μ-benchmarky v Java

Karel Rank - @karl82

Co je to *μ*-benchmark

- jako “normální” benchmark
- měří malou část kódu ~ < 1ms
- *profiling?*
- nezatížený systém
- FIKTIVNÍ/UMĚLÝ

Rada č.1,2 & 3

- nepište a nepoužívejte *μ*-benchmarky
    - -> ~~optimializacím~~
- pište Readable&Well Designed Code™
- použít caliper https://code.google.com/p/caliper

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

Děkuji za pozornost!

μ-benchmark struktura

1. zavolat JVM Warm-up melody
2. zavolat měřený metody
3. vypsat výsledky

- nedělat nic přímo v `main`

Vlivy

- env - OS, JRE, libs
- JVM/JIT/GC
- CPU věci
- OS scheduler

- metoda měření
- data
  - reálné
  - náhodné

Prostředí

- OS
  - Windows/Linux/Solaris/AIX/HP-UX/Android
  - verze
- JRE/JDK
- knihovny třetích stran

Metody měření

- delta mezi dvěmi hodnotami z HR zdroje času
- t1 - před testem; t2 - po testu

  dt = t2 - t1
- System.nanoTime()
  - přesnost X ~~rozlišení~~
  - Windows vs Linux

Metody měření

- ~~System.currentTimeMillis()~~

https://blogs.oracle.com/dholmes/entry/inside_the_hotspot_vm_clocks

JVM

- JVM interní vlákna
  - VM vlákno
  - Vlákno pro periodické úlohy
  - GC vlákna
  - Vlákna překladače
  - Signal dispatcher

- classloader and classloading
  - on demand/lazy
  - unzipping JARs
  - bytecode ověření
  - init of classes
- neměřte I/O

JIT

- client (C1)/server (C2)
- tiered (C1+C2)
- vlákna překladače
- hot code
  - ~ 3.000 calls C1
  - ~ 10.000 calls C2

JIT optimalizace C2

- hodně profilování v interpretu bytekódu
  - závisí na datech
  - statistiky větvení
- chyba -> deoptimalizace
- tvůj kámoš https://github.com/AdoptOpenJDK/jitwatch

JIT vnořování

- milujeme vnořování
  - hluboko
  - dynamické/virtuální metody
  - tisíce bytekódů
- nakopává další optimalize

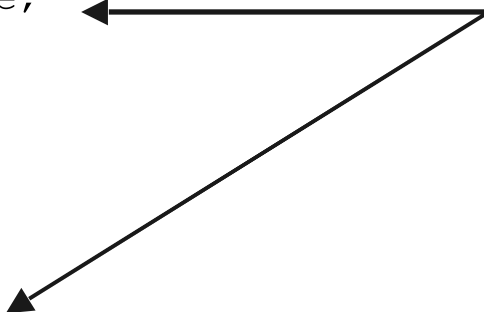
http://www.azulsystems.com/blog/cliff/2011-04-04-fixing-the-inlining-problem

JIT inlining

```
int hashCode() {
    int hash = 0;
    for (Integer i : integers) {
        hash += i.hashCode();
    }
}
class Integer {
    int hashCode() {
        return value;
    }
}
```

JIT inlining

```
int hashCode() {
    int hash = 0;
    for (Integer i : integers) {
        hash += i.value;
    }
}
class Integer {
    int hashCode() {
        return value;
    }
}
```



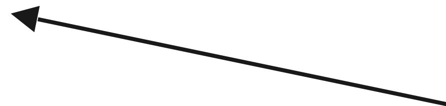
JIT unlooping

- rozbalování smyček!
- méně větvení
- víc vnořování

pouze C2

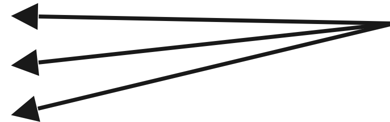
JIT unlooping

```
private static final String[] fooBar = { “Foo,
“Bar”, “FooBar” };

private void unrollFooBar() {
   for (final String s : fooBar) {
         printVerySpecial(s);
    }
}
```



