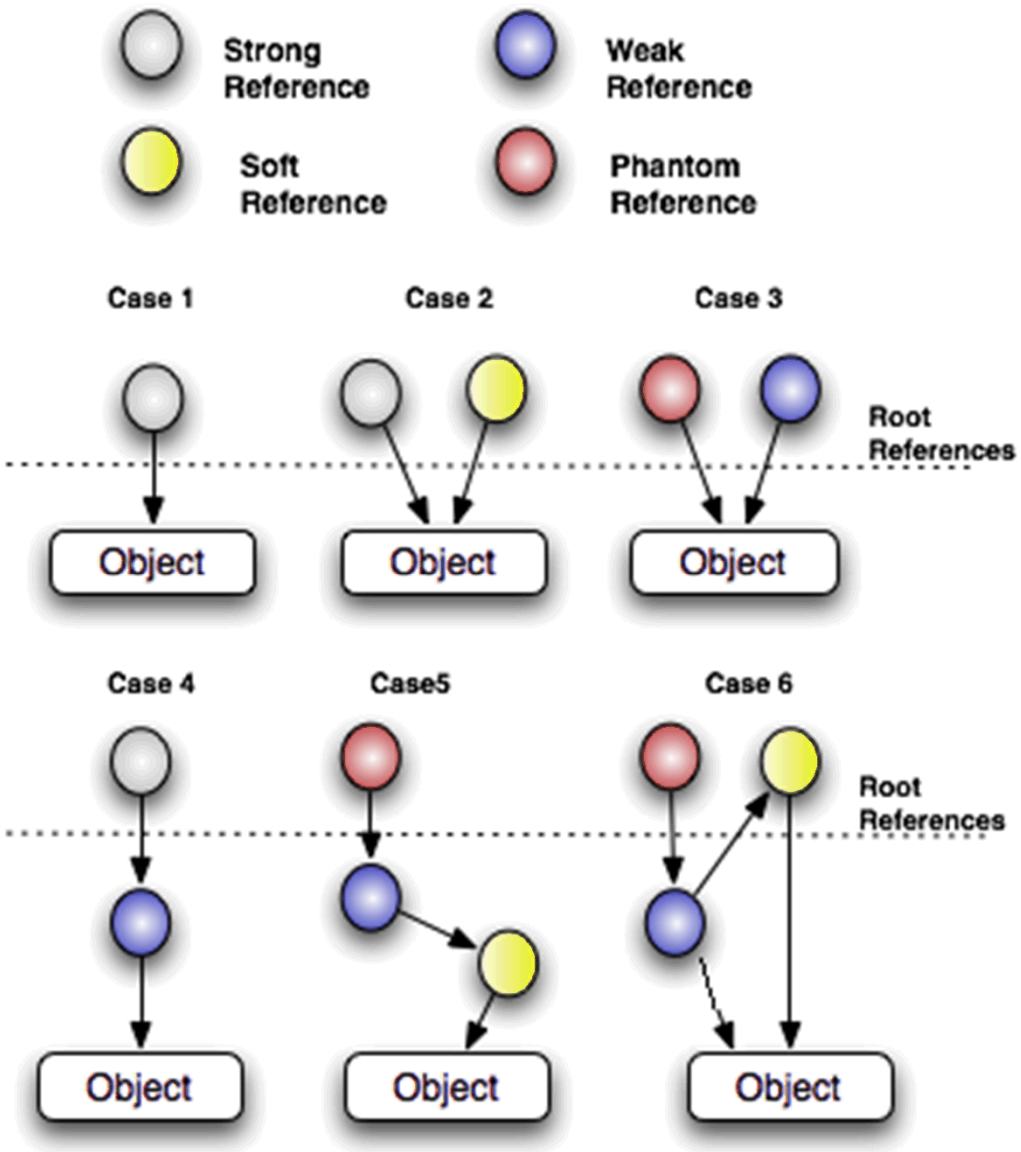
Performance Cost for References

- **creation** cost
  - allocation instance
  - synchronization with tracking of Reference (strong references)
- **garbage collection** cost (-XX:+PrintReferenceGC –XX:+PrintGCDetails)
  - tracking live not follow referents
  - construct list of live References each GC cycle
    - discovered field in Reference
  - per-reference traversal overhead regardless referent is collected or not
    - softly, weakly + finalizable, phantomly
  - Reference Object itself are subject for garbage collection
- **enqueue** cost
  - reference handler enqueue with synchronization
- **reference queue processing** cost
  - synchronized queue consumption

Reachability of Object



Reachability of Object



Effective Software

Lecture 12: Memory Management in JVM – Memory Layout, Garbage Collectors

David Šišlák
david.sislak@fel.cvut.cz

[1] Jones, R., Hosking, A., Moss, E.: The Garbage Collection Handbook. CRC Press, USA 2012.
[2] JVM source code - http://openjdk.java.net
[3] Oaks, S.: Java Performance: The Definitive Guide. O'Reilly, USA 2014.

Automatic Memory Management

» **advantages** over explicit memory management
- no crashes due to errors – e.g. usage of de-allocated objects
- no memory leaks

» *components*
- parts in **application code**
  - allocation
  - read/write references
- **garbage collector**
  - discover unreachable objects (not transiently reachable from **roots** – variables and stack operands in frames, static fields, special native references from JNI)
  - reclaim storage

```
New():
    ref ← allocate()
    if ref = null
        collect()
        ref ← allocate()
        if ref = null
            error "Out of memory"
    return ref
```

Automatic Memory Management

» *desired characteristics*

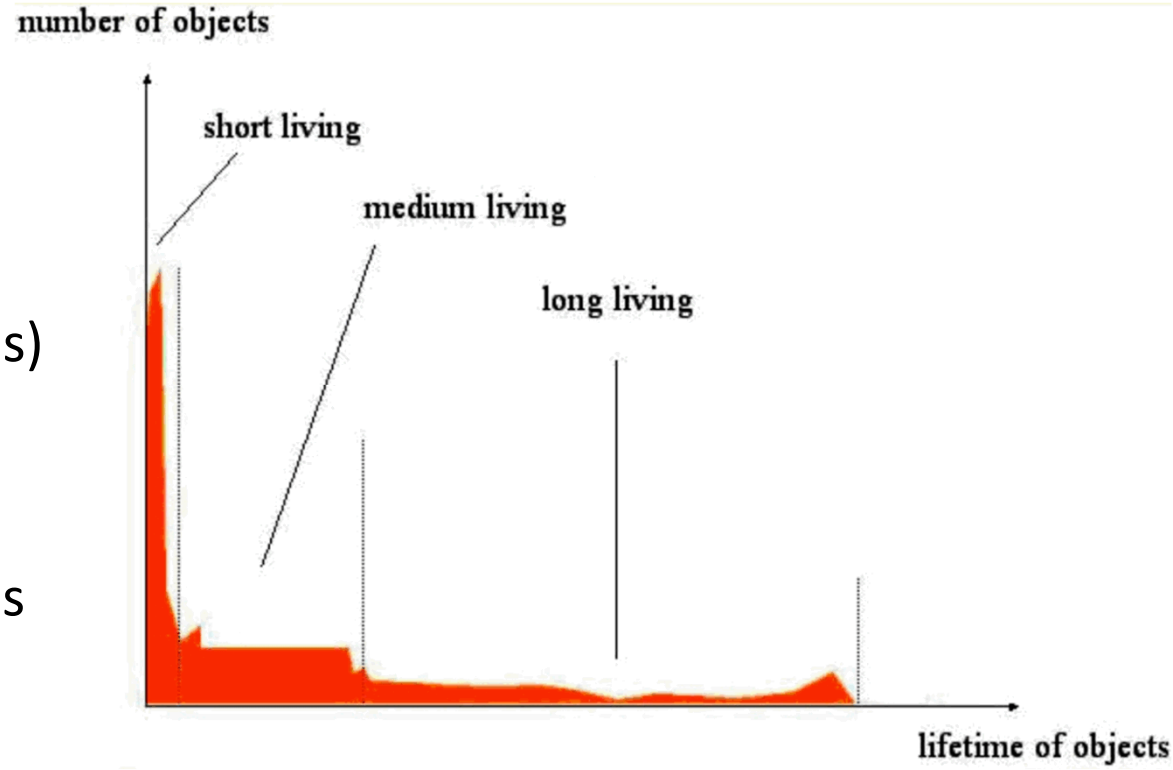
- **safety** – never reclaim space of reachable objects, thread safe
- **throughput** – application code performance
  - allocation performance – avoid fragmentation
  - *handles* or *direct references*
  - expensive reference counting or *cross-region reference tracking*
    - read/write barriers – e.g. added compiled code
  - later reads affected by re-ordering – breaking data locality, false sharing
- **completeness and promptness**
  - eventually all garbage
  - promptness of reclamation – how long garbage occupy memory
- **pause time** – stop the world (global safe point)
- **space overhead**
  - additional cost per capacity/reference
  - double heap for copying
- **scalability and portability -** multicore, large heaps

Generational Concept

» **generational hypothesis**
- **weak** – most objects die young
  - there exist few references from older to younger objects
- **strong** – younger object dies earlier then older

» segregate objects by age into **generations** (JAVA use 2 generations) to **minimize pause time**
- **young**
  - small size
  - frequent fast **minor** collections (milliseconds)
- **old** (tenured)
  - large size
  - rare slow **full** collections (seconds)

» promotion of objects during minor collections



Identify Reachable Objects

» **reference counting**

- additional counter for every object – number of references to the object
- a lot of atomics operations to have it thread-safe
  - slow down application code
- doesn't support cyclic references
- pollute cache a lot with additional memory operations
- can remove objects when counter is 0 immediately with further decreasing counts on reference objects

Identify Reachable Objects

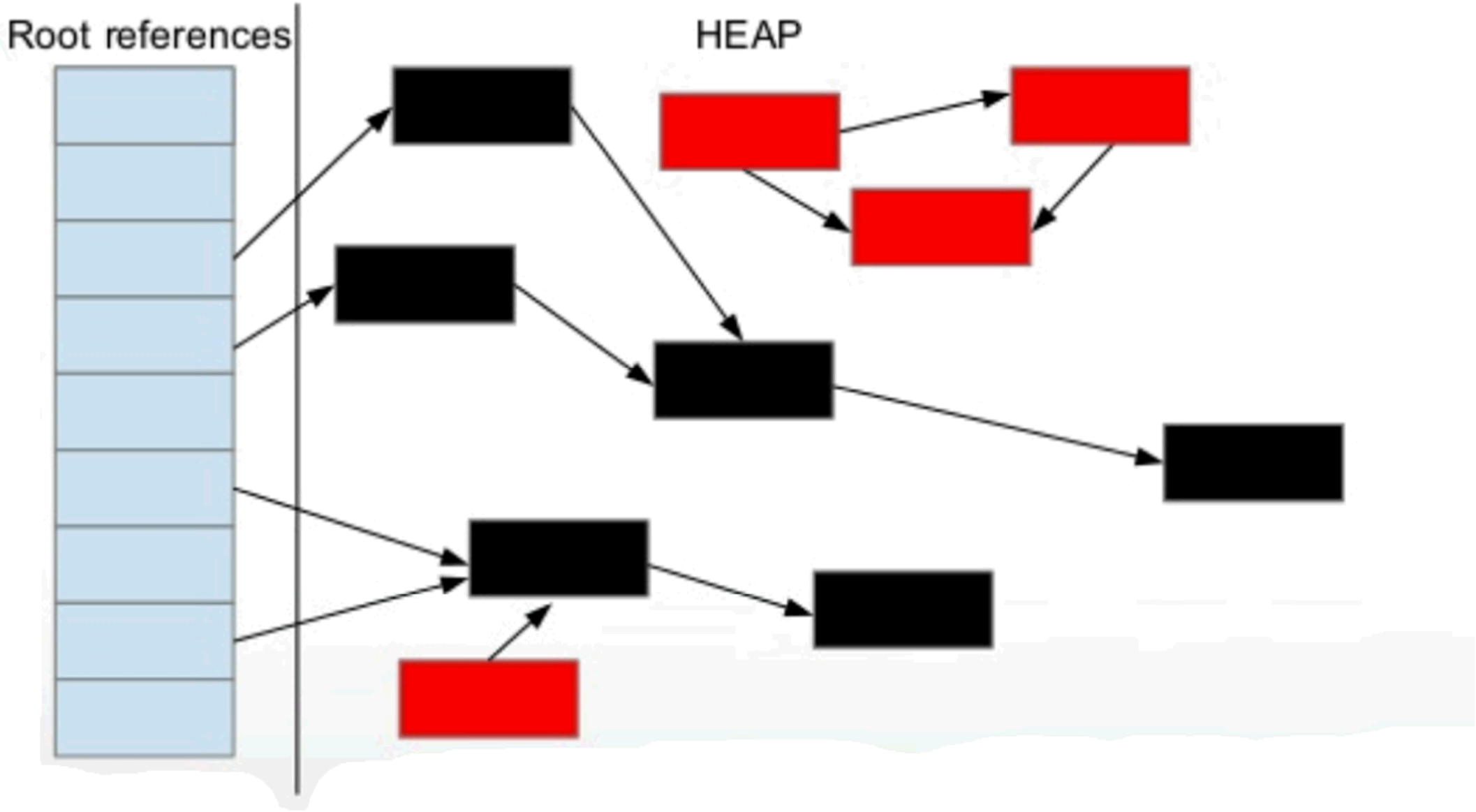
» **reference tracing approach**

- no direct slow down of application code
- **find** references
  - root in frames (stack, variables incl. parameters) using **OopMaps**
    - compiled maps for every possible *safepoint* entry

```
OopMap{rsi=Oop [48]=Oop rdx=Oop [72]=Oop off=1734}
```

