Optimizations performed
by the Java HotSpot Compiler

Huib van den Brink

Center for Software Technology, Universiteit Utrecht
http://www.cs.uu.nl/groups/ST/

June 11, 2006
Outline

1. Introduction
2. Traditional optimizations
3. Supporting techniques
4. Common optimization techniques
5. Stages performed by the Java HotSpot compiler
6. Optimizations without transformations
7. Mustang
8. Prospects
Introduction

Java HotSpot Compiler

1. Java to bytecode
2. Bytecode to machine instructions

Modes

1. Client Compiler
   - Low startup cost
   - Not fully optimized
2. Server Compiler
   - High startup cost
   - Some part of performance used for analysis
   - Optimized execution
Traditional optimizations

Traditional optimizations applied

1. Method inlining
2. Constant propagation
3. Dead code elimination
Supporting techniques

Program analysis

1. Use-define chains
2. Data-flow analysis
3. Reaching definition analysis
4. Static single assignment form
Static single assignment form

SSA characteristics

1. Helps other optimizations to perform their job
   - Sparse conditional constant propagation
   - Dead code elimination
   - Global Value Numbering

2. Solves scoping and shadowing issues

3. Uses Reaching definition analysis
Static Single Assignment form

SSA

1. Every variable is assigned a value only once
2. Existing variables are renamed
**Static Single Assignment form**

- Every variable is assigned a value only once
- Existing variables are renamed

```c
int a = 1;
a = 2;
int b = a;
```
Static Single Assignment form

1. Every variable is assigned a value only once
2. Existing variables are renamed

```
1 int a = 1;
2 a = 2;
3 int b = a;
```

```
1 int a_1 = 1;
2 a_2 = 2;
3 int b_1 = a_2;
```
Static Single Assignment form

A closer look

1. Three address code IR
   `'1 + 2 * 3' into 't1 ← 2 * 3' and 't2 ← 1 + t1'`

2. Control flow and Phi
Static Single Assignment form

\begin{figure}
\centering
\begin{tikzpicture}
  \node [circle, draw] {x = 2;}
  \node [circle, draw, below of=x] {x = x + 3}
  \node [circle, draw, below of=x] {x > 4}
  \node [circle, draw, below of=x] {y = 2 \times x;}
  \node [circle, draw, below of=x] {z = y}
  \node [circle, draw, below of=x] {y = x + 2}
  \node [circle, draw, below of=x] {z = x - y}
\end{tikzpicture}
\end{figure}
Static Single Assignment form

```
x = 2;
x = x + 3

x > 4

y = 2 * x;
z = y

y = x + 2

z = x - y

x_1 = 2;
x_2 = x_1 + 3

x_2 > 4

y_1 = 2 * x_2;
z_1 = y_1

y_2 = x_2 + 2

z_2 = x_2 - y_2
```
Static Single Assignment form

```plaintext
x_1 = 2;
x_2 = x_1 + 3

x_2 > 4

y_1 = 2 * x_2;
z_1 = y_1

y_2 = x_2 + 2

y_3 = φ(y_1, y_2)
z_2 = x_2 - y_3
```
Optimization techniques

Supporting techniques

1. Loop invariant hoisting
2. Common subexpression elimination
3. Global Value Numbering
4. Null-check & range-check elimination
Loop invariant hoisting

1. Locate static information
2. Don’t recompute if not strictly necessary

Before

```c
while (i < n-1)
{
    j += (n+1) * array[i] * (pi+2);
    i++;
}
```
Loop invariant hoisting

1. Locate static information
2. Don’t recompute if not strictly necessary

Before

```c
while(i < n-1)
{
    j += (n+1) * array[i] * (pi+2);
    i++;
}
```

After

```c
int max = n-1;
int tmp = (n+1) * (pi+2);
while(i < max)
{
    j += tmp * array[i];
    i++; }
```
Loop invariant hoisting

Remarks
Suppose the loop contains $i \leftarrow a \oplus b$
Where $a$ and $b$ never change

1. May not effect final result
2. Don’t recompute if not strictly necessary
3. Multiple definitions of $i$ in loop (scoping)
4. $i$ should dominate all exit paths
Common subexpression elimination

Idea

1. Identify instances of identical expressions
2. Often pieces of expressions evaluate to the same value
3. Use temporary variables instead
Common subexpression elimination

Initial situation

1. `int x = v * w + z;
2. int y = y * v * w;`
Common subexpression elimination

Initial situation

1. `int x = v * w + z;`
2. `int y = y * v * w;`

After optimization

1. `int tmp = v * w;`
2. `int x = tmp + z;`
3. `int y = y * tmp;`
Common subexpression elimination

Remarks

1. Excessive number of temporary values
2. Decide if the recalculation is faster than the memory allocation
3. Array indexing calculations do benefit
4. Global CSE relies on dataflow analysis
Global Value Numbering