JIT unlooping

```
private static final String[] fooBar = { “Foo”,
“Bar”, “FooBar” };


private void unrollFooBar() {
  printVerySpecial(s[0]);
  printVerySpecial(s[1]);
  printVerySpecial(s[2]);
}
```



JIT DCE

- hodně c00l
- hodně zrádný
- odstraní kód bez vedlejších efektů
  - nepoužitá hodnota
  - samo přiřazení

JIT DCE

Starting point

```
public void newMethod() {
  y = b.get();
  ...do stuff...
  z = b.get();
  sum = y + z;
}
```

Inline method

```
public void newMethod() {
  y = b.value;
  ...do stuff...   <--
  z = b.value;     <--
  sum = y + z;
}
```

Remove redundant loads

```
public void newMethod() {
    y = b.value;
    ...do stuff...
    z = y;   <--
    sum = y + z;
}
```

Copy propagation

```
public void newMethod() {
    y = b.value;
    ...do stuff...
    y = y;   <--
    sum = y + y;
}
```

JIT DCE

Eliminate dead code

```
public void newMethod() {
    y = b.value;
    ...do stuff...
    y = y;  <----
    sum = y + y;
}
```

http://docs.oracle.com/cd/E15289_01/doc.40/e15058/underst_jit.htm

JIT On-Stack Replacement

- spuštěná metoda se nikdy neukončí
- nahradí aktuální metodu, která se stala hot
  - smyčky
  - back-branching
- nepoužívá všechny optimalizace
  - není měřená top výkonost

http://www.azulsystems.com/blog/cliff/2008-03-07-another-round-micro-benchmark-advice

http://www.azulsystems.com/blog/cliff/2011-11-22-what-the-heck-is-osr-and-why-is-it-bad-or-good

GC

- zabránit
- stop-the-world
  - i s CMS
- je to možné?
  - před každým testem System.gc()
  - co nejvíc paměti pro JVM `-Xmx`/`-Xms`

