Goal

- Create a VM that is more easily extended with a just in time (JIT) compiler.
- Enable more languages to see benefits of JIT
- Trying to “reduce impedance mismatch” between interpreter and JIT. (anonymous reviewer)

- Build prototype in Java as proof-of-concept
Goal

- Create a VM that is more easily extended with a just in time (JIT) compiler.
- Enable more languages to see benefits of JIT
- Trying to “reduce impedance mismatch” between interpreter and JIT. (anonymous reviewer)

- Build prototype in Java as proof-of-concept
OUTLINE

- Introduction
  - Why compiling entire methods is hard
- Our Approach
- Background
- Implementation
- Experimental Results.
Run portably by High Level Language Virtual Machine
**Interpreter emulates virtual program**

- **Virtual instruction body** emulates instruction at vPC.
- **Dispatch** is mechanism to transfer control from body to body in virtual program order.
- **Systems often code bodies as cases in a big switch**
Compile Entire Methods

Hot Method

```java
int f(boolean);
Code:
   iload a
   iload b
   iconst 1
   iadd
   iadd
   istore c
```

- To run method must compile every virtual instruction.
### Hot Method

```java
int f(boolean);

Code:
- iload a
- iload b
- iconst 1
- iadd
- iadd
- istore c
```

### Native code

```
01010101110101
11010101110100
10101010111011
00010101110100
111010101110111
01010101110101
11010101110100
10101010111011
00010101110100
111010101110111
```

- To run method must compile every virtual instruction.
Method based compilation and cold code

Hot method

```java
fhot(){
    if(c){
        new Hot();
        h.hot();
    }else{
        new Cold();
        c.cold();
    }
}
```

JIT compiled code

```
01010101110101
11010101110100
10101010111011
00010101110100
111010101110111
01010101110101
11010101110100
10101010111011
00010101110100
111010101110111
```

- Cold portions of hot methods complicate runtime
Method based compilation and cold code

Hot method

```java
fhot()
    if (c) {
        new Hot();
        h.hot();
    } else {
        new Cold();
        c.cold();
    }
```

JIT compiled code

01010101110101
11010101110100
10101010111011
00010101110100

- Cold portions of hot methods complicate runtime
Method based compilation and cold code

Hot method

```java
fhot()
{
  if(c) {
    new Hot();
    h.hot();
  }
  else {
    new Cold();
    c.cold();
  }
}
```

JIT compiled code

- Cold portions of hot methods complicate runtime
Method based compilation and cold code

Hot method

```java
fhot(){
  if(c){
    new Hot();
    h.hot();
  }else{
    new Cold();
    c.cold();
  }
}
```

JIT compiled code

- `resolve Cold`
- `invoke c.cold`

- Cold portions of hot methods complicate runtime
Method based compilation and cold code

Hot method

```java
fhot(){
  if(c){
    new Hot();
    h.hot();
  }else{
    new Cold();
    c.cold();
  }
}
```

JIT compiled code

```
01010101110101
11010101110100
10101010111011
00010101110100
```

- Cold portions of hot methods complicate runtime

- resolve Cold
- invoke c.cold
- PY_ADD b,c
Integrate JIT & interpreter more closely

| Hot virtual code | Translated code |

```java
int f(boolean);
Code:
  iload a
  iload b
  iconst 1
  iadd
  iadd
  istore c
```

- Compile only subset of virtual instructions
Integrate JIT & interpreter more closely

Hot virtual code

```java
int f(boolean);
Code:
  iload a
  iload b
  iconst 1
  iadd
  iadd
  istore c
```

Translated code

```java
mov $1,(%vsp)
inc %vsp
```

- Compile only subset of virtual instructions
Integrate JIT & interpreter more closely

Hot virtual code

int f(boolean);
Code:
   iload a
   iload b
   icast 1
   iadd
   iadd
   istore c

Translated code

   iload
   call iload
   call iload
   mov $1,(%vsp)
   inc %vsp
   call iadd
   call iadd
   call istore c

- Compile only subset of virtual instructions
Avoid Cold Code

```java
fhot(){
    if(c){
        new Hot();
        h.hot();
    }else{
        new Cold();
        c.cold();
    }
}
```

- Compiling hot paths avoids problems of cold code
Avoid Cold Code

Suppose c is usually true

```java
fhot()
{
  if(c){
    new Hot();
    h.hot();
  }else{
    new Cold();
    c.cold();
  }
}
```