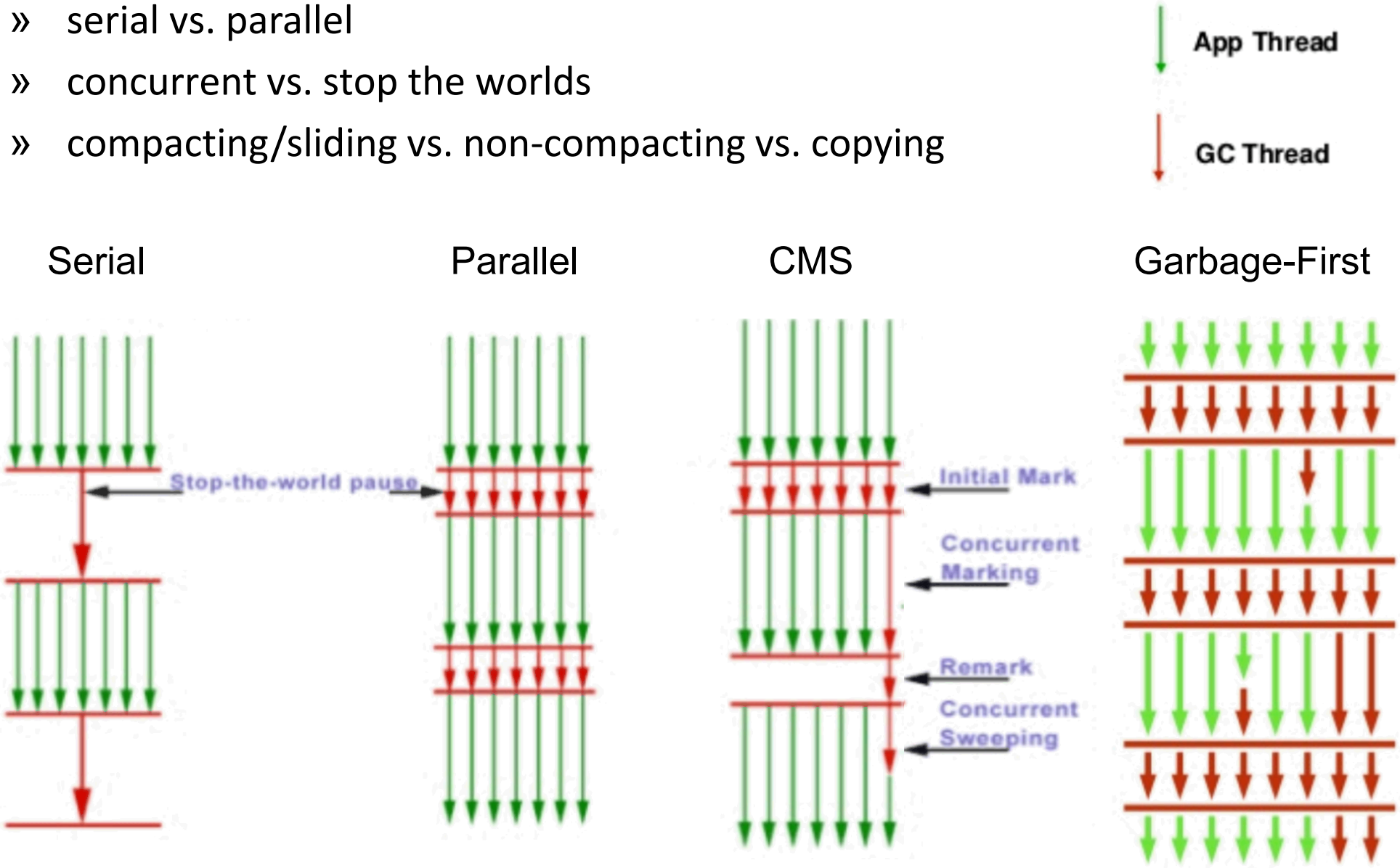
  - in different object using object type
    - reference positions in klass VM structure
- **marking phase** traverse all objects from **roots**
  - depth-first search, breath-first search
  - dominates collection time due to random access to memory
    - cache prefetching to reduce cost
- use mark flags to avoid cycles
  - in object header – standard writes with possible partial re-traversal
  - side bitmaps (1 bit for 64 bits) – improving cache operations, atomics

Identify Reachable Objects – Reference Tracking

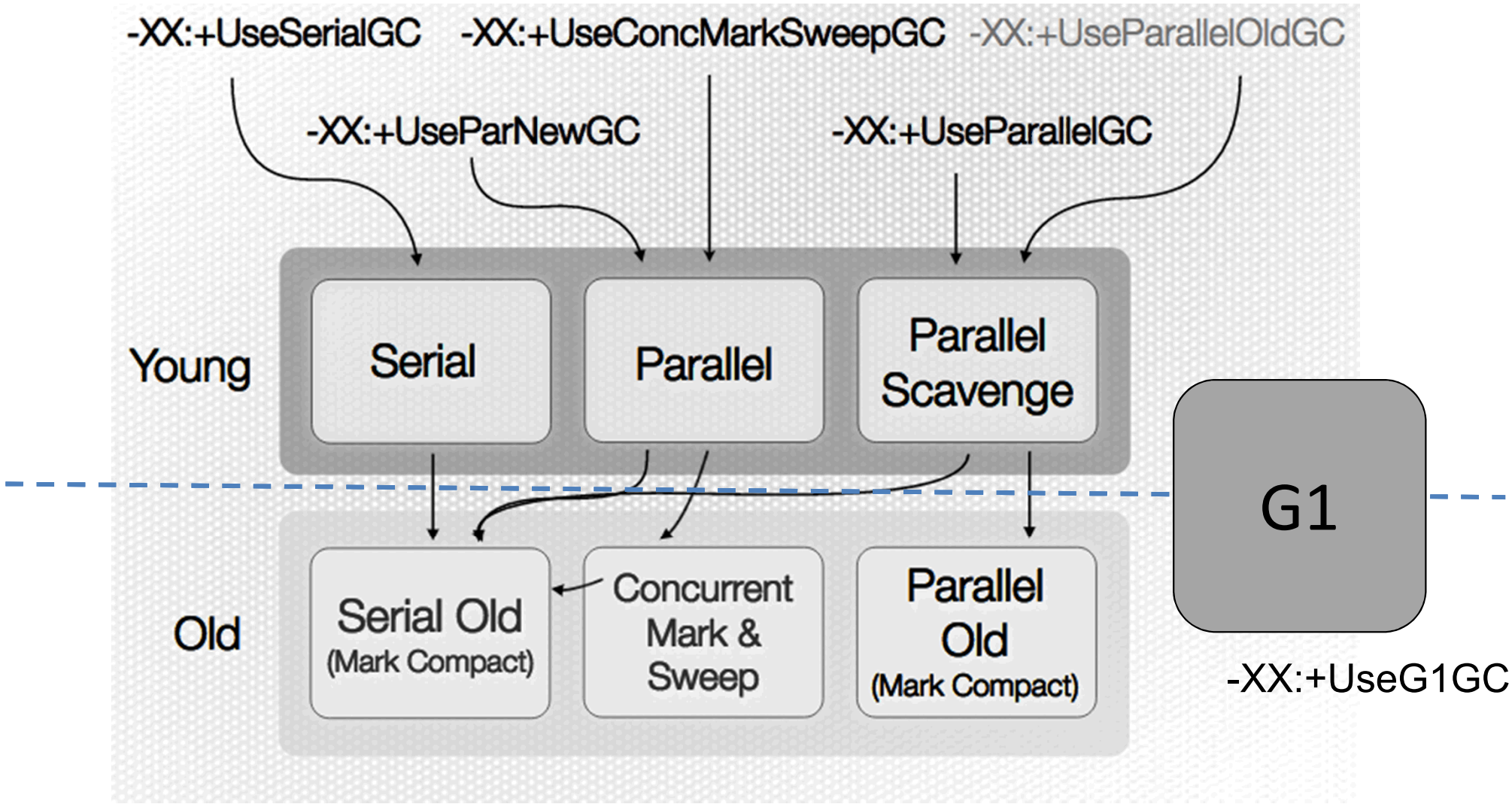


Garbage Collector Design Architecture

» serial vs. parallel
» concurrent vs. stop the worlds
» compacting/sliding vs. non-compacting vs. copying

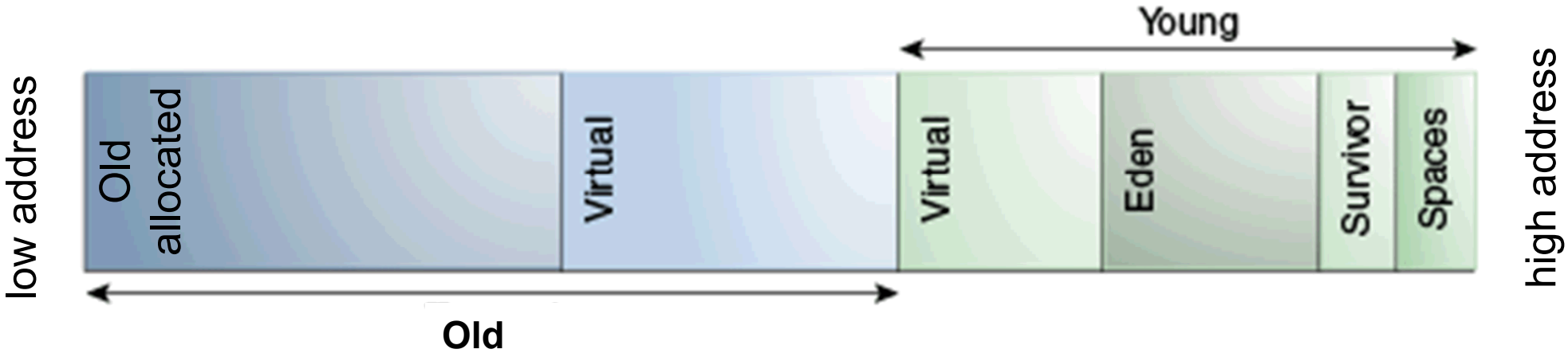


Garbage Collector Design Architecture



Parallel Collector

» **JVM heap layout** supporting adaptive resizing (*virtual* has no physical pages)



» **max heap** size (virtual space allocated) –Xmx

- default ¼ RAM up to 32 GB if there is >=128 GB RAM

» **initial heap** size (really allocated) –Xms

- default 1/64 RAM up to 1 GB if there is >=128 GB RAM

» **young** vs. **old** ratio –XX:NewRatio=<n>

- default 2 – thus old is 2x larger than young

» **survivor** spaces vs. **eden** ratio –XX:SurvivorRatio=<n>

- default 8 – thus eden is 8x larger than one survivor space

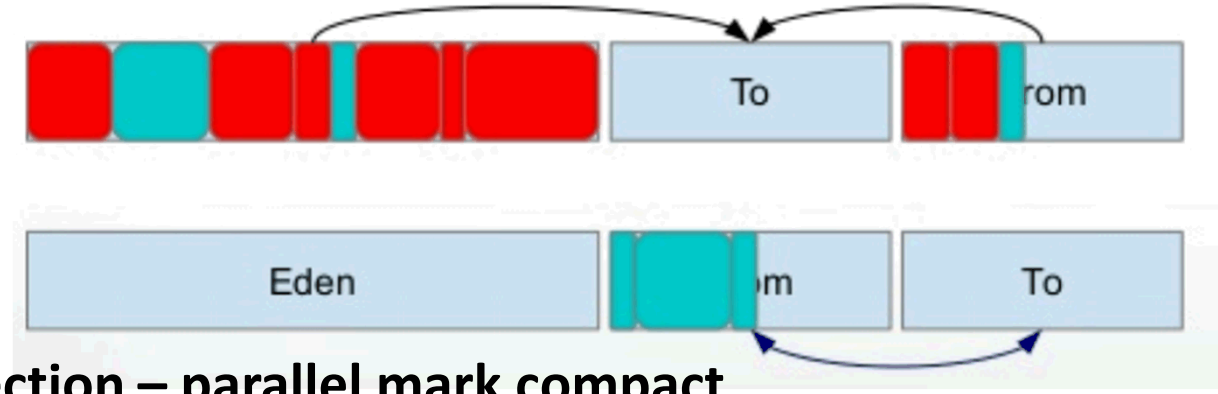
Parallel Collector

» **object allocations**

- in TLAB inside eden - no space in TLAB left, new TLAB allocated
- in eden directly for objects larger than TLAB
- in old directly for objects larger than eden

» **minor collection – parallel scavenge**

- triggered when no space for new TLAB/object in eden
- collection in young generation only, promote to survivor or old
- results into clean eden, **swap** of survivor spaces (one empty)



» **full collection – parallel mark compact**

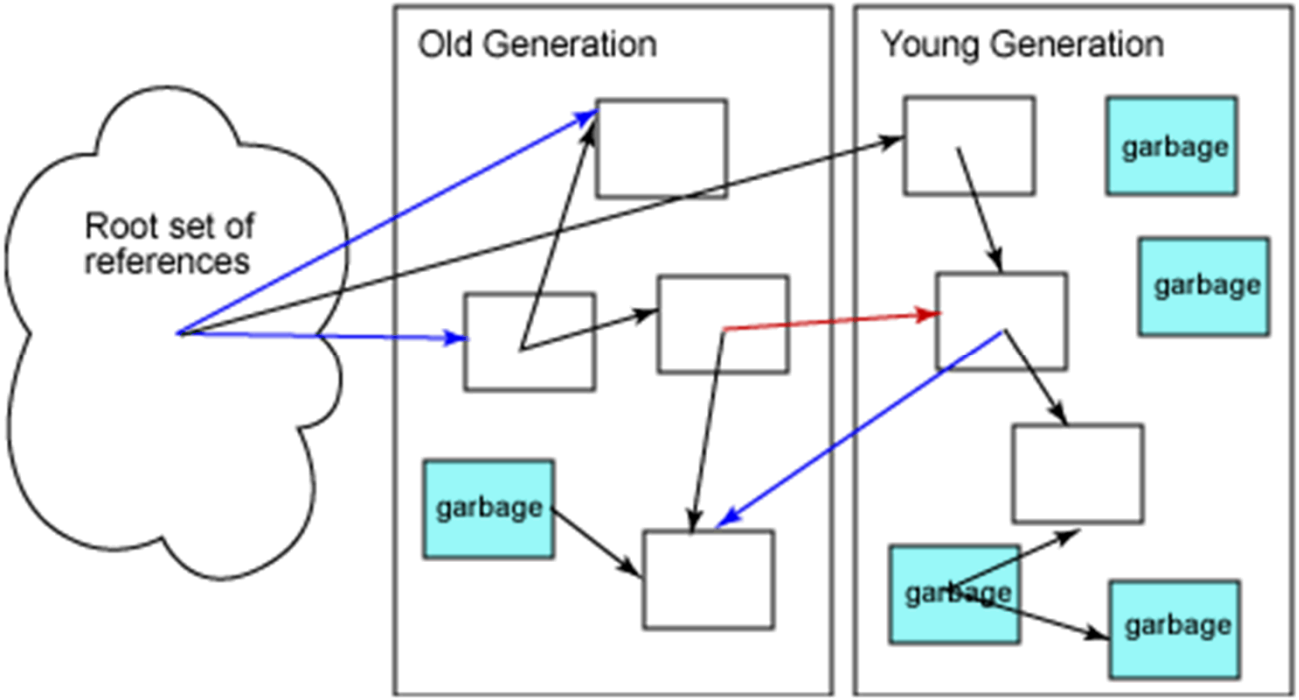
- triggered when there is no space for promotion or new object in old
- collection in young and old generations
- results into completely clean young (eden, both survivor spaces)

Remembered Set

» **track old-to-young references**

» **speed-up** frequent identification of reachable objects **for minor collection**

- marking starts from **roots** and **references old-to-young**
- do not traverse objects out of young generation
  - fast bit operations using generation size 2^n



red – old-to-young, blue – to old (*don't need trace during minor collection*)

Card Table Compressed Remembered Set

» whole heap divided to **512 Bytes chunks** (8 cache lines of 64 Bytes)
  - each chunk has one card table slot

» thread-safe **card table** is Byte based
  - avoid expensive atomic read-update-write for bit operations
  - standard byte writes
    - **dirty** (0) – possibly contain reference to young (has false positive)
    - **clean** – cannot contain reference to young (no false negatives)
  - 100 GB heap => 200 MB card table (<0.2%)
    - one cache line holds cards for 32kB of heap

