

# Inhaltsverzeichnis

# Kapitel 1

## Bytecode, JVM, Dynamische Compilierung

Am Beispiel des IBM Jalapeno-Compilers (besser als SUN!)<sup>1</sup>

---

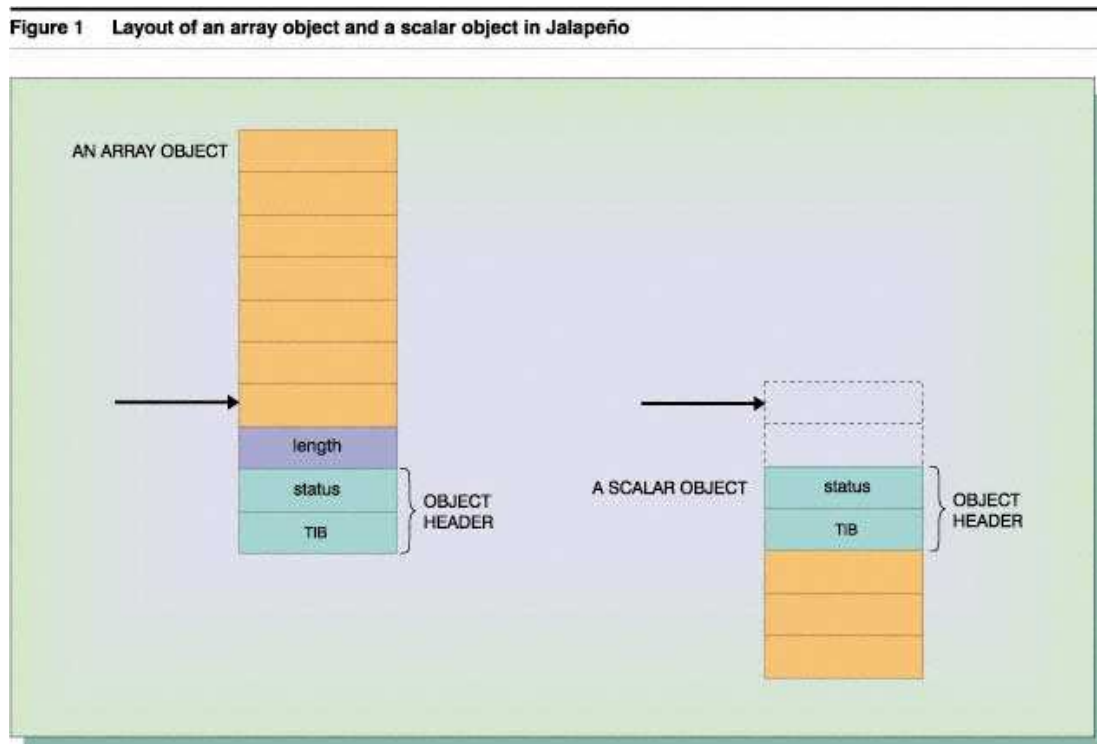
<sup>1</sup>Abbildungen aus: IBM Systems Journal, Vol 39 Nr 1

## 1.1 Laufzeitorganisation

Laufzeit-Datenbereiche der JVM:

- Program Counter: 1 PC-Register pro Thread. Enthält die Adresse der gerade ausgeführten Bytecode-Instruktion
- JVM Stack: jeder Thread hat eigenen Stack. Enthält die Activation Records der Methodenaufrufe
- Heap: enthält die Objekte. Alle Threads benutzen denselben Heap (shared memory)
- Method Area: enthält für jede Klasse Konstantentabelle sowie Bytecode für Methoden / Konstruktoren
- Operandenstack: zur Auswertung von (arithmetischen, logischen, ...) Ausdrücken

## Objektlayout:



ersten 12 Byte: Länge (für Arrays; für Nicht-Arrays nicht belegt)