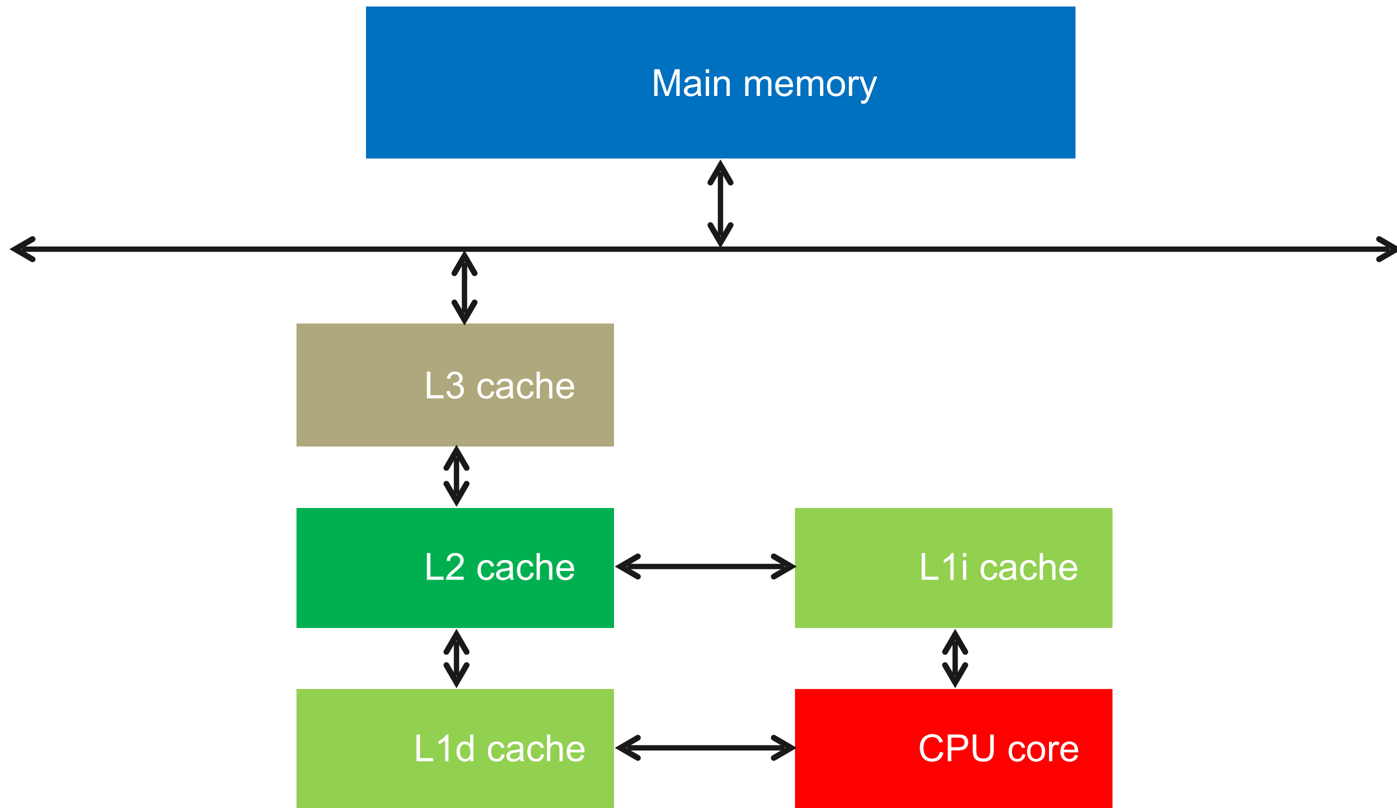
JVM warm-up

- “řeší” přechozí problémy
- stejný kód pro warm-up i test

1. spustit warm-up melody
2. změřit smysluplný hodnoty
3. nezapomenout GC před každým testem

CPU cache

- instruction cache
- data cache
- TLB
  - virtual address -> physical address


Main memory
L3 cache
L2 cache
L1i cache
L1d cache
CPU core

CPU cache

http://www.akkadia.org/drepper/cpumemory.pdf

CPU Branch Predictions

- pipelines
- co když je skok?
- předpovídej!
- omyl
  - -> zahoď pipeline
- RANDOMIZE

Instr No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

CPU power saving

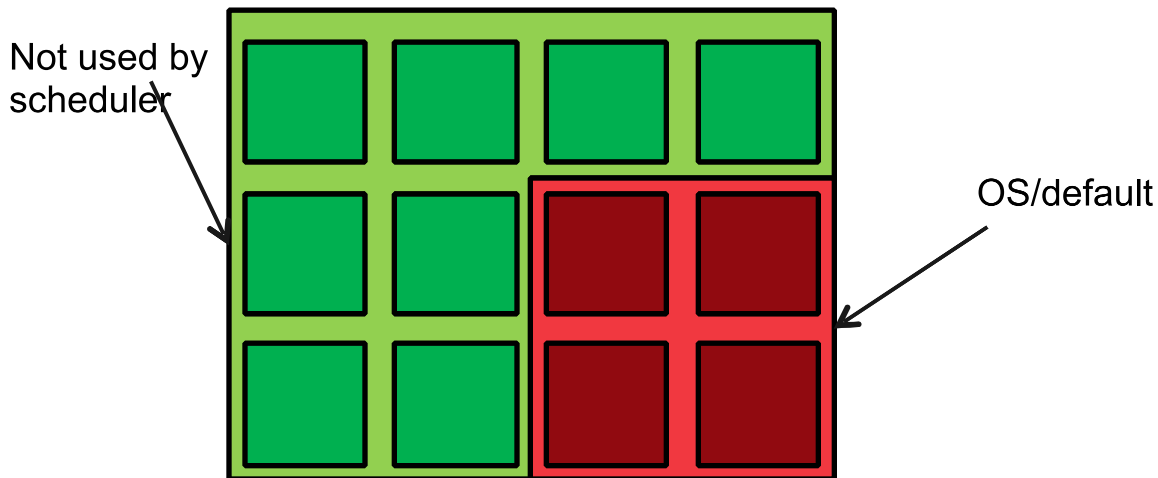
- počkat… co?