- Compiling hot paths avoids problems of cold code
Avoid Cold Code

fhot()
{
    if(c)
    {
        new Hot();
        h.hot();
    }
    else
    {
        new Cold();
        c.cold();
    }
}

- Compiling hot paths avoids problems of cold code
Avoid Cold Code

Compiling hot paths avoids problems of cold code
Avoid Cold Code

\[ \text{fhot()} \{ \]
\[ \quad \text{if(c)} \{ \]
\[ \quad \quad \text{new Hot();} \]
\[ \quad \quad \text{h.hot();} \]
\[ \} \text{else} \{ \]
\[ \quad \quad \text{new Cold();} \]
\[ \quad \quad \text{c.cold();} \]
\[ \} \]

- Compiling hot paths avoids problems of cold code

Translated path

\[ \text{c ifne exit new Hot invoke h.hot() new Cold invoke c.cold()} \]
We need

- Interpreter with callable virtual instruction bodies.
- Profiling infrastructure to identify hot paths.
- A way to dispatch compiled regions.
- A JIT that can generate code for some virtual instructions and fall back on emulation for others.
OUTLINE

• Introduction
• Background
  • Direct Call Threading (interpreter)
  • Dynamo Trace Selection (hot paths)
• Implementation
• Experimental Results.
**Direct Call Threaded Interpreter**

Interpreter function:

```c
interp(t_vpc *vPC) {
    vPC = rep;
    while(1)
        (*vPC)();

    iload:
        //push local *vPC++
        vPC++;
    iconst:
        asm ("ret"); //x86
    iadd:
    istore:
...```

Body also can be called from code generated by JIT
Direct Call Threaded Interpreter

```
interp(t_vpc *vPC){
  vPC = rep;
  while(1)
    (*vPC)();
}

iload:
  //push local *vPC++
  vPC++;
  asm ("ret");//x86
iconst:
iadd:
istore:
```

Body also can be called from code generated by JIT
Direct Call Threaded Interpreter

```
interp(t_vpc *vPC){
vPC = rep;
while(1)
    (*vPC)();

    iload:
        //push local *vPC++
        vPC++;
    iconst:
    iadd:
    iadd:
    istore:
    ..
```

Body also can be called from code generated by JIT
Direct Call Threaded Interpreter

Body also can be called from code generated by JIT
Direct Call Threaded Interpreter

Body also can be called from code generated by JIT
Direct Call Threaded Interpreter

**Code Example:**

```
interp(t_vpc *vPC){
  vPC = rep;
  while(1) {
    (*vPC)();
  }
  iload: //push local *vPC++
  vPC++;
  asm ("ret"); //x86
  iconst:
iadd:
istore:
```

- **iload a**
- **iload b**
- **iconst 1**
- **iadd**
- **iadd**
- **istore c**

» Body also can be called from code generated by JIT
Dynamo Traces

Traces are interprocedural paths through program

for(;;){
    if (c){
        b1;
    } else {
        b2;
    }
    b3;
}

hot reverse branch hint that hot loop body follows
Dynamo Traces

```
for(;;){
    if (c){
        b1;
    } else {
        b2;
    }
    b3;
}
```

▸ Traces are interprocedural paths through program
Dynamo Traces

Traces are interprocedural paths through program

for(;;){
    if (c){
        b1;
    } else {
        b2;
    }
    b3;
}
Traces are interprocedural paths through program
for(;;){
  if (c){
    b1;
  } else {
    b2;
  }
  b3;
}
Traces are interprocedural paths through program
Synopsis of our Approach

- Everything is callable

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

 fhot()
    { 
        flg = x || y
        if(flg)
            { 
                new Hot();
                h.hot();
            }
        else{
            new Cold();
            c.cold();
        }
    }
```
Synopsis of our Approach

Everything is callable

```c
Interpreter

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

fhot(){
    flg = x || y
    if(flg){
        new Hot();
        h.hot();
    }else{
        new Cold();
        c.cold();
    }
}
```
Synopsis of our Approach

- Everything is callable

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

fhot()
{
    flg = x || y
    if(flg)
    {
        new Hot();
        h.hot();
    }
    else
    {
        new Cold();
        c.cold();
    }
}
```
Synopsis of our Approach

Everything is callable

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

fhot()
{
    flg = x || y
    if(flg)
    {
        new Hot();
        h.hot();
    }
    else
    {
        new Cold();
        c.cold();
    }
}
```
OUTLINE

• Introduction
• Implementation
  • Linear Blocks
  • Traces
  • Simple Trace JIT
• Experimental Results.
Trace Compilation - 3 stage process

1. Dispatch instructions, identify *linear blocks* (LB)
   - LB is a sequence of virtual instructions, ending with branch.