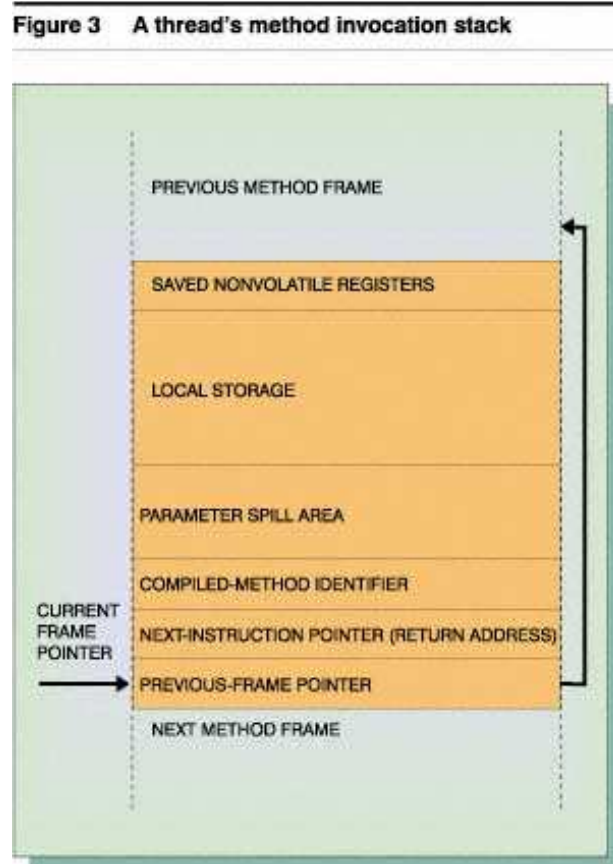
Status: Lock-Bits, Hash-Bits, Garbage-Collect-Bits

TIB: Type Information Block = vptr. JVM enthält in vtable zusätzlich Klassendeskriptor (vgl. Reflection-Interface)

Nullpointer-Zugriff erzeugt Hardware-Interrupt, da das length-Feld Offset -4 hat

Typische JVMs opfern Speicher, um Performance zu gewinnen!