» *write reference to object* imply **assembly code write barrier**
  - no tracking for null writes or reference writes in newly allocated
  - track **standard object start address** CARD_TABLE[object address >> 9] = 0;
  - track **real element address for native reference arrays**

    CARD_TABLE[array slot address >> 9] = 0;
  - **imprecise but very fast** without any condition
    - cards for young, all reference writes

Card Table Compressed Remembered Set – Write Barriers

write non-null reference in RAX to **standard object** at R11, standard oop, 64-bit:

```
mov     %rax,0x10(%r11)
mov     %r11,%r8
shr     $0x9,%r8
movabs  $0x215153000,%r9
movb    $0x0,(%r9,%r8,1)
```

store reference in RAX to the first field in object

compute card offset from obj. start (R11) directly

card table start address to R9

store **dirty** to card table

write non-null reference in RAX to **array** at R10 index EBP, standard oop, 64-bit:

```
movslq %ebp,%r11
shl    $0x3,%r11
lea    0x18(%r10,%r11,1),%r11
mov    %rax,(%r11)
shr    $0x9,%r11
movabs $0x215153000,%r8
movb   $0x0,(%r8,%r11,1)
```

count address of slot in array to R11

store reference in RAX to array slot

compute card offset from slot address (R11)

card table start address to R9

store **dirty** to card table

<u>Native Object array structure standard OOP, 64-bit:</u>

0x00:	mark word	
	Klass ref.	
0x10:	array length	empty padding
	object reference on index 0	
0x20:	object reference on index 1	

⋮

Card Table Compressed Remembered Set – Write Barriers

» **no optimization** for multi reference writes to the same object (which is fast due to already cached part of card table)

- object can overlap over chunk boundary

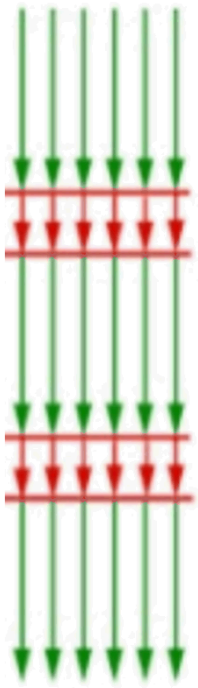
» **false sharing** in contended multi-thread writes (even worse on multi-CPU)

- 64B cache line implies sharing of cards for 32kB (64*512)
- speed-up with **conditional card table updates** (–XX:+UseCondCardMark)

```
if (CARD_TABLE [address >> 9] != 0) CARD_TABLE [address >> 9] = 0;
```

  – for highly contended reference writes up to 7 times faster

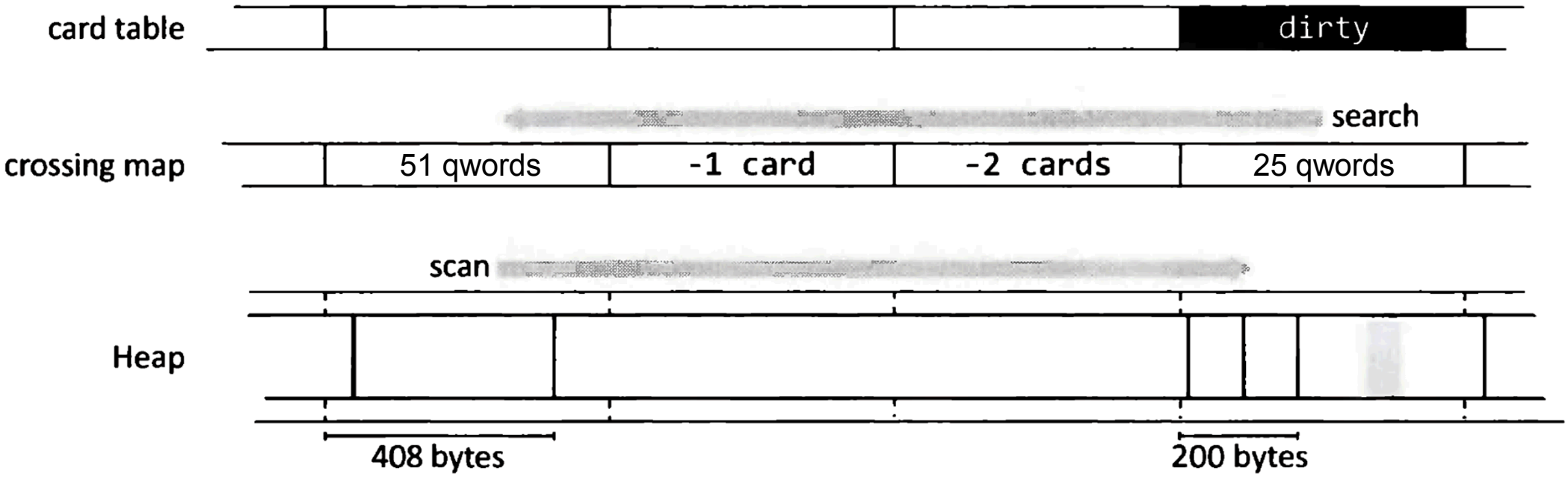
Minor Collector – Parallel Scavenge



- known also as **throughput garbage collector**
- currently default for Oracle JVM 8
- utilize more cores/CPUs (-XX:ParallelGCThreads=<N>)
  - default #HW threads for <= 8
  - 3+5/8 of #HW threads otherwise (e.g. 13 for 16 threads)
- stop-the-world manner
- **copying** with survivor spaces ("from" and "to", swapped)
  - relocate reachable objects in young generation to "to" survivor
    - if no space, relocate them to old (or trigger full collection)
  - eden and from survivor space is empty after minor collection

- **parallel processing** of **task queue** initially filled with
  - add stripes of cards for scanning for old-to-young references (only allocated)
  - add JNI handles and VM internals
  - add frames from stacks
  - add static references

Minor Collector – Scan Old for References to Young



» **crossing map** - Byte per 512 Bytes chunk like card table, for **old only**

- updated during allocation/promotion of object and full collection
- **speed-up search for object start**

  N>0 object start offset in align positions of the last object in the card

  N<0 object start offset start –N cards back or the there is the next –N

» **clean cards** before **DFS** queuing of **processing of addresses of old-to-young refs**

- already **forwarded objects** are updated immediately without queuing
- -XX:PrefetchScanIntervalInBytes=576 (9 cache lines)

Minor Collector – Process Address of –to-Young Reference

» target is already **marked/forwarded** – mark word (forwarding address | 0b11)
  - **update reference** to forwarding address

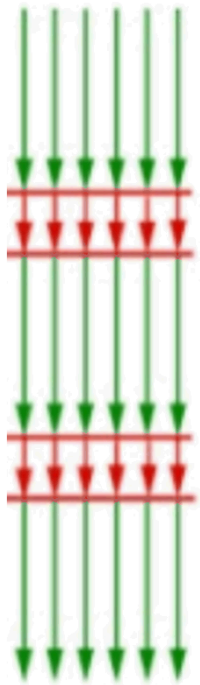
» target **not marked** yet
  - current **age** < tenuring threshold
    - copy object to "to" survivor using 32k **PLAB** (-XX:YoungPLABSize=4096)
  - older or no space in young
    - copy object to old using 8k **PLAB** (-XX:OldPLABSize=1024)
  - **mark** previous object with **forwarding address** using CAS
    - failed – de-allocate back, read other thread forwarding address
    - success
      - for forwarding in young update **age** of new object
      - DFS queuing of processing of object's addresses of **old-to-young refs**
  - **update reference** to forwarding address

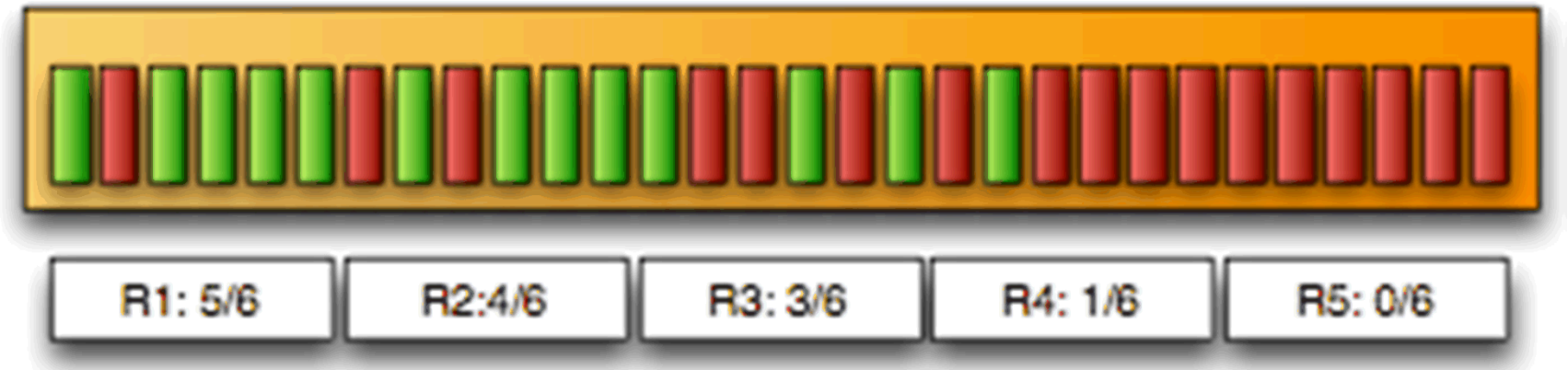
Note: all reference changes update card table if in "to" survivor

all PLAB or object re-allocations are **NUMA** aligned to speed-up collection

Full Collector – Parallel Mark Compact

- » default for Oracle JVM 8
- » stop-the-world manner
- » multiple threads as parallel scavenge
- » old generation logically divided into fixed-size regions
- » use **sliding compaction** - clean eden and both survivors as well
  - doesn't need additional memory, but is slower than copying
- » **parallel mark** phase
  - initiated with all roots (not using card table)
  - track all references not just those targeting to young
  - info about reachable objects (location & size) are propagated to corresponding region data

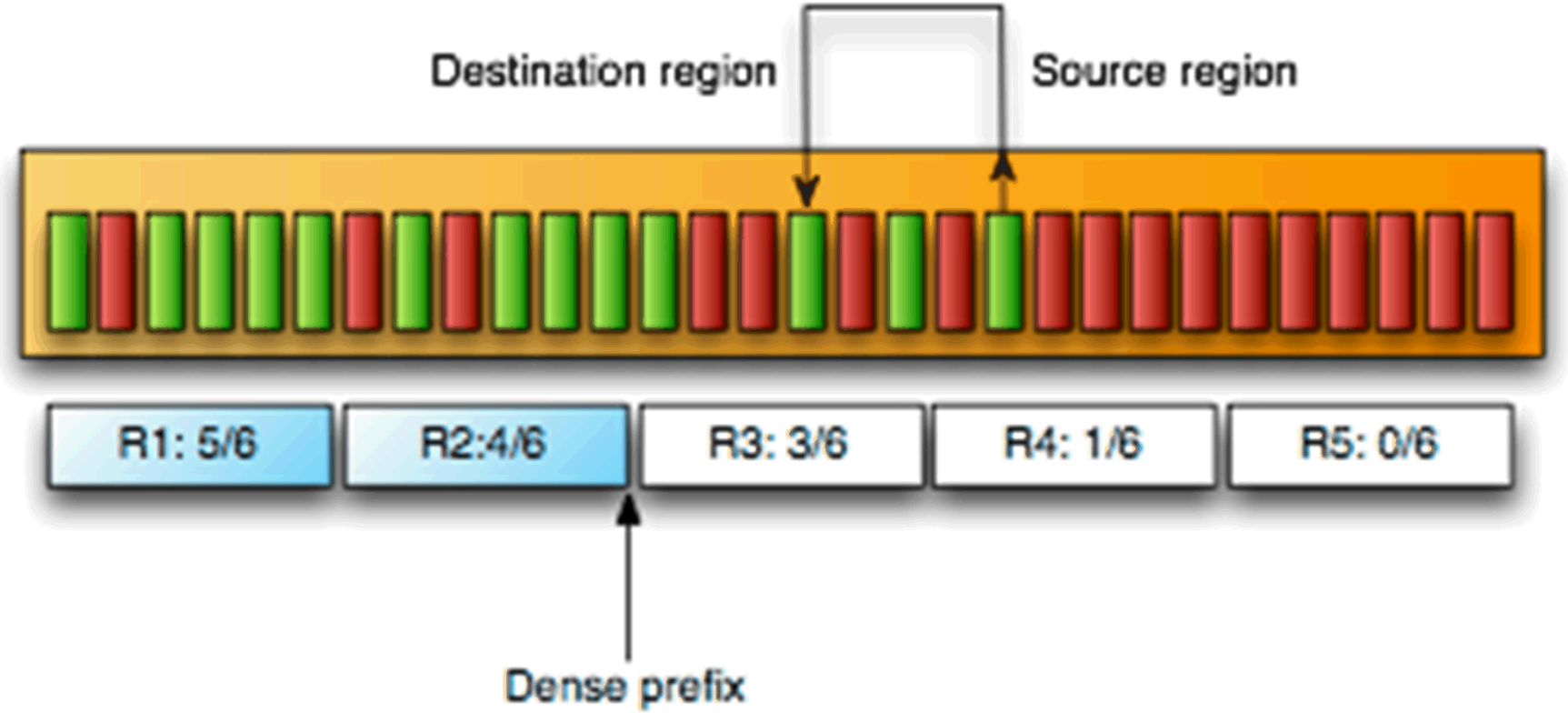




Full Collector – Parallel Mark Compact

» **serial summary** phase

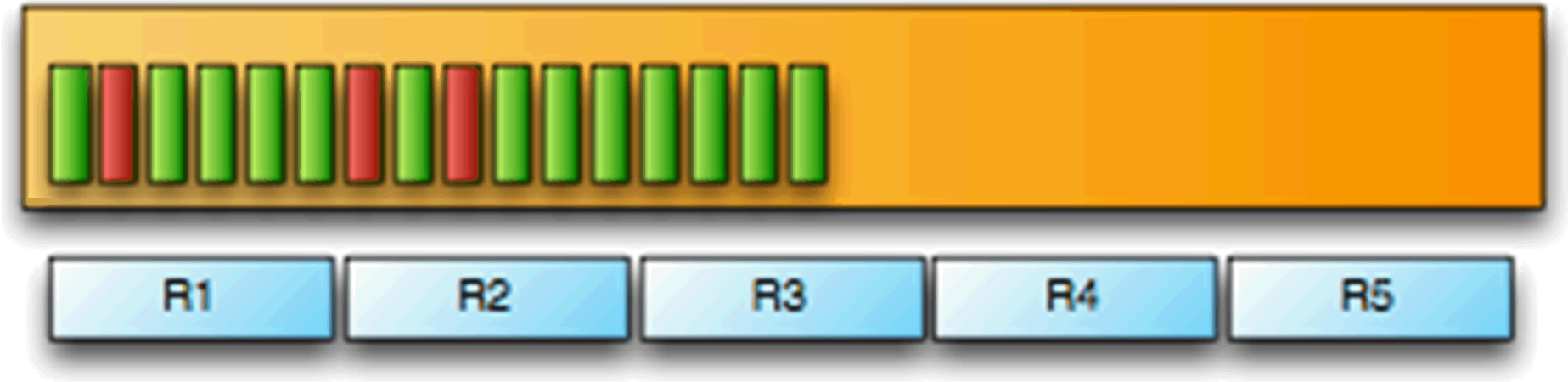
- identify density of regions (due to previous compactions, older objects should be on the left, younger to right side)
- find from which region (starting from the left side) it has sense to do compaction regarding recovered from a region
  - *dense prefix* – left regions which are not collected
- calculate new location of each live data for each regions; **most right regions will fill most left ones**; pretend data locality keeping their order



