Integrated Instruction Selection and Register Allocation for Compact Code Generation Exploiting Freeform Mixing of 16- and 32-bit Instructions

Tobias Edler von Koch
Igor Böhm and Björn Franke

PASTA
Processor Automated Synthesis by Iterative Analysis
Does Code Size Matter?

128-Mbit Flash
27.3mm² at 0.13μm
Does Code Size Matter?

128-Mbit Flash
27.3mm\(^2\) at 0.13\(\mu m\)

ARM Cortex M3
0.43mm\(^2\) at 0.13\(\mu m\)
Does Code Size Matter?

128-Mbit Flash
27.3mm² at 0.13µm

ARM Cortex M3
0.43mm² at 0.13µm

ENCORE Calton
0.15mm² at 0.13µm

Thumb-2 ISA

ARCompact ISA
Does Code Size Matter?

128-Mbit Flash
27.3mm² at 0.13μm

ARM Cortex M3
0.43mm² at 0.13μm

ENCORE Calton
0.15mm² at 0.13μm

RISC Architectures

sacrifice code density in order to simplify implementation circuitry and decrease die area
Solution to Code Size Problem

• Dual instruction sets providing 32-bit and 16-bit instruction encodings:
Solution to Code Size Problem

- Dual instruction sets providing 32-bit and 16-bit instruction encodings:
  - ARM Thumb-2
  - ARM Thumb
  - microMIPS
  - ARCompact
Solution to Code Size Problem

- Dual instruction sets providing 32-bit and 16-bit instruction encodings:

  ARM Thumb-2
  ARM Thumb
  microMIPS
  ARC compact

- There’s no such thing as a free lunch!
Solution to Code Size Problem

• Dual instruction sets providing 32-bit and 16-bit instruction encodings:

  ARM Thumb-2
  microMIPS
  ARCompact

• There’s no such thing as a free lunch!

• 16-bit instructions come with constraints!
Common Compact Instruction Format Constraints

• Only subset of registers accessible
Common Compact Instruction Format Constraints

- Only subset of registers accessible
- Explicit instructions necessary to switch between 16-bit and 32-bit modes
Common Compact Instruction Format Constraints

- Only subset of registers accessible
- Explicit instructions necessary to switch between 16-bit and 32-bit modes
- Reduced size of immediate operands
Common Compact Instruction Format Constraints

- Only subset of registers accessible
- Explicit instructions necessary to switch between 16-bit and 32-bit modes
- Reduced size of immediate operands
- Not every 32-bit instruction has a 16-bit counterpart
Common Compact Instruction Format Constraints

- Only subset of registers accessible
- Explicit instructions necessary to switch between 16-bit and 32-bit modes
- Reduced size of immediate operands
- Not every 32-bit instruction has a 16-bit counterpart
- Free mixing of 16- and 32-bit encodings not always possible
ARC 16-bit Instruction Format

Constraints

- Only subset of registers accessible
- Explicit instructions necessary to switch between 16-bit and 32-bit modes
- Reduced size of immediate operands
- Not every 32-bit instruction has a 16-bit counterpart
- Free mixing of 16- and 32-bit encodings not always possible
Motivating Example

32-Bit Only

```
ld r2,[sp,0]
ld r3,[sp,4]
ld r4,[sp,8]
add r2,r2,r3
asl r2,r2,2
sub r2,r2,r4
```

24 bytes
### Motivating Example

<table>
<thead>
<tr>
<th>Basic Block</th>
<th>32-Bit Only</th>
<th>Mixed Mode Aggressive</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r2,[sp,0]</td>
<td>ld_s r2,[sp,0]</td>
<td></td>
</tr>
<tr>
<td>ld r3,[sp,4]</td>
<td>ld_s r3,[sp,4]</td>
<td></td>
</tr>
<tr>
<td>ld r4,[sp,8]</td>
<td>ld_s rx,[sp,8]</td>
<td></td>
</tr>
<tr>
<td>add r2,r2,r3</td>
<td>add_s r2,r2,r3</td>
<td></td>
</tr>
<tr>
<td>asl r2,r2,2</td>
<td>asl_s r2,r2,2</td>
<td></td>
</tr>
<tr>
<td>sub r2,r2,r4</td>
<td>sub_s r2,r2,rx</td>
<td></td>
</tr>
</tbody>
</table>

