YETI: Gradually Extensible Trace Interpreter

Mathew Zaleski,
University of Toronto

matz@cs.toronto.edu

(thesis proposal)
Overview

- Introduction
  - Background
  - Efficient Interpretation
  - Our Approach to Mixed-Mode Execution
  - Results and Discussion
Why so few JIT compilers?

- Complex JIT infrastructure built in “big bang”, before any generated code can run.
- Rather than incrementally extend the interpreter, typical JITs is built alongside.
- The code generator of current JIT compilers makes little provision to reuse the interpreter.
- The method-orientation of most JITs means that cold code is compiled with hot.

- Interpreters should be more gradually extensible to become dynamic compilers.
Problems with current practice

• Packaging of virtual instruction bodies is:
  • Inefficient: Interpreters slowed by branch misprediction
  • Non-reusable: JIT compilers must implement all virtual instructions from scratch
  • Method orientation of a JIT compiler forces it to compile cold code along with hot.
  • Code compiled cold requires complex runtime to perform late binding if it runs.
  • Recompiling cold code that becomes hot requires complex recompilation infrastructure.
Our Approach

• Branch prediction problems of interpretation can be addressed by calling the virtual bodies.
• Can speed up interpretation significantly.
• Enables generated code to call the bodies.
• JIT need not support all virtual instructions.
• Complexity of compiling cold code can be side stepped by compiling dynamically selected regions that contain only hot code.
• We describe how compiling traces allows us to compile only hot code and link on newly hot regions as they emerge.

▶ Enables gradual enhancement of interpreter
Overview of Contribution

- Callable bodies make for efficient interpretation.
- Reuse of callable bodies from generated code smooths “big bang”.
- A trace oriented JIT compiler is a simple and promising architecture.
Gradual extension of VM

- Larger regions. More instructions compiled.
Result preview - Efficient Interpretation

- Branch misprediction dealt with by calling the bodies from region of generated code.
- Relative to Direct Threaded VM
- Geo mean
  - Java SpecJVM98 benchmarks
  - Ocaml benchmarks

![Subroutine Threading Graph]

Relative to Direct Threading

- java/P4: 0.81
- java/PPC: 0.82
- ocaml/P4: 0.90
- ocaml/PPC: 0.63
Result preview - Trace based JIT

- Geom mean SPECJVM98 relative to Sun Hotspot JIT
- SABVM
  - Selective inlining
- Modified JamVM.
  - TR-LINK = traces, no JIT
  - JIT = trace, JIT
    - Only 50 integer bytecodes
    - Promising start

![Graph showing Java/PPC970 relative to Sun Hotspot]
## Background & Related Work

<table>
<thead>
<tr>
<th>Author(s)</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ertl &amp; Gregg</td>
<td>Branch misprediction</td>
</tr>
<tr>
<td>Piumarta &amp; Riccardi</td>
<td>Selective inlining</td>
</tr>
<tr>
<td>Parrot (perl6)</td>
<td>Callable bodies</td>
</tr>
<tr>
<td>Vitale, Abdelrahman</td>
<td>Catenation, Tcl</td>
</tr>
<tr>
<td>Bala, Duesterwald, Banerjia</td>
<td>Dynamo</td>
</tr>
<tr>
<td>Bruening, Garnett, Amarasinghe</td>
<td>Dynamo Rio</td>
</tr>
<tr>
<td>Whaley</td>
<td>Partial methods</td>
</tr>
<tr>
<td>Gal, Probst, Franz</td>
<td>Hotpath, Trace-based JIT</td>
</tr>
<tr>
<td>Suganuma, Yasue, Nkatani</td>
<td>Region based compilation</td>
</tr>
<tr>
<td>Hozle, Chambers, Ungar</td>
<td>Self</td>
</tr>
<tr>
<td>Many Java, JVM and JIT authors</td>
<td>Java</td>
</tr>
</tbody>
</table>
Overview

• Introduction
  ‣ Background:
    ‣ Dynamo & Traces
    ‣ Interpretation
  • Our Approach to Mixed-Mode Execution
  • Results and Discussion
HP Dynamo

- Trace-oriented dynamic optimization system.
  - HP PA-8000 computers.
- Counter-Intuitive approach:
  - Don’t *execute* optimized binary *interpret* it.
  - Count transits of reverse branches.
  - Trace-generate (next slide).
  - Dispatch traces when encountered.
- Soon, most execution from trace cache.
  - *faster* than binary on hardware of the day!
Trace with if-then-else

```c
//c => b2
if (c)
    b1;
else
    b2;
b3;
```

- Trace is path followed by program
- Conditional branches become trace exits.
- Do *not* expect trace exits to be taken.
Overview

