Compiler Construction

Chapter 14: Runtime Optimization

Dittaya Wanvarie

Department of Mathematics and Computer Science
Chulalongkorn University

Second semester, 2024

Runtime optimization

Classic compilers

- Ahead-of-time compilation (AOT)
- Link time, loaded time knowledge
- No runtime knowledge
  - E.g. heavy used loops

Just-in-time compiler (JIT)

- Interpreter-like
- Compilation just before the execution
- Heavily used in scripting languages e.g JavaScript

Pros and cons

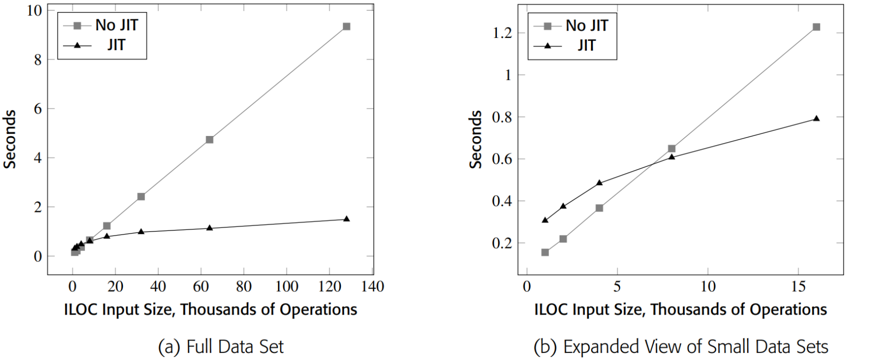
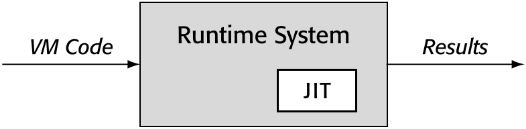


FIGURE 14.1 Scalability of a JAVA Application.

- Overhead cost in JIT

Execution model

- VM-code execution



  - More compact, and abstract than native codes
  - Requires emulation/translation
- Native-code execution
  - Faster than VM-code

Compilation trigger

Only compile (or optimize) **hot** codes

- Otherwise, it would be the same as AOT
- Requires a tracking/measurement of hot code

Finding hot code data profile

- Add codes to count the number of invocations
- Interrupts or special hardware to count the number of invocations

VM-code execution

- Mixed-mode environment: VM emulated + optimized translated native hot codes

Native-code execution

- Jump to optimized version when the code portion is hot

Granularity of optimization

Hot traces

- A sequence of basic building blocks that are frequently invoked
- A trace can cross procedure-call boundaries
- Local or regional optimization e.g. code generations, allocation, scheduling

Hot methods

- A method that is frequently invoked
- Non-local optimization e.g. code motion, regional instruction scheduling, dead-code elimination, global redundancy elimination, strength reduction, inline substitution

Methods and techniques

Finding redundancy, constant folding, simplifying identities

- Trace optimizer: Local-Value-Numbering
- Method optimizer: global redundancy algorithm

(Register) Allocation

- Trace optimizer: local register allocator
- Method optimizer: global register allocator

Sources of improvement

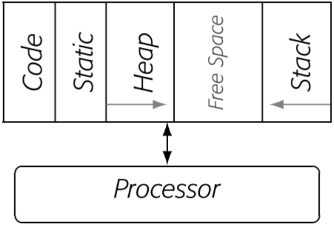
Runtime fact

- Profile information
- Object type
- Data structure sizes
- Loop bounds
- Constant values

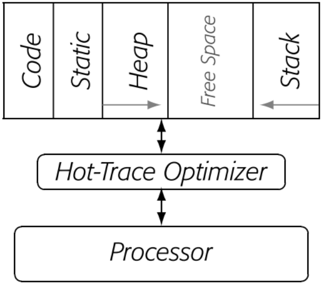
Possible improvement

- Eliminating VM overhead: VM emulation $\rightarrow$ native codes
- Improve code layout: E.g. removing some branch/jumps
- Eliminate redundancy: Value numbering
- Reduce call overhead: inline substitution
- Tailor code to the system: machine-dependent optimization
- Capitalize on runtime information: E.g. utilizing frequently used data types

Hot-trace optimization



Normal Execution



Execution Mediated by a Runtime Optimizer

Finding a trace

Candidate trace-entry blocks

- Loop header blocks
- Loop exit blocks

Counting the number of times the block is executed

- Set a threshold

Flow of execution

L1: emulate code until a taken branch or jump

lookup target address in the entry table
if address is not in the table then
if address < emulated PC then
create table entry for address
set target address' counter to 1

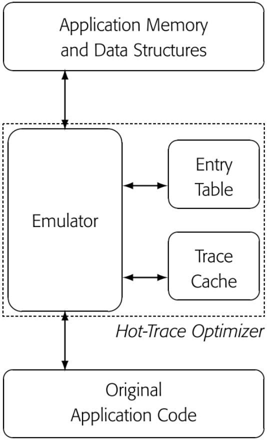
else // address is in the entry table
if fragment has been compiled then
jump to the compiled fragment

bump the target address' counter

if counter > hot threshold then
build, compile, & execute the trace

// if here, block runs in emulation mode
set emulator's PC to target address
jump back to L1

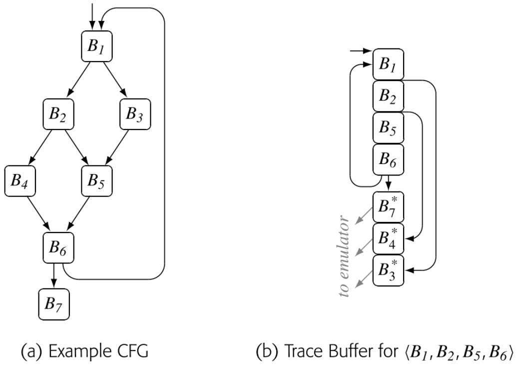
(a) High-level Algorithm



(b) Components of the Trace Optimizer

■ **FIGURE 14.2** Conceptual Structure of a Hot-Trace Optimizer.

