Ausgabe 02/2020

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 02

Java aktuell

iJUG Verbund
www.ijug.eu

High Performance
Ein Blick hinter die Kulissen von Java

Immutability
Vorteile von unveränderlichen Datenstrukturen

Zeitreise
Die Highlights der Java-Versionen 8 bis 13 im Überblick

Das Herz des Java-Universums

DW *Made for minds.*

Coding for freedom.

Mehr Berufung als Beruf:
Dein IT-Job bei der DW.

JETZT BEWERBEN!
dw.com/it-karriere

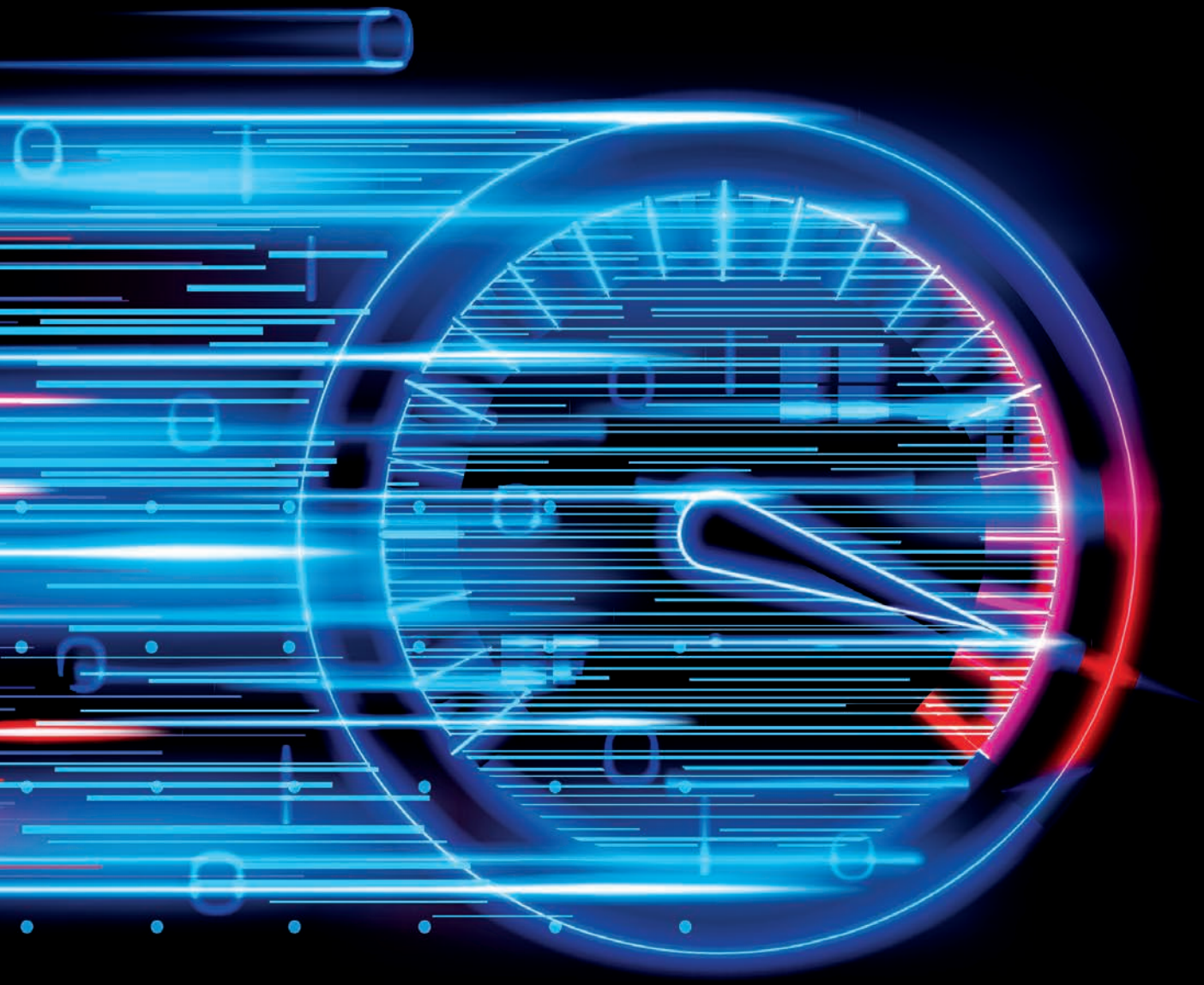
Du entwickelst passgenau? Dann fordere uns auf der **JavaLand** am Klask heraus. Du willst was footern? Komm mit uns bei einer Tüte Popcorn ins Gespräch.

Du findest uns auf der **JavaLand** direkt gegenüber vom Haupteingang.

High Performance Java – Hinter den Kulissen von Java

René Schwietzke, Xceptance GmbH

Die Zeiten, in denen Java der langsame Bytecode-Interpreter war, sind lange vorbei. Die JVM nutzt viele Tricks, um Code effizient auszuführen, und transformiert und optimiert Java-Code auf die ausführende Hardware. Mit etwas Wissen über diese Prozesse kann man vermeiden, dass man gegen die JVM arbeitet, und gleichzeitig mehr Geschwindigkeit erreichen. Selbst wenn man nicht die letzte Mikrosekunde jagt, ist es interessant zu sehen, welche Techniken die JVM einsetzt, um die Ausführungsumgebung besser zu verstehen.