• Introduction
  ‣ Background:
    • Dynamo & Traces
    ‣ Interpretation
  • Efficient Interpretation
  • Our Approach
  • Selecting Regions
  • Results and Discussion
int f(boolean parm) {
    if (parm) {
        return 42;
    } else {
        return 0;
    }
}

int f(boolean);
Code:
0: iload_1
1: ifeq 7
4: bipush 42
6: ireturn
7: iconst_0
8: ireturn
Interpreter

Execution Cycle

- fetch
- dispatch
- execute
- LoadParms

Loaded Program

Internal Representation

Bytecode bodies
Switched Interpreter

vPC = internalRep;

while(1){
    switch(*vPC++){
    case iload_1:
        ...
        break;

    case ifeq:
        ...
        break;

    //and many more..
    }
} }

- slow. Burdened by switch and loop overhead.
Call Threaded Interpreter

```c
void iload_1(){
    //push load 1
    vPC++;
}

void ifeq(){
    //change vPC
    vPC++;
}

static int *vPC = internalRep;

interp(){
    while(1){
        (*vPC)();
    }
};
```

- slow. burdened by function pointer call
Direct Threaded Interpreter

int f(boolean);
Code:

0: iload_1
1: ifeq 7
4: bipush 42
6: ireturn
7: iconst_0
8: ireturn

-Execution of virtual program “threads” through bodies

» Good: one dispatch branch taken per body
int f(boolean);  

Code:  
\[
\begin{align*}
0: & \text{ iload}_1 \\
1: & \text{ ifeq 7} \\
4: & \text{ bipush 42} \\
6: & \text{ ireturn} \\
7: & \text{ iload}_1 \\
8: & \text{ ireturn}
\end{align*}
\]

Virtual PC predicts destination.

Hardware PC insufficient context

Bad: hardware has no context to predict dispatch
Overview

✓ Introduction
✓ Background
  ▸ Efficient Interpretation
  • Our Approach to Mixed-Mode Execution
  • Results and Discussion
int f(boolean parm) {
    if (parm) {
        return 42;
    } else {
        return 0;
    }
}

int f(boolean);
Code:
0: iload_1
1: ifeq 7
4: bipush 42
6: ireturn
7: iconst_0
8: ireturn
Direct Threaded Interpreter

VPC

... iload_1
ifeq 7
bipush 42
ireturn
iconst_0
ireturn
...

DTT

&iload_1
&ifeq
+4
&bipush
42
&ireturn
&iconst_0
&ireturn

iload_1:
  ...
goto *vPC++;

ifeq:
  if () vPC=
goto *vPC++;

bipush:
  ...
goto *vPC++;

Virtual
Program

DTT - Direct
Threading Table

C implementation
of each body

▶ DTT maps vPC to implementation
Context Problem

Virtual PC predicts destination.

Hardware PC insufficient context

Bad: hardware has no context to predict dispatch

int f(boolean);

Code:

0: iload_1
1: ifeq 7
4: bipush 42
6: ireturn
7: iload_1
8: ireturn

ifeq:
   if () vPC=
goto *vPC++;

bipush:
   ...
goto *vPC++;

ireturn:
   ...
goto *vPC++;

iconst_0:
   ...
goto *vPC++;

ireturn:
   ...
goto *vPC++;
Essence of Subroutine Threading

Context Threading Table

<table>
<thead>
<tr>
<th>DTT</th>
<th>CTT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>call iload_1</td>
</tr>
<tr>
<td></td>
<td>call ifeq</td>
</tr>
<tr>
<td></td>
<td>call bipush</td>
</tr>
<tr>
<td></td>
<td>call ireturn</td>
</tr>
<tr>
<td></td>
<td>call iconst_0</td>
</tr>
<tr>
<td></td>
<td>call ireturn</td>
</tr>
</tbody>
</table>

- **iload_1:**
  ```
  ...  
  asm(ret);
  ```

- **ifeq:**
  ```
  vPC=..  
  goto *vPC;
  ```

- Points to generated code
- Virtual branches as direct
- Package bodies as subroutines and call them
- Ret terminated
Context Threading (CT) -- Generating specialized code in CTT

Specialized bodies can also be generated in CTT!
CT Performance

CT is an efficient interpretation technique
Overview

✓ Introduction
✓ Background: Interpretation & traces
✓ Efficient Interpretation
  ▸ Our Approach to Mixed-Mode Execution
    • Selecting Regions
    • Results and Discussion
Gradually Extensible Trace Interpreter

Three main enablers:

1. Bodies organized as callable routines so executable regions can efficiently mix compiled code and dispatched bodies.
2. The DTT can point to variously shaped execution units.
3. Efficient, flexible instrumentation.
1. Bodies are callable

Packaging bytecode bodies as lightweight subroutines means that they can be efficiently called from generated code.