Entry table and trace cache



■ **FIGURE 14.3** Building a Trace.

Linking traces

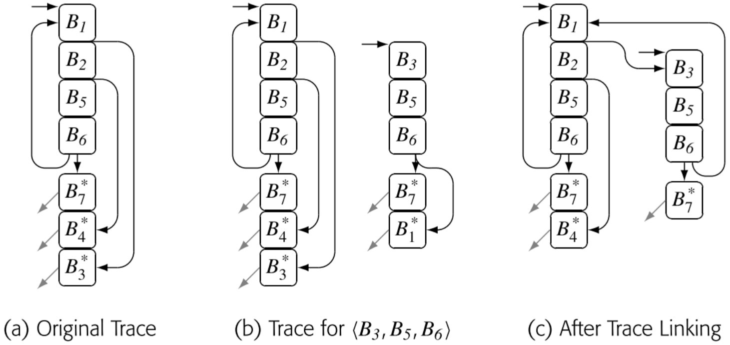


FIGURE 14.4 Adding a Second Trace.

Hot-method in mixed-mode environment

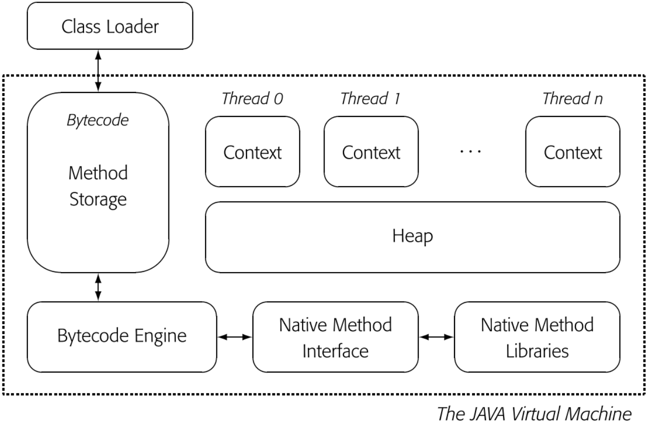


FIGURE 14.5 The JAVA Runtime Environment.

Adding JIT to JRE

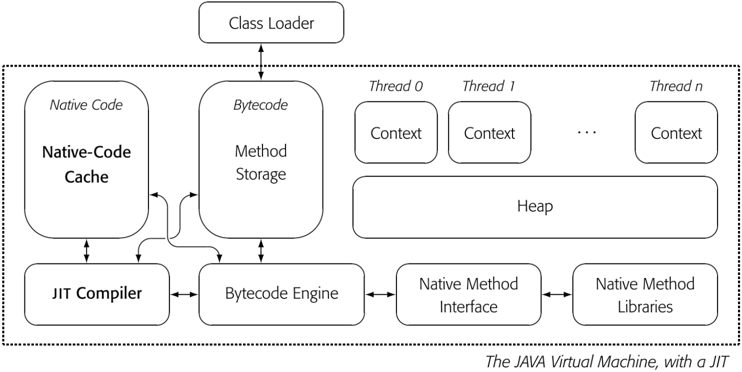


FIGURE 14.6 Adding a JIT to the JAVA Runtime Environment.

Getting runtime profile data

Instrumented VM code

- Insert VM code for counters
- Count: method invocations, loop iterations

Instrumented engine

- The emulator contains the counters

Compiling hot method

Similar to AOT compilation

- Parsing, instruction selection and scheduling, register allocation

Mixed-mode environment

- Native codes run faster than VM code $\longrightarrow$ benefit from heavy optimization
- Scalar optimization such as value numbering, constant propagation, dead-code elimination, code motion

Specialization to runtime data

Guessing the data type

$$
\begin{array}{l}
// \ x \leftarrow y + z \\
\textit{if actual\_type}(y) = \textit{32\_bit\_integer and} \\
\quad \textit{actual\_type}(z) = \textit{32\_bit\_integer then} \\
\qquad x \leftarrow \textit{32\_bit\_integer\_add}(y, z) \\
\textit{else} \\
\qquad x \leftarrow \textit{generic\_add}(y, \textit{actual\_type}(y), z, \textit{actual\_type}(z));
\end{array}
$$

■ **FIGURE 14.7** Code with a Fast Path for the Expected Case.

Hot-Methods in a native-code environment

Compile-on-call

1. Locates the VM code for the method
2. Invokes the JIT to produce native code for the method
3. Re-links the call site to point to the newly compiled native code

Optimization level

- Low: low compilation cost, simple optimization
- High: trade-off between the compilation time and the potential benefit of faster execution

Getting runtime profile data

Instrumented code

- Insert native codes for counters
- Count: method invocations, loop iterations

Interrupted-driven profiles

- Time interrupt to approximate the execution time