Im Rahmen von Performancetests wurde beobachtet, dass viele Java-Entwickler bei Profiling und Debugging eher naiv an Probleme herangehen und oft 50 Prozent bessere Performance erwarten und auch lokal messen, aber am Ende in Produktion keinerlei Unterschiede existieren. Häufig ist das Grundverständnis für die JVM-Laufzeitumgebung nicht ausgeprägt. Zusätzlich ist das Wissen über CPU, Speicher und Hardware im Allgemeinen gering, da moderne Programmiersprachen dies nicht mehr erfordern. Wenn man sich die Effizienzbetrachtungen zu Programmiersprachen *[1]* anschaut, sieht man, dass Java gleich hinter den klassischen kompilierten Sprachen wie C in der Effizienz steht. Wie schafft Java das, obwohl es nicht plattformspezifisch kompiliert wird?

Lernen durch Beobachten

An einem einfachen Beispiel kann man gut erkennen, wie Java mit Code umgeht. Zuerst der Hinweis, dass man natürlich jeden Microbenchmark mit dem Java Measurement Harness (JMH) *[2]* durchführen sollte, das Beispiel in *Listing 1* macht es allerdings bewusst anders.

In *Listing 1* wird eine Methode *append(int)* durch eine einfache Klammer um den Methodenaufruf vermessen (vollständiger Code: *[3]*). Bei Ausführung erhält man Messwerte, die starke Laufzeitunterschiede zwischen der ersten und der hunderttausendsten Ausführung der Methode für verschiedene Laufzeitumgebungen zeigen *(siehe Tabelle 1)*.

```
public class PoorMansBenchmark
{
    // just burn time
    private static String append(final int i)
    {
        final StringBuilder sb = new StringBuilder();
        sb.append(System.currentTimeMillis());
        sb.append(i);
        sb.append("lkJAHJ AHSDJHF KJASHF HSAKJFD");

        char[] ch = sb.toString().toCharArray();
        Arrays.sort(ch);

        return new String(ch);
    }

    public static void main (final String[] args)
    {
        final int SIZE = 100_000;
        final long[] result = new long[SIZE];

        int sum = 0;
        for (int i = 0; i < SIZE; i++)
        {
            final Timer t = Timer.startTimer();

            sum += append(i).length();

            result[i] = t.stop().runtimeNanos();
        }

        // pretend we need it but we really don't
        // just to avoid optimizations (spoilers, duh!)
        if (sum < 0)
        {
            System.out.println(sum);
        }

        // more code here to output the measurements
    }
}
```

Listing 1: Ein naiver Benchmark – org/sample/PoorMansbenchmark.java

Der Java-Skeptiker wird natürlich sofort die Garbage Collection (GC) als Grund ausmachen, während der C-Fan auf den Interpreter zeigen wird. Um diese Skepsis teilweise zu beseitigen, schließt Spalte 2 „NoOp" die GC mit dem Epsilon GC aus *[10]* und für Spalte 3 „Graal AOT" wurde das Programm mit der GraalVM nativ kompiliert. Ein kurzer Blick auf *Tabelle 1* zeigt, dass der Java-Code schneller wird, je länger er läuft. Die GC hat keinen Einfluss und der Code läuft sogar etwas langsamer. Der nativ kompilierte Code via Graal AOT ist anfangs schneller, wird aber am Ende von der Java Virtual Machine (JVM) trotzdem unterboten. Schaut man sich die Spalte 1 „G1" im Chart an, sieht man, wie stark die Laufzeiten schwanken *(siehe Abbildung 1)*.

Man sieht, dass die Laufzeit diskontinuierlich sinkt, gelegentlich aber sogar steigt. Java kompiliert und optimiert den Code wiederholt, bis ein optimales Laufzeitverhalten vorliegt. Dies führt zu den drei Teilthemen für diesen Artikel:

- Compiler
- Javas Trickkiste
- CPU, Memory und Cache

Compiler

Bei dem Begriff „Java Compiler" fällt vielen nur *javac* ein. Dieser ist allerdings nur einer von drei Compilern im Standard OpenJDK mit Hotspot. Hotspot ist der Just-in-time-Compiler (JIT), der wiederum aus zwei Compilern (C1 und C2) besteht und vier verschiedene Möglichkeiten besitzt, Code zu kompilieren. Bei dieser Übersetzung handelt es sich auch um die Transformation von Bytecode zu Maschinencode, in diesem Fall erfolgt diese jedoch erst während der Laufzeit des Programms.

javac übersetzt den Java-Code in plattformunabhängigen Bytecode. Wichtig ist, dass javac nur minimale Optimierungen vornimmt, und zwar nur, wenn Ergebnisse bereits zur Compile-Zeit berechenbar sind. Hierzu zählen Arithmetik, String-Operationen und Konstanten.

C1 und C2 übersetzen nur ganze Methoden und Schleifen (for, while), abhängig von der Häufigkeit der Verwendung. Je öfter Code ausgeführt wird, desto „heißer" und damit wahrscheinlicher ist eine Übersetzung in Maschinencode. Die Übersetzung kostet Zeit, deswegen wird nicht pauschal alles übersetzt. Man kann Java zwingen, mehr zu übersetzen, das ist jedoch ein eigener Forschungsgegenstand, da immer eine Kosten-Nutzen-Betrachtung erforderlich ist.

