The Ptolemy C Code Generator

Ankush Varma
Shuvra Bhattacharyya
University of Maryland, College Park

5th Biennial Ptolemy Miniconference
Berkeley, CA, May 9, 2003

Why C?

• Fast
  – fewer runtime checks.
  – no JVM overhead.

• Small
  – full library not needed.

• Retargetability
  – ANSI C source code is portable.
  – No JVM needed.
  – Can run on embedded systems with no JVM and no OS.

• More optimized
  – C compilers have highly configurable, well-studied optimizations.
Cracking the Code

• **Problem:** Java Bytecode is stack-based. C uses local variables.
  – Use Soot to unroll the stack into local variables while transforming it into intermediate representation.

• **Problem:** Java is Object-Oriented. C is not.
  – Use hashing to give each method/field a different name.
  – Use structures for Objects/Classes.
  – Dare to use indirect function pointers (scary, but it works).

• **Problem:** Java relies heavily on exceptions. C has no support for exceptions.
  – Build support for exceptions in C using setjmp-longjmp.

• **Problem:** Java has automatic garbage collection, C does not
  – Use a C-based GC as a plugin.

• **Problem:** Java relies on JVM or OS for some functionality
  – Build a runtime library to provide the required functionality.

The Dark Art of Java-to-C Compilation

```
<table>
<thead>
<tr>
<th>Java Code</th>
<th>Code Generator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portable Bytecode</td>
<td>Libraries, C code and headers</td>
</tr>
<tr>
<td>( class file )</td>
<td>C Compiler</td>
</tr>
<tr>
<td>Soot</td>
<td>Automatically generated makefile</td>
</tr>
<tr>
<td>Jimple Intermediate Representation</td>
<td>Executable Application</td>
</tr>
<tr>
<td>Optimizations</td>
<td></td>
</tr>
</tbody>
</table>
```

C Code Generation 3

C Code Generation 4
Trimming the Tree

- Set up MethodCallGraph
- Start with main class and all its methods as required.
- Start worklist-based algorithm
  - If you see a class:
    • look at its initialization methods.
    • look at its superclasses.
  - If you see a field:
    • look at the class declaring it.
    • look at the class of its type.
  - If you see a method:
    • look at all fields it references.
    • look at all methods it calls.
    • look at its class.
    • look at the classes of exceptions it throws/catches.

Strategies for Speeding Up Compilation

- Create single untrimmed shared static library
  - too simplistic.
  - generating library takes too long.
  - code bloat.
- Generate separate library trimmed for each application
  - still takes a long time.
  - mostly spending time figuring out polymorphic method calls within library classes.
Strategies for Speeding Up Compilation (continued)

- Divide code into application and library classes:
  - cache library method targets in a disk file.
  - compile-time analysis is simplified.
  - extensible to any library classes (Ptolemy?)

Faster ...

- 

![Graph showing performance improvement for different methods using various compilers and techniques.](image-url)
... and much, much smaller.

Roadmap

- Automatic makefile generation.
- Generated fully ANSI-compliant C.
- Inheritance.
- Exceptions.
- Multidimensional arrays.
- Interfaces.
- Automatic Garbage Collection.
- Plugging in to ptolemy.
- Further generic optimizations.
- Further ptolemy-specific optimizations.
- Run java code on a DSP.