Thoughts On
The Java Performance Model

Klaus Kreft
klaus.kreft@siemens.com

Angelika Langer
www.AngelikaLanger.com

objective

- what is the idea of a performance model?
  - "C is the last programming language"
- how is it in Java?
  - complex, counter-intuitive, unpredictable
- how do we measure performance in Java?
  - the art of benchmarking
- where do we find reliable information?
  - looking for the needle in the haystack
- how do we cope without information?
  - tools, tips & techniques

agenda

- programming languages and their performance model
- Java performance model and the `String`’s operator `+`
- performance investigation and micro benchmarking
- performance related information is hard to find
- techniques and tools for performance investigations

“… the Last Programming Language”

- controversial article by Richard P. Gabriel:
  “The End of History and the Last Programming Language”,
  JOOP, July 1993

- a theory of what makes a successful general-purpose programming language

what makes a successful programming language

- languages are accepted and evolved by a social process, not a technical or technological one
  - available on a wide variety of hardware
  - easy to learn
  - it’s implementation should be simple for it to spread
  - it should be similar to existing language

(cont.)

- successful languages must ...
  - not require that users have ‘mathematical sophistication’
  - have modest or minimal computer resource requirements
  - have a simple performance model

Richard Gabriel’s ideas were inspired ...

- by the time the article was published
  - C was the most widespread language
- by his own experiences
  - enthusiastic and experienced LISP and CLOS programmer
  - disappointed about LISP’s lack of success

simple performance model

"One of the things I learned by watching people try to learn Lisp is that they could write programs that worked pretty well but they could not write fast programs.
The reason was that they did not know how things were implemented and assumed that anything in the language manual took a small constant time."

"What customers really care about is price and speed. If your program costs too much, forget it; if it's too slow, forget it."

C performance model

- original Kernighan-Ritchie-C:
  - each statement has similar CPU cost
- relaxed with ANSI-C

C performance model - example

- definition of a **struct**:

```
struct s {
  int a;
  char b;
}
```

- original Kernighan-Ritchie-C:

```
...
struct s myS;
myS.a = 2
myS.b = ‘b’;
...
```

- ANSI-C:

```
...
struct s myS = { 2, ‘b’ };
...
```

reason for the title of the article

“The End of History and the Last Programming Language”

- C is the last programming language
  - with the advent of the OO-paradigm in the early 90ies new programming languages will never have a performance model as simple the one of C
  - hence no new widespread general-purpose programming language

- debate with Andrew Koenig
  - who defended C++ as having a sufficiently simple performance model to be successful

agenda

- programming languages and their performance model
- Java performance model and the `String`’s operator +
- performance investigation and micro benchmarking
- performance related information is hard to find
- techniques and tools for performance investigations

`String` and operator +

since the early days of Java you could do the following:

```
String foo(String s1, String s2) {
  return “result: “ + s1 + “ + “ + s2;
}
```

the compiler converts:

```
   “result: ” + s1 + “ + “ + s2;
```

to:

```
new String(“result”).concat(s1).
                     concat(“ + “).concat(s2);
```

(cont.)

- `String` is an immutable type
- each call to `concate()` creates a new object
- which is immediately discarded
- `->` two objects overhead

recommended alternative

use `StringBuffer`:

```
String foo(String s1, String s2) {
  StringBuffer buf = new StringBuffer (“result”);
  buf.append(s1).append(“ + “).append(s2);
  return buf.toString();
}
```

- only one object overhead (the `StringBuffer` itself)

conclusion:

- do not follow your intuition to use what looks simpler and more elegant

recent trends (> JDK 1.4)

compiler used to convert:

```
   “result: ” + s1 + “ + “ + s2;
```

to:

```
new String(“result”).concat(s1).
                     concat(“ + “).concat(s2);
```

- no longer true for newer compilers (JDK 1.4.x)
  - compiler does the `String` concatenation with `StringBuffer`
  - advantages of explicit usage of `StringBuffer`
    - JDK platform independent
    - easier to understand, e.g. during debugging

recent trends (> JDK 5.0)

- new class: `StringBuilder`
- same API and functionality as `StringBuffer`
  - but no synchronization
    - i.e. cannot be accessed by multiple threads concurrently
- performance improvement in most situations
- JDK 5.0 compiler uses `StringBuilder` for `String` concatenation

(cont.)

- no JDK version independent ‘best’ solution
  - if you used String’s operator +
    - it was slow with JDK 1.1
    - but has become fast with JDK 5.0
    - better than anything else that you could implement with 1.1 and run now under 5.0

lessons learned from this example

- Java performance model is highly complex
  - what looks simple and elegant is not necessarily fast
  - no JDK version independent ‘best’ solution
- Java compiler and runtime system try to iron out the wrinkles of the performance model for you
  - progressively over the JDK versions

(initially) suggested performance paradigm

- programmer cares only about high-level (design) alternatives
  - e.g. which type of collection to use

- the low-level stuff is handled by the Java compiler and runtime system
  - progressively optimized over the JDK versions

limitation of the paradigm

- your software might run into performance problems
- high level changes do not solve these problems
- you are forced to deal with the low-level stuff

agenda

- programming languages and their performance model
- Java performance model and the `String`'s operator +
- performance investigation and micro benchmarking
- performance related information is hard to find
- techniques and tools for performance investigations

micro-benchmark

- !!! you are forced to deal with the low-level stuff !!!
- micro-benchmark
  - typical approach to determine the performance costs of different low-level implementation alternatives
- idea of the micro-benchmark
  - small program that captures the essence of the respective implementation
  - all time spend in this few lines of code

idea of the micro-benchmark

- essential code executed in a loop
  - to compensate for different underlying system effects
  - to increase the time to be measured
    - counter precision problems
    - more precise calculation

example - micro-benchmark loop

```
Writer os = new BufferedWriter
            (new FileWriter(FILENAME), bufsize);

long start = System.currentTimeMillis();

for (int i=0; i<LOOP_SIZE; i++)
      os.write( (int) ‘A’);

long diff = System.currentTimeMillis() - start;

os.close();
```

note that setup and cleanup are not measured

problems

not Java specific

- wrong assumptions
  - e.g.: cost of serialization of a string array
    - measured for an array that holds the same `String`
    - while in the application it typically holds different `Strings`
- wrong implementation
  - e.g.: input generation included into the test loop

Java specific

- understand the JVM behavior

understand the JVM behavior:

- most problematic to understand:
  HotSpot JVM behavior
- uses runtime profiling to optimize the compilation of the executed code
- techniques used are:
  - complex
  - context dependent
  - intertwined

guidelines for HotSpot

- hardly any guidelines
  - no simple rules for behavior
  - technology is still changing
- keep an eye on:
  - monomorphic call transformation
  - warm-up
  - on-stack replacement
  - dead code elimination
  - loop unrolling, lock coalescing, ...
  - garbage collection
  - compiler differences

monomorphic call transformation

- without HotSpot
  - public, non-final methods can*not* be inlined
    - life was simple in pre-HotSpot times
- with HotSpot
  - also public non-final methods are inlined
    - based on global program analysis
    - if methods are found to be used monomorphically
  - potential deoptimization:
    - when new classes are loaded
    - which might use the inlined method polymorphically

example - dynamic deoptimization

```
abstract class A {
  public static double x = 2.0;
  abstract void aMethod();
}
class B extends A {
  public void aMethod() { x = x + 1.1; }
}
class C extends A {
  public void aMethod() { x = x - 1.1; }
}
class D extends A {
  public void aMethod() { x = x + 2.2; }
}
```

example - dynamic deoptimization (cont.)

```
public class DeoptimizationDemo {
  public static void run(A bar) {
    for (int i = 0; i < 1000000; i++)
      bar.aMethod();
  }

  public static void main(String[] args) {
    A b = new B();
    A c = new C();
    A d = new D();
    run(b); // warmup
    run(b); run(c); run(d);
  }
}
```

example - results / reason

- with server HotSpot JVM:
  - `B`: not measured, `B`: 20, `C`: 790, `D`: 180

- reason:
  - first loop (with `B`) gets optimized and compiled
    - inlines call monomorphic
  - second loop (with `B`) is extremely fast
  - third loop (with `C`) is extremely slow
    - call needs to be polymorphic now -> deoptimization
  - forth loop with polymorphic dispatch

example - monomorphic call (i)

