

Jikes™ Research Virtual Machine

IBM T. J. Watson Research

Christoph Bockisch

Outline

- Java Virtual Machines (JVM)
- Jikes Research Virtual Machine (RVM) overview
- Jikes implementation
 - Parts required for a JVM
 - Just-in-time compilation
 - Magic
- My work with Jikes: Steamloom

2

Christoph Bockisch

Jikes™ Research Virtual Machine

25.06.2003

JVMs

Java Virtual Machine

- Abstract computer executing Java bytecodes
 - Stack machine
 - Object oriented
 - Method invocation (static, virtual, interface)
 - Exceptions
 - Type checking
 - Multithreading
 - Memory management / garbage collection
 - Dynamic class loading
 - Reflection
 - Native methods

3

Christoph Bockisch

Jikes™ Research Virtual Machine

25.06.2003

Jikes overview

What is Jikes?

- Java Virtual Machine for Java servers
 - Executes large subset of Java
 - GNU Classpath
- Developed at IBM T. J. Watson Research
- Formerly Jalapeño
- Implemented in Java
- Based on just-in-time (JIT) compilation
- Meant for research
 - 3 compiler systems
 - 5 memory managers / garbage collectors
 - 4 object models
- Nothing really to do with Jikes sourcecode to bytecode compiler

4

Christoph Bockisch

Jikes™ Research Virtual Machine

25.06.2003

Impacts of using Java

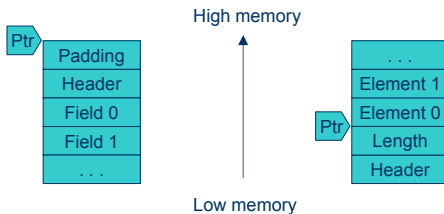
- Joint optimization of application and virtual machine
- No transformation between call conventions needed
- Jikes is not portable
 - Generates machine code

Bootstrapping

- Source JVM loads / executes Jikes RVM
 - Jikes again loads and executes itself
- Bootimage writer
 - Transform object layout
 - Translate object pointers
 - Writes image to file
- Bootimage runner
 - Written in C
 - Load image
 - Branch to executable code of `boot()` method

Object Layout

- Scalar object
- Array object



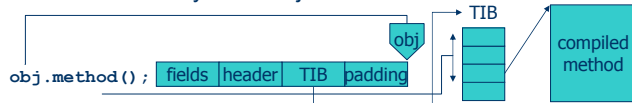
- Cheap null-pointer check
- Header differs from model to model
 - Java header
 - Garbage collection header
 - Miscellaneous header

Method invocation

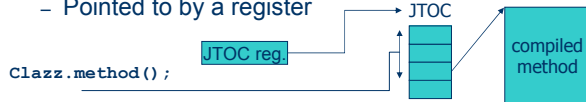
- Methods are late bound (virtual methods)
- Virtual method table
 - Contains pointers to executable code
 - Indexed by method's id
- Due to Java's single inheritance
 - Method's index can be the same for all sub classes (class inherits all virtual method from only one super class)
- Copy method pointers for inherited methods from super class
- Multiple inheritance for interfaces
 - ⇒ More complicated

Method dispatch

- **Virtual methods:** Type Information Block (TIB)
 - Exists once per class
 - Pointed to by each object

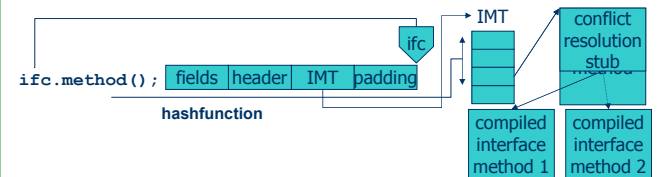


- **Static methods:** Jikes Table of Contents (JTOC)
 - Contains static members, all TIBs and constants
 - Pointed to by a register