Full Collector – Parallel Mark Compact

» **parallel compaction/sweeping** phase

- divide regions with some targets (start of objects)
- each thread first compact the region itself and fill it by designated right regions
  - all references are updated based on summarized data (read only)
  - crossing map is updated to track the last object start in chunk
- *no synchronization* needed, only one thread operate per each region
- update root references and clean empty in parallel
- finally heap is packed and large empty block is at the right end



Full Collector – Parallel Mark Compact

» support **strong generational hypothesis** – younger object dies earlier then older

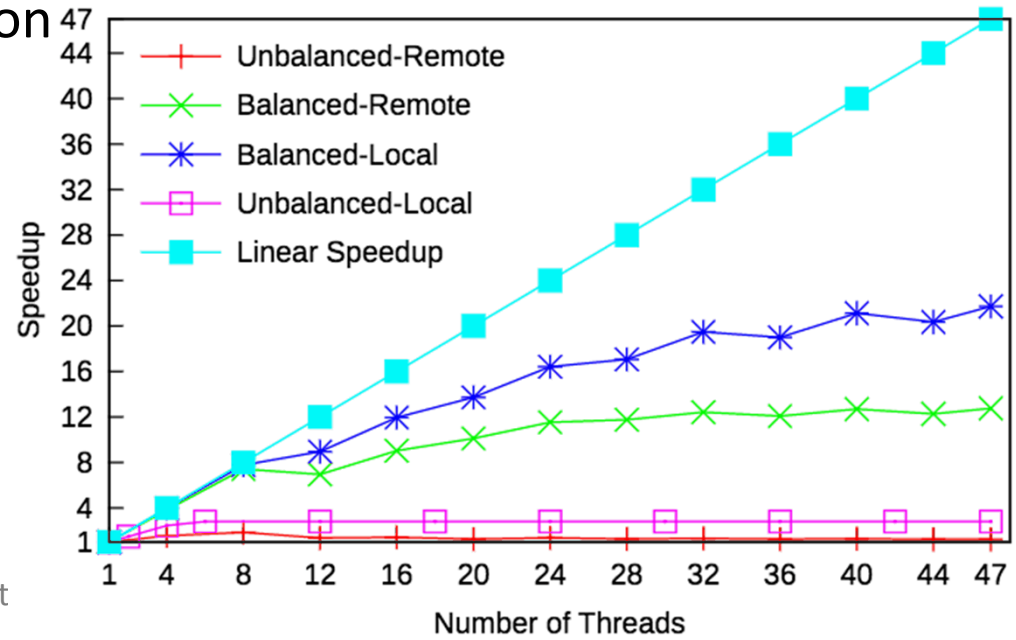
- the objects with highest probability to survive are located on the left side (because of previous GC runs)
- **dense prefix** completely *avoid their costly copying*
- 50% of full collection work reclaim 82% of garbage
- reclaim of additional 18% of garbage cost as much as previous work

» dense prefix is adaptively updated

- considering *used* to *total heap* ratio
- affects pause time of full collection

» after full collection

- whole young is empty
- card table is cleaned (there are no references to young)



Parallel Collector - Ergonomics

» **adaptive mechanism resizing** generations (-XX:+UseAdaptiveSizePolicy)

- **max pause time** goal (-XX:MaxGCPauseMillis=<undef>)
  - if not met - shrink generation size where the pause time is longest and at least above the goal
- **throughput** goal (-XX:GCTimeRatio=99) – applied when previous is met
  - if not met – increase both generations
    - young increased according to its time portion in total time
- **minimum footprint** goal – applied if all previous are met
  - shrink heap size

-XX:YoungGenerationSizeIncrement=20 ; -XX:TenuredGenerationSizeIncrement=20

-XX:AdaptiveSizeDecrementScaleFactor=4 (default 5%)

-XX:YoungGenerationSizeSupplement=80 (similar for tenured)

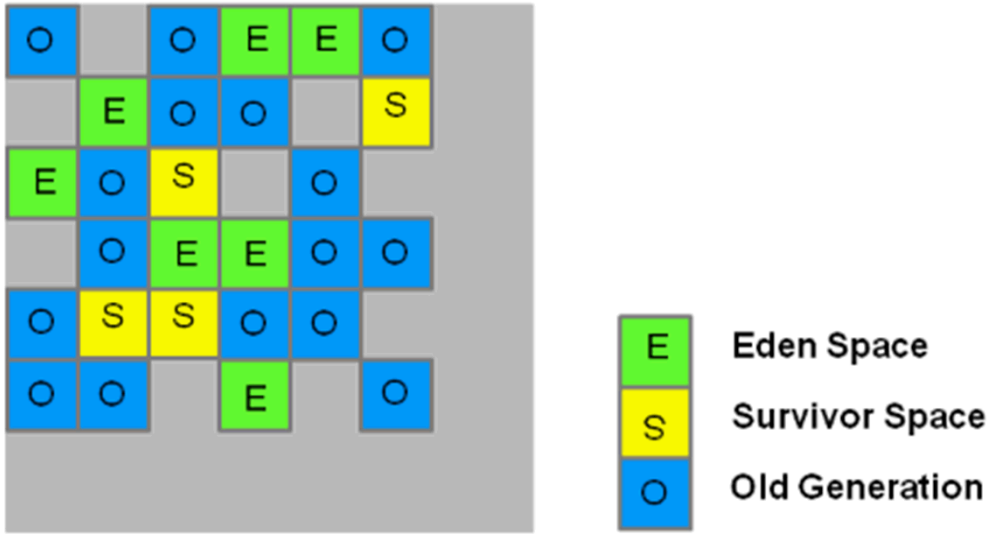
-XX:YoungGenerationSizeSupplementDecay=8 (8 times added)

-XX:TenuredGenerationSizeSupplementDecay=2 (2 times added)

Garbage First (G1) Collector

- **dynamic generational collector** called G1GC (-XX:+UseG1GC)
- **concurrent** collector for large heaps (replacement for older CMS)
- whole heap divided into **regions** (by def. to be close 2048 regions 1-32MB)
- no explicit separation between generations, only regions are mapped to generational spaces (generation is set of regions, changing in time)

- set of regions defines
  - young generation
  - old generation

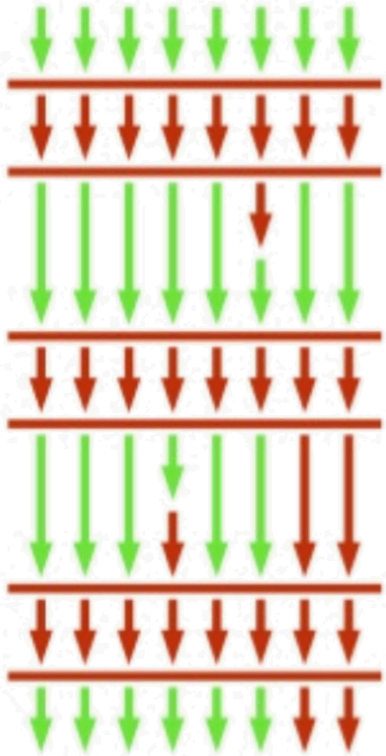


- compacting -> enables bump-the-pointer, TLABs, uses CAS
- **copying** = copy live from a region to an empty region
- keep **Humongous regions** (sequence) for objects >=50% regions size
- maintain list of free regions for constant time

Garbage First (G1) Collector

» **activities** in garbage first collector – **minor**, **mixed**, **full** collections

- parallel with **global safe point** (stop the world)
  - initial marking pause
  - final marking pause with data counting
    - prepare candidate regions for mixed
  - copying
- **concurrent** with multiple threads
  - remember set refinement
  - marking
  - clean-up ?



» **major speed-up** is that **fast copying** collection applied incrementally to old

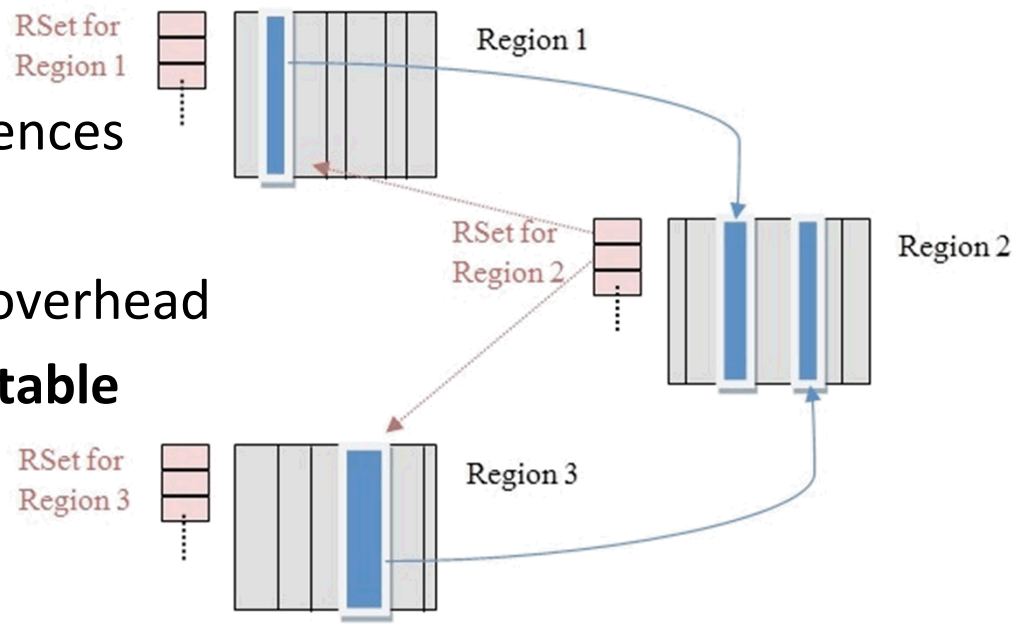
- requires more heap than parallel due to concurrent activities

» poor handling of larger objects (humongous objects)

» not NUMA aware

» default since JVM 9

Garbage First (G1) Collector – Remember Set



» **track references into a region**
  - **ignore** null and inter-region references
  - old-to-young and old-to-old

» additional structures with ~5% heap overhead

» use **per-region-table** (PRT) with **card table** updated *asynchronously* using *update thread log buffers*
  - processed by refinement threads -XX:G1ConcRefinementThreads=<n> (max threads)
  - filled by compiled write barrier (pseudo code shown for simplification)

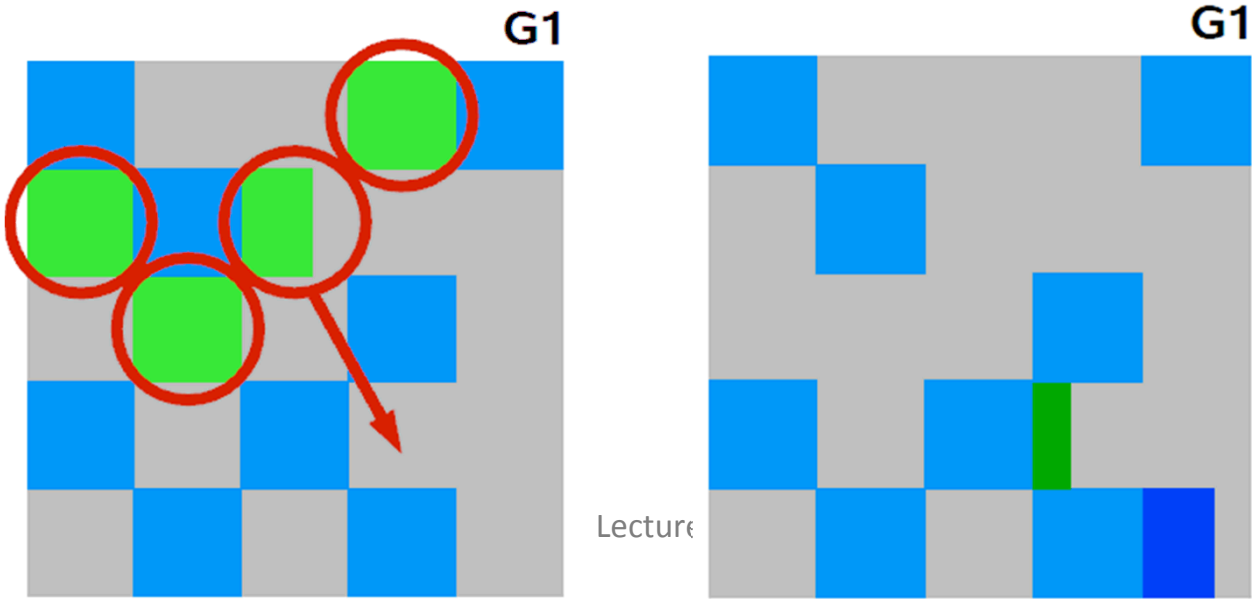
```
oop oldFooVal = this.foo;
if (GC.isMarking != 0 && oldFooVal != null){
  g1_wb_pre(oldFooVal);
}
this.foo = bar;
if ((this ^ bar) >> 20) != 0 && bar != null) {
  g1_wb_post(this);
}
```

log2 of region size (1MB)

-XX:+G1SummarizeRSetStats -XX:G1SummarizeRSetStatsPeriod=1

Garbage First (G1) Collector – Minor and Mixed Collection

- stop-the-world approach with **parallel threads**
- **triggered** when no more allocation in Young regions possible
- **collection set** (CSet)
  - eden and from survivor regions for pure **minor collection**
  - eden, from survivor and *candidate old* regions for **mixed collection**
- reachable objects identified from roots + Rset for the regions + card table
- reachable objects are copied (from eden and survivor regions) into one or more new survivor regions
  - using forwarding address with marking similar to parallel scavenge
- if aging threshold is met => promoted into old regions (optionally new)



Garbage First (G1) Collector – Marking Phase

» **triggered by** heap occupancy percent (-XX:InitiatingHeapOccupancyPercent=45)

» outcomes

- **candidate old regions** with a lot of garbage for mixed collection
- cleanup completely empty old regions

» **initial mark** – done right after minor collection *utilizing global safe point (GSP)*