2. Dispatch linear blocks, identify traces.
   - A trace is a sequence of linear blocks.

3. JIT compile hot traces.
   - Compile only selected virtual instructions.

- Prototype built on top of Lougher’s JamVM 1.3.3
1. Dispatch instructions, Identify Linear Blocks

```c
interp()
{
    while(1)
    {
        pre_work(vPC);
        (*vPC)();
        post_work(vPC);
    }
};

fhot()
{
    c = a + b + 1;
    if(c)
    {
        new Hot();
        h.hot();
    }
    else
    {
        new Cold();
        c.cold();
    }
}
```

- When branch reached the history list contains LB
1. Dispatch instructions, Identify Linear Blocks

```cpp
interp()
{
    while(1)
    {
        pre_work(vPC);
        (*vPC)();
        post_work(vPC);
    }
}

fhot()
{
    c = a + b + 1;
    if(c)
    {
        new Hot();
        h.hot();
    }
    else{
        new Cold();
        c.cold();
    }
}

When branch reached the history list contains LB
```
1. Dispatch instructions, Identify Linear Blocks

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

```
fhot(){
c = a + b + 1;
if(c){
    new Hot();
h.hot();
}else{
    new Cold();
c.cold();
}
}
```

- When branch reached the history list contains LB
1. Dispatch instructions, Identify Linear Blocks

```cpp
fhot()
{
  c = a + b + 1;
  if(c){
    new Hot();
    h.hot();
  }else{
    new Cold();
    c.cold();
  }
}

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

- When branch reached the history list contains LB
Use History List to generate LB

New region body will run from now on
Use History List to generate LB

- New region body will run from now on

- LB region body is subroutine threaded

```java
rep
iload
a
iload
b
iconst
1
iadd
iadd
istore
c
goto
42
```

```java
iload:
generated code

- call iload
- call iload
- call iload
- call iload
- call iload
- call iload
- call iload
- call iload
- call iload
- call goto
- ret
```
Use History List to generate LB

New *region body* will run from now on

- **lb**
  - a
  - iload
  - b
  - iconst
    - 1
  - iadd
  - iadd
  - istore
  - c
  - goto
  - 42

- **generated code**
  - call iload
  - call iload
  - call iconst
  - call iadd
  - call istore
  - call goto
  - ret

**LB region body**
- is subroutine
- threaded

**install in slot of LB entry**
Execute LB

```
while(1) {
    pre_work(vPC);
    (*vPC)();
    post_work(vPC);
}
```

- vPC set by region body
Execute LB

```
while(1){
    pre_work(vPC);
    (*vPC)();
    post_work(vPC);
}
```

- vPC set by region body
### Execute LB

**vPC set by region body**

<table>
<thead>
<tr>
<th>rep</th>
<th>lb</th>
<th>a</th>
<th>iload</th>
<th>b</th>
<th>iconst</th>
<th>1</th>
<th>iadd</th>
<th>iadd</th>
<th>istore</th>
<th>c</th>
<th>goto</th>
<th>42</th>
</tr>
</thead>
</table>

```
while(1) {
    pre_work(vPC);
    (*vPC)();
    post_work(vPC);
}
```

<table>
<thead>
<tr>
<th>generated code</th>
</tr>
</thead>
<tbody>
<tr>
<td>call iload</td>
</tr>
<tr>
<td>call iload</td>
</tr>
<tr>
<td>call iload</td>
</tr>
<tr>
<td>call iload</td>
</tr>
<tr>
<td>call iload</td>
</tr>
<tr>
<td>call iload</td>
</tr>
<tr>
<td>call iload</td>
</tr>
<tr>
<td>call iload</td>
</tr>
<tr>
<td>call iload</td>
</tr>
<tr>
<td>call iconst</td>
</tr>
<tr>
<td>call iadd</td>
</tr>
<tr>
<td>call iadd</td>
</tr>
<tr>
<td>call istore</td>
</tr>
<tr>
<td>call goto</td>
</tr>
<tr>
<td>call goto</td>
</tr>
<tr>
<td>call goto</td>
</tr>
<tr>
<td>ret</td>
</tr>
</tbody>
</table>
Execute LB

```
while(1)
{
    pre_work(vPC);
    (*vPC)();
    post_work(vPC);
}
```

- vPC set by region body
2. Run LB, identify traces

