ANALYSIS AND OPTIMIZATION OF PARADIGM MICROPROGRAMS

Victor Winter, James McCoy, Dominic Montoya, and Greg Wickstrom
OUTLINE

• Motivation – The SCore Processor
• Microprogramming Concepts Related to Analysis and Optimization
• Paradigm Analysis
  • Views
  • Heat Maps
  • Estimation Tables
• Paradigm Optimization
  • User-defined Optimization Rules
  • Paradigm’s Timing Constraint Language
The SCore Processor

MOTIVATION
THE SCORE PROCESSOR

- The **Scalable Core (SCore)** is a hardware implementation of a subset of the JVM, designed and developed at **Sandia National Laboratories (SNL)**, for use in resource constrained high-consequence embedded systems.

- Within the Score:
  - **Supported Java bytecodes** are realized in microcode.
  - **Native methods** used in the JVM and supported by the SCore are also implemented in microcode.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Relevant Keywords</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating point</td>
<td>strictfp, float, double</td>
<td>unsupported</td>
</tr>
<tr>
<td>Threading</td>
<td>synchronized, volatile</td>
<td>unsupported</td>
</tr>
<tr>
<td>Serialization</td>
<td>transient</td>
<td>unsupported</td>
</tr>
<tr>
<td>Assertions</td>
<td>assert</td>
<td>unsupported</td>
</tr>
<tr>
<td>Multi-dimensional Arrays</td>
<td></td>
<td>unsupported</td>
</tr>
</tbody>
</table>
## VM RESTRICTIONS

<table>
<thead>
<tr>
<th>Feature</th>
<th>Relevant Keywords</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native methods</td>
<td>native</td>
<td>Limited support</td>
</tr>
<tr>
<td>Garbage collection</td>
<td></td>
<td>Limited support</td>
</tr>
<tr>
<td>Reflection</td>
<td></td>
<td>Unsupported</td>
</tr>
<tr>
<td>(dynamic) class loading</td>
<td></td>
<td>Unsupported</td>
</tr>
</tbody>
</table>
Concepts related to analysis and optimization

MICROPROGRAMMING
MICROPROGRAMMING

• The purpose of microprogramming is to orchestrate the behavior of resources in a CPU.
  • It is a software-based alternative to hardware-based logic boxes.

• A microprogram
  • is a specification of how the resources within a CPU are to be controlled.
  • consists of a sequence of microinstructions.

• A microinstruction consists of a set of microoperations.

• A microoperation consists of a set of fields.

• A field consists of one or more bits whose binary values correspond to control lines.
MICROCODE OPTIMIZATION

\[ W \times B \]
MICROCODE OPTIMIZATION

- **High-level** – discover parallelism inherent in the algorithm (includes compiler optimizations)

- **Low-level** – maximize the parallelism of each microinstruction (i.e., concurrent execution of microoperations)
  - Terms: local/global compaction, packing
  - Notes: NP-complete
Views, Heat Maps, Estimation Tables

PARADIGM ANALYSIS
View

- A view shows the change register usage between a selected set of methods.
- For example, between the methods `mult32` and `+c`, the Paradigm compiler locally allocates 8 registers.
Heat Map

- 12 attributes are output to a file in a CSV format.
- Method type
- Method arity
- Method call size
- Method inline size
- Method reference frequency
- Inlined size – called size
- Etc.

Group 1: Method Type
Group 2: Inlined size – called size
Color: reference frequency
Size: called size
STATIC CALL FREQUENCY ESTIMATION

- Developers can use an *inline directive* to instruct the compiler to inline a method.
- A *method call* involves (1) *move instructions* to handle parameters, (2) a *jump instruction* to transfer control to the method declaration, and a (3) *return instruction* transferring control from the method body back to the point of call.

Scenario:
- Method m1 has 3 parameters and is called from 10 syntactically distinct locations
- Method m2 has 2 parameters and is called from 20 syntactically distinct locations

<table>
<thead>
<tr>
<th>Method</th>
<th>Move Instruction Overhead</th>
<th>Static Overhead Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>3*10 = 30</td>
<td>(3+2) * 10 = 50</td>
</tr>
<tr>
<td>m2</td>
<td>2*20 = 40</td>
<td>(2+2) * 20 = 80</td>
</tr>
</tbody>
</table>
EXECUTION PATH ESTIMATION

- Static call-frequency analysis is coarse grained. It does not account for execution paths that result in a distinct method call being evaluated multiple times (e.g., a method call within a loop).

```plaintext
interface call( LabelType toLabel ) { aux_call(); }
interface return() { aux_return(); }

subroutine m1() returns void { return(); }

subroutine m2() returns void {
    m1(); m1(); return();
}

subroutine m3() returns void {
    m2(); m2(); m2(); return();
}

microcode {
    m1(); m1(); m1(); m1(); m1();
    m2(); m2(); m2(); m2(); m2(); m2();
    m3(); m3(); m3(); m3(); m3();
}
```
EXECUTION PATH ESTIMATION

- Note: Accounts for the mandatory inlining of macros, interfaces, and conditions.