- Needn’t build all virtual instructions all in one shot.
2. DTT always points to implementation

..of corresponding execution unit

DTT indirection enables any shape of execution unit to be dispatched.
3. Flexible, Efficient Instrumentation

A dispatcher describes an execution unit

while(1) { // dispatch loop
    d = vPC->dispatcher;
    pay = d->payload;
    (*d->pre)(vPC,pay,&tcs);
    (*d->body)();
    (*d->post)(vPC,pay,&tcs);
}

Our runtime active before and after each dispatch
Overview

✓ Introduction
✓ Background: Interpretation & traces
  • Our Approach to Mixed-Mode Execution
    • Selecting Regions
      ‣ Basic Blocks
    • Traces
  • Results and Discussion
Basic Block Detection

Java Source

```
int f(boolean parm){
    if (parm){
        return 42;
    } else{
        return 0;
    }
}
```

Java Bytecode

```
int f(boolean);

Code:
0: iload_1
1: ifeq 7
4: bipush 42
6: ireturn
7:  iconst_0
8:  ireturn
```
while(1) { //dispatch loop
    d = vPC->dispatcher;
    pay = d->payload;
    (*d->pre)(vPC, pay, &tcs);
    (*d->body)();
    (*d->post)(vPC, pay, &tcs);
}
Generated code for a basic block

- Could JIT the bb, currently we generate “subroutine threading style” code for it.

Basic block is a run-time superinstruction
C Code for interp function

- all in one C function
- thread private are C local variables
- loader initializes DTT to static dispatchers
- dispatch loop calls instrumentation specific to dispatcher

```c
static t_dispatcher init[256]={};
interp(t_dispatcher *dtt){
    t_dispatcher *vPC = dtt;

    t_thread_context tcs;
    iload:
        //real work of iload here..
        vPC++; //to next instruction
        asm volatile("ret");
    iadd: //and many more bodies

    //dispatch loop
    while(1){
        d = vPC->dispatcher;
        pay = d->payload;
        (*d->pre)(vPC,pay,&tcs);
        (*d->body)();
        (*d->post)(vPC,pay,&tcs);
    }
```
Overview

✓ Introduction
✓ Background: Interpretation & traces
✓ Our Approach to Mixed-Mode Execution
  • Selecting Regions
    ✓ Basic Blocks
    ▶ Traces
  • Results and Discussion
Detecting Traces

- Use Dynamo’s trace detection heuristic.
- Instrument reverse branches until they are hot.
  - Postworker of basic block dispatcher
- Trace generate starting from hot reverse branch:
  - Much like bb’s were recorded
  - Postworker of each basic block region adds each *bb* to thread private history list.
  - Eventually creates new trace dispatcher
  - Hold off generating code until after trace has “trained” a few times.
Trace with if-then-else

//c => b2
if (c)
    b1;
else
    b2;
b3;

- Trace is path followed by program
- Conditional branches become trace exits.
- Do not expect trace exits to be taken.
Subroutine threading style code for a Trace

- Dispatch virtual instructions for trace

```
DTT

bb0
iload_1
ifeq
4

bb1
iconst_0
ireturn

trace-b0-b1
call iload_1
trace_exit_eq
call iconst_0
trace_exit_iret
..caller code..
...

trace exit handlers

TEH
..
ret

TEH
..
ret

Trace is super-super instruction
```
Trace Exits

(a) Two-way: from conditional branches
- one leg on trace
- other leg off trace

(b) Multiway: from invokes and returns
- one leg on trace
- potentially many legs off trace
Trace Exit Handlers

- Code runs when trace exit is taken before return to interpreter
- Record which trace exit has occurred in thread context
- Return to dispatch loop
- Housekeeping roles:
  - flush state of JIT code
  - Trace linking
Trace linking

- When trace exit is hot and destination is a trace
- Rewrite ret at end of trace exit handler as jmp to destination trace
- Only use of code rewriting in system
Trace JIT

• Generate native code for trace exits
  • A lot like branch inlining from CT system.
• Optimize invokevirtual when call and return occur in same trace.
• Naive register allocation scheme
• Only handle 50 integer/object virtual instructions
  • Do virtual instructions one-by-one
    • Relatively easy debugging
• Floating point should be easy.
Overview

✓ Introduction
✓ Background: Interpretation & traces
✓ Our Approach to Mixed-Mode Execution
  ▸ Results and Discussion
    • Data
    • Discussion
    • Remaining Work in this dissertation
Implementation

• Modify JamVM 1.1.3 to be SUB threaded
• Gradually extend it to:
  • Detect, execute subroutine style basic blocks
  • Detect, execute subroutine style traces
  • Link traces
  • Compile traces.
Region Shape