- conceivable benchmark design:
  - compare two alternative implementations of an algorithm
    - implemented as two classes that implement the alternatives
  - run the alternatives in a loop for measurement
    - implemented as test driver that calls the alternatives

example - monomorphic call (ii)

```
interface Algorithm { void doIt(); }
class Algorithm_1 implements Algorithm { ... }
class Algorithm_2 implements Algorithm { ... }

class MicroBenchmark {
  private static long test(Algorithm alg) {
    long start = System.nanoTime();
    for(long i = 0; i < 10000000L; i++) alg.doIt();
    return System.nanoTime() - start;
  }
  public static void main(String[] args) {
    long time_1 = test(new Algorithm_1());
    long time_2 = test(new Algorithm_2());
    ... print statistics ...
  }
}
```

polymorphic method call

example - monomophic call (iii)

- problem: algorithm is a polymorphic method
  - falls victim of monomorphic call transformation
  - first execution of `test()` is optimized
    - only one class has been loaded and `doIt()` is still monomorphic
    - leads to inlining
  - second execution of `test()` is deoptimized
    - second class is loaded and `doIt()` is now polymorphic
    - roll back inlining
- observed effect:
  - the alternative that runs first, looks faster

warm-up

- hot spot optimization
  - happens after a fixed number of method invocations
    - typically 100 000
  - followed by periodical re-optimizations
    - including potential deoptimizations
  - stops after an unspecified amount of time
    - when all possible optimizations have been applied

- conclusion: benchmark must have a *warm-up* phase
  - repeat benchmark runs
    - to make sure HotSpot has settled down and produces steady results

warm-up threshold

- finding the warm-up threshold is difficult
  - early JIT compilers had a perceivable threshold value
  - more recent JIT compilers perform optimization at less predictable times
  - new techniques add to the complications
    - e.g. *on-stack replacement*

(cont.)

- run benchmark with `-XX:+PrintCompilation`
  - to find warm-up period
  - after a while JIT should calm down

- since 5.0: compiler management beans
  - `CompilationMXBean`
    - `long getTotalCompilationTime()`
    - returns the approximate accumulated elapsed time spent in JIT compilation

recap: keep an eye on

- monomorphic call transformation
- warm-up
- on-stack replacement
- dead code elimination
- loop unrolling, lock coalescing, ...
- garbage collection
- compiler differences

how do we cope with HotSpot ?

- use JVM options as you would do for the real system

- run different sized iterations
  - try to determine warm-up phase
  - with an eye on the real situation

- mean values may be misleading
  - measure variance, too

Java does a lot for you performance wise

- downside:
  - complex functionality
    - typical developers are not supposed to deal with it
  - information is hard to find
    - little published (books, magazine, internet, …)
    - high-level
    - incomplete
  - sometimes you are on your own

agenda

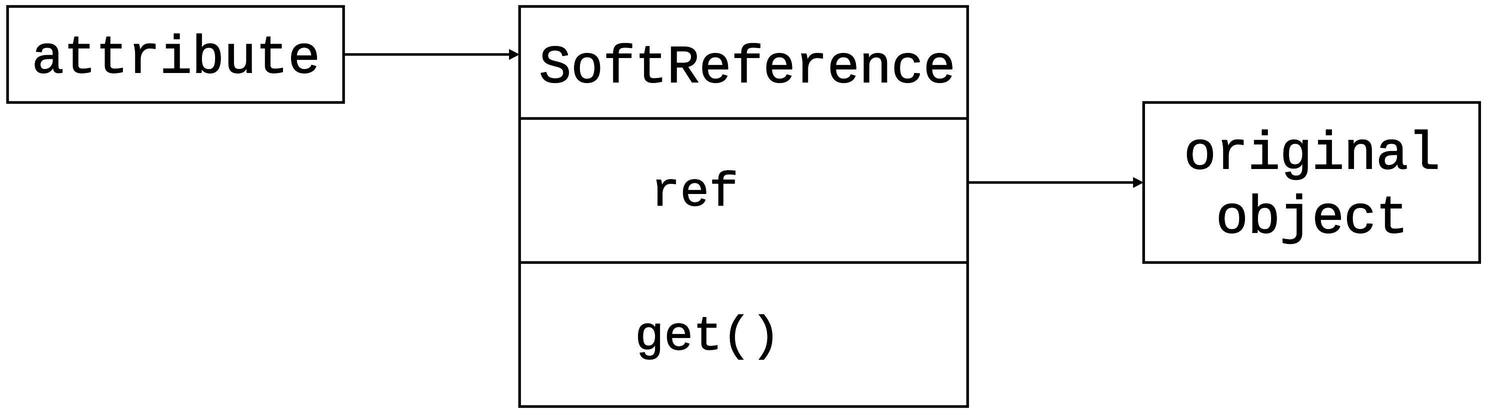
- programming languages and their performance model
- Java performance model and the `String`'s operator +
- performance investigation and the micro benchmarking
- performance related information is hard to find
- techniques and tools for performance investigations

situation

- database cache based on
  `java.lang.ref.SoftReference`
- details
  - some data in the database can only be changed by a system administrator
  - this is infrequently done
  - idea: cache queries that target this data in memory
    to increase the performance

how references work, pt. 1

- you do not refer to the object directly
  - but wrap it as a ***reference*** (e.g. a `SoftReference`)
  - the ***reference*** has a method `get()` which returns the reference to the original object



how references work, pt. 2

- when the object is not strongly reachable anymore
  - the garbage collector might clear the `SoftReference`
    - because of memory demands
  - method `get()` returns then `null`
  - optionally (depending on a ctor parameter) the cleared `SoftReference` is place into a queue
    - to process additional cleanup

from the `SoftReference` JavaDoc

Soft references are most often used to implement memory-sensitive caches.

Direct instances of this class may be used to implement simple caches; this class or derived subclasses may also be used in larger data structures to implement more sophisticated caches.

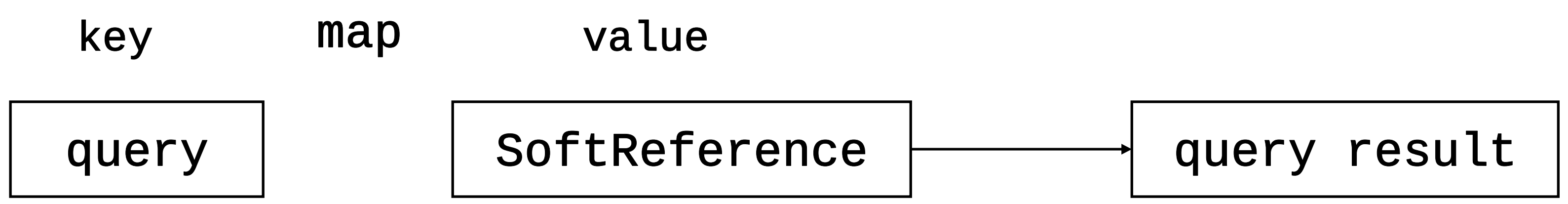
more from the `SoftReference` JavaDoc

All soft references to softly-reachable objects are guaranteed to have been cleared before the virtual machine throws an `OutOfMemoryError`.

Otherwise no constraints are placed upon the time at which a soft reference will be cleared or the order in which a set of such references to different objects will be cleared. Virtual machine implementations are, however, encouraged to bias against clearing recently-created or recently-used soft references.