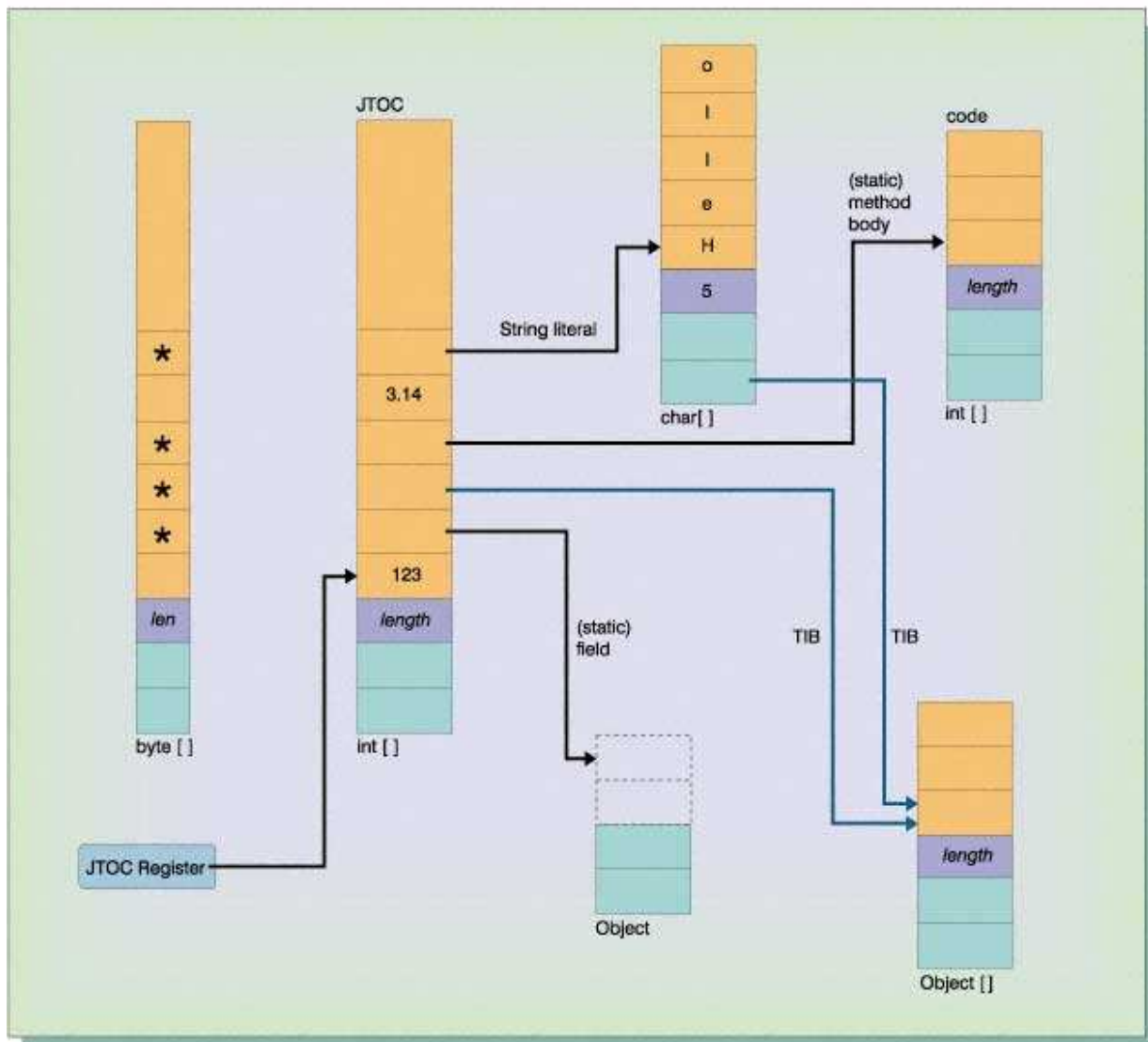
Aufbau des AR: analog C



JVM-Operandenstack wird in Hardwareregistern + Spillarea realisiert

Globale JTOC: Array mit (Verweisen auf) Konstanten + Klassendeskriptoren

Figure 2 The Jalepeño Table of Contents and other objects



## 1.2 Bytecode

- typische abstrakte Stack-Maschine

historisches Vorbild: Pascal P-Code

- Neben dem AR-Stack gibt es speziellen Operandenstack  
Arithmetische Codes: arbeiten auf Operanden-Stack.
- Unäre Operatoren (zB Typkonversion) wirken auf Topstack  
binäre verknüpfen die beiden obersten und  
schreiben Ergebnis wieder auf Stack.  
Ferner Lade/Speicherinstruktionen (push/pop)

- Alle Bytecodes kommen in verschiedenen typisierten Varianten:

| <i>opcode</i>    | <i>byte</i>    | <i>short</i>   | <i>int</i>       | <i>long</i>    | <i>float</i>   | <i>double</i>  | <i>char</i>    | <i>reference</i> |
|------------------|----------------|----------------|------------------|----------------|----------------|----------------|----------------|------------------|
| <i>Tipush</i>    | <i>bipush</i>  | <i>sipush</i>  |                  |                |                |                |                |                  |
| <i>Tconst</i>    |                |                | <i>iconst</i>    | <i>lconst</i>  | <i>fconst</i>  | <i>dconst</i>  |                | <i>aconst</i>    |
| <i>Tload</i>     |                |                | <i>iload</i>     | <i>lload</i>   | <i>fload</i>   | <i>dload</i>   |                | <i>aload</i>     |
| <i>Tstore</i>    |                |                | <i>istore</i>    | <i>lstore</i>  | <i>fstore</i>  | <i>dstore</i>  |                | <i>astore</i>    |
| <i>Tinc</i>      |                |                | <i>iinc</i>      |                |                |                |                |                  |
| <i>Taload</i>    | <i>baload</i>  | <i>saload</i>  | <i>iaload</i>    | <i>laload</i>  | <i>faload</i>  | <i>daload</i>  | <i>caload</i>  | <i>aload</i>     |
| <i>Tastore</i>   | <i>bastore</i> | <i>sastore</i> | <i>iastore</i>   | <i>lastore</i> | <i>fastore</i> | <i>dastore</i> | <i>castore</i> | <i>aastore</i>   |
| <i>Tadd</i>      |                |                | <i>iadd</i>      | <i>ladd</i>    | <i>fadd</i>    | <i>dadd</i>    |                |                  |
| <i>Tsub</i>      |                |                | <i>isub</i>      | <i>lsub</i>    | <i>fsub</i>    | <i>dsub</i>    |                |                  |
| <i>Tmul</i>      |                |                | <i>imul</i>      | <i>lmul</i>    | <i>fmul</i>    | <i>dmul</i>    |                |                  |
| <i>Tdiv</i>      |                |                | <i>idiv</i>      | <i>ldiv</i>    | <i>fdiv</i>    | <i>ddiv</i>    |                |                  |
| <i>Trem</i>      |                |                | <i>irem</i>      | <i>lrem</i>    | <i>frem</i>    | <i>drem</i>    |                |                  |
| <i>Tneg</i>      |                |                | <i>ineg</i>      | <i>lneg</i>    | <i>fneg</i>    | <i>dneg</i>    |                |                  |
| <i>Tshl</i>      |                |                | <i>ishl</i>      | <i>lshl</i>    |                |                |                |                  |
| <i>Tshr</i>      |                |                | <i>ishr</i>      | <i>lshr</i>    |                |                |                |                  |
| <i>Tushr</i>     |                |                | <i>iushr</i>     | <i>lushr</i>   |                |                |                |                  |
| <i>Tand</i>      |                |                | <i>iand</i>      | <i>land</i>    |                |                |                |                  |
| <i>Tor</i>       |                |                | <i>ior</i>       | <i>lor</i>     |                |                |                |                  |
| <i>Txor</i>      |                |                | <i>ixor</i>      | <i>lxor</i>    |                |                |                |                  |
| <i>i2T</i>       | <i>i2b</i>     | <i>i2s</i>     |                  | <i>i2l</i>     | <i>i2f</i>     | <i>i2d</i>     |                |                  |
| <i>l2T</i>       |                |                | <i>l2i</i>       |                | <i>l2f</i>     | <i>l2d</i>     |                |                  |
| <i>f2T</i>       |                |                | <i>f2i</i>       | <i>f2l</i>     |                | <i>f2d</i>     |                |                  |
| <i>d2T</i>       |                |                | <i>d2i</i>       | <i>d2l</i>     | <i>d2f</i>     |                |                |                  |
| <i>Tcmp</i>      |                |                |                  | <i>lcmp</i>    |                |                |                |                  |
| <i>Tcmpl</i>     |                |                |                  |                | <i>fcmpl</i>   | <i>dcmpl</i>   |                |                  |
| <i>Tcmpg</i>     |                |                |                  |                | <i>fcmpg</i>   | <i>dcmpg</i>   |                |                  |
| <i>if_TcmpOP</i> |                |                | <i>if_icmpOP</i> |                |                |                |                | <i>if_acmpOP</i> |
| <i>Treturn</i>   |                |                | <i>ireturn</i>   | <i>lreturn</i> | <i>freturn</i> | <i>dreturn</i> |                | <i>areturn</i>   |