Wie bereits erwähnt, haben C1 und C2 vier mögliche Übersetzungsergebnisse. Ein Blick in den Sourcecode des JDK (in src/share/vm/runtime/advancedThresholdPolicy.hpp) erklärt, welche Varianten existieren:

Durchlauf	1 - G1	2 - NoOp	3 - Graal AOT
1	595.433 ns	549.924 ns	8.791 ns
100	40.456 ns	34.017 ns	1.989 ns
1.000	6.923 ns	7.365 ns	1.620 ns
10.000	2.743 ns	2.738 ns	1.309 ns
100.000	755 ns	990 ns	1.278 ns

Tabelle 1: Vergleich der unterschiedlichen Umgebungen

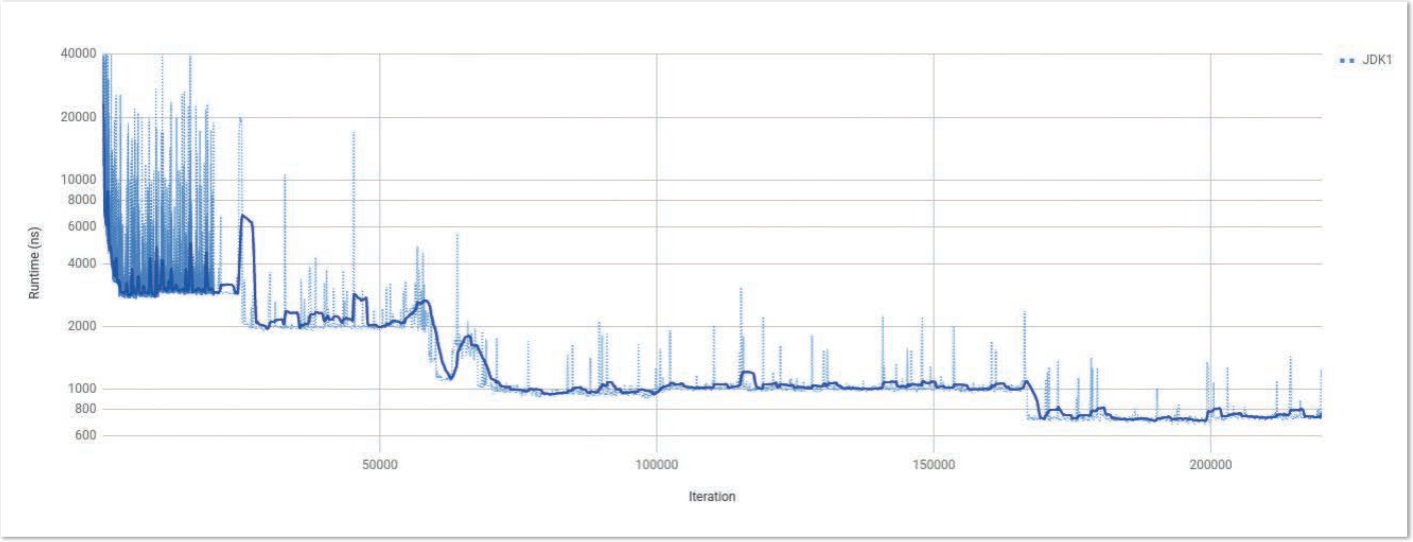


Abbildung 1: Laufzeitentwicklung von append(int) (© René Schwietzke, [4])

- Level 0: Interpreter
- Level 1: C1 with full optimization (no profiling)
- Level 2: C1 with invocation and back-edge counters
- Level 3: C1 with full profiling (level 2 + MDO)
- Level 4: C2 fully optimized

Während Level 0 lediglich Bytecode interpretiert, produziert C1 Code, der mit verschiedenen Instrumentierungen versehen wird, um messen zu können, ob der gebaute Code effizient ist oder weiter verfeinert werden kann.

Im Idealfall springt die JVM von Level 0 nach 3 und nach einer weiteren Beobachtung wird finaler Code mit C2 optimiert produziert. Aber natürlich ist die Welt nicht so einfach, deswegen kann es sein, dass C2 gerade beschäftigt ist und der C1 zunächst Level-2-Code produziert, bevor Level 3 und 4 genutzt werden können. Eventuell ist der Code so trivial, dass Level 1 ausreichend ist. Darum erklärt sich auch, warum Code schneller wird, je länger er läuft. Die JVM beobachtet kontinuierlich die Performance und kompiliert den Code gegebenenfalls nochmals mithilfe der gewonnenen Informationen.

Im Zusammenhang mit dem Wissen aus dem folgenden Kapitel kann es aber zu unerwarteten Seiteneffekten kommen, die den Compiler zwingen, seine Code-Optimierungen zurückzunehmen.

Javas Trickkiste

Nachfolgend einige Beispiele für Optimierungen, die die JVM während der Programmausführung durchführt.

Dead Code Elimination

Das folgende Beispiel von Douglas Hawkins zeigt, wie die JVM Laufzeitinformationen nutzt, um überflüssigen Code zu entfernen *(siehe Listing 2)*. Bei der Ausführung entsteht folgendes Bild *(siehe Abbildung 2)*.

Die Laufzeit ist zunächst hoch, verbessert sich dramatisch mit jeder Iteration, bis zur Iteration 400, wo ein einziges Mal die Variable `trap` nicht `null` ist. In diesem Moment verwirft der Compiler den bisherigen Code. Kurz wird der Interpreter wieder benutzt, bis der Compiler neuen nativen Code verfügbar hat. Dieser erreicht nicht wieder das vorherige Laufzeitniveau.

```
public static void main(String[] args)
{
    Object trap = null;
    Object o = null;

    for (int i = 0; i < 1_000; i++)
    {
        final Timer t = Timer.startTimer();

        for (int j = 0; j < 1_000; j++)
        {
            // burn time and train that null is normal
            o = new Object();

            if (trap != null)
            {
                System.out.println("Got you." + o);
                trap = null;
            }
        }

        // Give me a Non-Null, Vasily.
        // One Non-Null only, please.
        if (i == 400)
        {
            trap = new Object();
        }

        System.out.println(
            MessageFormat.format("{1} {0, number, #}",
            t.stop().runtimeNanos(), i));
    }
}
```

Listing 2: AllocationTrap.java

Der Compiler hat über die Zeit erkannt, dass `trap` immer `null` ist, und entschieden, ab circa 200 Iterationen das `IF` zu streichen. Er verlässt sich stattdessen auf das Betriebssystem und riskiert einen Segmentation Fault, um sich benachrichtigen zu lassen, falls `trap` doch einmal einen anderen Wert annimmt. Das ist die übliche Vorgehensweise in Java, weil sonst jeder Objekt-Zugriff mit einem `if != null else NPE` umgeben werden müsste. Die Iteration 400 zeigt dem Compiler jedoch, dass die der Optimierung zugrunde liegende Annahme falsch war, sodass die aggressive Optimierung zurückgenommen wird.