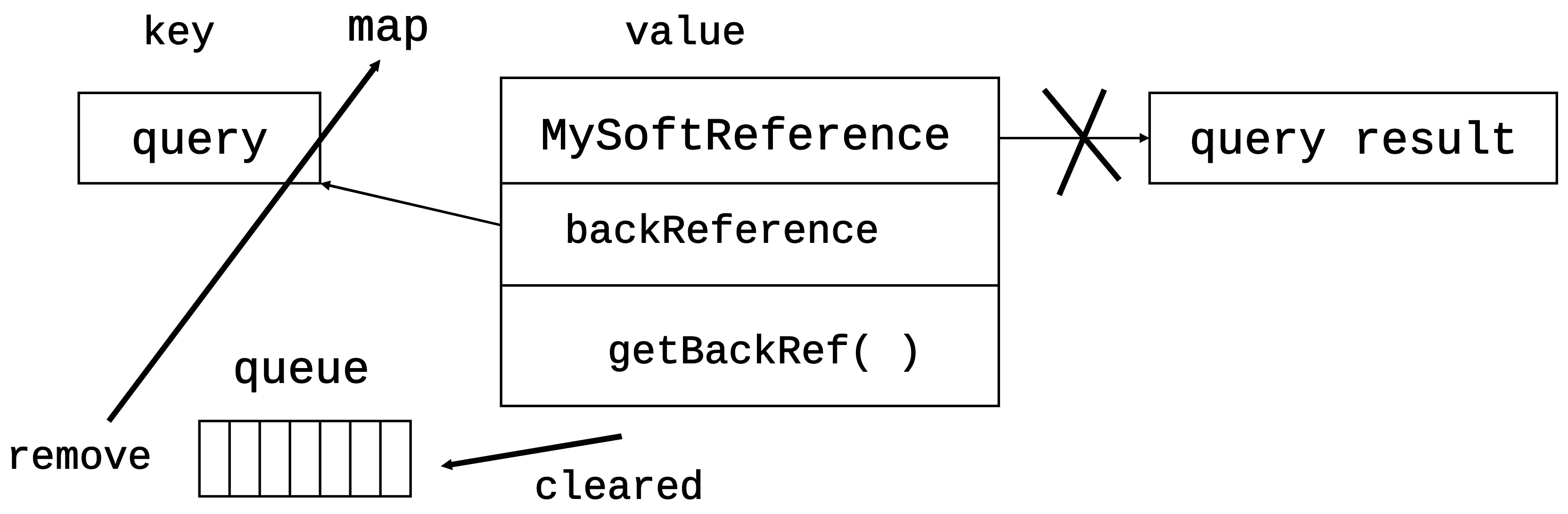
cache solution, pt. 1

- store
  - query, and
  - soft reference of the query result
  - as key and value in a map
- cache
  - before querying the database query the map



cache solution, pt. 2

- `MySoftReference`
  - derived from `SoftReference`
  - stores back reference to the query
  - allows to remove cleared `SoftReference` from the queue and use the back reference to clear the entry from the map



situation

- a small test (including the database) proved the functionality of the solution
- but with the real system the solution was a disaster
  - cache did not contain the requested data in almost all cases
  - even if it the same query was processed less than one minute before

and then?

- we assumed that our implementation was erroneous
  - despite the small test before that was successful
  - we looked for some error that was introduced when we put the cache into the whole system
- but our investigations showed that the `SoftReferences` had been cleared
  - and that was the reason for the cache misses
- we re-read the Java doc

`SoftReference` JavaDoc

All soft references to softly-reachable objects are guaranteed to have been cleared before the virtual machine throws an `OutOfMemoryError`.

Otherwise <u>no constraints</u> are placed upon the time at which a soft reference will be cleared or the order in which a set of such references to different objects will be cleared. Virtual machine implementations are, however, encouraged to bias against clearing recently-created or recently-used soft references.

and then?

- we increased the value of the JVM’s `–Xmx` option to absurdly high values
  - only to find out that this option seemed to have no effect on our problem
- we started to look for information about `SoftReferences`
  - to find something that could be related to our problem

information about SoftReference

- a lot of articles in print or online magazines
  - mostly dating back to the time references were introduced in Java (JDK 1.2)
  - content:
    - explanation how references work in general
    - some simple example implementation (even simpler that our small test)

information about SoftReference

- books
  - similar situation
    - text books only explain the intent of soft references, if at all, but never address practical problems
    - not even "The Java Programming Language" by Arnold, Gosling & Holmes has any information to offer
  - today, "Hardcore Java" by Robert Simmons points to the problem, but it was published years later (in 2004)

(cont.)

- online information from Sun about their JVM
  - official specs (for language and JVM) do not cover soft references
    - GC is an implementation detail
  - a lot of information about GC
    - but no details about GC and `SoftReference`
- online forums
  - wildest speculations, and
  - most contradicting statements

online forums example, pt. 1

- in a thread about problems with references there was one contribution that consisted of just one sentence:
  - *“Don’t you know that the whole reference stuff is broken !!!”*
  - no additional detail, nothing
  - when asked the only thing the poster disclosed was that:

    “This is a well known fact.”

online forums example, pt. 2

- on Sun’s ‘Java bug parade’ we found an error description:
  - ‘SoftReferences are cleared to eagerly’ (Bug ID: 4230645)
  - which looked quite similar to our problem
- but
  - it was against JDK 1.3 beta and closed in May 1999 with the comment: *“We added a LRU policy and stopped immediately clearing soft refs on each gc.”*
  - our problem appeared with JDK 1.3.2 in October 2002
  - comments added to this bug in June and August 2001 (JDK 1.3.1) showed that other people still had similar (?) problems

and then?

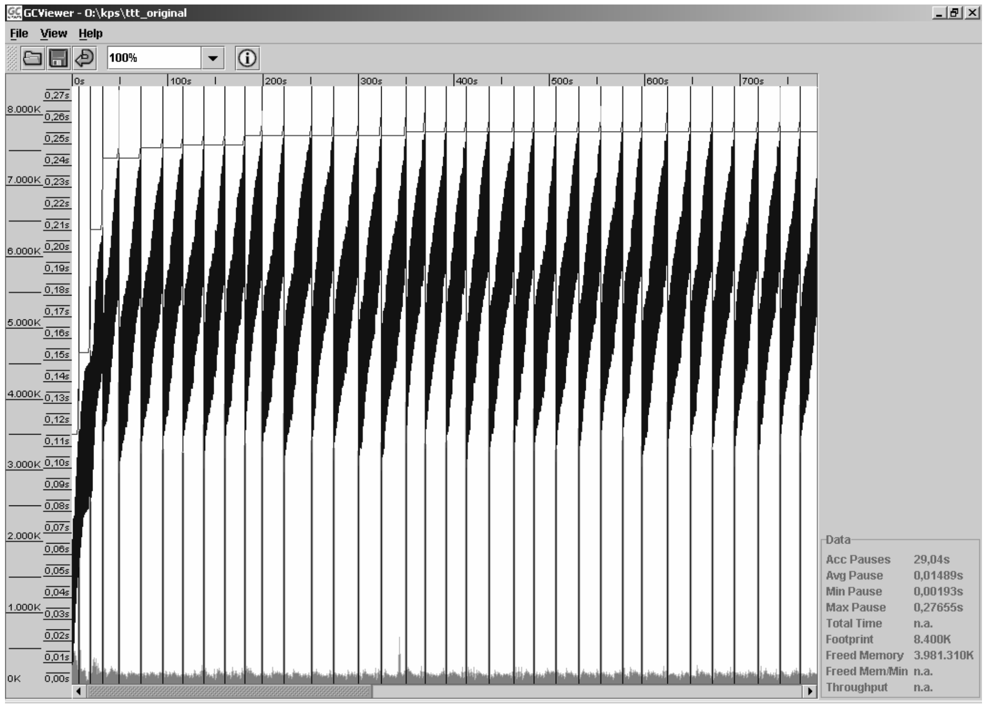
- the info from the ‘bug parade’ showed us that there were problems with the aggressive clearing of `SoftReferences`
  - which might have been fixed
  - but other people had still problems
- questions that remained for us:
  - when are the `SoftReferences` cleared in our system
    - which condition(s) trigger the GC to clear them
  - is it possible to change this behavior
- start our own investigations

start our own investigations

- we added a thread to our system which
  - does the same read access (via the cache) to the database repeatedly
    - (timer configurable, start with: 200 msec)
  - additionally we traced whether this was a cache hit or miss
- we started the JVM with the option `–verbose:gc`
  - to trace GC activates
  - to find the relationship between a certain GC activity and the moment the database access changes from cache hits to a miss

result

- each major collection cleared the `SoftReference`
  - happens when the remaining objects from the ‘young generation’ get copied to the ‘old generation’
  - pretty normal situation (nothing close to out of memory)