Idea

1. Looks at values stored in the different variables
2. Assigns a value number to variables and expressions
3. Is used to eliminates redundant code
4. Enables the optimization of the use of variables, values and computations
5. Improves the effect of constant propagation
Global Value Numbering

Initial situation

1. `int v = 2;`
2. `int w = 2;`
3. `int x = v + 3;`
4. `int y = w + 3;`
Global Value Numbering

Initial situation

1 int v = 2;
2 int w = 2;
3 int x = v + 3;
4 int y = w + 3;

\[ v \mapsto 1, \ w \mapsto 1, \ x \mapsto 2, \ y \mapsto 2 \]
Global Value Numbering

Initial situation

1 int v = 2;
2 int w = 2;
3 int x = v + 3;
4 int y = w + 3;

[v ↦→ 1, w ↦→ 1, x ↦→ 2, y ↦→ 2]

Result after substitution

1 int v = 2;
2 int w = v;
3 int x = v + 3;
4 int y = x;
GVN compared with CSE

1. CSE matches lexically identical expressions
2. GVN Determining the underlying equivalence

Example

```c
1 int x = v * w;
2 int y = v;
3 int z = y * w;
```
GVN compared with CSE

1. CSE matches lexically identical expressions
2. GVN Determining the underlying equivalence

Example

```c
1 int x = v * w;
2 int y = v;
3 int z = y * w;
```

GVN gives: \([v \mapsto 1, w \mapsto 2, x \mapsto 3, y \mapsto 1, z \mapsto 3]\)
GVN compared with CSE

1. CSE matches lexically identical expressions
2. GVN Determining the underlying equivalence

Example

```c
1 int x = v * w;
2 int y = v;
3 int z = y * w;
```

GVN gives: \([v \mapsto 1, w \mapsto 2, x \mapsto 3, y \mapsto 1, z \mapsto 3]\)

CSE would not discover this
Global Value Numbering

Remarks

1. Based on Static Single Assignment form representation
2. Preventing false variable name-value mappings
Null-check & range-check elimination

1. JVM performs implicit boundary checks at array access
2. Do not perform such checks if not strictly necessary
3. When $i$ and $a$ never change in $a[i]$ usages; only check it the first time
4. Data-flow used for detecting locations of $i$
5. Loop unrolling
6. Detect locations where variables become not null and null
Stages performed by the Java HotSpot compiler

1. Parsing
2. Machine independent optimizations
3. Instruction selection
4. Global code motion and scheduling
5. Register allocation
6. Peephole optimization
7. Code generation
Parsing

1. Pass over bytecode constructing internal representation
2. Constructing definition-use edges
3. Global Value Numbering is applied
4. \((constant + (constant + variable)) \) becomes \((variable + (constant + constant))\)
5. \(constant + 0\) is transformed in just \(constant\)
Machine independent optimizations

1. Class hierarchy analysis
2. Profiling
   - Inline when single receiver
   - Ensure hierarchy does not override method
3. Uncommon traps
   - Back to interpreted mode
   - Class not loaded or initialized by interpreter
   - Deoptimize all dependent methods too
4. Forward value propagation
5. Checking for identity situations
6. Constant propagation
7. Global dead code elimination
Instruction selection

1. Replace machine-independent instructions by machine instructions
2. Bottom up
3. Divide nodes into possibly overlapping subtrees
4. Construct machine specific nodes
5. Postorder walk for each root node
6. Lowest cost native instruction by DFA
7. For every parent, use the state vector
Global code motion and scheduling

1. Get the instructions into an optimal order
2. Rearrange generated machine instructions
3. Hardware platform specific
4. Dominator information is retrieved
5. Construct basic blocks
   - Earliest legal block (deepest in dominator tree)
   - Base cases like Region, Phi, Goto, Return
   - Definition-use gives latest legal block
6. Independent loads and stores are identified
7. Local ordering within basic blocks
   - Delay loads
   - Prefer instructions with many input
Register allocation

1. Global live analysis on annotated CFG
2. Gather legal set of registers for each live range
3. Interference graph
4. Copy merging
5. Indicator value per block for high/low register pressure
Peephole optimization

1. Inspects each adjacent instructions
2. Reorganization of instructions
3. Machine dependent
   - MOV dest_reg src_reg
   - INC dest_reg
   - IA32 allows: LEAL dest_reg dest_reg + 1
Code generation

1. Generate executable machine code
2. Produce debug info
3. Construct exception tables
4. Create implicit-null check table
5. Calculate the size of the executable
6. Give branch instructions their offset
7. Placed into a buffer for use by runtime system
Optimizations without transformations

Act smart

Not only optimization by transformation

1. Efficiently using the heap
   - Objects situated on heap
   - 95% very short lived
   - New objects placed in nursery
   - Contiguous allocated
   - Check on nursery overflow
   - Don’t reclaim the memory of the dead
   - Move surviving objects
   - Classes also on heap