**24 bytes** vs. **12 bytes**

**Instruction Selection**
Motivating Example

<table>
<thead>
<tr>
<th>Basic Block</th>
<th>32-Bit Only</th>
<th>Mixed Mode Aggressive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ld r2,[sp,0]</td>
<td>ld_s r2,[sp,0]</td>
</tr>
<tr>
<td></td>
<td>ld r3,[sp,4]</td>
<td>ld_s r3,[sp,4]</td>
</tr>
<tr>
<td></td>
<td>ld r4,[sp,8]</td>
<td>mov r4,r1</td>
</tr>
<tr>
<td></td>
<td>add r2,r2,r3</td>
<td>add_s r2,r2,r3</td>
</tr>
<tr>
<td></td>
<td>asl r2,r2,2</td>
<td>asl_s r2,r2,2</td>
</tr>
<tr>
<td></td>
<td>sub r2,r2,r4</td>
<td>sub_s r2,r2,r1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov r4,r1</td>
</tr>
</tbody>
</table>

24 bytes

20 bytes

Instruction Selection

Register Allocation
## Motivating Example

<table>
<thead>
<tr>
<th>Basic Block</th>
<th>32-Bit Only</th>
<th>Mixed Mode Aggressive</th>
<th>Mixed Mode Integrated</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r2,[sp,0]</td>
<td>ld_s r2,[sp,0]</td>
<td>ld_s r2,[sp,0]</td>
<td>ld_s r2,[sp,0]</td>
</tr>
<tr>
<td>ld r3,[sp,4]</td>
<td>ld_s r3,[sp,4]</td>
<td>ld_s r3,[sp,4]</td>
<td>ld_s r3,[sp,4]</td>
</tr>
<tr>
<td>ld r4,[sp,8]</td>
<td>mov r4,r1</td>
<td>ld_s r1,[sp,8]</td>
<td>ld_s r4,[sp,8]</td>
</tr>
<tr>
<td>add r2,r2,r3</td>
<td>add_s r2,r2,r3</td>
<td>add_s r2,r2,r3</td>
<td>add_s r2,r2,r3</td>
</tr>
<tr>
<td>asl r2,r2,2</td>
<td>asl_s r2,r2,2</td>
<td>asl_s r2,r2,2</td>
<td>asl_s r2,r2,2</td>
</tr>
<tr>
<td>sub r2,r2,r4</td>
<td>sub_s r2,r2,r1</td>
<td>sub_s r2,r2,r1</td>
<td>sub_s r2,r2,r4</td>
</tr>
</tbody>
</table>

- 24 bytes
- 20 bytes
- 16 bytes
Efficient Compact Code Generation is an Integrated Instruction Selection and Register Allocation Problem!
Compact Code Generation Approach

**ECC - EnCore C Compiler** based on commercial **CoSy** compiler by **ACE©**.

**Powerful Backend**

**BEG - Backend Generator**
Compact Code Generation Approach

**ECC - EnCore C Compiler** based on commercial **CoSy** compiler by **ACE**©.

Compiler backend supports **two** compact code generation strategies

- Opportunistic
- Feedback-Guided

---

**Pasta** Processor Automated Synthesis by Iterative Analysis
http://groups.inf.ed.ac.uk/pasta/
Compact Code Generation Approach

**ECC - EnCore C Compiler** based on commercial **CoSy** compiler by **ACE**.

Compiler backend supports **two** compact code generation strategies:

<table>
<thead>
<tr>
<th>Opportunistic</th>
<th>Feedback-Guided</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>MIR ↓</td>
<td></td>
</tr>
<tr>
<td><strong>Match / Cover</strong></td>
<td>32-bit patterns only</td>
</tr>
<tr>
<td>LIR ↓</td>
<td></td>
</tr>
<tr>
<td><strong>Register Allocation</strong></td>
<td></td>
</tr>
<tr>
<td>LIR ↓</td>
<td></td>
</tr>
<tr>
<td><strong>Code Emission</strong></td>
<td>32-bit/16-bit instructions</td>
</tr>
<tr>
<td>ASM ↓</td>
<td></td>
</tr>
</tbody>
</table>