Acc Pauses	29,04s
Avg Pause	0,01489s
Min Pause	0,00193s
Max Pause	0,27655s
Total Time	n.a.
Footprint	8.400K
Freed Memory	3.981.310K
Freed Mem/Min	n.a.
Throughput	n.a.

what do we learn from this

- books, magazines, etc.
  - standard information
  - often lack the amount of detail to help with specific performance issues
- internet
  - quality of information varies
  - sometimes: unclear / confusing / contradicting
- sometimes you are forced to do your own investigations
  - because you do not get a coherent answer

agenda

- programming languages and their performance model
- Java performance model and the `String`'s operator +
- performance investigation and the micro benchmarking
- performance related information is hard to find
- techniques and tools for performance investigations

previous example

tools/techniques used in the previous example:

- JVM’s GC profiling functionality
  - to see when GC activity is in progress
- tracing
  - to see if the db access is a cache hit or miss

general rules to pick tools/techniques

- favor the approach ...
  - ... that provides you with the most information related to the open problem
- favor the least time consuming approach
  - to keep the costs low
  - hard to determine what is time consuming in the long run since you are investigating an open problem
    - base your decision on what is time consuming in the short run

(cont.)

- favor the least intrusive approach
  - because often intrusive means effort
    - keep the effort and with this the costs low
  - because often intrusive means different behavior
    - keep as close to the original problem as possible
- favor the most flexible approach
  - since you are investigating an open problem
    - things might turn out differently from what you assumed before the test
    - you might need to run different (but similar) tests additionally

leads to

- start with profiling and monitoring tools/techniques related to your problem
  - because you do not have to do any changes in your software
    - little intrusive
    - little bring-up effort
  - provides a broad range of different information
    - relative flexible
- if this is not sufficient - collect trace information
  - more intrusive
  - but provides information that is otherwise not explicitly visible
    - example: change from cache hit to miss

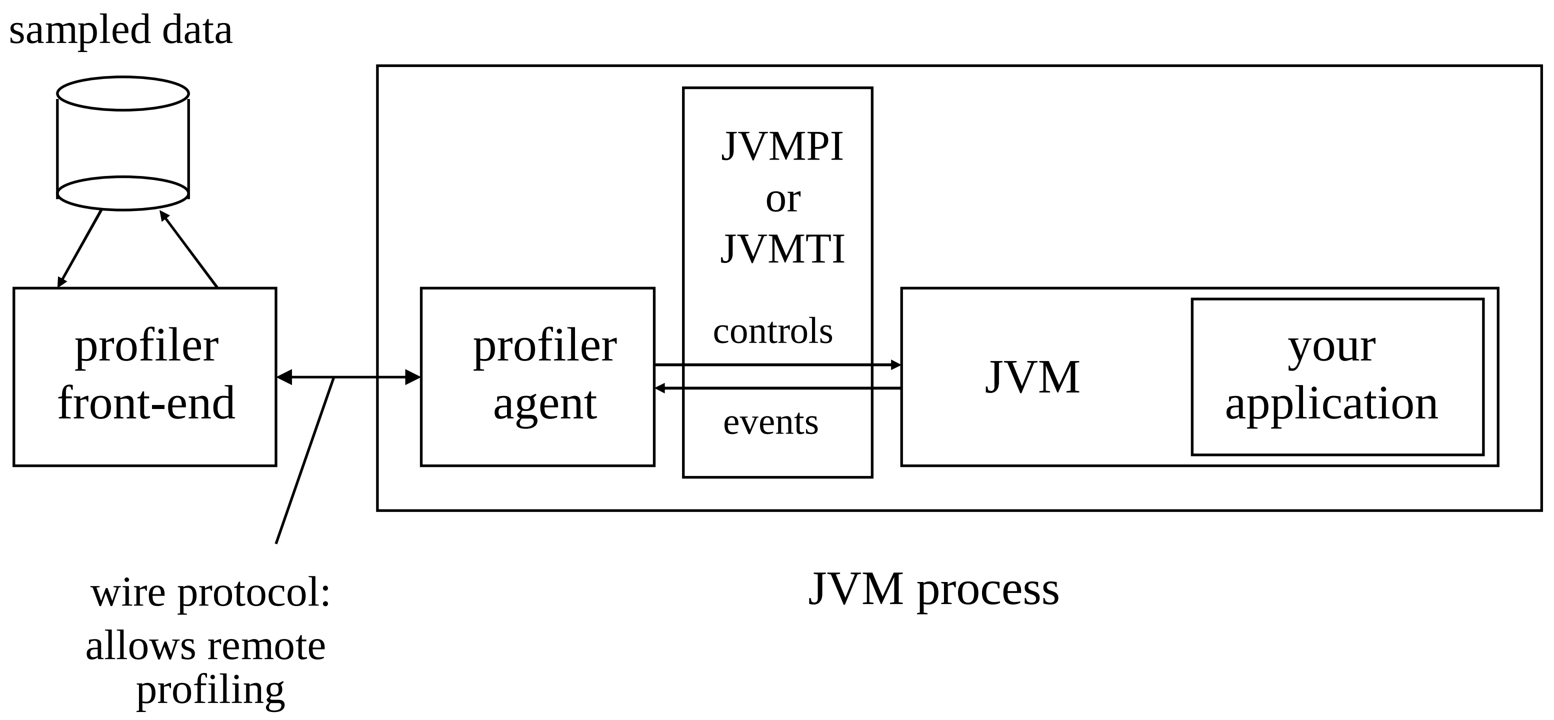
profiling and monitoring tools/techniques

- non-Java specific
  - system
    - e.g. xosview (Linux), Task Manager (Win)
    - information: CPU usage, memory usage / swapping, network traffic (per process and for the whole computer)
  - other technologies (e.g. database, EJB container, …)

- Java specific
  - profiler tools
    - commercial & freeware; based on JVMPI/JVMTI
  - JVM diagnostics
    - via JVM options
    - via programming API (in 5.0)