```
//c mostly false
if (c) {
    b1;
} else {
    b2;
}
```

- LB’s in trace recorded in history list
2. Run LB, identify traces

```c
//c mostly false
if(c){
    b1;
} else {
    b2;
}
b3;
```

LB’s in trace recorded in history list
2. Run LB, identify traces

```c
//c mostly false
if(c){
    b1;
} else {
    b2;
}
b3;
```

- LB’s in trace recorded in history list
Use history list to generate trace

▶ Trace predicts path through virtual program
Use history list to generate trace

- Trace predicts path through virtual program
Use history list to generate trace

Trace predicts path through virtual program
Use history list to generate trace

Trace predicts path through virtual program
Use history list to generate trace

Trace predicts path through virtual program
Details of an (Interpreted) Trace

Interpreted traces run on PPC, x86 (June).

```
call
call
call ifne
cmpl vPC,b2
jne TEH_c
call
call
call
texit..
```

```
TEH_C
..
..
ret;
```
Details of an (Interpreted) Trace

"Interpreted" - all real work done in bodies

Interpreted traces run on PPC, x86 (June).
Details of an (Interpreted) Trace

“Interpreted” - all real work done in bodies

Compare vPC to hardwired address of the on trace destination (b2|b3)

Interpreted traces run on PPC, x86 (June).
Details of an (Interpreted) Trace

- Interpreted traces run on PPC, x86 (June).

"Interpreted" - all real work done in bodies

Compare vPC to hardwired address of the on trace destination (b2|b3)

Trace Exit Handler (TEH) does housekeeping, then returns to dispatch loop

- Interpreted traces run on PPC, x86 (June).
Details of an (Interpreted) Trace

Interpreted traces run on PPC, x86 (June).

- Interpreted” - all real work done in bodies
- Compare vPC to hardwired address of the on trace destination (b2|b3)
- Trace Exit Handler (TEH) does housekeeping, then returns to dispatch loop
- Rewrite as branch to link traces

Generated code:
- call
- call
- call ifne
- cmpl vPC,b2
- jne TEH_c
- call
- call
- texit..

“Interpreted” block:
- TEH_C
- ...
- ...
- ret;
3. JIT compiled Trace

Traces are easy to compile (PPC only)

generated code

<table>
<thead>
<tr>
<th>call</th>
<th>100101000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>011001100</td>
</tr>
<tr>
<td></td>
<td>011001011</td>
</tr>
<tr>
<td>bne  TEH_C</td>
<td></td>
</tr>
<tr>
<td>101010101</td>
<td></td>
</tr>
<tr>
<td>100101000</td>
<td></td>
</tr>
<tr>
<td>011001100</td>
<td></td>
</tr>
<tr>
<td>texit..</td>
<td></td>
</tr>
</tbody>
</table>
3. JIT compiled Trace

- Traces are easy to compile (PPC only)

- Some virtual instructions still emulated

- Generated code:
  - `call 100101000 011001100 011001011 bne TEH_C 101010101 l00101000 011001011 texit..`
  - `TEH_C
    ..
    ..
    ret;`
3. JIT compiled Trace

- Traces are easy to compile (PPC only)

Some virtual instructions still emulated

No control flow merges.
No forward branches.

Generated code:
- call
- 100101000
- 011001100
- 011001011
- bne TEH_C
- 101010101
- 100101000
- 011001100
- texit..

- TEH_C
  -..
  -..
  - ret;
3. JIT compiled Trace

- Traces are easy to compile (PPC only)

- Some virtual instructions still emulated

- No control flow merges.
- No forward branches.

- Trace Exit Handler (TEH) flushes any values in registers to stack.

---

- generated code:
  - call
  - 100101000
  - 011001100
  - 011001011
  - bne TEH_C
  - 101010101
  - 100101000
  - 011001100
  - texit..

- TEH_C:
  - ..
  - ..
  - ret;
Simple Trace JIT

- Much simpler than Jikes style baseline compiler.
  - No control flow merges, no forward branches.
- Optimize virtual method invocation:
  - Exploit fact that traces are interprocedural.
  - Convert to trace exit - check class of invoked upon object.
- Similar effect to polymorphic inline cache.
Choose which instructions to compile

- Attempt only selected virtual instructions:
  - All conditional branches e.g. `ifnull`
  - 50 integer and object instructions e.g. `iadd`
- Can address specific performance challenges.
  - i.e. compile a few bytecodes that really matter.
- Or avoid compiling nasty corner cases..
  - i.e. bail on a instruction when going gets tough
OUTLINE

• Introduction