*Beispiel 1:* Übersetzung von  $x = x+y*z;$

int-Variablen x, y, z haben Offset 42, 43, 44 im AR

Bytecode:

```
    iload 43
    iload 44
    imul
    iload 42
    iadd
    istore 42
```

*Weitere Bytecodes:*

Objekterzeugung, Memberzugriff: **new**, **newarray**,  
**anewarray**, **multianewarray**, **getfield**, **putfield**,  
**getstatic**, **putstatic**

Arrayzugriff: **Taload**, **Tastore**, **arraylength**

Typetest: **instanceof**, **ckeckcast**

bedingte Sprünge: **ifeq**, **iflt**, **ifnull**, **if\_icmpeq**,  
**if\_acmpeq**, ..., **tableswitch**, **lookupswitch**

unbedingte Sprünge: **goto**, **goto\_w**

Methodenaufruf: **invokevirtual**, **invokeinterface**,  
**invokespecial**, **invokestatic**, **Treturn**

Exceptions: **athrow**, **jsrm jsr\_w**, **ret**

Synchronisation: **monitorenter**, **monitorexit**

## Beispiel 2: Fibonacci-Berechnung

```

static void calcSequence() {
    long fiboNum = 1;
    long a = 1;
    long b = 1;
    for (;;) {
        fiboNum = a + b;
        a = b;
        b = fiboNum;
    }
}

```

Bytecode:

```

0 lconst_1 // Push long constant 1
1 lstore_0 // Pop long into local vars 0 & 1:
           // long a = 1;
2 lconst_1 // Push long constant 1
3 lstore_2 // Pop long into local vars 2 & 3:
           // long b = 1;
4 lconst_1 // Push long constant 1
5 lstore_4 // Pop long into local vars 4 & 5:
           // long fiboNum = 1;
7 lload_0  // Push long from local vars 0 & 1
8 lload_2  // Push long from local vars 2 & 3
9 ladd     // Pop two longs, add them, push result
10 lstore_4 // Pop long into local vars 4 & 5:
           // fiboNum = a + b;
12 lload_2  // Push long from local vars 2 & 3
13 lstore_0 // Pop long into local vars 0 & 1: a = b;
14 lload_4  // Push long from local vars 4 & 5
16 lstore_2 // Pop long into local vars 2 & 3:
           // b = fiboNum;
17 goto 7 // Jump back to offset 7: for (;;) {}