Java profiler tools

- use JMVPI/JVMTI interface



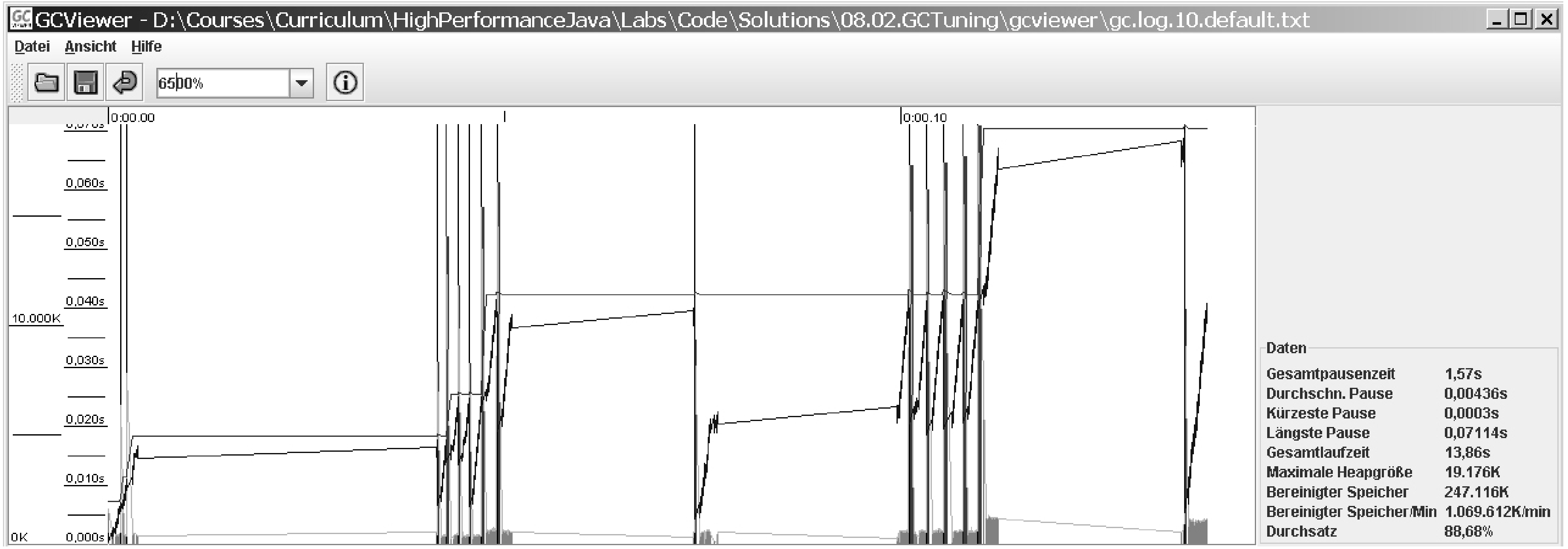
JVMPI / JVMTI functionality

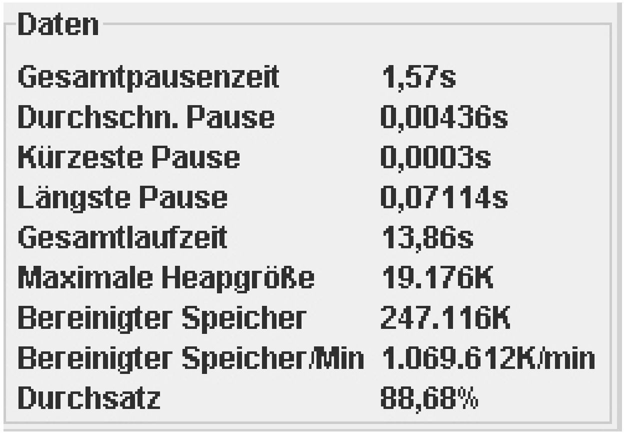
- functions that allow to enquire certain information
  - runtime information:
    - all loaded classes, all threads, current stack trace, field values, …
  - meta data:
    - fields and methods belonging to a class, …
  - other information:
    - time, thread specific CPU time, …
- callbacks that are triggered when certain events occur:
  - method entry, method exit,
  - frame pop, exception catch, field access,
  - thread start, thread end, monitor enter, monitor exit,
  - object allocation,
  - …

Java profiler tools

- time profiler
  - measures execution paths of an application on the method level
- space profiler
  - provide insight into development of the heap
  - such as which methods allocate most memory
- thread profiler
  - analyze thread synchronization issues

garbage collection (GCViewer - freeware)

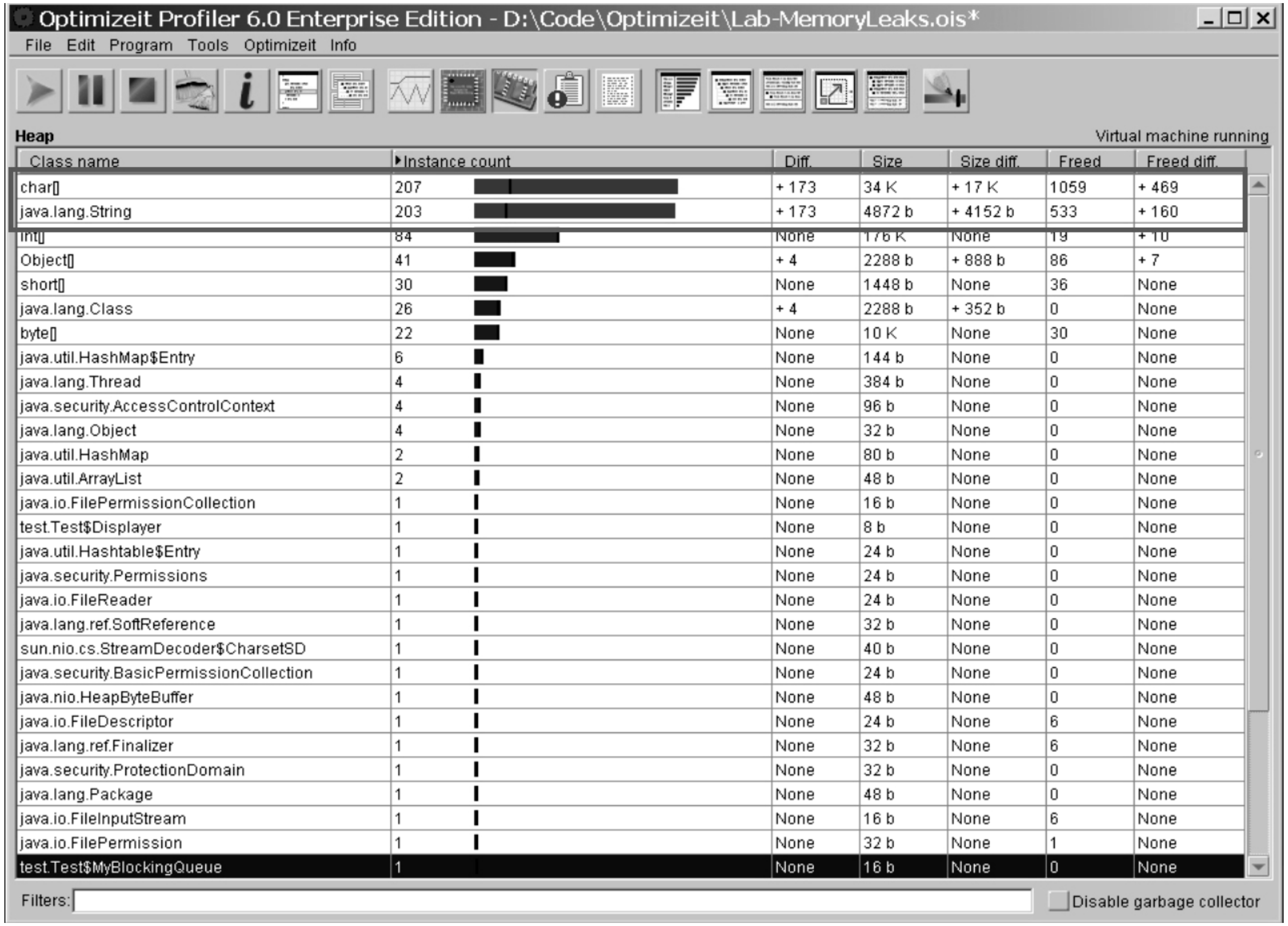

GCViewer - D:\Courses\Curriculum\HighPerformanceJava\Labs\Code\Solutions\08.02.GCTuning\gcviewer\gc.log.10.default.txt
Datei
Ansicht
Hilfe
6500%
0:00.00
0:00.10
0,060s
0,050s
0,040s
10.000K
0,030s
0,020s
0,010s
0K
0,000s



Daten

Gesamtpausenzeit	1,57s
Durchschn. Pause	0,00436s
Kürzeste Pause	0,0003s
Längste Pause	0,07114s
Gesamtlaufzeit	13,86s
Maximale Heapgröße	19.176K
Bereinigter Speicher	247.116K
Bereinigter Speicher/Min	1.069.612K/min
Durchsatz	88,68%

memory leaks (OptimizeIt - commercial)