Bei der Optimierung des eigenen Codes ist daher wichtig, dass Ausnahmen dieser Art nicht vorkommen oder früher behandelt werden.

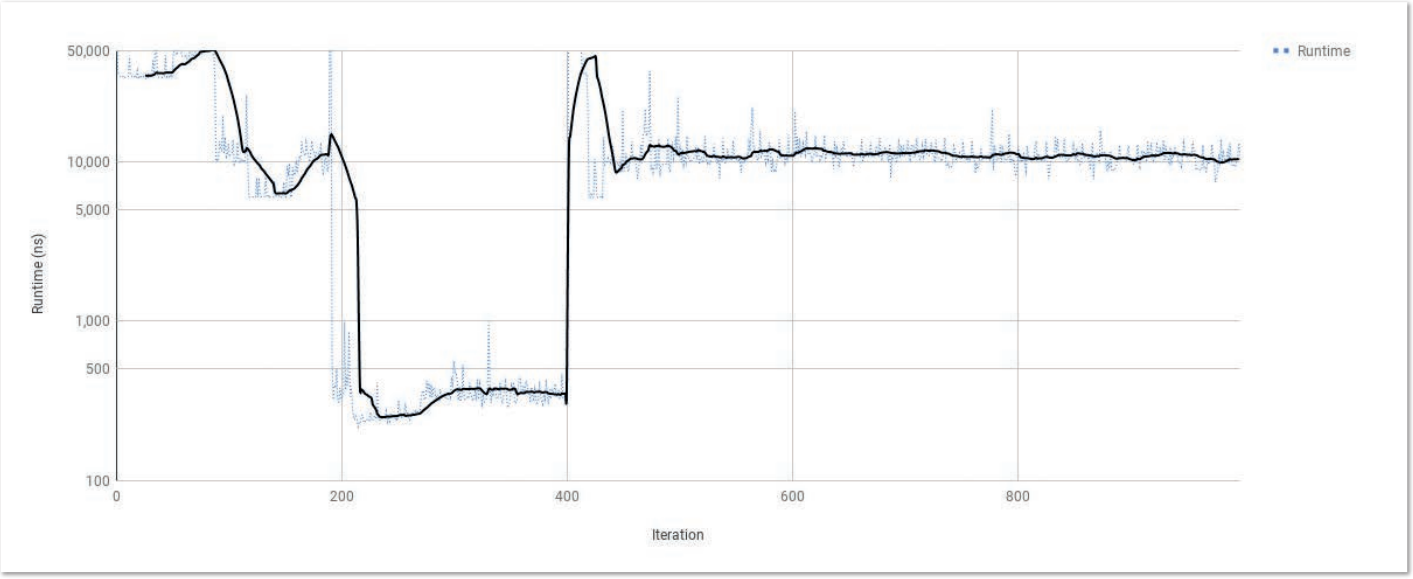


Abbildung 2: AllocationTrap.java Laufzeiten (© René Schwietzke, [4])

Oft sind dazu algorithmische Änderungen nötig, aber so kann man das stark optimierte Laufzeitverhalten erhalten.

Loop Unrolling

Dieses Beispiel zeigt, wie Java Schleifen durch Zerlegen optimiert. Dabei reduziert der Compiler die Anzahl der Schleifendurchläufe und baut dafür die in der Schleife auszuführenden Anweisungen mehrfach in den Anweisungsblock des Schleifenkörpers ein. So kann die JVM Bereichsüberprüfungen und Sprünge einsparen *(siehe Listing 3)*. Die Laufzeiten zeigen, dass die Methode `variable()` etwas mehr als 30 Prozent langsamer ist. Die Gegenprobe mit dem Interpreter beweist, dass der Gesamtcode gleich ist, wenn der Compiler nicht optimiert *(siehe Listing 4)*.

```
private int next()
{
    int i = r.nextInt(1) + 1;
    return i;
}

@Benchmark
public int classic()
{
    int sum = 0;
    int step = next();
    for (int i = 0; i < ints.length; i = i + 1)
    {
        sum += ints[i];
        step = next();
    }
    return sum + step;
}

@Benchmark
public int variable()
{
    int sum = 0;
    int step = next();
    for (int i = 0; i < ints.length; i = i + step)
    {
        sum += ints[i];
        step = next();
    }
    return sum + step;
}
```

Listing 3: Optimizing Loops Example (LoopUnroll.java)

Der JIT-Compiler nutzt hier die Information der konstanten Laufweite von 1 und baut den Code entsprechend um. Für den interessierten Nutzer ist ein Blick in den Code von Hotspot (src/share/vm/opto/loopTransform.cpp) beziehungsweise der GraalVM (org/graalvm/compiler/loop/phases/LoopTransformations.java) zu empfehlen. Die VM zerlegt die Schleife übrigens noch weiter, inklusive der Möglichkeit der Auto-Vectorization von Schleifen (siehe *[12]*).