**PASTA** Processor Automated Synthesis
by Iterative Analysis
http://groups.inf.ed.ac.uk/pasta/
Compact Code Generation Approach

**ECC - EnCore C Compiler** based on commercial **CoSy** compiler by **ACE©**.

Compiler backend supports **two** compact code generation strategies

- **Opportunistic**
  - MIR
    - Match / Cover
      - 32-bit patterns only
    - LIR
    - Register Allocation
    - LIR
    - Code Emission
      - 32-bit/16-bit instructions
    - ASM

- **Feedback-Guided**
Compact Code Generation Approach

**ECC - EnCore C Compiler** based on commercial **CoSy** compiler by **ACE©**.

Compiler backend supports **two** compact code generation strategies

1. **Opportunistic**
   - **Match / Cover**
     - 32-bit patterns only
   - **Register Allocation**
   - **Code Emission**
     - 32-bit/16-bit instructions
   - **ASM**

2. **Feedback-Guided**
Compact Code Generation Approach

**ECC - EnCore C Compiler** based on commercial **CoSy** compiler by **ACE©**.

Compiler backend supports **two** compact code generation strategies

- **Opportunistic**
  - MIR
  - Match / Cover
    - 32-bit patterns only
  - LIR
  - Register Allocation
  - LIR
  - Code Emission
    - 32-bit/16-bit instructions
  - ASM

- **Feedback-Guided**
Compact Code Generation Approach

**ECC - EnCore C Compiler** based on commercial **CoSy** compiler by **ACE©**.

Compiler backend supports **two** compact code generation strategies

<table>
<thead>
<tr>
<th>Opportunistic</th>
<th>Feedback-Guided</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIR</td>
<td>MIR</td>
</tr>
<tr>
<td>Match / Cover 32-bit patterns only</td>
<td>Match / Cover 32-bit &amp; 16-bit patterns preferred</td>
</tr>
<tr>
<td>LIR</td>
<td>MIR &amp; LIR</td>
</tr>
<tr>
<td>Register Allocation</td>
<td>Register Allocation with constrained register sets for 16-bit instructions</td>
</tr>
<tr>
<td>LIR</td>
<td>LIR</td>
</tr>
<tr>
<td>ASM</td>
<td>ASM</td>
</tr>
</tbody>
</table>

**Match / Cover**
- 32-bit patterns only
- 32-bit/16-bit instructions

**Register Allocation**
- with constrained register sets for 16-bit instructions

**Code Emission**
- 32-bit/16-bit instructions

**ASM**
- Tape-out

**MIR Annotation**
- selectivity deactivate 16-bit
Compact Code Generation Approach

**ECC - EnCore C Compiler** based on commercial **CoSy** compiler by **ACE©**.

Compiler backend supports **two** compact code generation strategies

```
<table>
<thead>
<tr>
<th>Opportunistic</th>
<th>Feedback-Guided</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIR ↓</td>
<td>MIR ↓</td>
</tr>
<tr>
<td><strong>Match / Cover</strong></td>
<td><strong>Match / Cover</strong></td>
</tr>
<tr>
<td>32-bit patterns only</td>
<td>32-bit &amp; 16-bit patterns preferred</td>
</tr>
<tr>
<td>LIR ↓</td>
<td>MIR &amp; LIR ↓</td>
</tr>
<tr>
<td><strong>Register Allocation</strong></td>
<td><strong>Register Allocation</strong></td>
</tr>
<tr>
<td>LIR ↓</td>
<td>MIR &amp; LIR ↓</td>
</tr>
<tr>
<td><strong>Code Emission</strong></td>
<td><strong>Code Emission</strong></td>
</tr>
<tr>
<td>32-bit/16-bit instructions</td>
<td>32-bit/16-bit instructions</td>
</tr>
<tr>
<td>ASM ↓</td>
<td>ASM ↓</td>
</tr>
</tbody>
</table>
```

**PASTA**

Processor Automated Synthesis
by Iterative Analysis
http://groups.inf.ed.ac.uk/pasta/
Compact Code Generation Approach

**ECC - EnCore C Compiler** based on commercial **CoSy** compiler by **ACE©**.

Compiler backend supports **two** compact code generation strategies

**Opportunistic**

1. Match / Cover
   - 32-bit patterns only
2. Register Allocation
3. Code Emission
   - 32-bit/16-bit instructions

**Feedback-Guided**

1. Match / Cover
   - 32-bit & 16-bit patterns preferred
2. Register Allocation
   - with constrained register sets for 16-bit instructions
3. Code Emission
   - 32-bit/16-bit instructions