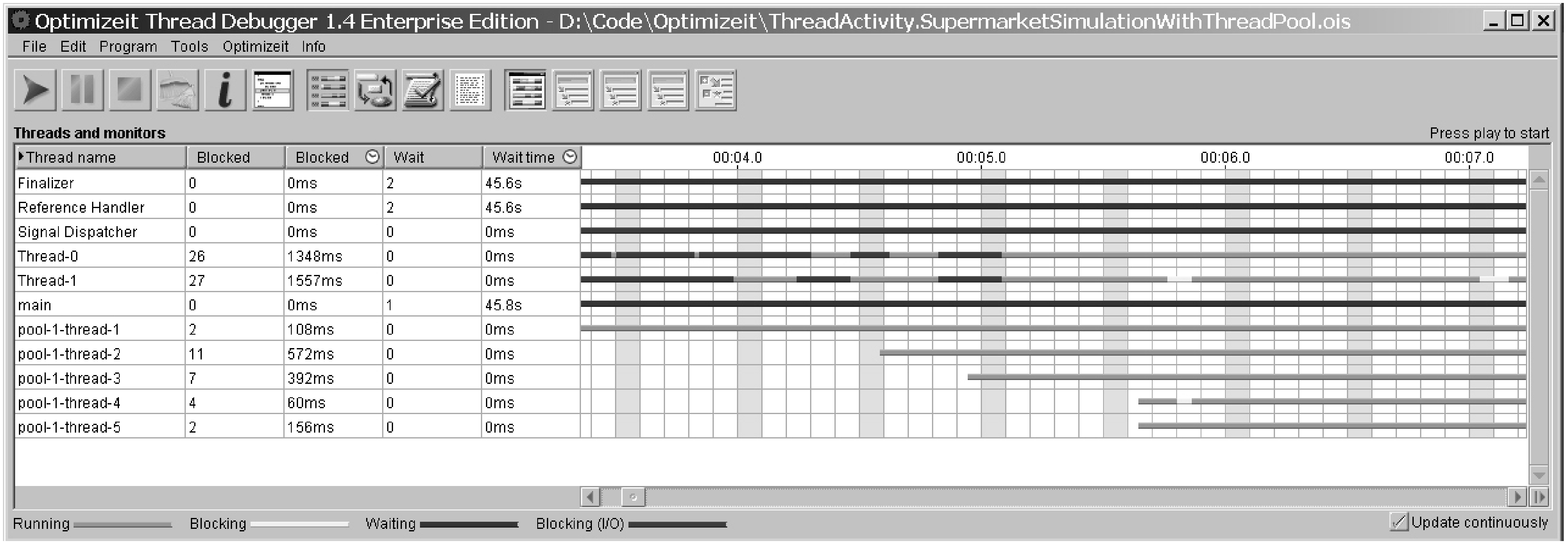

Optimizeit Profiler 6.0 Enterprise Edition - D:\Code\Optimizeit\Lab-MemoryLeaks.ois*
File Edit Program Tools Optimizeit Info
Filters:
Disable garbage collector

Heap

Virtual machine running

Class name	Instance count	Diff.	Size	Size diff.	Freed	Freed diff.
char[]	207	+ 173	34 K	+ 17 K	1059	+ 469
java.lang.String	203	+ 173	4872 b	+ 4152 b	533	+ 160
int[]	84	None	176 K	None	19	+ 10
Object[]	41	+ 4	2288 b	+ 888 b	86	+ 7
short[]	30	None	1448 b	None	36	None
java.lang.Class	26	+ 4	2288 b	+ 352 b	0	None
byte[]	22	None	10 K	None	30	None
java.util.HashMap$Entry	6	None	144 b	None	0	None
java.lang.Thread	4	None	384 b	None	0	None
java.security.AccessControlContext	4	None	96 b	None	0	None
java.lang.Object	4	None	32 b	None	0	None
java.util.HashMap	2	None	80 b	None	0	None
java.util.ArrayList	2	None	48 b	None	0	None
java.io.FilePermissionCollection	1	None	16 b	None	0	None
test.Test$Displayer	1	None	8 b	None	0	None
java.util.Hashtable$Entry	1	None	24 b	None	0	None
java.security.Permissions	1	None	24 b	None	0	None
java.io.FileReader	1	None	24 b	None	0	None
java.lang.ref.SoftReference	1	None	32 b	None	0	None
sun.nio.cs.StreamDecoder$CharsetSD	1	None	40 b	None	0	None
java.security.BasicPermissionCollection	1	None	24 b	None	0	None
java.nio.HeapByteBuffer	1	None	48 b	None	0	None
java.io.FileDescriptor	1	None	24 b	None	6	None
java.lang.ref.Finalizer	1	None	32 b	None	6	None
java.security.ProtectionDomain	1	None	32 b	None	0	None
java.lang.Package	1	None	48 b	None	0	None
java.io.FileInputStream	1	None	16 b	None	6	None
java.io.FilePermission	1	None	32 b	None	1	None
test.Test$MyBlockingQueue	1	None	16 b	None	0	None

thread behavior (OptimizeIt - commercial)



Optimizeit Thread Debugger 1.4 Enterprise Edition - D:\Code\Optimizeit\ThreadActivity.SupermarketSimulationWithThreadPool.ois

Thread name	Blocked	Blocked	Wait	Wait time
Finalizer	0	0ms	2	45.6s
Reference Handler	0	0ms	2	45.6s
Signal Dispatcher	0	0ms	0	0ms
Thread-0	26	1348ms	0	0ms
Thread-1	27	1557ms	0	0ms
main	0	0ms	1	45.8s
pool-1-thread-1	2	108ms	0	0ms
pool-1-thread-2	11	572ms	0	0ms
pool-1-thread-3	7	392ms	0	0ms
pool-1-thread-4	4	60ms	0	0ms
pool-1-thread-5	2	156ms	0	0ms

Running ▬ Blocking ▭ Waiting ▬ Blocking (I/O) ▬

✓ Update continuously

diagnostics via JVM options

- standard options
  - e.g. `-verbose:class|gc|jni`
- JVM-specific options
  - e.g. in SUN's JVM:

`-Xprof`	(profiling data)
`-XX:+PrintGCDetails`	(GC details)
`-XX:+PrintCompilation`	(HotSpot)
...	

JVM option `-verbose:GC`

- writes a GC trace to standard out:

```
[GC 2368K->590K(3520K), 0,0148500 secs]
[GC 2638K->879K(3520K), 0,0154419 secs]
[GC 2927K->1148K(3520K), 0,0116911 secs]
[GC 3196K->1365K(3520K), 0,0110821 secs]
[GC 3413K->1508K(3648K), 0,0071192 secs]
[Full GC 3556K->1652K(4672K), 0,1310929 secs]
[GC 3583K->1785K(4672K), 0,0233162 secs]
```

before GC

after GC

time spend for GC

minor or major ?

total heap size

JVM option `-XX:+PrintCompilation`

- writes HotSpot compilation info to standard out:
  - useful to detect distortions in benchmarks and determine warm-up phase

```
BENCHMARK RESULTS (for 10000000 loops):

  6   b   java.lang.String::indexOf (74 bytes)
  1% !b   instanceofbenchmark.Test::main @ 472 (942 bytes)
Duration instanceof:             101
  7*  b   java.lang.Class::isInstance (0 bytes)
Duration isInstance():          1462
  8   b   java.lang.String::equals (89 bytes)
Duration class name comparison: 5918
  9   b   java.lang.Object::equals (11 bytes)
Duration classes comparison:    1823
```

JVM diagnostics via programming API

- JMX = Java Management Extension
  - agent-based architecture for a programming API
  - a MBean (= management bean) exposes monitoring and management functionality of a device or service
  - included in JDK since 5.0
- beans for monitoring and managing the JVM
  - called MXBean (= standard platform MBean)