- snapshot-at-the-beginning (SATB) – scan roots

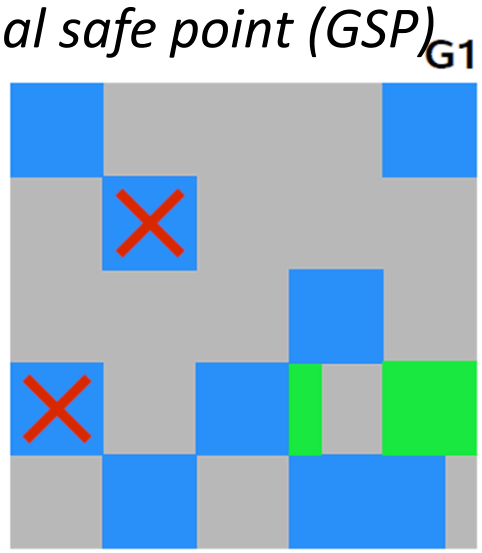
» **marking** and **region-based statistics collection** – *concurrent*

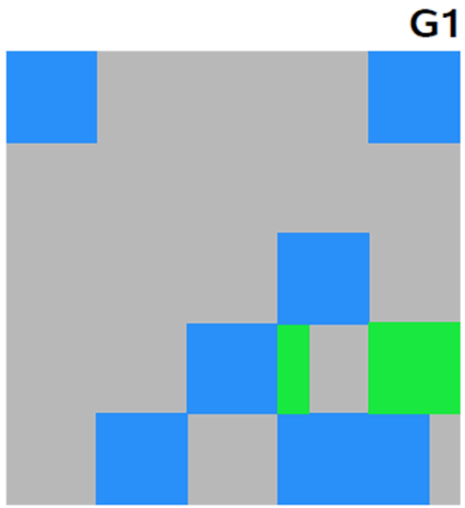
(-XX:ConcGCThreads=<n>)

- can be interrupted by minor GC
- pre-write barrier keeps previous reference in SATB

» **final mark -** right after the next minor collection *in GSP*

- reflect changes in previous minor collections and allocations utilizing modifications in card tables
- summarize and prepare ordered candidates





Garbage First (G1) Collector – Full Collection

» multiphase **full tracking with compact of all regions** during global safe point

» triggered by

- **concurrent mode failure** – old fill-up before concurrent complete
  - increase heap, decrease trigger threshold, more concurrent threads
- **promotion failure** – mixed collection but run-of space in old
  - trigger sooner
- **evacuation failure** – minor collection has no more space for promotion
  - increase heap
- **humongous allocation failure** – no space for large objects
  - avoid large objects (>50% of region size)
  - increase region size (alternatively increase heap)

Garbage First (G1) Collector – Humongous Objects

- objects larger than ½ of the region are considered as **humongous**
  - with 1MB region it is just 500kB -> there can be a lot of such objects
- **allocation**
  - check concurrent trigger and optionally start concurrent marking
  - one set of humongous regions contain just one such object
    - **waste** up to region size – 1 + allocated **out of Young** generation
  - not having sequence of free regions for allocation of a object trigger **expensive full collection**
- **reclamation** of non-reachable during (compacted during full collection only)
  - cleanup phase of concurrent cycle
  - full collection
- **debug** humongous allocations
  - -XX:+UnlockExperimentalVMOptions –XX:G1LogLevel=finest –XX:+PrintAdaptiveSizePolicy
  - use **Java Flight Recorder** in Java Mission Control
    - all allocations tracked in runtime routines like TLAB allocations

Garbage First (G1) Collection – Tuning Options ☺

G1ConcMarkForceOverflow
G1ConcMarkStepDurationMillis
G1ConcRSHotCardLimit
G1ConcRSLogCacheSize
G1ConcRefinementGreenZone
G1ConcRefinementRedZone
G1ConcRefinementServiceIntervalMillis
G1ConcRefinementThreads
G1ConcRefinementThresholdStep
G1ConcRefinementYellowZone
G1ConcRegionFreeingVerbose
G1ConfidencePercent
G1DummyRegionsPerGC
G1EvacuationFailureALot
G1EvacuationFailureALotCount
G1EvacuationFailureALotDuringConcMark
G1EvacuationFailureALotDuringInitialMark
G1EvacuationFailureALotDuringMixedGC
G1EvacuationFailureALotDuringYoungGC
G1EvacuationFailureALotInterval
G1ExitOnExpansionFailure
G1FailOnFPError
G1HRRSFlushLogBuffersOnVerify
G1HRRSUseSparseTable
G1HeapRegionSize
G1HeapWastePercent
G1MarkingOverheadPercent
G1MarkingVerboseLevel
G1MaxVerifyFailures
G1MixedGCCountTarget
G1PrintHeapRegions
G1PrintRegionLivenessInfo
G1RSBarrierRegionFilter
G1RSScrubVerbose
G1RSetRegionEntries
G1RSetRegionEntriesBase
G1RSetScanBlockSize
G1RSetSparseRegionEntries
G1RSetSparseRegionEntriesBase
G1RSetUpdatingPauseTimePercent
G1RecordHRRSEvents
G1RecordHRRSOops
G1RefProcDrainInterval
G1ReservePercent
G1SATBBufferEnqueueingThresholdPercent
G1SATBBufferSize
G1SATBProcessCompletedThreshold
G1ScrubRemSets
G1SecondaryFreeListAppendLength
G1StressConcRegionFreeing
G1StressConcRegionFreeingDelayMillis
G1SummarizeConcMark
G1SummarizeRSetStats
G1SummarizeRSetStatsPeriod
G1TraceConcRefinement
G1TraceHeapRegionRememberedSet
G1TraceMarkStackOverflow
G1UpdateBufferSize
G1UseAdaptiveConcRefinement
G1VerifyBitmaps
G1VerifyCTCleanup
G1VerifyHeapRegionCodeRoots
G1VerifyRSetsDuringFullGC
G1YoungSurvRateNumRegionsSummary
G1YoungSurvRateVerbose
PrintCFG1

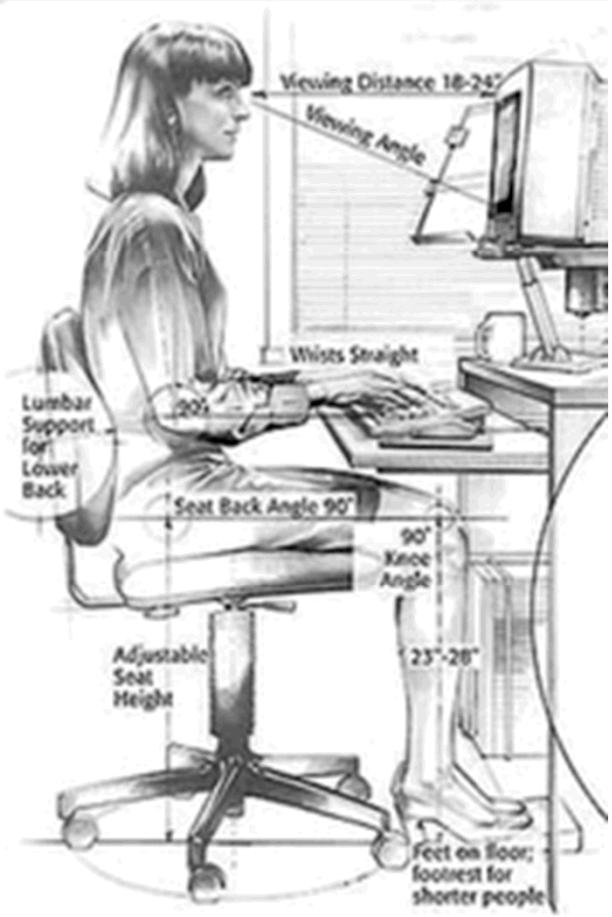
Conclusion

Google

When is the best time to do a GC?

When nobody is looking.

Using camera to track eye movement
When subject looks away do a GC.



B4M36ESW: Efficient software

Lecture 13: Virtualization

Michal Sojka
michal.sojka@cvut.cz

May 20, 2019

Outline

1. Virtualization basics
2. Hardware assisted virtualization
   - Nested paging
3. Example: Mini VMM with KVM
4. I/O virtualization
   - How do modern Network Interface Cards (NIC) work
   - Device emulation
   - Virtio
   - PCI pass-through
   - Single-Root I/O Virtualization
   - Inter-VM networking
5. Summary

Outline

1 Virtualization basics

2 Hardware assisted virtualization
- Nested paging

3 Example: Mini VMM with KVM

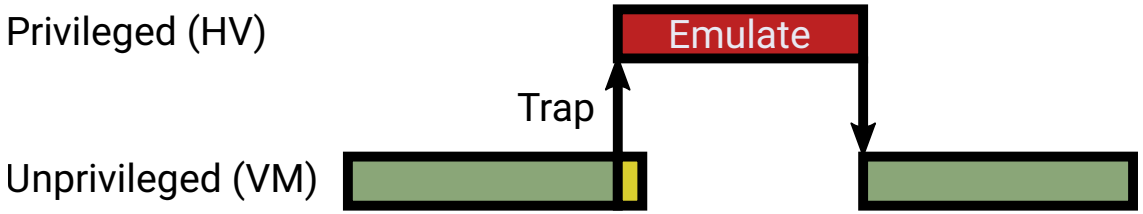
4 I/O virtualization
- How do modern Network Interface Cards (NIC) work
- Device emulation
- Virtio
- PCI pass-through
- Single-Root I/O Virtualization
- Inter-VM networking

5 Summary

Virtualization

- **Definition:** Virtualization of the whole computing platform – the operating system thinks it runs on real hardware, but the hardware is largely emulated by the hypervisor (HV) and/or virtual machine monitor (VMM).
- Virtual machine (VM) vs. Java VM
  - Java VM interprets Java byte code and interacts with an operating system
  - VM executes native (machine) code and interacts with a hypervisor.
- VMs used since '70s, mostly on IBM mainframes
  - Popek and Goldberg defined requirements for instruction set architecture (ISA) virtualization in their paper from 1974,
  - x86 became fully virtualizable in 2005.
- More detailed introduction to virtualization (from OSY course): `https://cw.fel.cvut.cz/b181/_media/courses/b4b35osy/lekce12_virt.pdf`

Trap-and-emulate

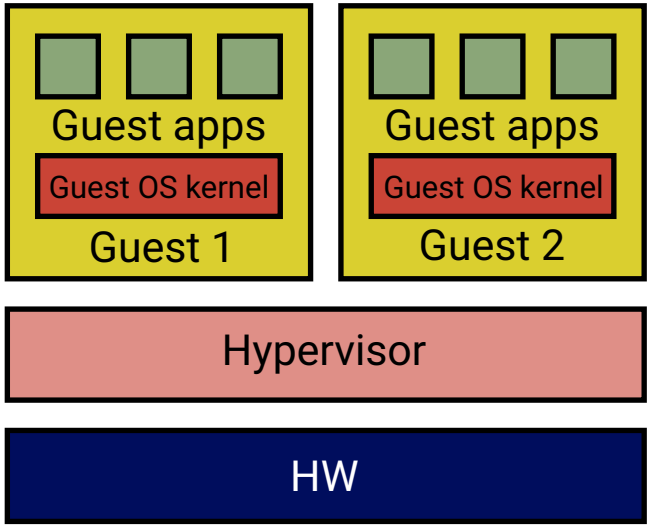


- Basic mechanism of virtualization
- Popek and Goldberg: "*All sensitive instructions must be privileged instructions*"
  - **Sensitive instruction**: Changes *global state*[1] or behaves differently depending on *global state* (e.g. cli, pushf on x86)
  - **Privileged instruction**: Unprivileged execution traps to the privileged mode (to the hypervisor via CPU exception)
  - on x86 `popf`, `pushf` and few other instructions were not privileged!
    - pushf stores all flags to stack including "global" interrupt flag (IF)
    - popf sets IF in privileged mode and ignores it in unprivileged mode (does not trap)
- Hypervisor (HV) can emulate the effect of sensitive instructions depending on the VM state (not the global state).

[1] Global state means a state that is common to all running VMs, not local to a single VM. For example, CPU reset signal is global

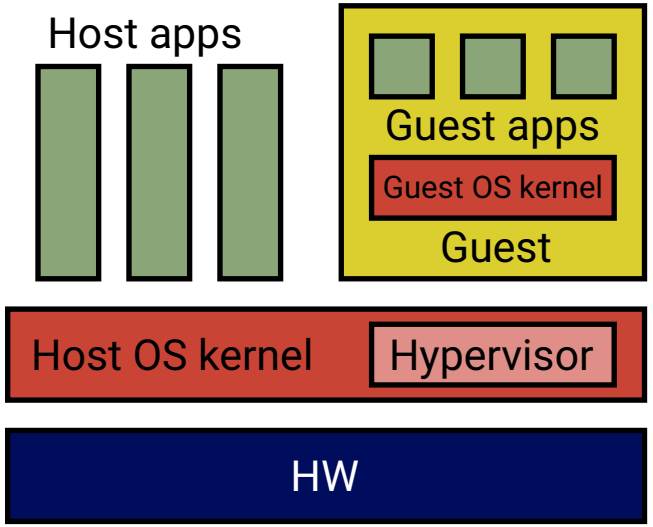
Hypervisor

- Privileged code that supervises execution of the VM, i.e. handles traps.
- Hypervisor types:



Bare-metal hypervisor

- Examples: Xen, VMware ESX, ...
- Idea: Avoid overheads of the general purpose OS.

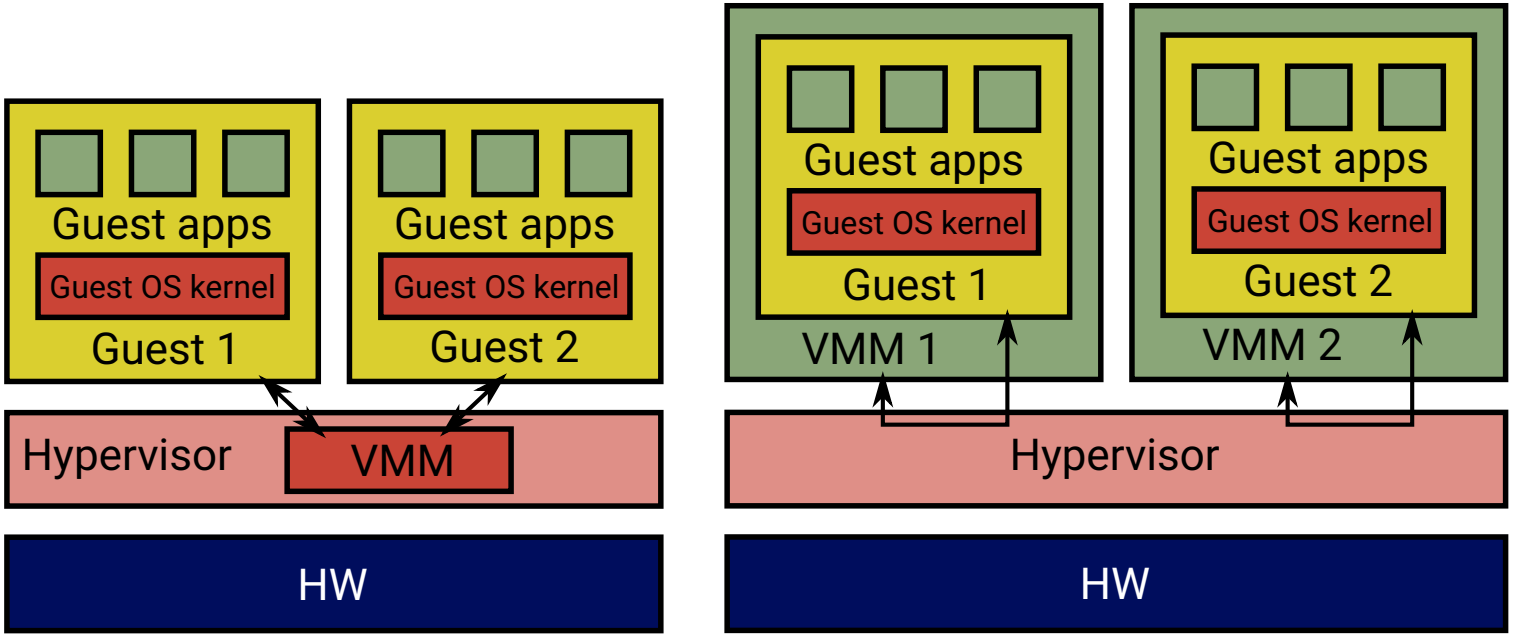


Hosted hypervisor

- Examples: KVM, VirtualBox, ...
- Idea: Why to reinvent the wheel?

