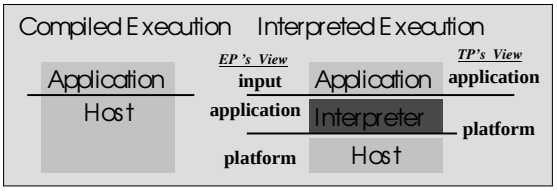
Performance Measurement of Interpreted Programs

Tia Newhall
newhall@cs.wisc.edu
Computer Sciences
University of Wisconsin
1210 W. Dayton St.
Madison, WI 53706

Interpreted Execution



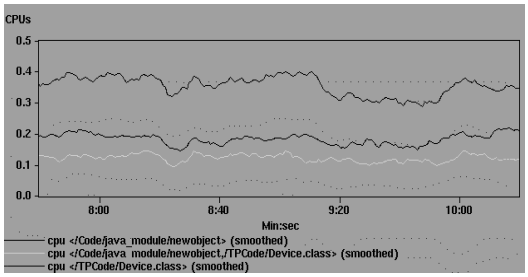
Interdependence Between TP & EP:

- Interpreter determines application's performance
- Application determines what EP code is executed

Performance Data

- That describes EP-TP interaction
  - time to fetch and decode instructions from TP method `foo`
- That allows:
  - TP developer to see inside EP and understand fundamental costs
  - EP developer to characterize performance in terms of TP workloads

Data that Explicitly Represents EP-TP interaction

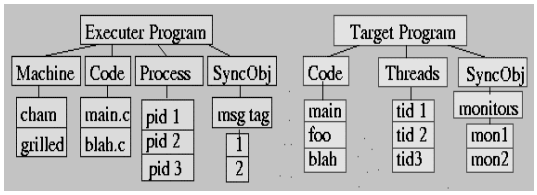


Outline

- Model for representing performance data
- Implementation:
  Measuring interpreted Java
- Results
- Conclusions

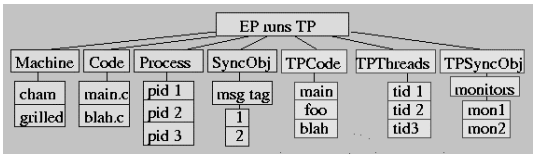
Interpreted Execution Model

- EP & TP are program executions



Interpreted Execution Model

- Relationship between EP & TP's execution: EP runs TP
  - EP resources interact w/ TP resources
  - EP runs TP: union of EP & TP resources



Representing Performance Data

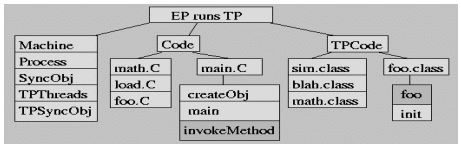
- Want:
  - from any part of execution of TP by EP: selectively constrain performance data to any part of execution
    amount of time for EP to fetch TP function foo code
  - in language that both the TP & EP developer can understand
- 2 mechanisms to constrain data: Metric & Focus
  - EP-specific metrics for TP developer
  - EP–TP foci for EP developer

Constraints

- Constraint: resource selection, restriction of hierarchy to sub-set of its resources
  - /Code/main.c/foo restricted to function foo
- "Active" is resource-specific
  - active process vs. active function
- Constraint Function: true when resource active
  - Code.constrain(/Code/main.C/foo)

Focus

- One constraint from each hierarchy
  </Machine, /Code/main.C/invokeMethod, /Process, /SyncObj, /TPCode/foo.class/foo, /TPThreads, /TPSyncObj>

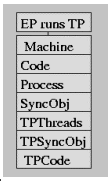


- Focus Active: AND of constraint functions
  Code.constrain(/Code/main.C/invokeMethod)
  AND TPCode.constrain(/TPCode/foo.class/foo)

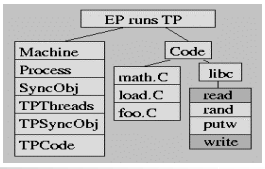
Metrics

•Functions of timer, counter, constraints & constraint operators (AND, OR, RANGE)

CPUtime:
[ ] processTime/sec



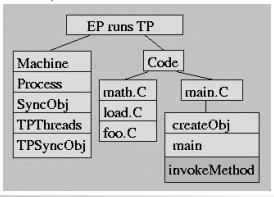
io_wait:
[Code.constrain(/Code/libc/read) **OR** Code.constrain(/Code/libc/write)]wallTime/sec



