Method Optimizations

Method Inlining

Method inlining replaces a function call site with the body of the callee.

Example:

```
int max(int a, int b) {
    if(a > b) return a;
    else return b;
}

int main() {
    int x = 3;
    int y = 5;
    int z = max(x,y);
}
```

$\Rightarrow$

```
int main() {
    int x = 3;
    int y = 5;
    int z;
    if(x > y) z = x;
    else z = y;
}
```

Method Inlining

Benefits:
- Less dynamic instructions
  - Call site removed, prologue and epilogue code eliminated
- Smaller dynamic memory needs since the activation record is eliminated
- Removal of control flow transfer helps eliminate branch penalties and improves instruction cache locality
- After inlining is performed, more code is available to the optimizer to improve

Disadvantages:
- Static code size increase is likely
- Code growth can impact instruction cache performance
- May increase register pressure

Method Inlining

In languages like C++, there is a keyword `inline` that hints to the compiler that a method should be inlined during compilation.

In C, this is one of the benefits of using a parameterized `#define` macro.

In OOPLs, we often have very small methods (usually acting as accessors and mutators) that can be inlined.

Inlining is not always the right thing to do, and so the compiler must use heuristics to decide to apply it or not.

Unknown depth recursion makes inlining difficult.

Tail Recursion Elimination

A **tail call** is a function call site that appears as the last statement in a function.

Example:

```
int factorial(int x) {
    if(x < 2) return 1;
    return x * fact(x-1);
}
```

Tail calls can be implemented without adding a new activation record to the stack.

The activation record of the original call is reused, substituting in the new parameter values as appropriate. The tail call is then replaced with a jump to the beginning of the function.

Optimization Order

A: Source Code or High-Level IR

- Scalar replacement of array references
- Data-cache optimization

B: Medium-Level IR

- Procedure integration
- Tail-call optimization
- Scalar replacement of aggregates
- Sparse conditional constant propagation
- Interprocedural constant propagation
- Procedure specialization and cloning

C:Medium/Low-level IR

- Global value numbering
- Local and global copy propagation
- Sparse conditional constant propagation
- Dead code elimination
- Local and global common subexpression elimination
- Loop-invariant code motion
- Partial redundancy elimination
- Code hoisting
- Induction-variable strength reduction
- Linear-function test replacement
- Induction-variable removal
- Unnecessary bounds-checking elimination
- Control-flow optimizations

D: Low-level IR/Machine code

- Inline expansion
- Leaf routine optimization
- Shrink wrapping
- Machine idioms
- Tail merging
- Branch optimizations and conditional moves
- Dead-code elimination
- Software pipelining: loop unrolling, variable expansion, etc.
- Basic-block scheduling
- Register allocation
- Intraprocedural I-cache optimization
- Instruction prefetching
- Data prefetching
- Branch prediction

Each Step

After A, B, C, and D,

- Constant folding
- Algebraic simplification

E: Link Time Optimization

- Interprocedural register allocation
- Aggregation of global references
- Interprocedural I-cache optimization

Runtime Optimization

Just-in-time Compilation

Just-in-time (JIT) compilers are software dynamic translators that convert one language into another at runtime, when a segment of code (often a method) is needed.

JIT compilers can be used to support dynamic languages which are not traditionally compiled such as JavaScript or to support languages compiled into platform-independent bytecode, like Java.

Since the JIT compiler serves as a runtime environment, we can access dynamic properties of the program in order to better optimize it.

Java

```java
class EvenOdd {
    public static void main(String args[]) {
        long evenSum=0, oddSum=0;
        for(int i=0;i<1000000;i++) {
            if(i%2 == 0) {
                evenSum+=i;
            }
            else {
                oddSum+=i;
            }
        }
        System.out.println("Even sum: " + evenSum);
        System.out.println("Odd sum: " + oddSum);
    }
}
```

Bytecode

```
00 : lconst_0
01 : lstore_1
02 : lconst_0
03 : lstore_3
04 : iconst_0
05 : istore              local.05
07 : iload               local.05
09 : ldc                 1
0B : if_icmpge           pos.2A
0E : iload               local.05
10 : iconst_2
11 : irem
12 : ifne                pos.1E
15 : lload_1
16 : iload               local.05
18 : i2l
19 : ladd
1A : lstore_1
1B : goto                pos.24
1E : lload_3
1F : iload               local.05
21 : i2l
22 : ladd
```

Bytecode

```
23 : lstore_3
24 : iinc                local.05, 1
27 : goto                pos.07
2A : getstatic           java.io.PrintStream java.lang.System.out
2D : new                 java.lang.StringBuilder
30 : dup
31 : invokespecial       void java.lang.StringBuilder.<init>()
34 : ldc                 "Even sum: "
36 : invokevirtual       java.lang.StringBuilder java.lang.StringBuilder.append(java.lang.String)
39 : lload_1
3A : invokevirtual       java.lang.StringBuilder java.lang.StringBuilder.append(long)
3D : invokevirtual       java.lang.String java.lang.StringBuilder.toString()
40 : invokevirtual       void java.io.PrintStream.println(java.lang.String)
43 : getstatic           java.io.PrintStream java.lang.System.out
46 : new                 java.lang.StringBuilder
49 : dup
4A : invokespecial       void java.lang.StringBuilder.<init>()
4D : ldc                 "Odd sum: "
4F : invokevirtual       java.lang.StringBuilder java.lang.StringBuilder.append(java.lang.String)
52 : lload_3
53 : invokevirtual       java.lang.StringBuilder java.lang.StringBuilder.append(long)
56 : invokevirtual       java.lang.String java.lang.StringBuilder.toString()
59 : invokevirtual       void java.io.PrintStream.println(java.lang.String)
5C : return
```

Adaptive Optimization

Consider a JIT-compiler as part of a Java Virtual Machine (JVM).

Java Bytecode is a stack-oriented machine language. The JVM can use interpretation to implement the virtual CPU for Java Bytecode.

However, this interpretation is slow.

One option is to use JIT compilation to convert the bytecode into machine code.

However, compilation can be slow as well.

We have two ideas:

1. Apply cheap yet effective optimizations
2. Apply optimization only to regions that will benefit from it

Cost-Benefit Analysis

Let us only apply optimization O to method M if the time gained from the optimization is greater than the cost of doing the optimization.

cost(M' = O(M)) < (cost(M) - cost(M'))

We need models of cost for both the optimization and the optimized method. How can we know?

For the cost of the optimization, we can base this on the complexity of the method and the average-case complexity of the algorithm O.

For the cost of the optimized method M', we must be able to predict the future.

To do this, we can look at the past.

Profiling

We can generate a profile of method M to identify how "hot" it is. That is, how much time has been spent in the method so far by inserting instrumentation into the method code during compilation or interpretation to see how much time is spent in that code.

Execution time typically follows the **Pareto Principle** (the 80/20 or 90/10 rule):
90%(80%) of the time is spent in 10%(20%) of the code.

Let us only optimize those methods where we are likely to gain the most.

Choice of Optimizations

Let us then define an adaptive optimization scheme. Each time a method reaches a certain level of hotness, recompile it at the next level.

For instance, a Java Adaptive Optimizer might have 4 or 5 levels:

Level 0: no optimization, potentially even interpretation
Level 1: Baseline JIT, naïve code that still mimics the stack-based nature of the bytecode
Level 2: Do register allocation to remove as many of the operand stack loads and stores as possible
Level 3: Apply some basic control and dataflow optimizations
Level 4: Aggressively optimize the code

This is how the Java HotSpot compiler and Jikes Research Virtual Machine work.