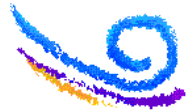
IBM's Mixed-Mode Interpreter

Dr. Robert Lougher
IBM Hursley, UK

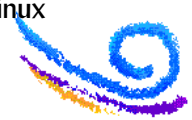
What is the Mixed-Mode Interpreter?

"The mixed-mode interpreter (MMI) is a new, high-speed profiling interpreter which completely replaces the existing interpreter within the Java Virtual Machine (JVM). It is separate but complementary to the JIT."



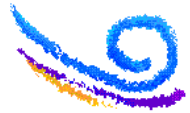
MMI Releases

- First appeared in JDK 1.1
  - Windows NT/98 JDK 1.1.7
  - OS/390 JDK 1.1.8
  - OS/2 JDK 1.1.8
- Java 2
  - Improved portability
  - Greater maintainability
  - Supported Platforms
    - AIX, Windows NT/98, OS/390, OS/2, Linux



Why "Mixed-Mode"?

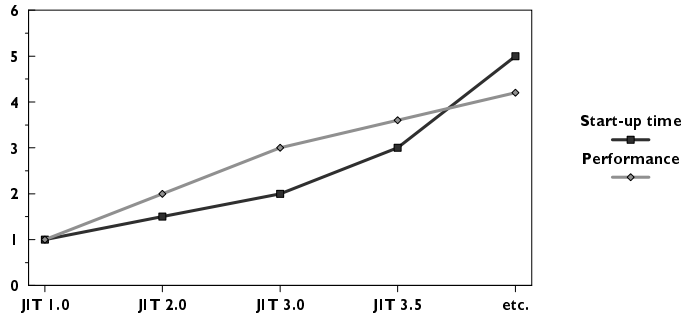
- C/Assembler Interpreter
  - All methods interpreted as bytecodes
- With JIT
  - <clinit> interpreted
  - All other methods compiled on first use
- MMI and JIT
  - Truly mixed-mode of execution
    - JIT-insensitive methods are interpreted
    - Hot methods compiled

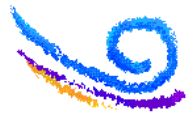


Why Do we Need MMI?

- The Start-up Time Problem
  - Each Technology Improvement
    - C interpreter, Assembler, JIT 1.0, JIT 2.0, JIT 3.0, JIT 3.5, etc.
  - Performance increases
  - Start-up time longer and longer

Start-up time versus Performance





Reducing Start-up Time

- Pre-compile bytecodes to native code
  - native code already available at start-up
  - Fat Classes
    - pre-compiled bytecodes contained within class file
  - JIT caching
    - native code cached to disk as JVM runs
- Selective Compilation
  - select the best sub-set of methods to compile
  - delays method compilation at start-up

Selective Compilation

- Select Methods Based on Usage
  - Profile Count
    - Decremented on each invocation
    - Compile method when count reaches zero
    - MMI uses initial value of 500
  - Problems
    - What do you do with methods that contain loops?
      - Method invocation count weights these methods too low

Loop Detection

- When MMI encounters a bytecode of the form
  - opc_if*xx*
  - opc_if_icmp*xx*
- Checks if it is a loop backedge by comparing with specific bytecode pattern
- Modifies profile count depending on
  - computed loop count
  - distance of backward branch

Loop Detection Example

Bytecodes

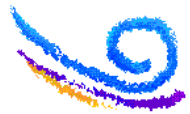
```
...
loop:
  ...
  iinc 1 1
  iload_1
  sipush 1000
  if_icmplt loop
  ...
```

pattern match

Java

```
...
for(i = 0; i < 1000; i++) {
  ...
}
...
```

Predict iteration count = 1000

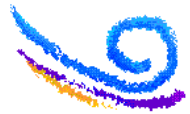


Loop Detection (Cont)

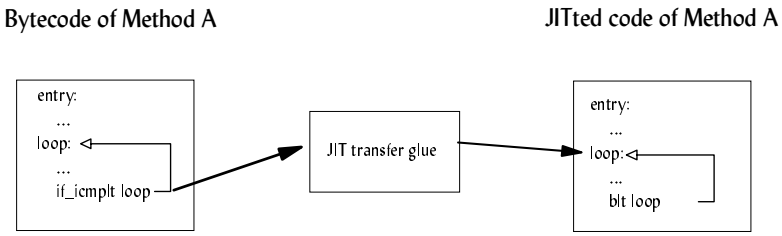
- MMI Loop Thresholds
  - Count < 3 => Profile count unchanged
  - Count < 50 => Profile count decreased
  - Count < 200 => Compile next Invocation
  -
  - Problem:
    - Methods with loops which never exit
      - Symantec benchmark
      - Scanning of last 20 bytecodes

Long-Running Loops

- JIT Compilation with Transfer Point
  - JIT compiler produces two entry points:
    - Method entry
    - Loop backedge
  - Glue code to compute compiler-generated local variables at transfer point
- Direct Control Transfer
  - MMI re-creates stack frame for JIT and jumps to glue code