**MIR Annotation**
- selectively deactivate 16-bit

---

**Processor Automated Synthesis**
by **Iterative Analysis**

http://groups.inf.ed.ac.uk/pasta/
Compact Code Generation Approach

**ECC - EnCore C Compiler** based on commercial **CoSy** compiler by **ACE©**.

Compiler backend supports **two** compact code generation strategies

**Opportunistic**
1. **Match / Cover**
   - 32-bit patterns only
2. **Register Allocation**
3. **Code Emission**
   - 32-bit/16-bit instructions

**Feedback-Guided**
1. **Match / Cover**
   - 32-bit & 16-bit patterns preferred
2. **Register Allocation**
   - with constrained register sets for 16-bit instructions
3. **Code Emission**
   - 32-bit/16-bit instructions

**MIR**

**LIR**

**ASM**

**PASTA**

**Processor Automated Synthesis**

by **Iterative Analysis**

[http://groups.inf.ed.ac.uk/pasta/](http://groups.inf.ed.ac.uk/pasta/)
Compact Code Generation Approach

**ECC - EnCore C Compiler** based on commercial **CoSy** compiler by **ACE**©.

Compiler backend supports **two** compact code generation strategies

**Opportunistic**

1. **Match / Cover**
   - 32-bit patterns only

2. **Register Allocation**

3. **Code Emission**
   - 32-bit/16-bit instructions

**Feedback-Guided**

1. **Match / Cover**
   - 32-bit & 16-bit patterns preferred

2. **Register Allocation**
   - with constrained register sets for 16-bit instructions

3. **MIR Annotation**
   - selectively deactivate 16-bit

4. **Code Emission**
   - 32-bit/16-bit instructions

---

**IC Layout**

**CC Match / Cover**

**LIR**

**MIR**

**CoSy GDS II**

**Feedback-Guided**

**32-bit/16-bit instructions**

**ASM**

---

**http://groups.inf.ed.ac.uk/pasta/**
Compact Code Generation Approach

**ECC - EnCore C Compiler** based on commercial **CoSy** compiler by **ACE©**.

Compiler backend supports **two** compact code generation strategies

**Opportunistic**
- **Match / Cover**
  - 32-bit patterns only
- **Register Allocation**
- **Code Emission**
  - 32-bit/16-bit instructions

**Feedback-Guided**
- **Match / Cover**
  - 32-bit & 16-bit patterns preferred
- **Register Allocation**
  - with constrained register sets for 16-bit instructions
- **Code Emission**
  - 32-bit/16-bit instructions
- **MIR Annotation**
  - selectively deactivate 16-bit

---

Processor Automated Synthesis  
by Iterative Analysis  
http://groups.inf.ed.ac.uk/pasta/
Compact Code Generation Approach

**ECC - EnCore C Compiler** based on commercial **CoSy** compiler by **ACE©**.

Compiler backend supports **two** compact code generation strategies

<table>
<thead>
<tr>
<th>Opportunistic</th>
<th>Feedback-Guided</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Match / Cover</strong>&lt;br&gt;32-bit patterns only</td>
<td><strong>Match / Cover</strong>&lt;br&gt;32-bit &amp; 16-bit patterns preferred</td>
</tr>
<tr>
<td><strong>Register Allocation</strong>&lt;br&gt;LIR</td>
<td><strong>Register Allocation</strong>&lt;br&gt;MIR &amp; LIR</td>
</tr>
<tr>
<td><strong>Code Emission</strong>&lt;br&gt;32-bit/16-bit instructions</td>
<td><strong>Code Emission</strong>&lt;br&gt;32-bit/16-bit instructions</td>
</tr>
<tr>
<td>ASM</td>
<td>ASM</td>
</tr>
</tbody>
</table>