- The boundary is blurry – many bare-metal hypervisors support native apps

Virtual Machine Monitor (VMM)



- Software that emulates the HW platform (network, graphics, storage, ...)
- Often implemented inside hypervisor (left) ⇒ people confuse VMM with hypervisors
- Today's platforms are complex (e.g. PC bears 40 years heritage, millions lines of emulation code)
- It is more secure to execute the VMM in user mode, outside of privileged mode (right, example: KVM & qemu)
- It is also slower, but see NOVA microhypervisor (TU Dresden), which implements this faster.

Questions

- How many privilege levels we need to implement virtualization?
  - Two are sufficient, but then, every guest system call, page fault etc. traps from the guest app to the hypervisor, which then arranges switch to the guest kernel – this is slow.
  - Hardware assisted virtualization – introduces more privilege levels (and other features) to make virtualization faster – see later.
- Why is virtualization needed at all? (My personal rant)
  - To some extent because the design of mainstream operating systems is not up to the current needs.
  - Current OSes do not offer sufficient isolation of applications and groups of applications. Many things such as user permissions, apply implicitly to the whole system.
  - Microkernel OSes, which solve this problem, were designed in the past without much success.
  - Now, people are adding "containers" to mainstream OSes, which is painful and often with security problems.
    - Making a microkernel from a monolithic kernel[2] is more difficult that starting with the microkernel from scratch.

[2] `https://marc.info/?l=linux-mm&m=154952428103397`

Outline

1 Virtualization basics

2 Hardware assisted virtualization
- Nested paging

3 Example: Mini VMM with KVM

4 I/O virtualization
- How do modern Network Interface Cards (NIC) work
- Device emulation
- Virtio
- PCI pass-through
- Single-Root I/O Virtualization
- Inter-VM networking

5 Summary

Hardware assisted virtualization

- Accelerates virtualized execution
- Differences between vendors (Intel, AMD, ARM, ...), core principles similar:
  - More privilege levels (x86 – root/non-root, ARMv8 EL0–3)
    - HW emulates things faster than SW
  - Nested paging
  - IO virtualization

Intel VMX

- VMX root operation
  - host rings 0–3
- VMX non-root operation
  - guest rings 0–3
  - in non-root mode, x86 ISA is fully virtualizable
- Transition root→non-root = **VM Enter**
  - instructions: vmlaunch, vmresume
- Transition non-root→root = **VM Exit**
  - instructions: vmresume, vmcall
  - faults (e.g. I/O)

VMCS (up to 4 KiB – e.g. 1024 B)

VM-execution control fields
Host state
Guest state
VM-exit information fields
VM-entry control fields
VM-exit control fields

- VM Control Structure (VMCS)
  - Data structure in memory that controls VMX execution (managed by the hypervisor/VMM)
  - (Re)stores host/guest state
  - “Large structure” ⇒ VM Enter/Exit has overhead
  - The overhead depends on what is (re)stored from/to VMCS (configurable)

Outline

1 Virtualization basics

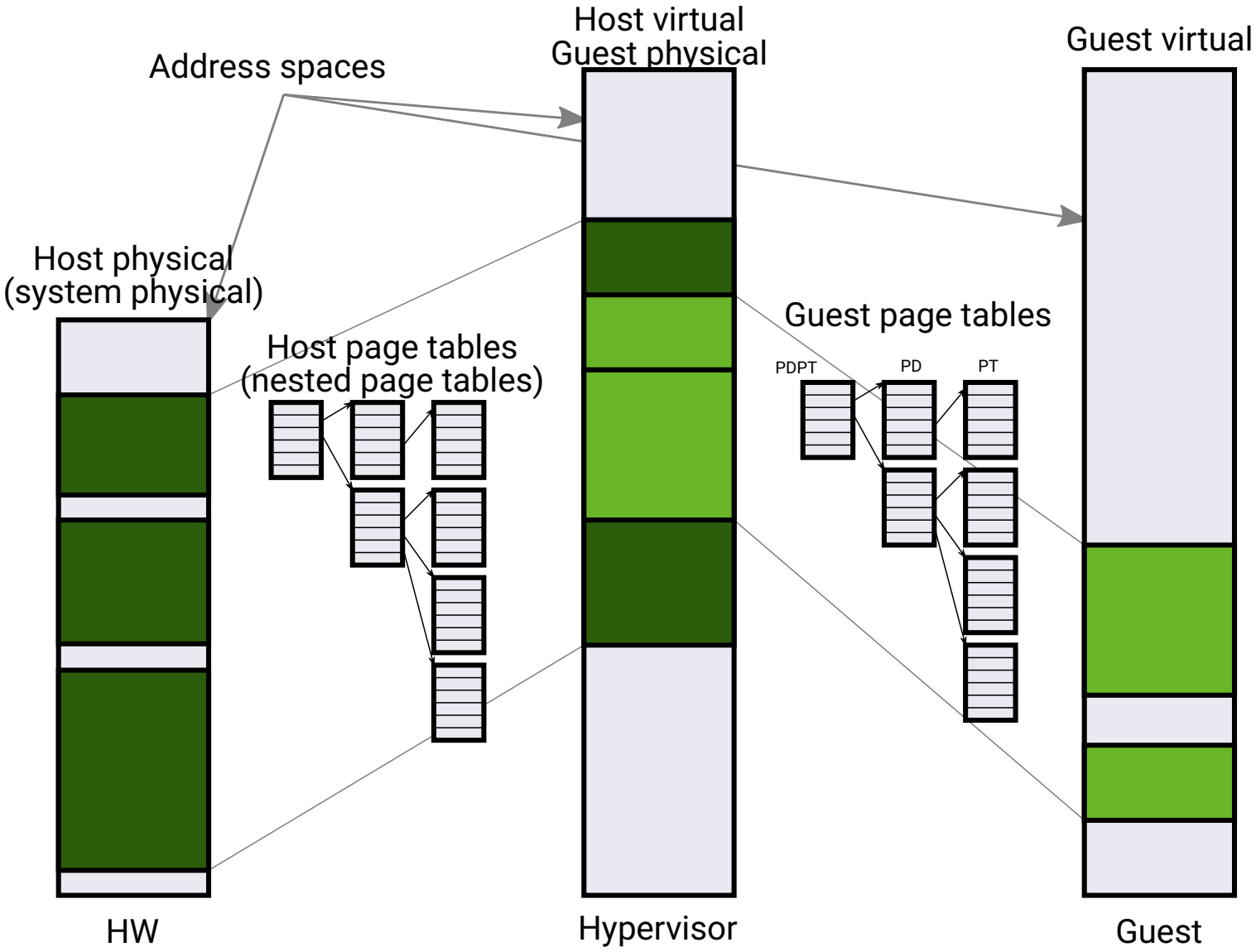
2 Hardware assisted virtualization
- Nested paging

3 Example: Mini VMM with KVM

4 I/O virtualization
- How do modern Network Interface Cards (NIC) work
- Device emulation
- Virtio
- PCI pass-through
- Single-Root I/O Virtualization
- Inter-VM networking

5 Summary

Nested paging



Memory access overhead

- TLB misses and page faults are more expensive inside VM!
- Page walk in a VM (4-level page tables):

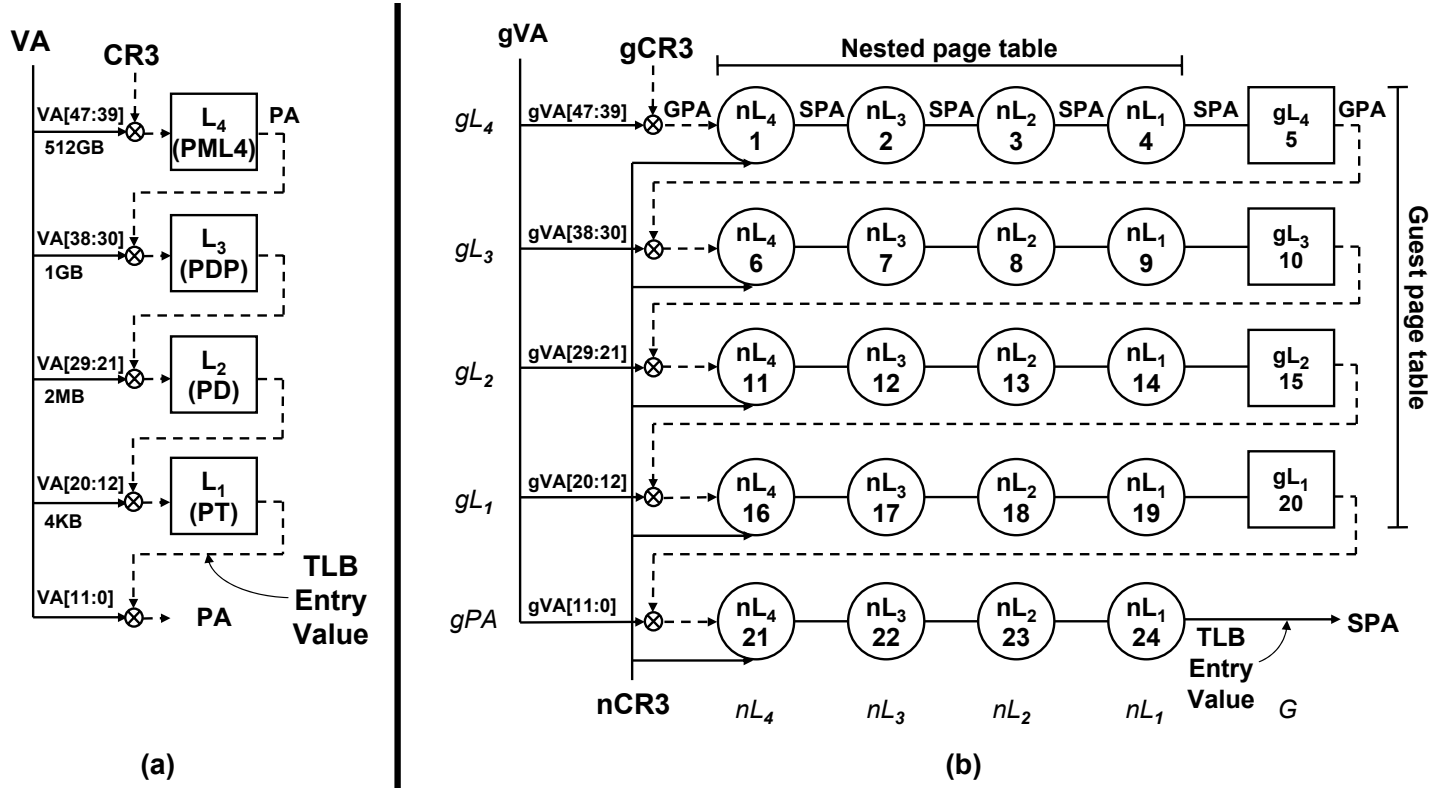


Figure 1. (a) Standard x86 page walk. (b) Two-dimensional page walk. Italics indicate column and row names; notations such as {nL_1,gPA} and {G,gL_1} indicate entries in the indicated columns and rows.

Source: Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating two-dimensional page walks for virtualized systems. SIGOPS Oper. Syst. Rev. 42, 2 (March 2008), 26-35.
DOI: `https://doi.org/10.1145/1353535.1346286`

Page walk in a VM

4-level page tables (64-bit systems)

- 2D page walk – translation of guest virtual address to system physical address:
  - Access to each level of the guest page table has to be translated to system physical addresses via nested page table.
  - TLB miss in virtualized systems results in 24 memory accesses (non-virtualized systems need only 4 memory accesses).
  - If we are lucky, some of these memory accesses are served from a cache rather than from slow main memory.
  - Performance drop up to 15% (Intel), 38% (AMD)[3]
- Tagged TLBs (HW feature of modern CPUs)
  - No need to flush TLBs on process (or VM) switches (good)
  - Applications share TLBs with the hypervisor and VMM (bad)
- Recommendation: Use huge pages (2 MB) if possible
  - Bigger pages ⇒ less TLBs for the same amount of memory
  - Note: Older implementation of huge pages in Linux had performance problems and people recommended not using them. Now, the situation is different.