Inlining und Virtual Methods

Methodenaufrufe stellen Overhead dar, da Parameter bestimmt und übergeben werden müssen und aus Sicht der CPU-Caches auch nicht optimal genutzt werden. Viele Methoden, gerade Getter und Setter sowie statische Hilfsmethoden, werden häufig aufgerufen. Aus diesem Grund bettet der JIT-Compiler Methodencode direkt ein. Er kopiert den Anweisungsblock des Aufrufziels (callee) in die Quelle ein (callsite). Diese Optimierung erfolgt je nach Aufruffrequenz und Größe der Zielmethode. Gleichermaßen können mehrfach verschachtelte Inlines erzeugt werden. Große Methoden werden nicht eingebettet, ebenso beschränkt sich Inlining auf Methoden am Ende des Callstack bis zu einer bestimmten Tiefe.

Da viele Methoden in Java virtuelle Methoden sind, deren Auflösung zur Laufzeit erfolgt, wendet Java hier ähnliche Optimierungen wie bei Dead Code an, um festzustellen, wie viele Varianten einer Methode es gibt. Sollte für lange Zeit keine weitere Variation der Methode existieren, dann wird der Code direkt eingebettet. Sollte der Code in mehr als den bisher bekannten Variationen auftreten, dann kann die JVM auch den Code wieder de-optimieren.

```
Benchmark            (size)  Mode  Cnt       Score  Units
LoopUnroll.classic    10000  avgt    2      18,871  ns/op
LoopUnroll.variable   10000  avgt    2      27,433  ns/op

# Verification of identical behavior with -Xint
Benchmark            (size)  Mode Cnt         Score  Units
LoopUnroll.classic    10000  avgt   2     2,650,812  ns/op
LoopUnroll.variable   10000  avgt   2     2,449,845  ns/op
```

Listing 4: Laufzeiten LoopUnroll.java

```
# -XX:+PrintCompilation
@ 54   java.lang.Math::min (11 bytes)   (intrinsic)
@ 57   java.lang.System::arraycopy (0 bytes)   (intrinsic)
```

Listing 5: Compiler-Debug-Output

Es gibt allerdings noch Grenzfälle, wenn es zwei, drei oder mehrere mögliche Methoden gibt. Normalerweise muss hier die Methode erst über eine Virtual Method Table (VMT) aufgelöst werden. Um diesen Overhead zu reduzieren, bettet Java bei einer Variante den Code direkt ein – bei zwei Varianten über ein einfaches `IF`, das via Branch Prediction optimiert werden kann. Sollte es drei oder mehr Varianten der Methode geben, dann geht Java den klassischen Weg über die VMT. Mehr zum Thema kann man bei Aleksey Shipilëv lesen *[11]*.

Intrinsics

Intrinsics sind optimierter und mitgelieferter Code, auf den Java zurückgreift, um Code schneller zu kompilieren. Wenn die Plattform den bereitliegenden Code unterstützt, dann wird der Intrinsics-Native-Code direkt an die notwendige Stelle kopiert, ohne dass Methoden-Aufrufe oder Native-Calls erfolgen. Intrinsics sind im Java-Quellcode nicht erkennbar. Einige Stellen mit nativem Code (Native-Keyword) werden genauso ersetzt wie purer Java-Code, zum Beispiel `Math.min()`.

Beim Start der JVM wird die Umgebung einschließlich der CPU ausgewertet, um zu entscheiden, welche Library-Methoden direkt ersetzt werden. Einzig der Debug-Output des Compilers gibt Aufschluss darüber *(siehe Listing 5)*.

Escape Analysis

Mit Blick auf CPU und Speicher wird klar, dass die Verwendung des lokalen Stacks effizienter ist als die Nutzung von Heap im Hauptspeicher. Auf dem Stack ist keine GC nötig. Die JVM versucht daher zu vermeiden, dass der Heap benutzt wird, und arbeitet mit CPU-Registern und Stack. Dabei besteht das Problem jedoch darin, zu erkennen, wann ein Objekt aus einer Methode entkommt, also von anderen Objekten auf dem Heap referenziert wird. Ein Escape Analysis genanntes Verfahren versucht, die Gültigkeit von Objekten zu erkennen. *Listing 6* zeigt ein einfaches Beispiel.

Die Methode `array64()` läuft circa 22 ns pro Aufruf, während `array65()` circa 42 ns läuft. Java erkennt, dass das Array „a" nicht außerhalb der Methode genutzt wird, also weder zurückgegeben noch über globale Objekte referenziert wird. Mit der „-prof gc"-Option des JMH sieht man, dass der Heap für `array64()` nicht genutzt wird *(siehe Listing 7)*. Die JVM besitzt ein Array-Limit für lokale Allokationen, das man mit `-XX:EliminateAllocationArraySizeLimit` anpassen kann.

CPU, Memory und Cache

Bei eigenen Optimierungsversuchen hat man sich gegebenenfalls schon gefragt, warum das rechenintensive, wiederholte Parsen ei-

```
@Benchmark
public long array64()
{
    int[] a = new int[64];

    a[0] = r.nextInt();
    a[1] = r.nextInt();

    return a[0] + a[1];
}
@Benchmark
public long array65()
{
    int[] a = new int[65];

    a[0] = r.nextInt();
    a[1] = r.nextInt();

    return a[0] + a[1];
}
```

Listing 6: Lokale Objekte – org/sample/EscapeAnalysis.java

```
array64                          avgt  2    22.804 ns/op
array64:·gc.alloc.rate           avgt  2    ≈ 10⁻⁴ MB/sec
array64:·gc.alloc.rate.norm      avgt  2    ≈ 10⁻⁶ B/op
array64:·gc.count                avgt  2      ≈ 0 counts

array65                          avgt  2    41.890 ns/op
array65:·gc.alloc.rate           avgt  2  5795.571 MB/sec
array65:·gc.alloc.rate.norm      avgt  2   280.000  B/op
array65:·gc.count                avgt  2    93.000 counts
```

Listing 7: GC Profile für Listing 6

```
final int SIZE = 1_000_000;
final int[] src = new int[SIZE];

@Benchmark
public int step1()
{
    int sum = 0;
    for (int i = 0; i < SIZE; i++)
    {
        sum += src[i];
    }

    return sum;
}

@Benchmark
public int step20()
{
    int sum = 0;
    for (int i = 0; i < SIZE; i = i + 20)
    {
        sum += src[i];
    }

    return sum;
}
```

Listing 8: Hauptspeicherzugriff (org.sample.ArraysAndHardware.java)

```
Benchmark  Mode  Cnt    Score  Units
step1      avgt    2  325,159  ns/op
step20     avgt    2   97,402  ns/op
```

Listing 9: Benchmark-Ergebnisse ArraysAndHardware

nes Strings schneller ist, als das Ergebnis zu cachen. Der verringerte CPU-Bedarf müsste doch zu einer kürzeren Laufzeit führen, oder?