EP-Specific Metrics

- Encode information about EP program in metric
- For TP developer

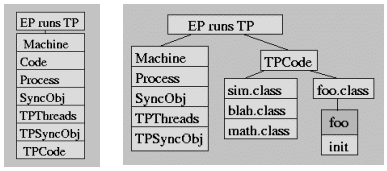
methodCallTime:
[Code.constrain(/Code/main.C/invokeMethod)] processTime/sec



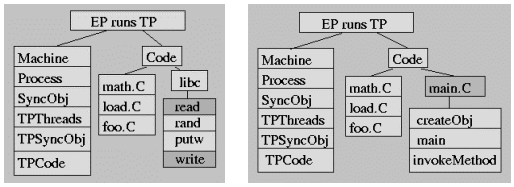
Representing Performance Data

• Metric AND Focus

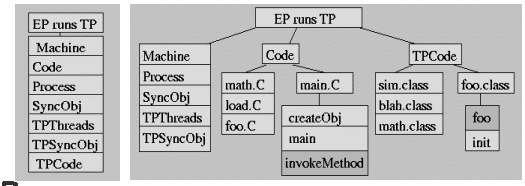
CPUTime, </TPCode/foo.class/foo>:
[] processTime/sec
AND [TPCode.constrain(/TPCode/foo.class/foo)]



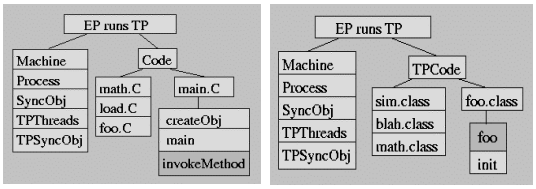
io_wait, </Code/main.C>:
[Code.constrain(/Code/libc/read) OR
Code.constrain(/Code/libc/write)] wallTime/sec
AND [Code.constrain_caller(/Code/main.C)]



CPUTime,
</Code/main.c/invokeMethod, /TPCode/foo.class/foo>:
[] processTime/sec
AND
[Code.constrain(/Code/main.C/invokeMethod) AND
TPCode.constrain(/TPCode/foo.class/foo)]



MethodCallTime, </TPCode/foo.class/foo>:
[Code.constrain(/Code/main.C/invokeMethod)] processTime/sec
AND
[TPCode.constrain(/TPCode/foo.class/foo)]



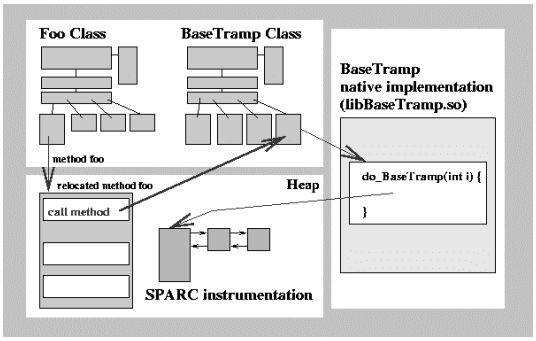
Measuring Interpreted Java

- Implemented for Sun's 1.0.2 Java VM
- Dynamically insert Java byte-code, and SPARC instrumentation into application and VM
  requires no changes to Java application or VM prior to execution
- Issues:
  - instrumenting VM code and Java byte-code in same image
  - Java .class files are demand loaded
  - Java IS has no instructions to save/restore execution stack frame or create new operand stack for instrumentation code

Solution: Transformational Instrumentation

Transformational Instrumentation

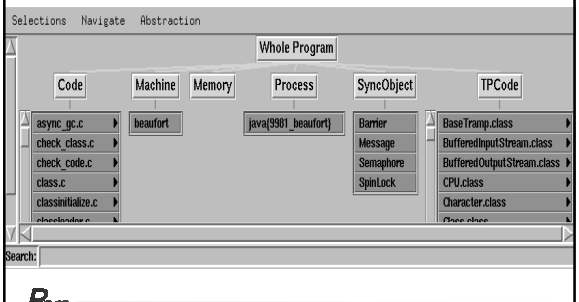
- Relocate Java method function to heap, expand w/ nops at instrumentation points
- Branch from original method to relocated one
- Generate SPARC instrumentation in heap: use native methods to execute it from Java application
- Insert method call instruction at instrumentation pts: forces VM to create new execution and operand stacks for instrumentation code