[3] Ulrich Drepper, The Cost of Virtualization, ACM Queue, Vol. 6 No. 1 – 2008

Outline

1 Virtualization basics

2 Hardware assisted virtualization
- Nested paging

3 Example: Mini VMM with KVM

4 I/O virtualization
- How do modern Network Interface Cards (NIC) work
- Device emulation
- Virtio
- PCI pass-through
- Single-Root I/O Virtualization
- Inter-VM networking

5 Summary

KVM

- Linux-based hosted hypervisor
- Abstracts hardware-assisted virtualization of different architectures behind `ioctl`-based API
- We will develop a miniature user-space VMM
  - Simplest hardware to virtualize: serial port

  1. Setup the VM's memory
  2. Load the code to execute
  3. Run the VM
  4. Handle the VM Exits and emulate serial port
  5. Goto 3

  - See also `https://lwn.net/Articles/658511/`

Outline

1 Virtualization basics

2 Hardware assisted virtualization
- Nested paging

3 Example: Mini VMM with KVM

4 I/O virtualization
- How do modern Network Interface Cards (NIC) work
- Device emulation
- Virtio
- PCI pass-through
- Single-Root I/O Virtualization
- Inter-VM networking

5 Summary

Outline

1 Virtualization basics

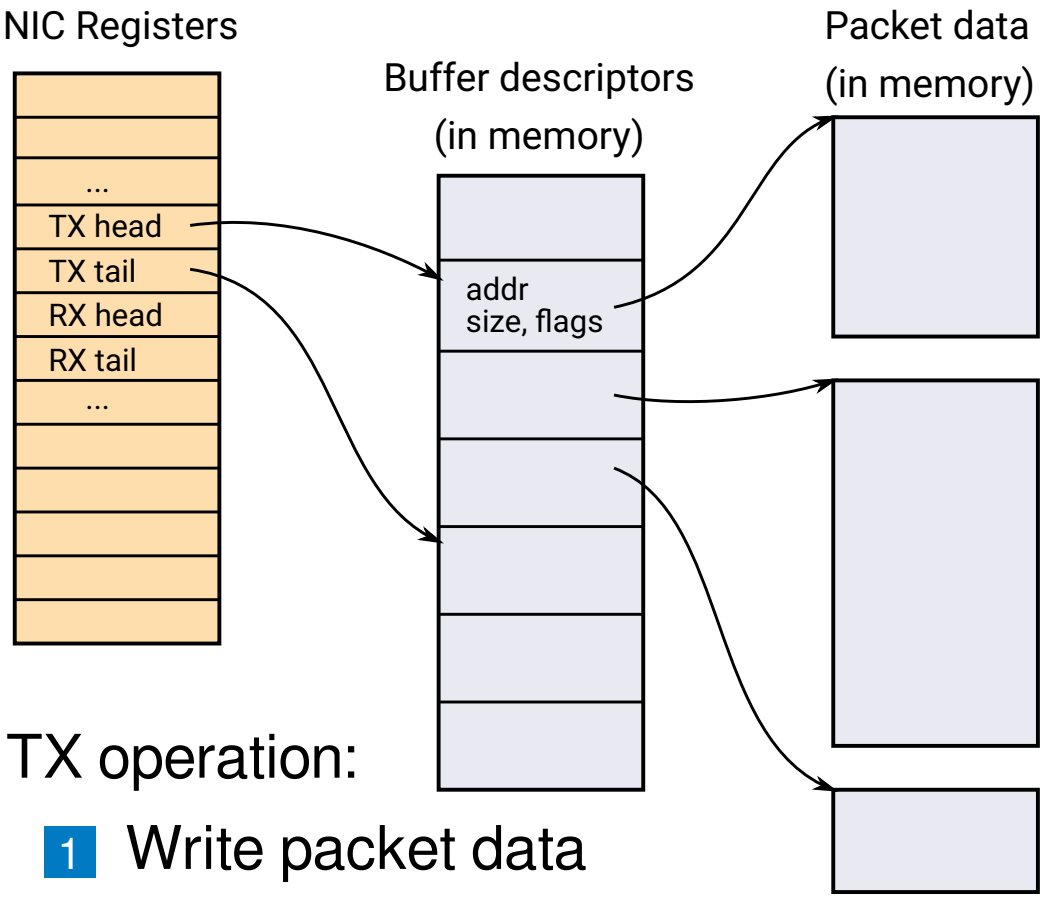
2 Hardware assisted virtualization
- Nested paging

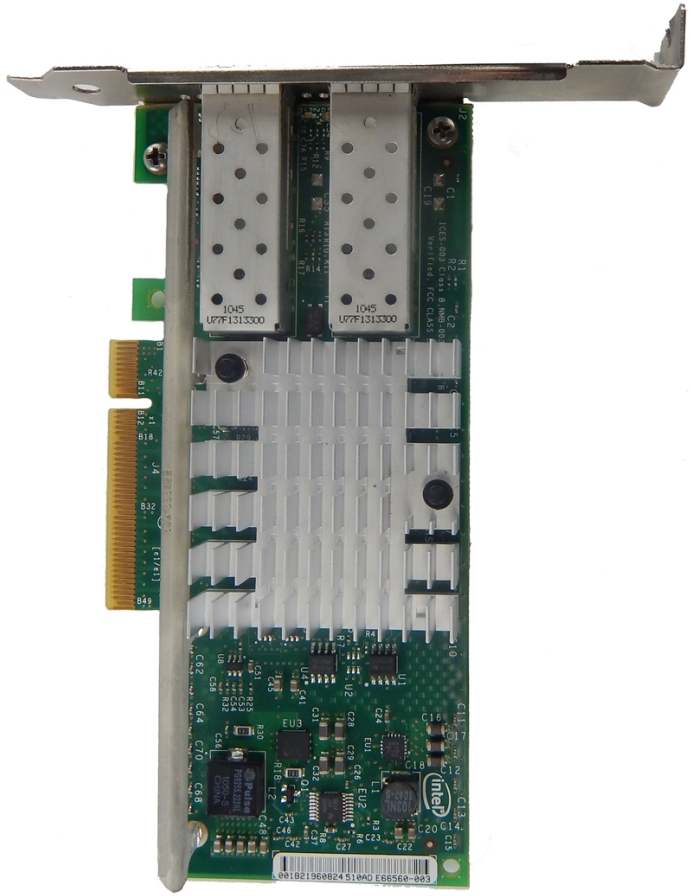
3 Example: Mini VMM with KVM

4 I/O virtualization
- How do modern Network Interface Cards (NIC) work
- Device emulation
- Virtio
- PCI pass-through
- Single-Root I/O Virtualization
- Inter-VM networking

5 Summary

Network Interface Card & transmit operation

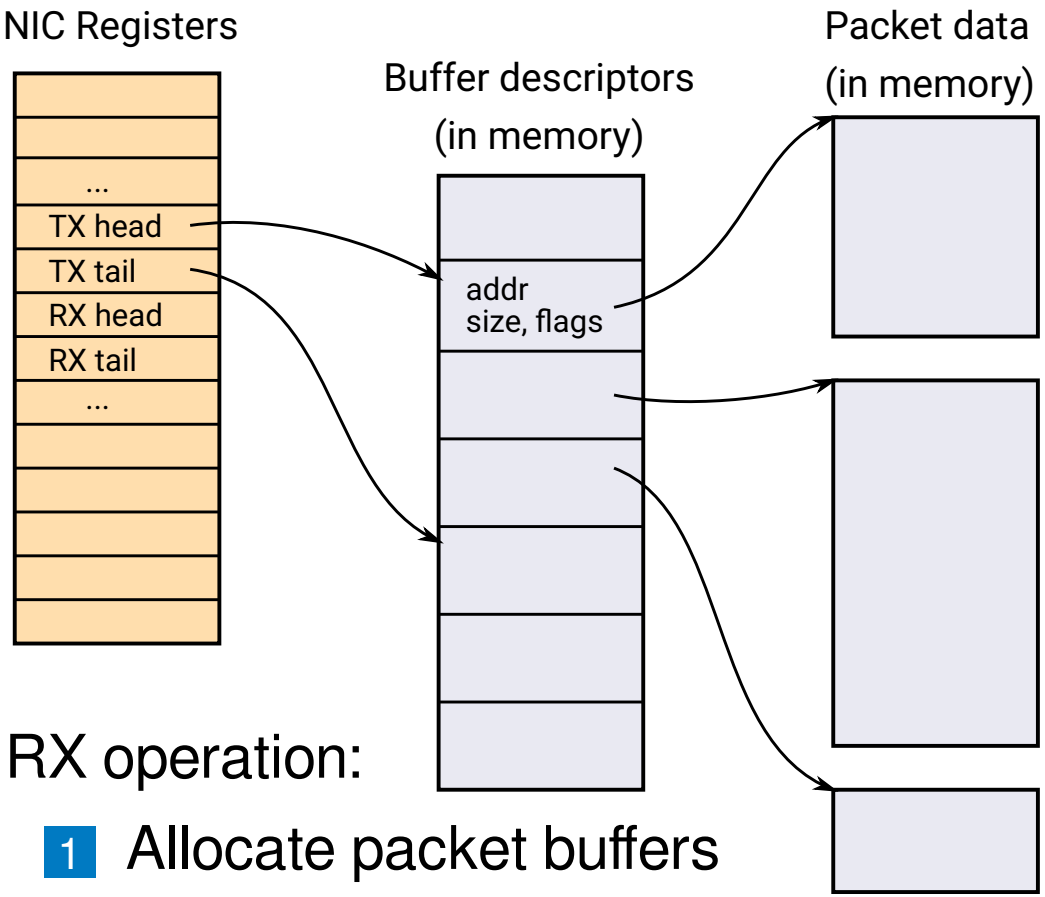


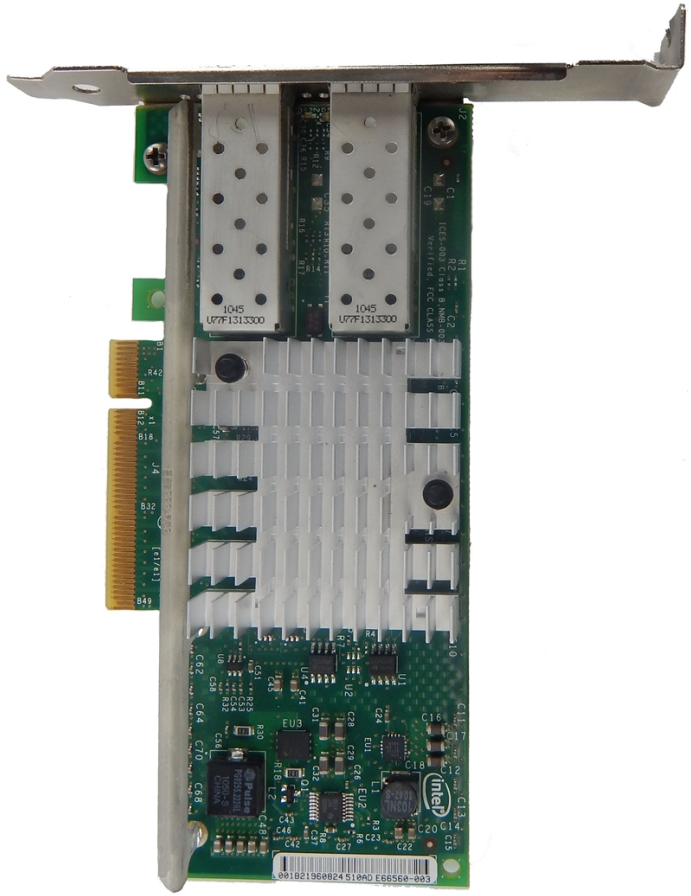


TX operation:

1. Write packet data
2. Fill in an empty buffer descriptor
3. Notify the NIC by writing TX tail reg

Network Interface Card & receive operation

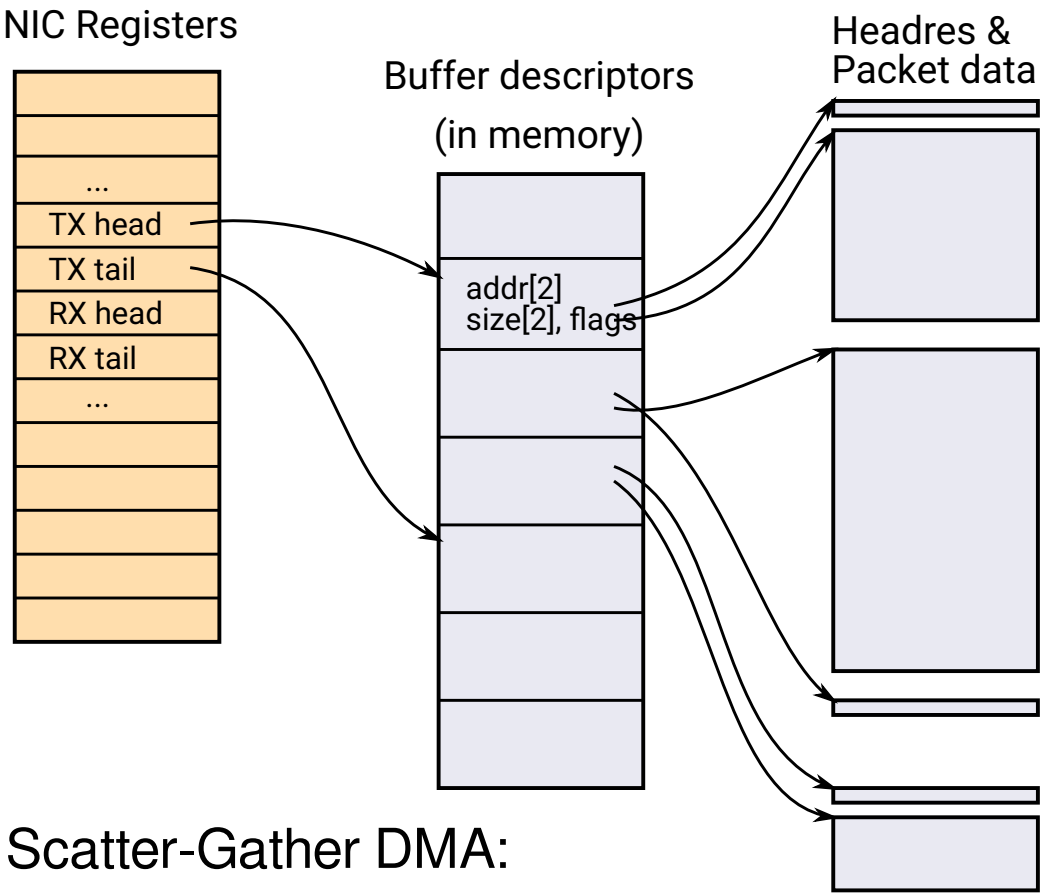


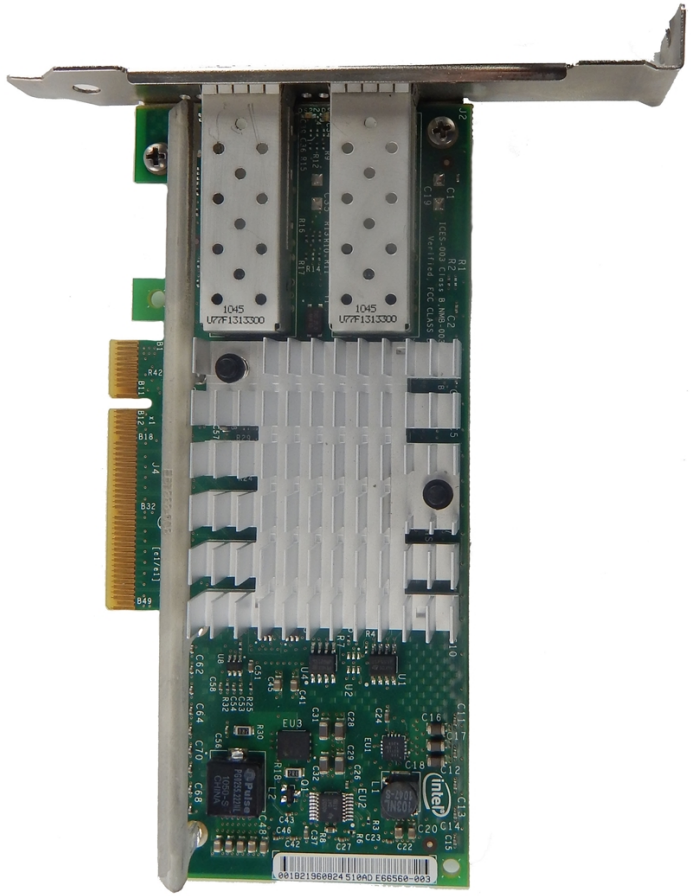


RX operation:

1. Allocate packet buffers and update buffer descriptors
2. Update RX head/tail regs
3. On packet reception, NIC stores the data to memory and generates an interrupt

Network Interface Card & SG DMA





Scatter-Gather DMA:

- NIC composes the final packet from several pieces scattered in memory
- Typically header (from OS) and data (from app)

Outline

1 Virtualization basics

2 Hardware assisted virtualization
- Nested paging

3 Example: Mini VMM with KVM

4 I/O virtualization
- How do modern Network Interface Cards (NIC) work
- Device emulation
- Virtio
- PCI pass-through
- Single-Root I/O Virtualization
- Inter-VM networking

5 Summary

NIC device emulation

- Trap accesses to NIC registers (memory-mapped IO)
- Upon write to TX tail, VMM iterates over queued buffers and sends them via real NIC (e.g. SOCK_RAW)
- Multiple packets can be sent during a single VM Exit ($\Rightarrow$ less overhead)
- Reception works similarly

- Not all hardware is “that nice” to virtualize
  - Several VM Exits per TX or RX
  - Registers that must be trapped are intermixed with non-sensitive (e.g. read-only) registers in a single page
    - $\Rightarrow$ Unnecessary VM Exits for some register accesses
- VMM must emulate not only RX/TX, but also management
  - Link negotiation, configuration, ...
  - More complex compared to RX/TX

Outline

1 Virtualization basics

2 Hardware assisted virtualization
- Nested paging

3 Example: Mini VMM with KVM

4 I/O virtualization
- How do modern Network Interface Cards (NIC) work
- Device emulation
- Virtio
- PCI pass-through
- Single-Root I/O Virtualization
- Inter-VM networking

5 Summary

Virtio

- It is neither easy nor necessary to emulate a real NIC
- Emulation of TX, RX and simple configuration (e.g. MAC address) is sufficient
- Why to implement different ring-buffer formats?