In modernen Architekturen ist die CPU um ein Vielfaches schneller als der Speicher. Um Daten aus dem Hauptspeicher zu bekommen, muss die CPU im ungünstigsten Fall bis zu 200 Zyklen warten. Bekommt sie die Daten aus einem L1- bis L3-Cache, dann dauert es zwei bis 60 Zyklen. Wenn man jetzt bedenkt, dass eine moderne CPU pro Zyklus (und das bedeutet nicht Multi-Core!) eine Vielzahl von parallelen Execution-Units nutzt (siehe *[6]*), sodass circa drei bis vier Microbefehle gleichzeitig ausgeführt werden können (solange es keine Datenabhängigkeit gibt), verliert man bei einem einzigen Hauptspeicherzugriff 600 bis 800 mögliche ausgeführte Instruktionen.

Für alle modernen Rechner gilt, dass der Speicherzugriff teuer, aber CPU-Zyklen günstig und reichlich vorhanden sind. Auch diese Tatsache lässt sich für die Optimierung nutzen. Das Beispiel in *Listing 8* zeigt das auf einfache Art. Es summiert die Inhalte eines Arrays. Dabei greift die Methode `step20()` nur auf jedes 20. Element zu. Rein theoretisch sollte also `step20()` etwa 20 Mal schneller sein als `step1()` *(siehe Listing 9)*.

`Step20()` ist jedoch nur etwas mehr als drei Mal schneller, obwohl 20 Mal weniger Speicherzugriffe und Additionen durchführt werden. Ein Blick auf die Hardware-Statistiken erklärt das Problem *(siehe Listing 10)*.

Es ist zu sehen, dass `step1` die CPU besser auslastet (`insn per cycle`), auch wenn die Auslastung vom Optimum (`3.5 insn per cycle`) weit entfernt ist. Nur 6,25 Prozent aller L1-Cache-Zugriffe treffen nicht (`L1-dcache-load-miss`). `Step20()` greift eigentlich immer am L1-Cache vorbei und löst mehr LLC-Zugriffe (Low Level Cache, L2/L3) aus. Ein L1-Hit kostet circa zwei Zyklen, ein L2/L3-Miss ungefähr 20 bis 60 Zyklen. Dieses einfache Beispiel zeigt, dass sich Code nicht immer so verhält, wie man es erwartet, wenn man die Laufzeitumgebung inklusive der CPU und des Speichers ignoriert.

Branch Prediction and Speculative Execution

Wenn man über CPUs redet, sind zwei Features wichtig: Branch Prediction und Speculative Execution – Fähigkeiten der CPU, Code im Voraus auszuführen, um die Pipelines und Execution Units besser auszulasten. Java selbst wendet ebenfalls Branch Prediction an, um Sprünge im Programmcode zu reduzieren und Verzweigungen umzusortieren. *Listing 11* zeigt ein Beispiel, wo die Branch Prediction von Java und der CPU zum Tragen kommen.

Der Test läuft über ein Array mit Zufallszahlen. Abhängig vom Vorzeichen wird Zweig A oder B durchlaufen. Sind die Zufallszahlen sortiert im Array vorhanden, läuft der Code nahezu doppelt so schnell *(siehe Listing 12)*.

Wenn man die technischen Informationen vergleicht, wird klar, dass ein unsortiertes Array mehr Branch-Prediction-Misses hat und damit langsamer läuft. Die Gesamtzahl der Instruktionen ist gleich, aber der Code auf dem unsortierten Array kann die CPU nur zur Hälfte auslasten. Die sortierten Arrays schaffen fast das Optimum an Auslastung der CPU mit 3,2 Instruktionen pro Cycle (IPC).

Aus dem reinen Benchmark geht übrigens nicht hervor, ob die CPU die Vorhersage vorgenommen hat oder ob der JIT-Compiler den

Code optimiert hat und die Reihenfolge der Branches geändert wurde, um Sprünge zu vermeiden (siehe *[9]*).

Synchronization

Der Autor möchte noch kurz auf zwei weitere Techniken, um Kontextwechsel der CPU und Memory-Zugriffe zu vermeiden, eingehen: Lock Elision und Lock Coarsening.

Lock Elision: Wenn der JIT-Compiler Objekte als lokal erkennt, diese jedoch Synchronized-Statements enthalten, dann werden diese Locks nicht ausgeführt, da ohnehin kein paralleler Zugriff möglich ist.

Lock Coarsening: Der JIT-Compiler versucht, nah zusammenliegende Locks des gleichen Objekts zusammenzufassen, um Kontextwechsel der CPU und Schreiboperationen auf den Hauptspeicher zu reduzieren.

Weiterhin darf der JIT-Compiler jederzeit auch Code von außerhalb eines Synchronized-Blocks hineinziehen, wenn es die Performance verbessert.