```

## 1.3 Methodenaufwurf

1. Bezugsobjekt+aktuelle Parameter auf Operandenstack pushen
2. `invokevirtual`-Befehl ausführen:
3. neues AR anlegen (Länge statisch bekannt);  
Program Counter+1 → Return Address; Current-  
Frame-Ptr → Previous-Frame-ptr; Register retten
4. `this`-Ptr → Offset 0 im Local Storage; Parameter → Offset 1...; Operandenstack poppen
5. Einsprungadresse des Bytecode für Methodenrumpf aus vtable → Program Counter
6. Code für Rumpf ausführen
7. **return**-Befehl ausführen:
8. Return-Value auf Operandenstack pushen
9. Return-Adress → PC;  
Previous-Frame-Ptr → Frame-ptr; AR freigeben

## Beispiel 3: Methodenaufruf + Exceptions

```
class A {
    Object f() {
        return this;
    }
}
class B extends A {
    Object f() {
        throw new Error();
    }
}
class C {
    public static void main(String s[]) {
        A a;
        Object o;

        if(s[0].equals("A"))
            a = new A();
        else
            a = new B();

        try {
            o = a.f();
        } catch(Exception e) {
            o = e;
        }
    }
}
```

```
bytecode A.f(->java.lang.Object)
```

```
0 aload 0  
1 areturn
```

```
bytecode A.<init>(->)
```

```
0 aload 0  
1 invokespecial java.lang.Object.<init>(->)  
4 return
```

```
bytecode C.main(java.lang.String[]->)
```

```
EH [#1e,#23) #26 java.lang.Exception
```

```
0 aload 0  
1 iconst 0  
2 aaload  
3 ldc string  
5 invokevirtual java.lang.String.equals(  
    java.lang.Object->.boolean)  
8 ifeq #16  
b new A  
e dup  
f invokespecial A.<init>(->)  
12 astore 1  
13 goto #1e  
16 new B  
19 dup  
1a invokespecial B.<init>(->)  
1d astore 1  
1e aload 1  
1f invokevirtual A.f(->java.lang.Object)  
22 astore 2  
23 goto #29  
26 astore 3  
27 aload 3  
28 astore 2  
29 return
```

```
bytecode C.<init>(->)  
0 aload 0  
1 invokespecial java.lang.Object.<init>(->)  
4 return
```

```
bytecode B.<init>(->)  
0 aload 0  
1 invokespecial A.<init>(->)  
4 return
```

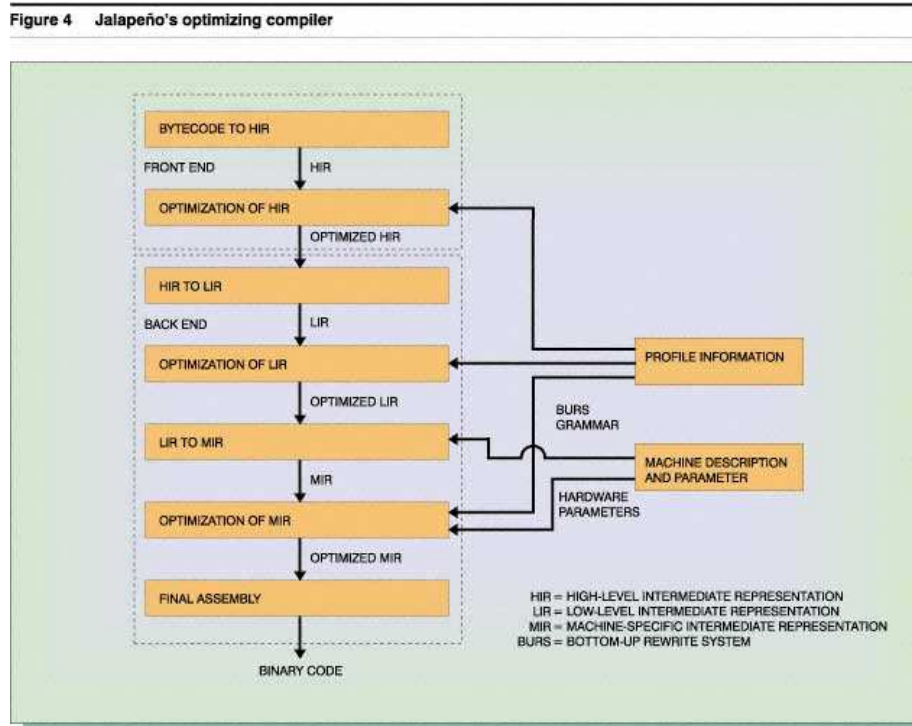
```
bytecode B.f(->java.lang.Object)  
0 new java.lang.Error  
3 dup  
4 invokespecial java.lang.Error.<init>(->)  
7 athrow
```