- Virtio[4]
  - Universal ring-buffer-based communication between VM and HV
  - Used for network, storage, serial line, ...
  - PCI-based probing & configuration – VMs can easily discover virtio devices

[4] R. Russell, virtio: Towards a De-Facto Standard For Virtual I/O Devices, ACM SIGOPS Operating Systems Review, 2008

Outline

1 Virtualization basics

2 Hardware assisted virtualization
- Nested paging

3 Example: Mini VMM with KVM

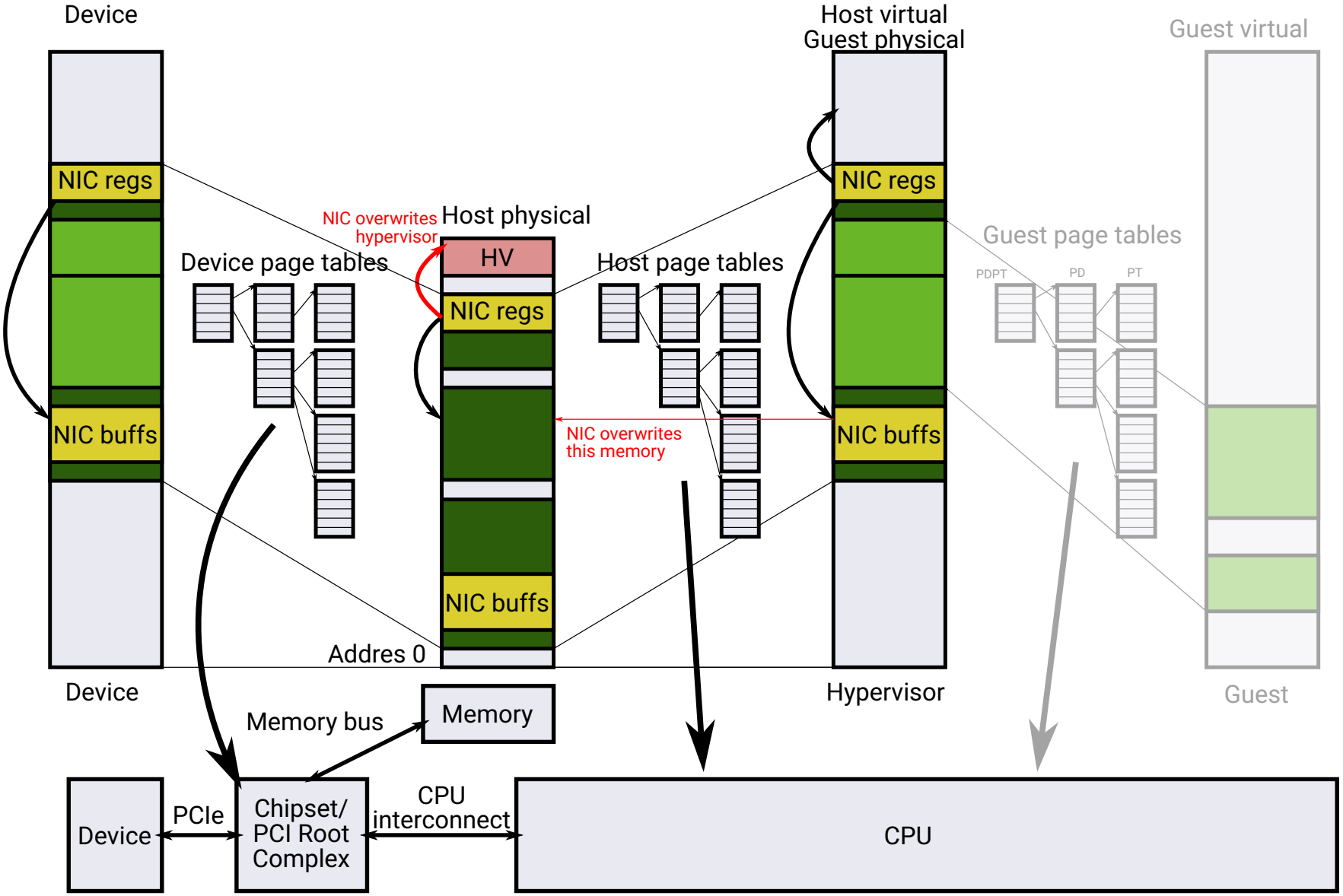
4 I/O virtualization
- How do modern Network Interface Cards (NIC) work
- Device emulation
- Virtio
- PCI pass-through
- Single-Root I/O Virtualization
- Inter-VM networking

5 Summary

PCI pass-through

- Even virtio needs one VM Exit per (a batch of) TX operation(s)
- If we don't want VM Exits, we may want to give a VM exclusive access to the NIC
- Few problems to solve...

PCI pass-through graphically



PCI pass-through

- Problems:
  1. Virtual address space (see previous slide)
     - Security: One VM could configure the NIC to read or write memory of other VM or even the hypervisor!
  2. Device interrupts: Host does not know how to acknowledge (silence) the interrupt – it has no driver for the device
     - Host injects the interrupt to the VM and returns from the IRQ handler
     - Host is interrupted again, because the VM didn't have chance to run and ack the interrupt
     - $\Rightarrow$ infinite loop
- Solution: Hardware support for direct use of devices in VMs
  1. IOMMU (AMD), VT-d (Intel), SMMU (ARM)
  2. Mask individual sources of interrupts without understanding the device
     - Hard with (parallel) PCI, where interrupt lines are shared between devices
     - Possible with Message Signaled Interrupts (MSI), PCI-express

Outline

1 Virtualization basics

2 Hardware assisted virtualization
- Nested paging

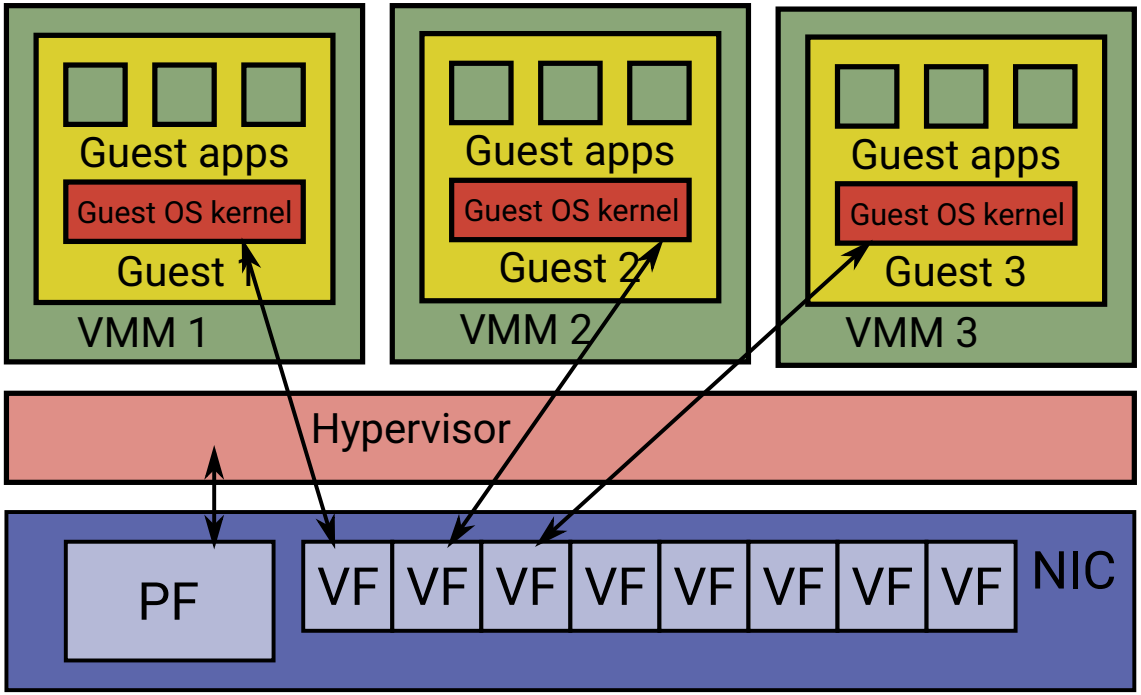
3 Example: Mini VMM with KVM

4 I/O virtualization
- How do modern Network Interface Cards (NIC) work
- Device emulation
- Virtio
- PCI pass-through
- Single-Root I/O Virtualization
- Inter-VM networking

5 Summary

Single-Root I/O Virtualization (SR-IOV)

- PCI pass-through is nice, but I have more VMs that want to communicate...
  - Each VM has an emulated NIC, VMM multiplexes the real NIC between VMs in software
  - What about performing the multiplexing in hardware?



- SR-IOV
  - Besides “classic” physical function (PF), NIC implements several virtual functions (VFs)
  - Each VF provides simplified PCI interface and its own RX/TX ring buffers

Outline

1 Virtualization basics

2 Hardware assisted virtualization
- Nested paging

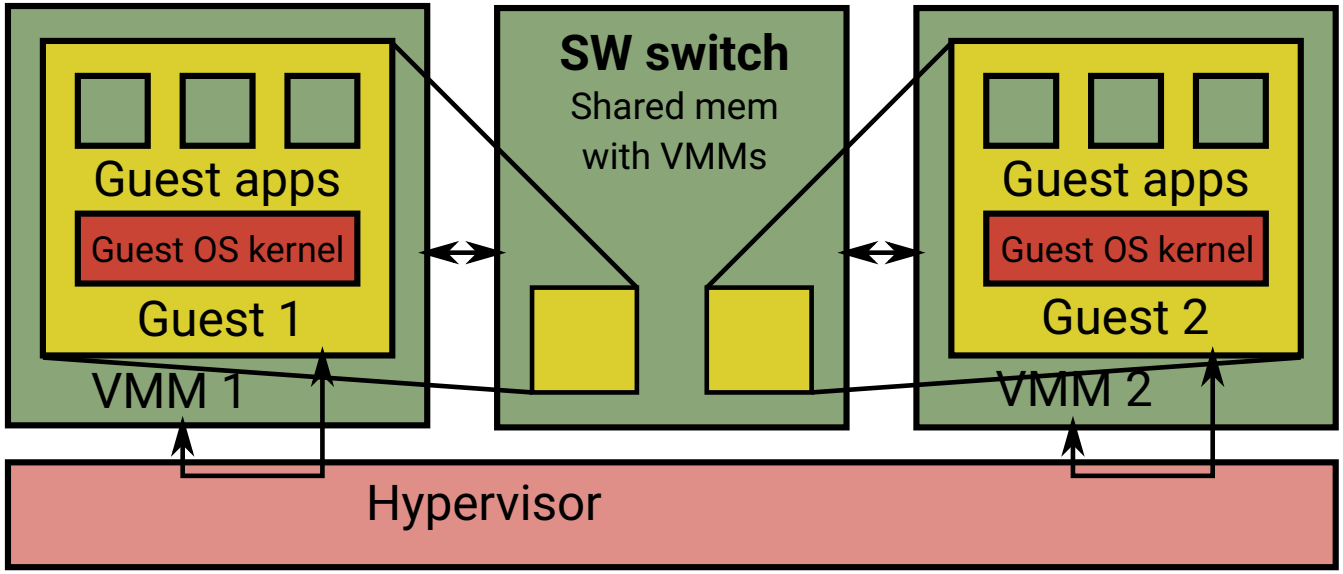
3 Example: Mini VMM with KVM

4 I/O virtualization
- How do modern Network Interface Cards (NIC) work
- Device emulation
- Virtio
- PCI pass-through
- Single-Root I/O Virtualization
- Inter-VM networking

5 Summary

Efficient inter-VM networking

Software-based Ethernet switch



- Packet stored in VM's memory
- VMM notified (VM Exit) e.g. via virtio's kick()
- VMM notifies the SW switch via standard IPC mechanism
- Switch does `memcpy()` of the packet from source VM to destination VM (into dest NIC ring buffer)
  - Note: The switch can see (`mmap()`) all VMs memory – the same as for real hardware NIC
- Dest VMM notifies the VM (injects an interrupt)

Optimizations

- OS networking stack is responsible for splitting application data to packets (e.g. TCP segmentation) and adding appropriate headers
- VMM sees many small packets and the switch does many small memcpy()s
- Receiver's networking stack strips packet headers and combines the payload to larger data chunks for application.

- TCP segmentation is not necessary for Inter-VM communication (overhead)!
- Modern NICs support TCP Segmentation Offload (TSO)/Large Receive Offload (LRO): Segmentation/reconstruction is done in hardware.
- If the virtual NIC supports TSO/LRO, Inter-VM communication is much faster, because whole TCP segments (in contrast to small packets) can be copied at once.

Outline

1 Virtualization basics

2 Hardware assisted virtualization
- Nested paging

3 Example: Mini VMM with KVM

4 I/O virtualization
- How do modern Network Interface Cards (NIC) work
- Device emulation
- Virtio
- PCI pass-through
- Single-Root I/O Virtualization
- Inter-VM networking

5 Summary

Summary

- Virtualization is just “another layer of indirection” and as such it adds overheads
- It is useful to know where the overheads are and how to mitigate them