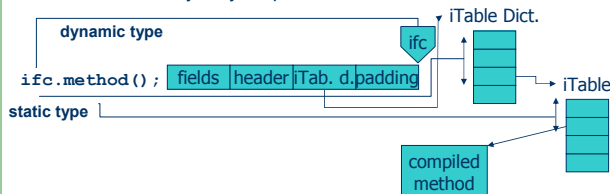
Interface method invocation

- Method inherited from interface
 - At different method's index in different TIBs
- Search method implementation
 - Interface Method Table (IMT) (embedded or indirect)
 - Fixed size interface method table per TIB
 - Hash function maps method descriptor to IMT entry
 - In case of conflict, IMT entry points to conflict resolution stub



Interface method invocation

- Search method implementation - continued
 - iTable
 - TIB points to iTable dictionary (one entry per loaded interface)
 - iTable dictionary (indexed by interface) points to iTable for class and interface
 - iTable (indexed by method id) points to executable code
 - Dictionary may be packed

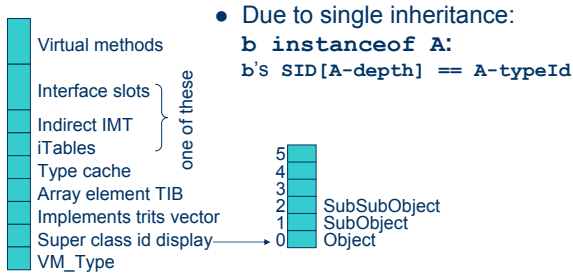


Exceptions

- Hardware exceptions delivered by operating system to C trap handler
- Trap handler modifies stack / registers of Java thread
 - `VM_Runtime.deliverException()` is called
- Throwing software exception calls `VM_Runtime.deliverException()`
- `deliverException()`
 - Unlock locked objects
 - Find exception handler / unwind stack
- Java's precise exception constrain constricts optimizations

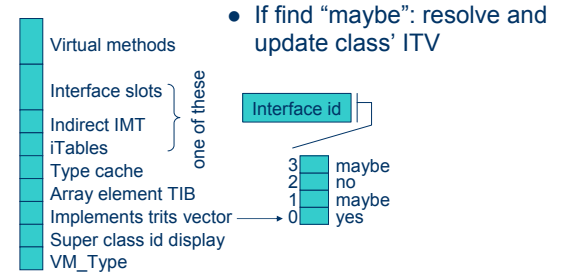
Dynamic type checking - SID

- Each type has own **VM_Type** instance with own id
- Each **VM_Type** stores number of super types (depth)
- Super class identifier display (SID) maps super class' depth to super class' type id



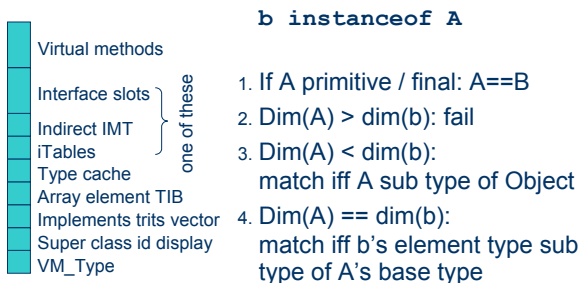
Dynamic type checking - ITV

- Implements trits vector (ITV) indexed by interface id
- Initial ITV: “maybe” for all interfaces
- If class implements no interfaces, share ITV with super class



Dynamic type checking - ETT

- Depending on dimension and element type
- **VM_Type** stores dimension
- Element Type TIB (ETT)

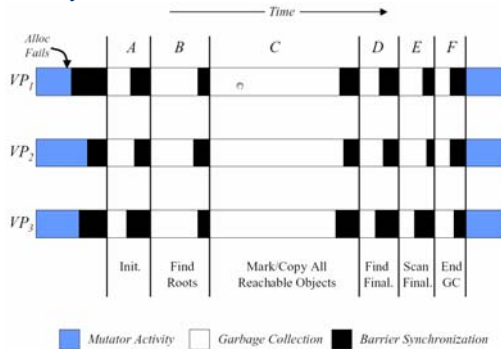


Memory management

- Memory allocation
- Stop-the-world garbage collectors
 - All suspended threads are at safe points
 - Safe points: possible memory allocation
 - Metadata about location of object references available for safe points
- Garbage collectors (GC)
 - Copying / non-copying
 - Generational / non-generational
 - Non-generational copy / mark-sweep hybrid