Wichtiger Hinweis

Die JVM ist absolut frei, mit neuen Versionen ihr Verhalten zu ändern. Der Compiler javac kann anderen Bytecode erzeugen, die JVM kann andere Intrinsics mitbringen, Inlining und Escape Analysis können sich ändern. Deswegen ist es wichtig, Tests mit neuen JDKs und neuer Hardware wiederholt auszuführen und das Ergebnis zu bewerten. In *Listing 13* sind drei Ergebnisse des gleichen Tests auf unterschiedlicher Hardware (IntrinsicsArrayCopy_AverageTime.java) zu sehen.

Garbage Collection kann die Performance verbessern

Garbage Collection wird meist als Problem gesehen, kann jedoch auch überraschend helfen. Der folgende Test zeigt, wie das Layout von Objekten im Speicher die Geschwindigkeit bestimmt (org.sample.GCAndAccessSpeed *[3]*). Im Beispiel werden Strings in der Reihenfolge ihrer Erzeugung in eine Liste gelegt sowie eine zweite Liste mit den gleichen Strings, aber in zufälliger Reihenfolge, erzeugt. Im Test wird die Liste abgelaufen und die Länge aller Strings aufsummiert. Zusätzlich werden zehn Mal mehr Strings erzeugt und gehalten, als in den Listen später genutzt werden.

In *Listing 14* zeigt die Spalte (drop), ob die temporären String-Referenzen aus dem Setup freigegeben wurden, und (gc) zeigt, ob nach dem Aufwärmen des Benchmarks eine Garbage Collection angestoßen wurde.

Reihenfolge: Man kann deutlich sehen, dass das geordnete Ablaufen des Hauptspeichers deutlich schneller ist (*Listing 14*, 1 vs. 2), denn die L1- bis L3-Caches sind optimal gefüllt. Werden die Referenzen gelöst, aber kein GC ausgeführt, dann bleibt das Ergebnis gleich, denn die Caches wissen nichts von den Inhalten des Speichers und darüber, ob die Daten wirklich benötigt werden (*Listing 14*, 5 und 6).

Unordnung: Wird eine GC ausgeführt, aber die temporären Strings weiter gehalten, dann wird auch der geordnete Zugriff langsamer, denn die GC hat die Ordnung, die nach dem Anlegen im Speicher existierte, nicht bewahrt. Damit wird die geordnete Liste aus Sicht des Speicherlayouts auch zu einer ungeordneten Liste (*Listing 14*, 3 und 4).

```
Linux Perf for step1
=============================================================
11,620,571,786  instructions         # 1.04 insn per cycle
 9,482,675,316  L1-dcache-loads      # 2114.210 M/sec
   593,181,935  L1-dcache-load-miss  # 6.26% of all hits
    21,448,491  LLC-loads            # 4.782 M/sec
     5,856,980  LLC-load-misses      # 27.31% of all hits

Linux Perf for step20
=============================================================
11,046,881,256  instructions         # 0.35 insn per cycle
   979,546,079  L1-dcache-loads      # 219.829 M/sec
   860,310,077  L1-dcache-load-miss  # 87.83% of all hits
   746,566,165  LLC-loads            # 167.544 M/sec
   330,798,682  LLC-load-misses      # 44.31% of all hits
```

Listing 10: Hardware-Statistiken JMH mit -prof perf

```
private static final int COUNT = 10_000;

// Contain random numbers from -50 to 50
private int[] sorted;
private int[] unsorted;
private int[] reversed;

public void doIt(int[] array, Blackhole bh1, bh2)
{
    for (int v : array)
    {
        if (v > 0)
        {
            bh1.consume(v);
        }
        else
        {
            bh2.consume(v);
        }
    }
}
```

Listing 11: org.sample.BranchPrediction1.java

```
Benchmark                        Mode         Score   Units
=============================================================
reversed                         avgt        35,182   ns/op
reversed:IPC                     avgt         3.257    #/op
reversed:branch-misses           avgt        13.767    #/op
reversed:branches                avgt    55,339.912    #/op
reversed:cycles                  avgt   110,993.223    #/op
reversed:instructions            avgt   361,508.018    #/op

sorted                           avgt        35,183   ns/op
sorted:IPC                       avgt         3.257    #/op
sorted:branch-misses             avgt        14.738    #/op
sorted:branches                  avgt    55,884.946    #/op
sorted:cycles                    avgt   112,912.763    #/op
sorted:instructions              avgt   367,820.131    #/op

unsorted                         avgt        65,770   ns/op
unsorted:IPC                     avgt         1.577    #/op
unsorted:branch-misses           avgt      3795.737    #/op
unsorted:branches                avgt    56,008.188    #/op
unsorted:cycles                  avgt   211,089.268    #/op
unsorted:instructions            avgt   333,105.231    #/op
```

Listing 12: Runtimes und Branch-Informationen BranchPrediction1.java

Umordnung: Werden allerdings die überflüssigen Objekte de-referenziert und eine GC angestoßen, dann führt die Umordnung der Daten im Hauptspeicher dazu, dass die Zugriffe optimal erfolgen und unnötige Daten (nicht benötigte Objekte) die Caches nicht mehr belasten. Die Objekte sind zusammengerückt.

```
# Lenovo T450s, JDK 11
Benchmark                    Mode  Cnt       Score  Units
IntrinsicsArrayCopy.manual   avgt    2  56,108.312  ns/op
IntrinsicsArrayCopy.system   avgt    2  55,748.067  ns/op
IntrinsicsArrayCopy.arrays   avgt    2  53,769.876  ns/op

# AWS c5.xlarge, JDK 11
Benchmark                    Mode  Cnt       Score  Units
IntrinsicsArrayCopy.manual   avgt    2  83,066.975  ns/op
IntrinsicsArrayCopy.system   avgt    2  63,422.029  ns/op
IntrinsicsArrayCopy.arrays   avgt    2  64,748.500  ns/op

# GCP n1-highcpu-8, JDK 11
Benchmark                    Mode  Cnt       Score  Units
IntrinsicsArrayCopy.manual   avgt    2  82,865.258  ns/op
IntrinsicsArrayCopy.system   avgt    2  62,810.793  ns/op
IntrinsicsArrayCopy.arrays   avgt    2  61,237.489  ns/op
```