2. Fast subtype checking
Fast subtype checking

```java
Volkswagen vw = new Volkswagen();
Vehicle v = id(vw);

if(v instanceof Volkswagen)
    vw = (Volkswagen)v;

public Vehicle id(Vehicle v)
{
    return v;
}
```
Optimizations without transformations

Inheritance

```java
Volkswagen vw = new Volkswagen();
Vehicle v = id(vw);

if(v instanceof Volkswagen)
    vw = (Volkswagen)v;

public Vehicle id(Vehicle v)
{
    return v;
}
```

1. Instanceof
2. Typecast
3. Storing objects into an array
Fast subtype checking

1. Single inheritance but multiple supertypes
2. $\text{Object}[][] \rightarrow \{\text{Clonable}, \text{Object}[], \text{Object}\}$
Fast subtype checking

1. Single inheritance but multiple supertypes
2. \( \text{Object}[][] \rightarrow \{\text{Clonable}, \text{Object}, \text{Object}[]\} \)

Primary types

\( \text{Integer}[][] \) has display:
\( \{\text{Object}, \text{Object}, \text{Object}[][], \text{Number}[][], \text{Integer}[]\} \)
Fast subtype checking

1. Depth equal to display length
2. Fixed subtype position
Fast subtype checking

Secondary types

1. Interfaces & arrays of interfaces
2. Linear search through list
3. Normally only short list
4. One element cache used
The prospects of Mustang

Introduced features

1. Escape analysis
2. Tiered compilation
Escape analysis

### Storage locations

1. Local variables (primitive type) on stack
   Pushing and popping on entry and exit
2. Static variables stored in static memory
3. Global non-static in heap
Escape analysis

Storage locations
1. Local variables (primitive type) on stack
   Pushing and popping on entry and exit
2. Static variables stored in static memory
3. Global non-static in heap

Complications
1. Call by reference
2. Many references from local vars to objects
Escape analysis

Objective

1. Detect when stack allocation is more appropriate
2. Propagate content of isolated objects
3. Hoist object fields into registers
# Escape analysis

## Objective

1. Detect when stack allocation is more appropriate
2. Propagate content of isolated objects
3. Hoist object fields into registers

## Requirements

1. Objects only referenced and used locally
2. Confined to single thread
3. Lifetime bounded by stack frame
A user container

class User
{
    private String name;
    private int age;

    User(String name, int age){
        this.name = name;
        this.age = age;
    }

    User(User u){
        this(u.name, u.age); }

    String getName(){
        return name; }

    int getAge(){
        return age; }
}
Escape analysis

User information containing a User object

class UserInformation
{
    private User u;

    User getUser(){ return new User(u); }

    boolean isEqual(UserInformation other)
    {
        User otherUser = other.getUser();

        boolean sameAge = otherUser.getAge() == u.getAge();
        boolean sameNames = otherUser.getName().equals(u.getName());

        return (sameAge && sameNames);
    }
}
Checking equality

```java
1   boolean isEqual(UserInformation other) {
2       User otherUser = new User(other.u.name, other.u.age);
3
4       boolean sameAge = otherUser.age == u.age;
5       boolean sameNames = otherUser.name.equals(u.name);
6
7       return (sameAge && sameNames);
8   }
```
Checking equality

```java
boolean isEqual(UserInformation other) {
    User otherUser = new User(other.u.name, other.u.age);
    boolean sameAge = otherUser.age == u.age;
    boolean sameNames = otherUser.name.equals(u.name);
    return (sameAge && sameNames);
}
```

`otherUser` never escapes its scope
not mutated anywhere

```java
boolean isEqual(UserInformation other) {
    int tmpAge = other.u.age;
    String tmpName = other.u.name;
    boolean sameAge = tmpAge == u.age;
    boolean sameNames = tmpName.equals(u.name);
    return (sameAge && sameNames);
}
```
Escape analysis

Benefits

1. Convert heap allocation to stack (or no) allocation
2. Faster average allocation time
3. Reduced memory usage
4. Less cache misses
5. Improved garbage collector behavior

While staying object oriented
Tiered compilation

1. Mustang provides tiered compilation
2. Both client- & servermode present
3. Minor optimizations performed by client
4. Servermode applied where desirable
5. Advanced optimizations computed at less time critical moments
The future

The path taken so far

1. First JVM interpreter only
2. Template generated code
3. Interpreter plus optimized code
4. Multiple modes present and cooperating within JVM
The future

The path taken so far

1. First JVM interpreter only
2. Template generated code
3. Interpreter plus optimized code
4. Multiple modes present and cooperating within JVM

1. Platform independent
2. Portability and security

The way to go

1. Shift optimizations to compiler creating bytecode
2. Enrich bytecode instruction format
3. Transport program analysis results
4. Mobile-code transportation format
The optimizations performed by the Java HotSpot compiler

1. Greatly evolved over time
2. Lot of effort put into it
3. Significant improvement of performance
   Current optimized execution at least 3 times faster than interpreted only
4. Not always as fast as C++
5. Still some opportunities for the future
Thank you for your attention