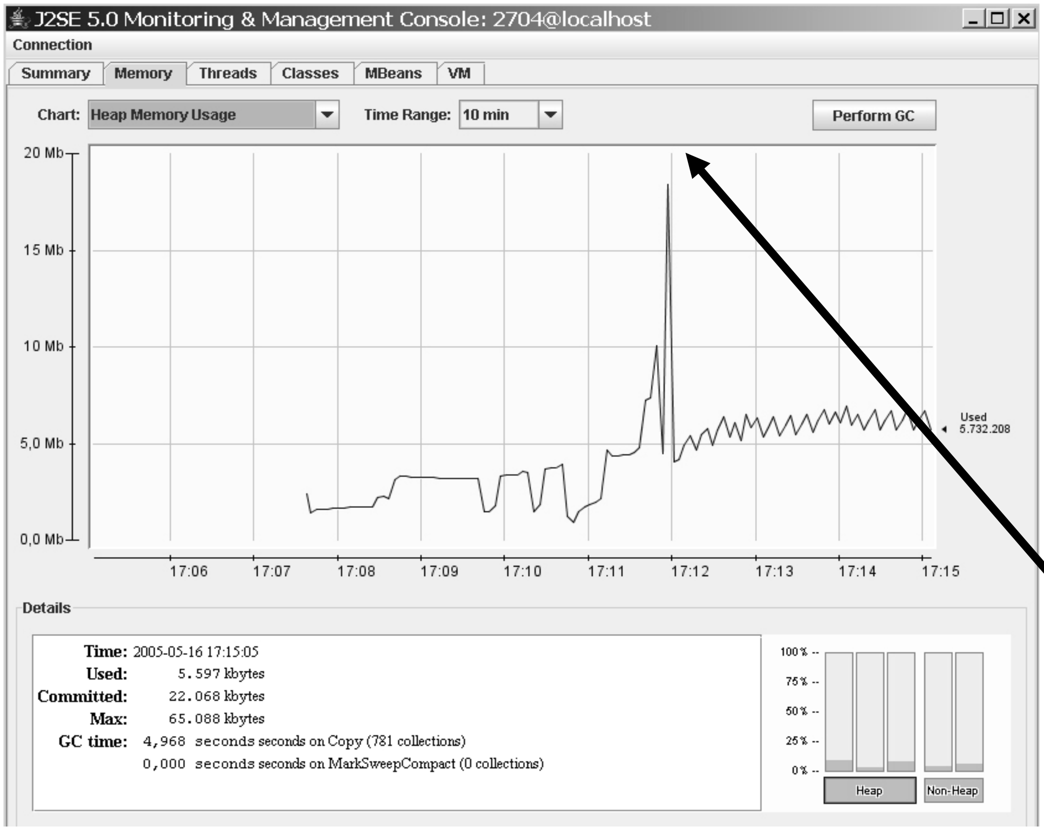
standard JVM beans (in 5.0)

- MXBean interfaces defined in `java.lang.management`:
  - `ClassLoadingMXBean`
  - `CompilationMXBean`
  - `GarbageCollectorMXBean`
  - `MemoryManagerMXBean`
  - `MemoryMXBean`
  - `MemoryPoolMXBean`
  - `OperatingSystemMXBean`
  - `RuntimeMXBean`
  - `ThreadMXBean`

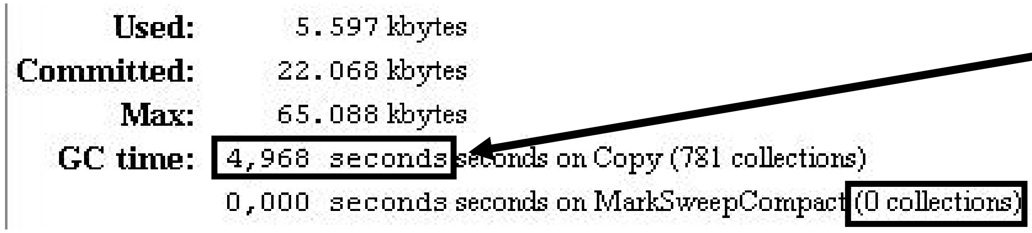
MXBeans

- allow programmatic access to JVM information
  - relatively intrusive
  - similar to tracing
- several "experimental" tools based on MXBeans
  - see JDK tool documentation
  - `jconsole`
    - simple console tool which allows to access the MXBeans
  - `jstat`
    - collects and logs statistics for a specified JVM
  - `jmap`
    - prints heap memory details
  - ...

heap monitoring using jconsole



```
      Time: 2005-05-16 17:15:05
      Used:      5.597 kbytes
 Committed:     22.068 kbytes
       Max:     65.088 kbytes
   GC time:  4,968  seconds seconds on Copy (781 collections)
             0,000  seconds seconds on MarkSweepCompact (0 collections)
```



```
      Used:      5.597 kbytes
 Committed:     22.068 kbytes
       Max:     65.088 kbytes
   GC time:  4,968  seconds seconds on Copy (781 collections)
             0,000  seconds seconds on MarkSweepCompact (0 collections)
```

- peak heap: ~20 MB
- GC time:
  - total: 4.968 ms
  - pauses: 781
  - per pause: 0.006 ms
- no full GC