**Processor Automated Synthesis**
by **Iterative Analysis**
http://groups.inf.ed.ac.uk/pasta/
Feedback-Guided Instruction Selection

vX ... Virtual Register
rX ... Physical Register

MIR

ld v2,x
ld v3,y
ld v4,z
add v5,v2,v3
asl v6,v5,2
sub v7,v6,v4
Feedback-Guided Instruction Selection

vX ... Virtual Register
rX ... Physical Register

MIR

ld v2,x
ld v3,y
ld v4,z
add v5,v2,v3
asl v6,v5,2
sub v7,v6,v4

MIR Annotation

Match/Cover

ld_s v2,[sp,0]
ld_s v3,[sp,4]
ld_s v4,[sp,8]
add_s v5,v2,v3
asl_s v6,v5,2
sub_s v7,v6,v4

Register Allocation & MIR Annotation

aggressively select 16-bit instructions
Feedback-Guided Instruction Selection

register allocator constrained to 16-bit accessible register set

aggressively select 16-bit instructions
Feedback-Guided Instruction Selection

register allocator constrained to 16-bit accessible register set
Feedback-Guided Instruction Selection

MIR Annotation

Match/Cover

ld v2, x
ld v3, y
ld v4, z
add v5, v2, v3
asl v6, v5, 2
sub v7, v6, v4
ld_s v2, [sp, 0]
ld_s v3, [sp, 4]
ld_s v4, [sp, 8]
add_s v5, v2, v3
asl_s v6, v5, 2
sub_s v7, v6, v4
mov r4, r1
ld_s r1, [sp, 8]
add_s r2, r2, r3
asl_s r2, r2, 2
sub_s r2, r2, r1
mov r1, r4

Register Allocation & MIR Annotation

ld_s r2, [sp, 0]
ld_s r3, [sp, 4]

Annotated MIR

ld v2, x
ld v3, y
ld v4, z
add v5, v2, v3
asl v6, v5, 2
sub v7, v6, v4

No 16-bit

Annotated MIR

No 16-bit

No 16-bit

No 16-bit
Feedback-Guided Instruction Selection

MIR Annotation

Match/Cover

Register Allocation & MIR Annotation

ld_s v2,[sp,0]
ld_s v3,[sp,4]
ld_s v4,[sp,8]
add_s v5,v2,v3
asl_s v6,v5,2
sub_s v7,v6,v4

ld_s r2,[sp,0]
ld_s r3,[sp,4]
move r4,r1
ld_s r1,[sp,8]
add_s r2,r2,r3
asl_s r2,r2,2
sub_s r2,r2,r1
mov r1,r4

Annotated MIR

LD T2,x
LD T3,y

ld v2,x
ld v3,y

ld v2,x
ld v3,y

Annotated MIR

 smarter selection of 16-bit instructions based on feedback
Feedback-Guided Instruction Selection

fewer constraints for register allocator

smarter selection of 16-bit instructions based on feedback

Processor Automated Synthesis by Iterative Analysis
http://groups.inf.ed.ac.uk/pasta/
Feedback-Guided Instruction Selection

MIR Annotation

Match/Cover

ld v2,x
ld v3,y
ld v4,z
add v5,v2,v3
asl v6,v5,2
sub v7,v6,v4

ld_s v2,[sp,0]
ld_s v3,[sp,4]
ld_s v4,[sp,8]
add_s v5,v2,v3
asl_s v6,v5,2
sub_s v7,v6,v4

ld_s r2,[sp,0]
ld_s r3,[sp,4]
ld_s r4,[sp,8]
add_s r2,r2,v3
asl_s r2,r2,2
sub_s r2,r2,r4

mov r4,r1
mov r4,r1
mov r4,r1

Register Allocation & MIR Annotation

ld_s r2,[sp,0]
ld_s r3,[sp,4]
ld_s r1,[sp,8]
add_s r2,r2,r3
asl_s r2,r2,2
sub_s r2,r2,r1
mov r1,r4

ld v2,x
ld v3,y
ld v4,z
add v5,v2,v3
asl v6,v5,2
sub v7,v6,v4

Annotated MIR

No 16-bit

Code Emission

Register Allocation

ld v2,x
ld v3,y
ld v4,z
add v5,v2,v3
asl v6,v5,2
sub v7,v6,v4

No 16-bit

Annotated MIR

Feedback Guided Code Generation

fewer constraints for register allocator

Processor Automated Synthesis
by iTerative Analysis
http://groups.inf.ed.ac.uk/pasta/
### Evaluation - Experimental Setup

