Compact Code Generation

Tobias Edler von Koch
Igor Böhm and Björn Franke

PASTA
Processor Automated Synthesis
by Iterative Analysis
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
Compact Code Generation

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
27.3mm² at 0.13μm

ARM Cortex M3
0.43mm² at 0.13μm
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
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
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  microMIPS  ARCompact
  ARM Thumb

• 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
  - ARM Thumb
  - microMIPS
  - ARCompact

- There’s no such thing as a free lunch!
- 16-bit instructions come with constraints!
Solution to Code Size Problem

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

  ARM Thumb-2
  ARM Thumb
  microMIPS
  ARCompact

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

- 16-bit instructions come with constraints!
Solution to Code Size Problem

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

   ARM Thumb-2

   ARM Thumb

   microMIPS

   ARCompact

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

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

Basic Block

24 bytes

http://groups.inf.ed.ac.uk/pasta/
### Motivating Example

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>ld r3,[sp,4]</td>
<td>ld_s r3,[sp,4]</td>
</tr>
<tr>
<td>ld r4,[sp,8]</td>
<td>ld_s rx,[sp,8]</td>
</tr>
<tr>
<td>add 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>
</tr>
<tr>
<td>sub r2,r2,r4</td>
<td>sub_s r2,r2,rx</td>
</tr>
</tbody>
</table>

**Basic Block**

- 24 bytes
- 12 bytes

**Instruction Selection**

- Processor Automated Synthesis
- by Iterative Analysis
- http://groups.inf.ed.ac.uk/pasta/
## 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>24 bytes</td>
<td>20 bytes</td>
<td>16 bytes</td>
<td></td>
</tr>
<tr>
<td><strong>ld r2,[sp,0]</strong></td>
<td><strong>ld_s r2,[sp,0]</strong></td>
<td><strong>ld_s r2,[sp,0]</strong></td>
<td></td>
</tr>
<tr>
<td><strong>ld r3,[sp,4]</strong></td>
<td><strong>ld_s r3,[sp,4]</strong></td>
<td><strong>ld_s r3,[sp,4]</strong></td>
<td></td>
</tr>
<tr>
<td><strong>ld r4,[sp,8]</strong></td>
<td><strong>mov r4,r1</strong></td>
<td><strong>mov r4,r1</strong></td>
<td></td>
</tr>
<tr>
<td><strong>add r2,r2,r3</strong></td>
<td><strong>add_s r2,r2,r3</strong></td>
<td><strong>add_s r2,r2,r3</strong></td>
<td></td>
</tr>
<tr>
<td><strong>asl r2,r2,2</strong></td>
<td><strong>asl_s r2,r2,2</strong></td>
<td><strong>asl_s r2,r2,2</strong></td>
<td></td>
</tr>
<tr>
<td><strong>sub r2,r2,r4</strong></td>
<td><strong>sub_s r2,r2,r1</strong></td>
<td><strong>sub_s r2,r2,r1</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>mov r4,r1</strong></td>
<td></td>
</tr>
</tbody>
</table>

The table compares three different modes of operation: 32-Bit Only, Mixed Mode Aggressive, and Mixed Mode Integrated. Each mode is represented by a set of instructions and the resulting size in bytes. The 32-Bit Only mode uses the fewest bytes (24 bytes), while the Mixed Mode Integrated uses the most (16 bytes). The Mixed Mode Aggressive is in the middle with 20 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
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
    - 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

![Diagram showing Opportunistic and Feedback-Guided strategies]

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

- **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:

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

- **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**

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

2. **Register Allocation**
   - LIR

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

---

**Annotation**

- **MIR**
- **LIR**
- **ASM**
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

---

**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

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

- **ASM**

**Annotated MIR**

- **MIR Annotation**
  - selectively 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

### 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

**ASM**
Compact Code Generation Approach

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

Compiler backend supports **two** compact code generation strategies.
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. **Annotated MIR**

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

---

**Processors 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. **Annotated MIR**
5. **LIR**
6. **ASM**
Feedback-Guided Instruction Selection

MIR

\[ \text{ld } v2, x \\
\text{ld } v3, y \\
\text{ld } v4, z \\
\text{add } v5, v2, v3 \\
\text{asl } v6, v5, 2 \\
\text{sub } v7, v6, v4 \]
Feedback-Guided Instruction Selection

MIR Annotation

Match/Cover

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

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]
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

Annotated MIR

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

No 16-bit

No 16-bit

No 16-bit

VX ... Virtual Register
RX ... Physical Register

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

Compiler
Design

Pre-processing/Compiling of micro-code definition

Static
Explicitly intended by the designer but are inherent in the layout of the circuit.

Assignment,
Clock

C
(RAM locations, as well as routing and placement obstructions.

Encountered
S
Layout
Design.

Power
Floorplanning
IC Layout
Database

RTL

Synthesis

Virtual Register
rX...
Physical Register

vx...