peak heap size: ~20 MB

GC time: 4.968 ms

no full GC

heap monitoring using jconsole

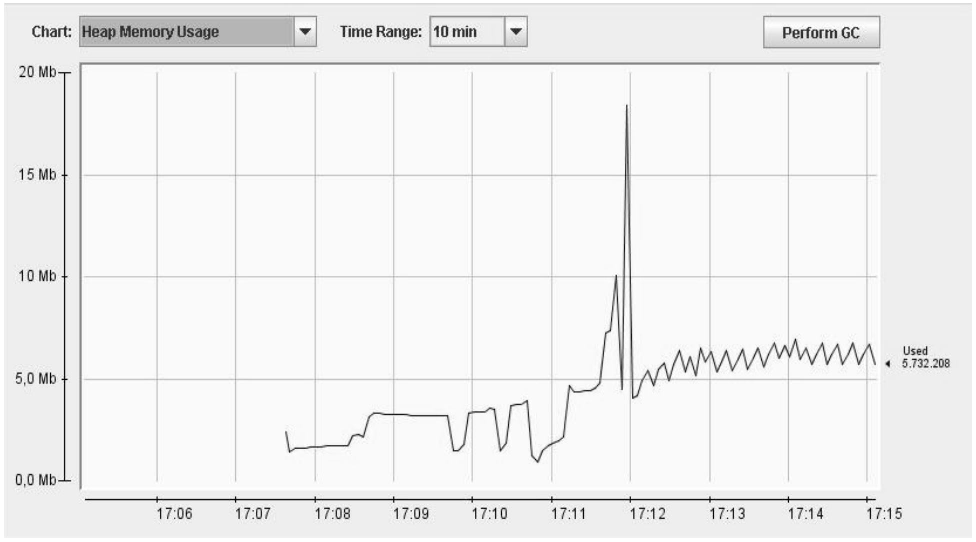


Chart: Heap Memory Usage Time Range: 10 min Perform GC

Used 5.732.208

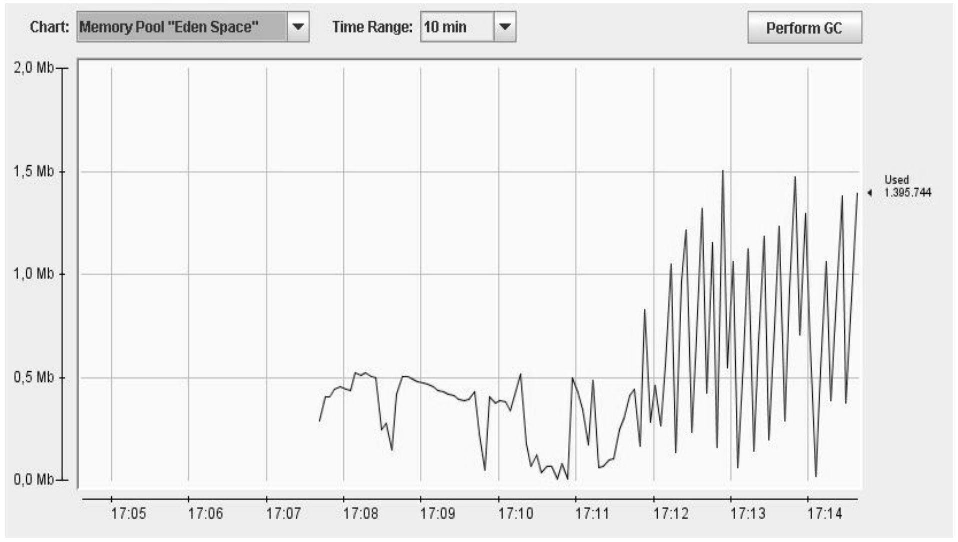


Chart: Memory Pool "Eden Space" Time Range: 10 min Perform GC

Used 1.395.744

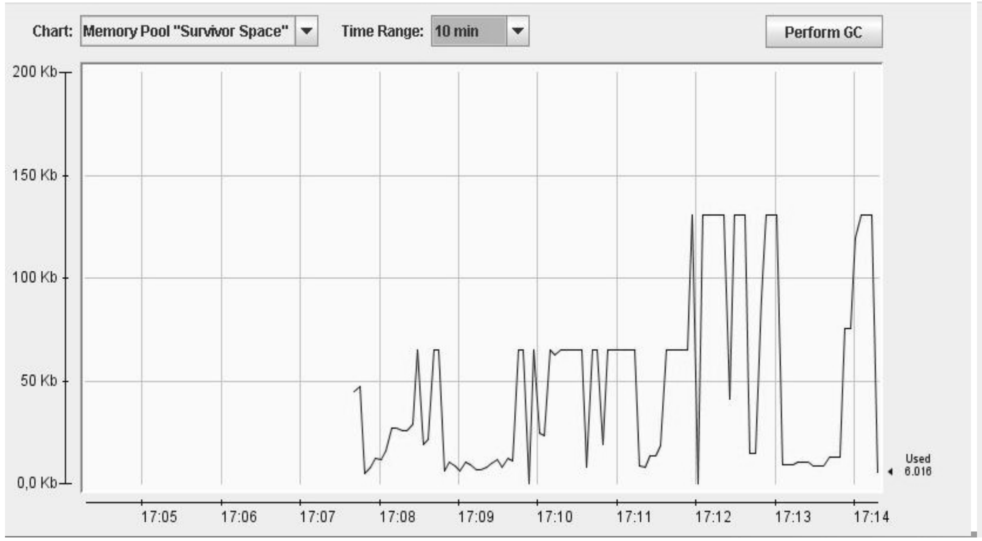


Chart: Memory Pool "Survivor Space" Time Range: 10 min Perform GC

Used 6.016

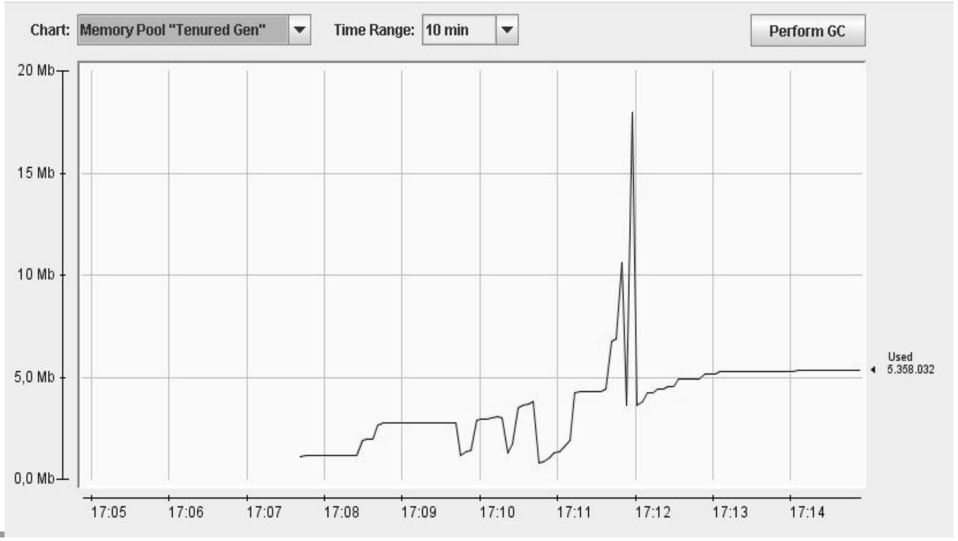


Chart: Memory Pool "Tenured Gen" Time Range: 10 min Perform GC

Used 5.358.032

tracing

- traces
  - indication + timestamps
  - more intrusive
  - either part of your software or ad hoc
  - indication of the situation that is otherwise not explicitly visible (example)
  - correlation of situations (in different process/JVMs, on different machines, …)

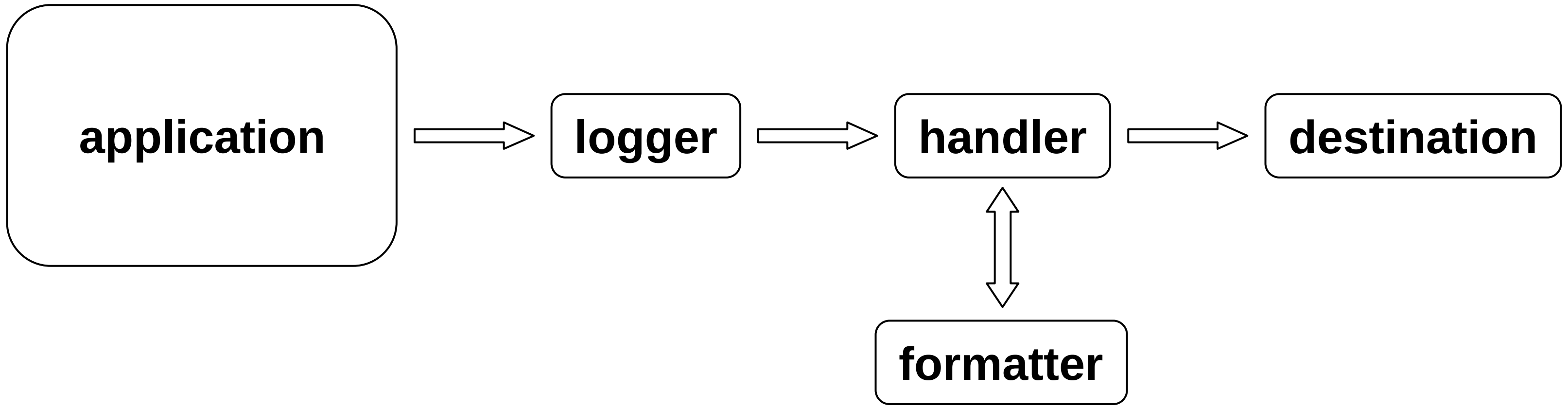
tracing techniques

- add trace output to your program
  - when profiling and monitoring does not suffice and application-specific information is needed

- on an ad-hoc basis
  - insert `System.out.println` where needed
- on a systematic basis
  - using a logging API
  - such as log4j (part of Apache) or JUL (=`java.util.logging`)
    - Log4j and JUL are almost conceptually identical
    - "Whatever JUL can do, Log4j can also do - and more."

Java logging (JUL)

- Java SDK supplies logging API (since 1.4)
  - loosely coupled components allow great flexibility



Java logging (JUL) - example

```
                          void basics()
                          {
logical log name ——→        Logger l = Logger.getLogger("com.develop.test");

output destination ——→      FileHandler h = new FileHandler("MyLog.txt");

output format ——→           SimpleFormatter f = new SimpleFormatter();

                            h.setFormatter(f);

                            l.addHandler(h);

write to log ——→            l.log(Level.WARNING, "just a warning");
                          }
```

`MyLog.txt` contents ——→

```
Sep 9, 2002 1:22:53 AM LogDemo basics
WARNING: just a warning
```

wrap-up

- Java does not have a simple performance model.
  - statements incur non-obvious performance complexities
  - numerous invisible optimizations
- Benchmarking is non-trivial.
  - HotSpot technology make performance characteristics rather unpredictable
- Performance related information is hard to find.
  - often you are on your own for investigation the root of a problem
- Various tools and techniques are available.
  - OS specific tools
  - Java profiling and monitoring
  - logging APIs

references - Richard Gabriel

Richard P. Gabriel published his thoughts about a successful programming language

first in the article:

`“The End of History and the Last Programming Language”, JOOP, July 1993, page 90-94`

and later in his book:

`“Patterns of Software: Tales from the Software Community”, Oxford University Press (ISBN 0195121236), page 111-122`

the book is also available from the internet

`http://www.dreamsongs.com/NewFiles/PatternsOfSoftware.pdf`

references

the SoftReference bug:

`http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4239645`

information on tools:

`http://www.AngelikaLanger.com/Conferences/PerfModelReferences.htm`

authors

Angelika Langer

Training & Mentoring

Object-Oriented Software Development in C++ & Java

Email: **contact@AngelikaLanger.com**
http: **www.AngelikaLanger.com**

Klaus Kreft

Siemens Enterprise Communications Gmbh & Co. KG, Munich, Germany

Email: **klaus.kreft@siemens.com**