OS scheduler

- liší se OS od OS (samozřejmě!!!)
- linux 2.6.32-29-server x86_64
  - context switch Intel L5630 http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html
    - proces ~ 3000ns
  - affinity
    - proces ~ 1600ns
    - vlákno ~ 1200ns

OS scheduler

- cache znečištění
- affinity
  - rezervuj jádra
  - upni vlákna
- kdy?
  - -> latency



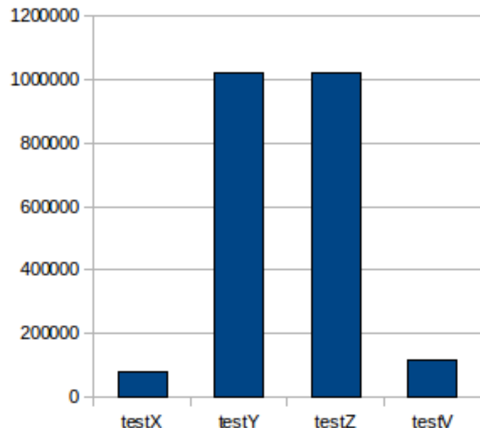
Examples

https://github.com/karl82/ubench

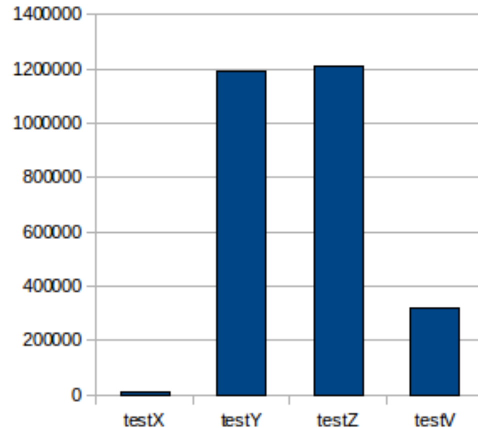
OSR Příklad

http://www.root.cz/clanky/pohled-pod-kapotu-jvm-zaklady-optimalizace-aplikaci-naprogramovanych-v-jave-5/

C1 a C2 mají stejnou výkonnost pro synchronizované bloky?

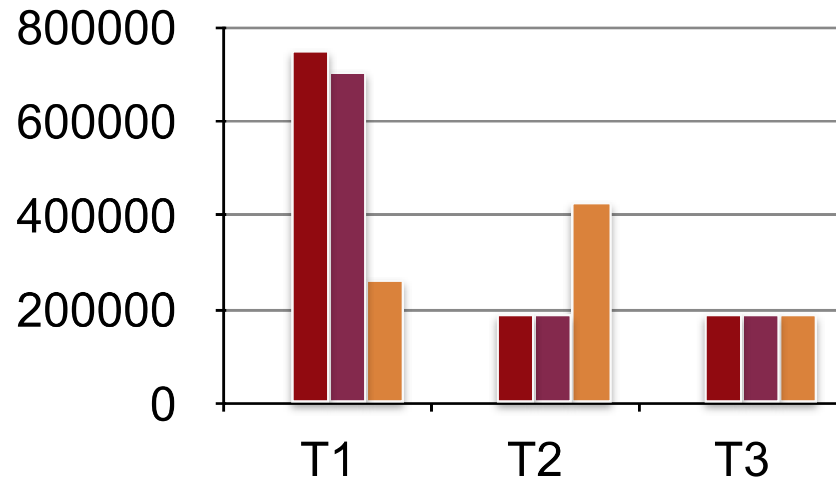


C1



C2

OSR Příklad



Kde dělám?

V Barclays!

http://www.barclays.com/barclays-careers.html

Děkuji!