<table>
<thead>
<tr>
<th>BenchMarks</th>
<th>EEMBC 1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Core</strong></td>
<td><strong>ARC750D</strong></td>
</tr>
<tr>
<td>Pipeline</td>
<td>7-stage interlocked</td>
</tr>
<tr>
<td>Execution Order</td>
<td>In-Order</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>Yes</td>
</tr>
<tr>
<td>ISA</td>
<td>ARCompact</td>
</tr>
<tr>
<td>Floating Point</td>
<td>Hardware</td>
</tr>
<tr>
<td><strong>Memory Subsystem</strong></td>
<td></td>
</tr>
<tr>
<td>L1 Cache</td>
<td>Yes</td>
</tr>
<tr>
<td>Instruction</td>
<td>8k/2-way associative</td>
</tr>
<tr>
<td>Data</td>
<td>8k/2-way associative</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>No</td>
</tr>
</tbody>
</table>

**Simulator**

| ArcSim |
|---|---|
| **Simulation Mode** | Full System, Cycle Accurate |
| **Accuracy** | Cycle accurate mode validated against real HW |
| **Options** | Default |
| **I/O & System Calls** | Emulated |
Evaluation - **Code Size Reduction**

*Improvement in Code Size (in %)*

Baseline: plain 32-bit code.

- Feedback-guided selection (avg: 16.7%)
- Opportunistic selection (avg: 15.4%)
- GCC (avg: 2.7%)
Evaluation - **Code Size Reduction**

**Improvement in Code Size (in %)**

Baseline: plain 32-bit code.

- **Feedback-guided selection** (avg: 16.7%)
- **Opportunistic selection** (avg: 15.4%)
- **GCC** (avg: 2.7%)

The chart illustrates the reduction in code size across different categories (automotive, consumer, net, office, telecom) for various benchmarks. The average reduction for opportunistic selection is 15.4%, while for feedback-guided selection, it is 16.7%. GCC shows the least reduction at 2.7% on average.
Evaluation - **Code Size Reduction**

Improvement in Code Size (in %)

- **Baseline:** plain 32-bit code.
  - Feedback-guided selection (avg: 16.7%)
  - Opportunistic selection (avg: 15.4%)
  - GCC (avg: 2.7%)

**Percentage Reductions:**
- **16.7%**
- **15.4%**

Legend:
- **Red:** Feedback-guided selection
- **Blue:** Opportunistic selection
- **Yellow:** GCC
Evaluation - Code Size Reduction

Improvement in Code Size (in %)

Baseline: plain 32-bit code.
- Feedback-guided selection (avg: 16.7%)
- Opportunistic selection (avg: 15.4%)
- GCC (avg: 2.7%)
Evaluation - **Code Size Reduction**

Improvement in Code Size (in %)

Baseline: plain32-bit code.

<table>
<thead>
<tr>
<th>automotive</th>
<th>consumer</th>
<th>net</th>
<th>office</th>
<th>telecom</th>
</tr>
</thead>
<tbody>
<tr>
<td>aifr01</td>
<td>aifr01</td>
<td>pnu</td>
<td>aifr01</td>
<td>aifr01</td>
</tr>
<tr>
<td>aifr01</td>
<td>aifr01</td>
<td>pnum</td>
<td>aifr01</td>
<td>aifr01</td>
</tr>
<tr>
<td>aifr01</td>
<td>aifr01</td>
<td>pnum</td>
<td>aifr01</td>
<td>aifr01</td>
</tr>
<tr>
<td>aifr01</td>
<td>aifr01</td>
<td>pnum</td>
<td>aifr01</td>
<td>aifr01</td>
</tr>
<tr>
<td>aifr01</td>
<td>aifr01</td>
<td>pnum</td>
<td>aifr01</td>
<td>aifr01</td>
</tr>
<tr>
<td>aifr01</td>
<td>aifr01</td>
<td>pnum</td>
<td>aifr01</td>
<td>aifr01</td>
</tr>
<tr>
<td>aifr01</td>
<td>aifr01</td>
<td>pnum</td>
<td>aifr01</td>
<td>aifr01</td>
</tr>
<tr>
<td>aifr01</td>
<td>aifr01</td>
<td>pnum</td>
<td>aifr01</td>
<td>aifr01</td>
</tr>
<tr>
<td>aifr01</td>
<td>aifr01</td>
<td>pnum</td>
<td>aifr01</td>
<td>aifr01</td>
</tr>
<tr>
<td>aifr01</td>
<td>aifr01</td>
<td>pnum</td>
<td>aifr01</td>
<td>aifr01</td>
</tr>
<tr>
<td>aifr01</td>
<td>aifr01</td>
<td>pnum</td>
<td>aifr01</td>
<td>aifr01</td>
</tr>
<tr>
<td>aifr01</td>
<td>aifr01</td>
<td>pnum</td>
<td>aifr01</td>
<td>aifr01</td>
</tr>
</tbody>
</table>