## 1.4 Just-in-Time Compiler

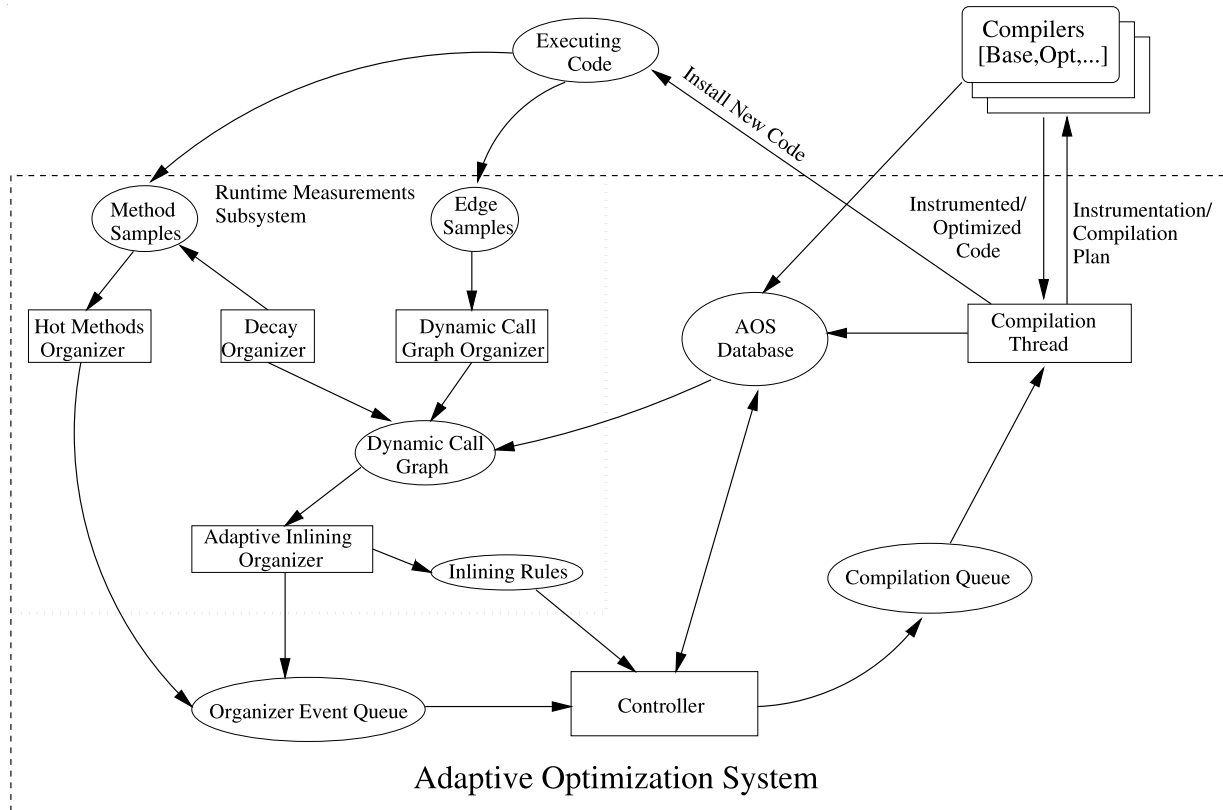
Compilervariationen:

- nur Bytecode-Generierung, JVM in C
- Just-in-time: Maschinencode für Methoden, sobald sie das erstmal aufgerufen werden
- Adaptive Compilation (Jalapeño): JVM größtenteils in Java, Generierung von Maschinencode und Optimierung aufgrund dynamischem Profiling

Grobaufbau des Jalapeño-Compilers: (Bytecode nach Maschinencode)



### Struktur der Compileroptimierung (Hotspot-Technologie)



Es wird sowohl Häufigkeit von Methodenausführungen als auch Zahl der Aufrufe  $A.f() \rightarrow B.g()$  gemessen

Falls Schwellwert überschritten: Maschinencode; für Kanten im dynamischen Call Graph: Inlining

Schwellwerte sind heuristisch adaptiv; alte Werte „verfaulen“; Datenbank mit alten Messwerten

## 1.5 Bytecode Verifier

...