• Implementation
• Experimental Results.
Experiments

- Suppose Direct Call Threading, LB, Traces, etc were incremental deployments of new VM.
- Would performance improve?
- Was development sufficiently incremental?
Experiments

• Suppose Direct Call Threading, LB, Traces, etc were incremental deployments of new VM.
  • Would performance improve?
  • Was development sufficiently incremental?

• Performance Evaluation:
  • Compare elapsed time to distro JamVM.
  • Average of time `-p` over three runs.
  • 2 CPU, 2 GHz PPC970 under OSX 10.4
Experiments

- Suppose Direct Call Threading, LB, Traces, etc were incremental deployments of new VM.
  - Would performance improve?
  - Was development sufficiently incremental?
- Performance Evaluation:
  - Compare elapsed time to distro JamVM.
  - Average of \texttt{time -p} over three runs.
  - 2 CPU, 2 GHz PPC970 under OSX 10.4
  - Benchmarks suite is the SPECjvm98 + scimark.
Virtual instructions emulated per dispatch

SPECjvm98 + scitest

- LINEAR BLOCK
- TRACE
- TRACE-LINK
Virtual instructions emulated per dispatch

<table>
<thead>
<tr>
<th>Geometric Mean (virtual instructions executed per region dispatch)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^5</td>
</tr>
<tr>
<td>10^4</td>
</tr>
<tr>
<td>10^3</td>
</tr>
<tr>
<td>10^2</td>
</tr>
<tr>
<td>10^1</td>
</tr>
</tbody>
</table>

**LINEAR BLOCK**

**TRACE**

**TRACE-LINK**

**SPECjvm98 + scitest**

**geomean**
Virtual instructions emulated per dispatch

- The dynamic average LB executes 6.3 virtual instructions between branches.

![Diagram showing virtual instructions executed per region dispatch](chart)

geomean
SPECjvm98 + scitest
Virtual instructions emulated per dispatch

- The dynamic average LB executes 6.3 virtual instructions between branches.
- Traces with linking disabled execute about 5 LB’s before trace exiting.
- 35 virtual instructions meaty enough to optimize.

<table>
<thead>
<tr>
<th></th>
<th>LINEAR BLOCK</th>
<th>TRACE</th>
<th>TRACE-LINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^5</td>
<td>6.3</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

geomean SPECjvm98 + scitest
The dynamic average LB executes 6.3 virtual instructions between branches.

Traces with linking disabled execute about 5 LB’s before trace exiting.

35 virtual instructions meaty enough to optimize.

Trace linking closes loop nests explaining strong effect.
Virtual instructions emulated per dispatch

SPECjvm98 + scitesc (sorted by LB)
Elapsed Time Relative to Direct Threading

- DIRECT CALL THREADING
- LINEAR BLOCKS
- TRACE
- TRACE-LINK
- JIT

geomean
SPECjvm98 + scitest
- Direct Call Threading about as fast as switch (on PPC).
Elapsed Time Relative to Direct Threading

- Direct Call Threading about as fast as switch (on PPC).
- LB faster as predicts straight-line dispatch perfectly.
Elapsed Time Relative to Direct Threading

- Direct Call Threading about as fast as switch (on PPC).
- LB faster as predicts straight-line dispatch perfectly.
- “Interpreted”, linked traces outperform direct threading.
- 4% faster than SableVM
Elapsed Time Relative to Direct Threading

- Direct Call Threading about as fast as switch (on PPC).
- LB faster as predicts straight-line dispatch perfectly.
- “Interpreted”, linked traces outperform direct threading.
- 4% faster than SableVM
- Simple trace JIT 32% faster
- Almost 2x direct threading.
- Hotspot still 4x faster.
Elapsed Time Relative to Direct Threading

- DIRECT CALL THREADING
- LINEAR BLOCKS
- TRACE-LINK
- JIT
- TRACE

SPECjvm98 + scitest

data for various benchmarks and performance analyses.
• Our approach offers more deployable milestones.
  ‣ A more gradual approach to building a mixed-mode system.
Our approach offers more deployable milestones.

- A more gradual approach to building a mixed-mode system.
Future

• Apply techniques to new languages
  • i.e. ones with no JIT.
  • Apply dynamic compilation to linear regions of run time typed languages.
  • Speculatively optimize polymorphic bytecodes (e.g. string, int, float add in Python).
• Investigate how a new shape of dynamic compilation unit might be built from the network of linked traces.
End