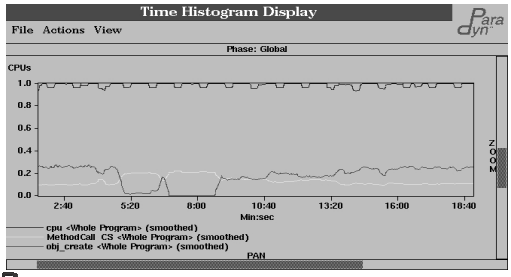
Performance Tuning Study

- CPU-simulator Java Application Program
  - 1200 lines of Java source code
  - 11 Java class files

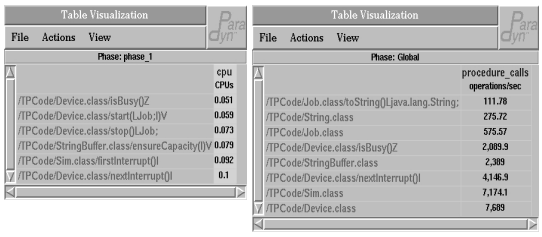
Where Axis with TPCode



Large amount of CPU time due to method call and object creation overheads

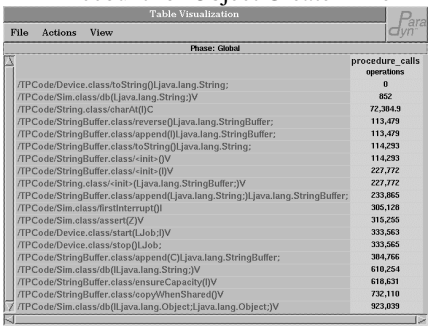


Account for Method Call time



⇒ in-line calls to Device.isBusy() & nextInterrupt()

Account for Object Create Time



=> reduce number of String object creates

Results

Optimization	Number of Method calls	Number of Obj. creates	Execution Time
Original Version	13,373,200	465,140	389.29 seconds
Method In-lining	9,213,400 (31% less)	465,140	343.99 (-12%)
Fewer Obj. Creates	9,727,800 (27% less)	17,350 (96% less)	234.13 (-40%)
Both	5,568,100 (58% less)	17,350	113.60 (-70%)

Conclusion

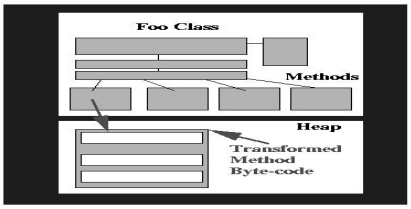
- Performance Data for interpreted executions
  - that explicitly represents TP-EP interaction
  - can represent any part of EP's execution of TP
  - in language for both TP & EP developer
- More Information:
  ftp://grilled.cs.wisc.edu/technical_papers/javaperf.ps.Z

More Metrics

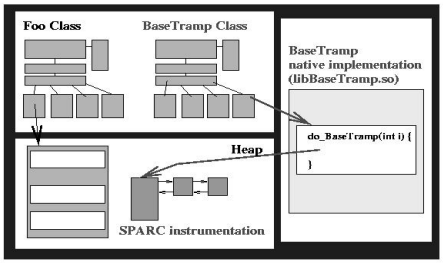
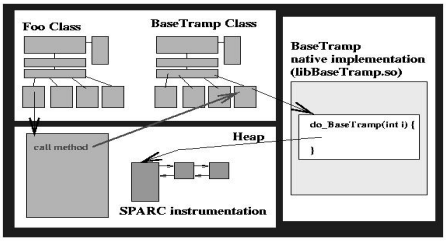
```
send_recv:
  [Code.constrain(/Code/libpvm/pvm_send/entry)
  : Code.constrain(/Code/libpvm/pvm_recv/exit)]
  processTime/sec

send_recv, </Process/pid_1>:
  [Code.constrain(/Code/libpvm/pvm_send/entry)
  : Code.constrain(/Code/libpvm/pvm_recv/exit)]
  processTime/sec
  AND  Process.constrain(/Process/pid_1)
```


Transformational Instrumentation

- Relocate Java method function to heap, expand w/ nops at instrumentation points
- Branch from original method to relocated one



- Generate SPARC instrumentation in heap: use native methods to execute it from Java application



- Insert method call instruction at instrumentation pts: forces VM to create new execution and operand stacks for instrumentation code