**Feedback-guided selection (avg: 16.7%)**

**Opportunistic selection (avg: 15.4%)**

**GCC (avg: 2.7%)**
Evaluation - **Code Size Reduction**

Improvement in Code Size (in %)

Baseline: plain 32-bit code.  
- Feedback-guided selection (avg: 16.7%)  
- Opportunistic selection (avg: 15.4%)  
- GCC (avg: 2.7%)
Evaluation - Code Size Reduction

Improvement in Code Size (in %)

Baseline: plain 32-bit code.
- Feedback-guided selection (avg: 16.7%)
- Opportunistic selection (avg: 15.4%)
- GCC (avg: 2.7%)
Why does the simple Opportunistic Mode perform so well?
Why does the simple Opportunistic Mode perform so well?

We exploit the fact that 16-bit accessible registers are already frequently used in 32-bit code.
Why does the simple Opportunistic Mode perform so well?

- register allocator selects registers with lower ID from set of possible registers

We exploit the fact that 16-bit accessible registers are already frequently used in 32-bit code.
Why does the simple Opportunistic Mode perform so well?

- register allocator selects registers with lower ID from set of possible registers
- calling conventions constrain register allocator

We exploit the fact that 16-bit accessible registers are already frequently used in 32-bit code
Evaluation - **Performance** Improvements

**Improve in Cycle Count (in %)**

Baseline: plain 32-bit code.

- **Feedback-guided selection**: (avg: 17.7%)
- **Opportunistic selection**: (avg: 16.6%)
- **GCC**: (avg: -2.08%)
Evaluation - **Performance Improvements**

Improvement in Cycle Count (in %)

Baseline: plain 32-bit code.

- **Feedback-guided selection** (avg: 17.7%)
- **Opportunistic selection** (avg: 16.6%)
- **GCC** (avg: -2.08%)
Identification of a document page containing a graph and some text. The graph shows performance improvements in cycle count for various benchmarks across different categories: automotive, consumer, net, office, and telecom. The baseline is plain 32-bit code. The graph highlights the improvements in terms of cycle count, with average improvements: feedback-guided selection (avg: 17.7%), opportunistic selection (avg: 16.6%), and GCC (avg: -2.08%).
Evaluation - **Performance Improvements**

Improvement in Cycle Count (in %)

Baseline: plain 32-bit code.

- **Feedback-guided selection** (avg: 17.7%)
- **Opportunistic selection** (avg: 16.6%)
- **GCC** (avg: -2.08%)

-6.36

-20.8%
Evaluation - **Performance** Improvements

Improvement in Cycle Count (in %)

Baseline: plain32-bit code.

- **Feedback-guided selection** (avg: 17.7%)
- **Opportunistic selection** (avg: 16.6%)
- **GCC** (avg: -2.08%)
Conclusions

- Compact Code Generation is an integrated Instruction Selection and Register Allocation problem.
Conclusions

• Compact Code Generation is an *integrated* Instruction Selection and Register Allocation problem.

• While our simple *opportunistic* mode works well, our *feedback-directed* mode produces more consistent results and does not rely on calling conventions or register-allocation implementations.
Conclusions

- Compact Code Generation is an integrated Instruction Selection and Register Allocation problem.
- While our simple opportunistic mode works well, our feedback-directed mode produces more consistent results and does not rely on calling conventions or register-allocation implementations.
- Our scheme is the first one demonstrating that small code size can be achieved whilst improving performance.
Thanks!

For more information visit our website or search for the term ‘PASTA project’.

http://groups.inf.ed.ac.uk/pasta/