Listing 13: Ergebnisse dreier Plattformen, jeweils Linux und JDK11

```
Benchmark          (drop)   (gc)      Score    Units
1:walkNonOrdered    false  false   1,903,731   ns/op
2:walkOrdered       false  false   1,229,945   ns/op

3:walkNonOrdered    false   true   2,026,861   ns/op
4:walkOrdered       false   true   1,809,961   ns/op

5:walkNonOrdered     true  false   1,920,658   ns/op
6:walkOrdered        true  false   1,239,658   ns/op

7:walkNonOrdered     true   true   1,160,229   ns/op
8:walkOrdered        true   true     403,949   ns/op
```

Listing 14: Ergebnisse GCAndAccessSpeed Tests mit G1-GC

Die GC kann das Hauptspeicherlayout verbessern und die CPU-Cache-Effizient steigern. Gleichzeitig werden jedoch Annahmen über die Speicherordnung (siehe *Listing 14*, Messungen 3 und 4) ungültig.

Fazit und Warnung

Performance-Tuning und Benchmarking machen viel Spaß, aber solange man nicht weiß, welche Probleme der Code hat, sollte man nicht auf diesem niedrigen Level optimieren. Um Donald Knuth zu zitieren: „Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%." *[8]*. In 90 Prozent der Fälle ist eine algorithmische Optimierung des Programmcodes hilfreicher, um Performanceprobleme zu beheben. Erst wenn diese Möglichkeiten ausgeschöpft sind, dann kann man sich Änderungen zuwenden, die das Wissen über den JIT, die JVM und die Hardware nutzen.

Weiterführende Themen und Material

Dieser Artikel erhebt keinen Anspruch auf Vollständigkeit und soll als Inspiration verstanden werden, sich näher mit den komplexen Themen Compiler, Memory, CPU und Caches zu beschäftigen. Viele Zusammenhänge lassen sich ohne umfangreiche Codelistings und Messergebnisse schwierig darstellen.

Empfohlen seien an dieser Stelle Douglas Q Hawkins Talks auf YouTube *[5]*. Dough ist ein Core-JVM-Entwickler. Das Buch „Optimizing Java" ist ebenfalls sehr empfehlenswert *[6]*, da es detailliert auf Themen rund um Compiler, GC und JVM- Interna eingeht. Die Ausgangsbasis dieses Artikels als vollständige Präsentation *[4]* liefert mehr Messergebnisse und mehr Codebeispiele *[3]*.

Quellen

[1] Energy Efficiency across Programming Languages: How does Energy, Time and Memory Relate?, accepted at the International Conference on Software Language Engineering (SLE) - Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva, 2017; *https://sites.google.com/view/energy-efficiency-languages/home*

[2] *https://openjdk.java.net/projects/code-tools/jmh/*

[3] GIT Repository für den Code dieses Artikels: *https://github.com/Xceptance/jmh-jmm-training*

[4] René Schwietzke, High Performance Java Präsentation, 2019: *https://training.xceptance.com/java/420-high-performance.html*

[5] Douglas Q Hawkins, Java Performance Puzzlers, Azul Systems, 2017: *https://www.youtube.com/watch?v=wgQBz2Ldhvk*

[6] Evans/Gough/Newland, Optimizing Java, O'Reilly Media, 2018

[7] *https://en.wikichip.org/wiki/intel/microarchitectures/kaby_lake*

[8] Donald Knuth, Turing Award Lecture 1974

[9] Java on Steroids: 5 Useful JIT Optimization Techniques; *https://blog.overops.com/java-on-steroids-5-super-useful-jit-optimization-techniques/*

[10] An Introduction to Epsilon GC, Attila Fejér, *https://www.baeldung.com/jvm-epsilon-gc-garbage-collector*

[11] Aleksey Shipilëv, JVM Anatomy Quark #16: Megamorphic Virtual Calls *https://shipilev.net/jvm/anatomy-quarks/16-megamorphic-virtual-calls/*

[12] Fasih Khatib, JVM JIT - Loop Unrolling, *http://fasihkhatib.com/2018/05/20/JVM-JIT-Loop-Unrolling/*

René Schwietzke
Xceptance GmbH
r.schwietzke@xceptance.com

René Schwietzke ist Mitgründer und Geschäftsführer der Xceptance GmbH. Er arbeitet seit der Version 1.0 mit Java und beschäftigt sich seit 2004 intensiv mit den JVM-Interna für Analyse und Performance-Tuning von SaaS-Commerce-Applikationen. Das Xceptance-Lasttest-Tool XLT basiert ebenfalls auf Java. Effiziente und zuverlässige Ausführungen großer Tests machten es nötig, die JVM genau zu verstehen.

HAST DU ES SCHON MIT EINEM NEUSTART VERSUCHT?

Come and join the CI Crowd.

JAVA DEVELOPER BI BERATER UX DESIGNER MOBILE DEVELOPER DU

Hier neu starten:
karriereportal.
cologne-intelligence.de

COLOGNE INTELLIGENCE

QAWARE
SOFTWARE ENGINEERING

IT-Probleme lösen.
Digitale Zukunft gestalten.

Mit Erfindergeist
und Handwerksstolz.

Great Place To Work.
2019
Beste Arbeitgeber™
Information & Telekommunikation

kununu.com/qaware
qaware.de/karriere