- As execution units become larger
- Trips around dispatch loop become less frequent
- Next show data to justify “step back” approach.
- Very simple experiment:
  - Modify dispatch loops to count iterations.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCT</td>
<td>Direct Call Threading</td>
</tr>
<tr>
<td>BB</td>
<td>CT-style Basic Blocks</td>
</tr>
<tr>
<td>TR</td>
<td>Traces (no link, no JIT)</td>
</tr>
<tr>
<td>TR-LINK</td>
<td>Linked Traces (no JIT)</td>
</tr>
</tbody>
</table>
Region Shape Effect on Dispatch Count

Spec JVM98 Benchmarks

Legend
- DCT
- BB
- TR
- TR-LINK

log dispatch count

1e10
1e8
1e6
1e4
1e2
1e0

compress
db
jack
javac
jess
mpeg
mtrt
ray
scistest
geomean
How efficient is profiling system?

• Run instrumentation without the JIT.
• Are the intermediate versions of Java viable?
• Include SUB threading in comparison:
  • Since it is an efficient dispatch technique.
• Report elapsed time relative to distro of JamVM
Profiling/Instrumentation Overhead

Elapsed time relative to jam-distro

Spec JVM98 Benchmarks

Legend
- SUB
- DCT
- BB
- TR
- TR-LINK

compress 0.73 1.87 0.95 0.78 0.66
db 0.87 1.45 1.05 0.90 0.80
jack 0.88 1.41 1.22 0.88 0.79
javac 0.87 1.38 1.15 0.87 0.77
jess 0.72 2.00 0.99 0.78 0.72
mpeg 0.84 1.11 1.42 0.82 0.81
ray 0.59 1.69 1.18 0.68 0.60
scitest 0.80 1.51 1.15 0.83 0.75
geomean 0.0 0.5 1.0 1.5 2.0
Performance of simple JIT

- Compare YETI performance WITHOUT JIT to selective inlining SableV
- Compare YETI with preliminary trace based JIT to Sun’s Hotspot optimizing compiler
- Not much basis for comparison to Hotpath

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR-LINK</td>
<td>Linked Traces</td>
</tr>
<tr>
<td>SABVM</td>
<td>SableVM selective inlining</td>
</tr>
<tr>
<td>JIT</td>
<td>Traces (JIT and Link)</td>
</tr>
</tbody>
</table>
JIT Performance relative to Sun Hotspot

Spec JVM98 Benchmarks

Legend
- SABVM
- TR-LINK
- JIT
Overview

✓ Introduction
✓ Background: Interpretation & traces
✓ Our Approach
✓ Selecting Regions

• Results and Discussion
  ✓ Data
  ‣ Discussion (and future work)
• Remaining Work in this dissertation
Gradual performance improvement

- Performance improves as effort invested.
- SUB very effective for lightweight bodies
- BB not viable by itself
- TR-LINK about same as CT/branch inlining.
- JIT preliminary.
Discussion

• We have demonstrated how to build an interpreter that is simple and yet as efficient as SableVM and JamVM.
• We have shown that our interpreter can be gradually extended to identify, link and compile traces.
• We have shown how generated code can reuse callable virtual instruction bodies.
• Our JIT, although it has no optimizer, only supports 50 Java virtual instructions, improves performance by 24%.
• More instructions, better performance?
2D vision of Incremental VM lifecycle..

- Basic blocks - just integer virtual instructions
- Traces - compile all virtual instructions
- Traces - just integer instructions
- Traces - sub dispatch

- SUB interpreter

- Have explored this space
Application

- If I had to build a new interpreter.
  - for “lightweight” bodies, so dispatch matters.
- Start with bodies that can be conditionally compiled to be either direct threaded or callable.
- Bring up the system using DCT because the dispatch loop makes it easier to debug.
  - e.g. Logging from dispatch loop is very helpful.
- Primary platforms would run SUB and secondary platforms would run direct threading.
- Gradually extend as described.
Future Work

• Work could go in many different directions.
• Apply to another language system
  • JavaScript? Python? Fortress?
• Deal with polymorphic bytecodes
• Extend JIT/Optimizer
  • Explore performance potential
• Need a lot more infrastructure (e.g. IR)
• Package infrastructure for others to apply.
Overview

✓ Introduction
✓ Background: Interpretation & traces
✓ Our Approach
✓ Selecting Regions
  • Results and Discussion
    ✓ Data
    ✓ Discussion
    ▸ Remaining work in this dissertation

- Infrastructure to measure:
  - Compile time;
  - Proportion of virtual instructions executed from compiled code.

- Add float register class.
  - scimark, ray, Linpack would likely benefit.

- Compile Basic Blocks
  - Long bb benchmarks will benefit.

- Write, write, write.
• Back 12
• Efficient 23
• Our 28
• BB 33
• TR 38
• RES 46
• CONC 54