<table>
<thead>
<tr>
<th>Method</th>
<th>Total Calls</th>
<th>Inlined Size</th>
<th>Called Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>41</td>
<td>0</td>
<td>83</td>
</tr>
<tr>
<td>m2</td>
<td>18</td>
<td>36</td>
<td>39</td>
</tr>
<tr>
<td>m3</td>
<td>4</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

\[41 = 5 + 6 \times 2 + 4 \times 3 \times 2; \ 83 = 41(\text{calls}) + 41(\text{returns}) + 1(\text{declaration})\]

<table>
<thead>
<tr>
<th>Method</th>
<th>Inlined Time</th>
<th>Called Time</th>
<th>% Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>0</td>
<td>82</td>
<td>100.0%</td>
</tr>
<tr>
<td>m2</td>
<td>36</td>
<td>72</td>
<td>50.0%</td>
</tr>
<tr>
<td>m3</td>
<td>12</td>
<td>20</td>
<td>40.0%</td>
</tr>
</tbody>
</table>
Transformation rules and timing constraints

OPTIMIZATION
TRANSFORMATION RULES

• The Paradigm compiler is transformation-based and implemented in the TL System.
• During compilation, a Paradigm program is passed through a number of canonical forms, each of which can be output in human-readable form.
• The Paradigm compiler is extensible in the sense that it supports the incorporation of user-defined transformation rules into the compilation process.
  • Such rules provide domain experts the opportunity to perform custom optimizations specific to a particular architecture or code design.
EXAMPLE (CAN BE MADE MORE GENERAL)

```c
...writeReg(T1Type.SOME, AType.$temp_reg 3);
copyReg(AType.$temp_reg 3, AType.$reg 2);
...writeReg(T1Type.SOME, AType.$reg 2);
...```
TIMING CONSTRAINT LANGUAGE (TCL)

- Paradigm provides a **declarative language**, called TCL, for specifying the timing constraints of a targeted hardware architecture.

- Timing constraints form the basis of a **local compaction** algorithm focusing on the compression of straight-line microcode (SLM).

- Timing-constraint based optimization does not involve commutative reordering of microoperations, instead it focuses on maximizing the compression of **adjacent (i.e., associative) microinstructions**.

- In the compilation stage where timing constraint based optimization occurs, the target program has a form where **all non-sequential control flows are expressed in terms of jumps to labels**. In this context, a straight-line microcode (SLM) is then simply the sequence of microinstructions occurring between consecutive labels.
A constraint is a pair of logical formulas:

\[
\text{constraint TC}_k \{
\begin{align*}
\text{first\_row:} & \quad \text{logical formula} \\
\text{second\_row:} & \quad \text{logical formula}
\end{align*}
\}
\]

An adjacent pair of microinstructions for which a constraint evaluates to true is said to satisfy the constraint.

Timing properties of hardware can be specified via a set of constraints.
COMPRESSION

- An adjacent pair of microinstructions may be compressed, iff they do not satisfy any timing constraint in the set of timing constraints for the target hardware.
- Compression consists of merging the contents of microinstructions.
- Merging is subject to hardware limitations (e.g., fields may not simultaneously hold multiple non-default values).
OPTIMIZATION METRICS

Standard Compiler Optimizations.
- Total number of temp register optimizations = 0
- Number of nop() statements removed = 0

Custom Optimizations.
- Total number of row reductions due to custom optimizations = 0

Constraint-based Optimizations.
- Number of row mergings prevented due to timing constraints = 1291
- Number of duplicate row mergings = 500
- Number of conflict-free row mergings = 1000
- Total number of constraint-based row mergings = 1500

Number of rows before any optimization = 2800
Number of rows after all optimization = 1300
Size of optimized file as a percentage of the unoptimized file = 46.43%
The size of the unoptimized file was reduced by = 53.57%
The End