Garbage collection phases

- Parallel and load-balancing
- Barrier synchronization



Threads

- One virtual processor (VP) for each operating system thread
 - Each VP has queues of Java threads
 - Ready, waiting, iowait, idle, gc, transfer
 - Use timer interrupt for scheduling (time-slice)
 - Load balancing
- JIT Compiler (JITC) generates yield points
 - Always at safe point
 - At method prologue and loop back edge
- Detect long-running native code
 - Transfer blocked Java thread to skeleton VP

Magic

- Not expressible in Java
 - Access to machine registers / memory
 - Transfer execution to arbitrary address
 - Coerce object reference to addresses
 - Invoke operating system services
- Class `VM_Magic` defines empty methods
 - Compilable by Java sourcecode compiler
 - JITC recognizes invocations of these methods
 - JITC generates appropriate machine code
- Object allocation, garbage collection, dynamic linking, exception handling, reflection, input / output

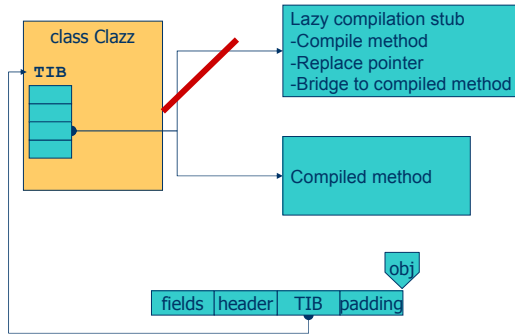
Class loading

- In Java: dynamic class loading
 - Must occur, when causing code is first executed
 - Possibly throws `ClassNotFoundException`
- Certain bytecodes refer to classes
 - Possibly need to load class
 - Preemptive class loading doesn't comply
- Class already loaded
 - Nothing special to do
- Class not loaded
 - Check if class loaded
 - If needed load class
 - Replace machine code with code for resolved class access

Just-in-time compiler

- Generate GC map for safe points
- Create machine code for line number mapping
- Create machine code for exception table mapping
- Baseline compiler
 - Emulates stack machine
 - No in-lining

Lazy compilation



Optimizing compiler

- Four transformations
 - Bytecode to high-level intermediate representation (HIR)
 - HIR to low-level intermediate representation (LIR)
 - LIR to machine-level intermediate representation (MIR)
 - Final assembly to machine code
- During transformation
 - Eliminate JVM stack abstraction
 - Eliminate redundant array bounds checks
 - Profile-driven (guarded) inlining
 - Inlining of bytecode subroutines
 - Devirtualization

Baseline vs. optimizing compiler

- Relative to the baseline compiler



Adaptive optimization system

- Samples taken at yield points
 - Coarse-grained
 - Low overhead
 - Occur at method prologues, epilogues and loop back edges
- Four optimization levels
- Extrapolate time expected to be spent executing a method at optimization level j (T_j)
- Estimate cost for compiling at level j (C_j)
- Choose level j minimizing $T_j + C_j$
 - Recompile if $j >$ current optimization level

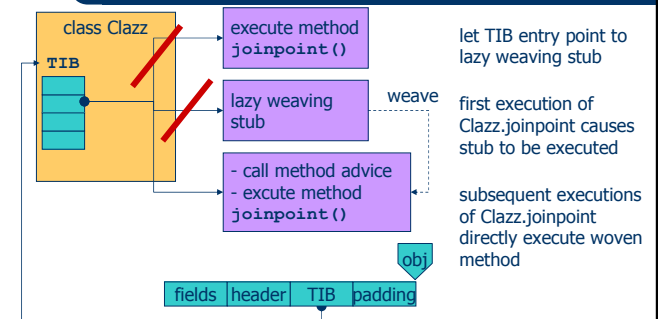
On-Stack-Replacement

- AOS may want to recompile currently executing method
- Compiled method must be replaced on stack:
- Capture JVM state
- Generate bytecode to
 - Restore state
 - Jump to instruction at which execution suspended
 - Append original bytecode
- Recompile optimized
- Continue execution at start of new compiled code