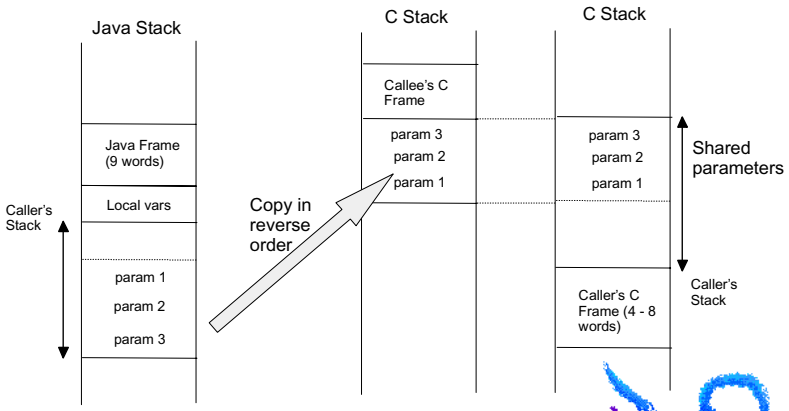
Direct Control Transfer



Implications of Mixed-Mode Execution

- Transfer Between Modes Must be Low-Cost
  - Interpreter and JIT uses different stacks
    - Interpreter uses "JavaStack"
    - JIT uses native stack
  - MMI must share stacks
- Low-Cost Exceptions
  - Interpreter and JIT use different exception handlers
    - Rethrowing of exceptions at boundary
  - MMI must share exception handlers
- High-Speed

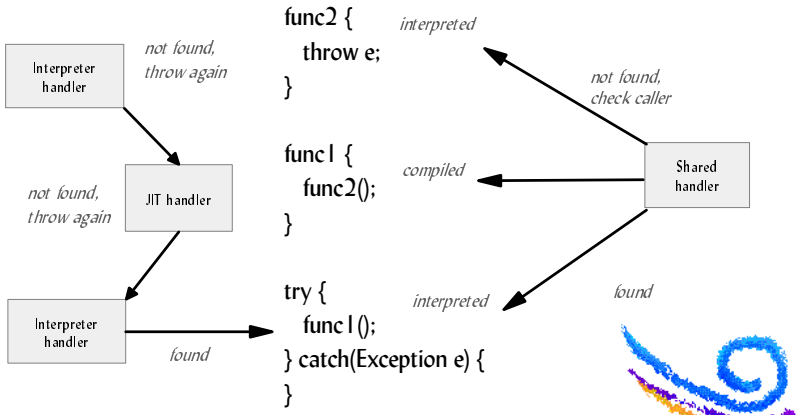
Shared Native Stack



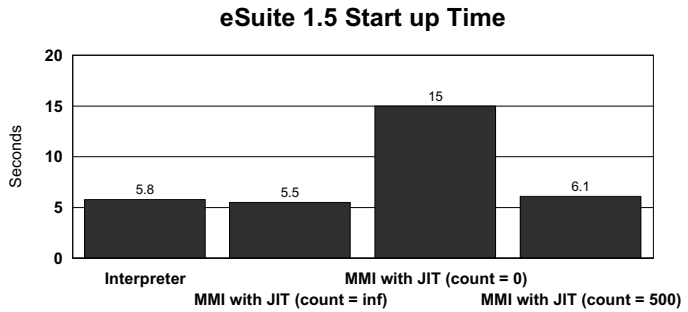
High-Speed

- Hand-written assembler
  - PowerPC approx 14,000 lines
  - Intel approx 18,000 lines
- Architecture tuning
  - PowerPC
    - Bytecode prefetch
    - 'Free' bytecode decode
  - Intel
    - Instruction cache balancing

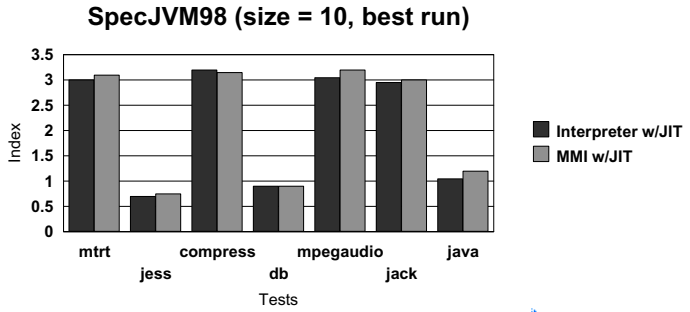
Shared Exception Handler



Start Up Times With MMI



MMI with JIT Performance



Conclusions

- MMI is extremely effective at reducing start-up times
  - MMI avoids JITting all methods
  - MMI delays compilation at load-time
- MMI improves performance
  - MMI is twice as fast as existing interpreter
  - MMI works as an efficient profiler to find JIT sensitive code
  - MMI generates execution profile that enables JIT to apply more effective optimisations