My current work: Steamloom

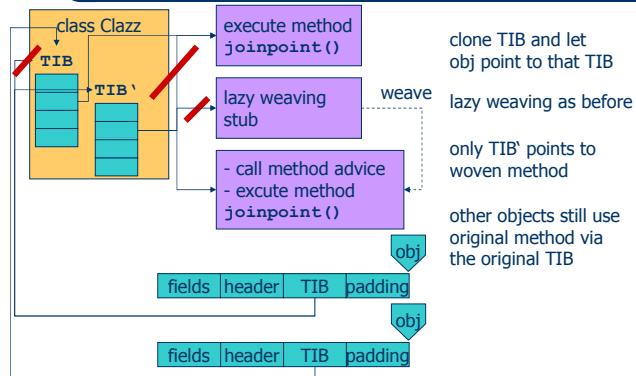
- Runtime environment for dynamic AOP
- In general:
 - Change methods' execution at runtime
 - Via API
 - Per class, per object, per thread
- Easy in Jikes RVM because
 - Method dispatchment via TIB / JTOC
 - Recomilation and machine code replacement already supported
 - Can easily change machine code for method dispatchment
 - API and AOP-engine written in Java

Lazy weaving



```
AdviceRegistry.setBeforeAdvice(Clazz.joinpoint, Aspect.advice);
```

Per-instance aspects



```
AdviceRegistry.setBeforeAdvice(obj, joinpoint, Aspect.advice );
```

Sources

- **The Design and Implementation of the Jalapeño Research VM for Java™**
International Conference on Parallel Architectures and Compilation Techniques, September 9, 2001 (PACT'01 Tutorial)
- **Optimizing Compilation of Java Programs (Revised version of PLDI'00 tutorial)**
ACM Java Grande, June 2001 (Java Grande'01 Tutorial)
- **Issues in High-Performance Programming in Java**
High Performance Computing System, June 13th, 1999 (HPCS99 Tutorial)
- **The Design and Implementation of the Jikes RVM Optimizing Compiler**
OOPSLA'02 Nov 5, 2002 (OOPSLA'02 Tutorial, expanded version of PLDI'02 tutorial)
- **Implementing Jalapeño in Java**
B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, S. Smith
1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99), Denver, Colorado, November 1, 1999.
- **A Comparative Evaluation of Parallel Garbage Collectors**
C. Attanasio, D. Bacon, A. Cocchi, S. Smith
Fourteenth Annual Workshop on Languages and Compilers for Parallel Computing, Cumberland Falls, Kentucky, Aug, 2001.
- **Efficient Implementation of Java™ Interfaces: Invokeinterface Considered Harmless**
B. Alpern, A. Cocchi, S. Fink, D. Grove, D. Lieber
ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Tampa, FL, USA, Oct 14-18, 2001.
- **Design, Implementation and Evaluation of Adaptive Recompile with On-Stack Replacement**
S. Fink, F. Qian
International Symposium on Code Generation and Optimization San Francisco, California, March 2003.
- **Adaptive Online Context-Sensitive Inlining**
K. Hazelwood and D. Grove
International Symposium on Code Generation and Optimization San Francisco, California, March 2003.
- **Adaptive Optimization in the Jalapeño JVM**
M. Arnold, S. Fink, D. Grove, M. Hind, P. F. Sweeney
ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000), Minneapolis, Minnesota, October 15-19, 2000.
- **The Java™ Virtual Machine Specification, Second Edition**
<http://java.sun.com/docs/books/vmspec/>