MIR

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

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

MIR Annotation

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

Annotated MIR

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

Feedback Guided Code Generation

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

fewer constraints for register allocator

fewer constraints for register allocator
fewer constraints for register allocator
## Evaluation - Experimental Setup

<table>
<thead>
<tr>
<th>BenchMarks</th>
<th>EEMBC 1.1</th>
</tr>
</thead>
</table>

---

**EnCore Processor Design Flow**

- **Time Tape-out**
- **Physical Optimisation, Placement and Routing**
- **Compilers**
- **Star-RCXT**
- **StarXtract**
- **EnCore Chip Layout**
- **Gate-level Synthesis**
- **Compiler's Analysis**
- **SDF**
- **SDF Extraction**
- **Physical Effects**
- **Power Minimisation**
- **Power Density**
- **Power Drop Minimisation**
- **Power Effects**
- **TL-to-Gates**
- **Virtual Floor Planning**
- **Final Design**
- **Check Setup**
- **Power Supply Voltage, and Temperature.**
- **Characteristics**
- **Floor Planning**
- **Floor planning**

---

*EnCore - Calton Chip Layout*

*Libraries*
Evaluation - Experimental Setup

<table>
<thead>
<tr>
<th>BenchMarks</th>
<th>EEMBC 1.1</th>
</tr>
</thead>
</table>

ArcSim
Cycle Accurate Instruction Set Simulator

### Simulator vs ArcSim

<table>
<thead>
<tr>
<th></th>
<th>ArcSim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation Mode</td>
<td>Full System, Cycle Accurate</td>
</tr>
<tr>
<td>Accuracy</td>
<td>Cycle accurate mode validated against real HW</td>
</tr>
<tr>
<td>Options</td>
<td>Default</td>
</tr>
<tr>
<td>I/O &amp; System Calls</td>
<td>Emulated</td>
</tr>
</tbody>
</table>

ArcSim
Cycle Accurate Instruction Set Simulator

<table>
<thead>
<tr>
<th></th>
<th>ArcSim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation Mode</td>
<td>Full System, Cycle Accurate</td>
</tr>
<tr>
<td>Accuracy</td>
<td>Cycle accurate mode validated against real HW</td>
</tr>
<tr>
<td>Options</td>
<td>Default</td>
</tr>
<tr>
<td>I/O &amp; System Calls</td>
<td>Emulated</td>
</tr>
</tbody>
</table>
Evaluation - Experimental Setup

<table>
<thead>
<tr>
<th>BenchMarks</th>
<th>EEMBC 1.1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Core</th>
<th>ARC750D</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

ArcsiM
Cycle Accurate Instruction Set Set Simulator

<table>
<thead>
<tr>
<th>Simulator</th>
<th>ArcSim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation Mode</td>
<td>Full System, Cycle Accurate</td>
</tr>
<tr>
<td>Accuracy</td>
<td>Cycle accurate mode validated against real HW</td>
</tr>
<tr>
<td>Options</td>
<td>Default</td>
</tr>
<tr>
<td>I/O &amp; System Calls</td>
<td>Emulated</td>
</tr>
</tbody>
</table>

PASTA 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**

<table>
<thead>
<tr>
<th><strong>ArcSim</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Simulation Mode</strong></td>
</tr>
<tr>
<td><strong>Accuracy</strong></td>
</tr>
<tr>
<td><strong>Options</strong></td>
</tr>
<tr>
<td><strong>I/O &amp; System Calls</strong></td>
</tr>
</tbody>
</table>

[Email: groups.inf.ed.ac.uk/pasta/]

[Image: http://groups.inf.ed.ac.uk/pasta/]

### Benchmarks

- **Core**
  - Pipeline: 7-stage interlocked
  - Execution Order: In-Order
  - Branch Prediction: Yes
  - ISA: ARCompact
  - Floating Point: Hardware

- **Memory Subsystem**
  - L1 Cache: Yes
  - Instruction: 8k/2-way associative
  - Data: 8k/2-way associative
  - L2 Cache: No

### Simulators

- **ArcSim**
  - Simulation Mode: Full System, Cycle Accurate
  - Accuracy: Cycle accurate mode validated against real HW
  - Options: Default
  - I/O & System Calls: Emulated
Baseline: plain 32-bit code using -O3.
Evaluation - Code Size Reduction

Improvement in Code Size (in %)

Baseline: plain 32-bit code using -O3.

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 using -O3.
Evaluation - Code Size Reduction

Improvement in Code Size (in %)

Baseline: plain 32-bit code using -O3.

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

Baseline: plain 32-bit code using -O3.

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

Improvement in Code Size (in %)
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
Baseline: plain 32-bit code using -O3.

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

Evaluation - Performance Improvements

Improvement in Cycle Count (in %)
Evaluation - **Performance Improvements**

** Improvement in Cycle Count (in %) **

Baseline: plain 32-bit code using -O3.

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

17.7%
Evaluation - Performance Improvements

<table>
<thead>
<tr>
<th>Improvement in Cycle Count (in %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O3 -fmixed-feedback</td>
</tr>
<tr>
<td>-O3 -fmixed-opportunistic</td>
</tr>
</tbody>
</table>

Baseline: plain 32-bit code using -O3.

- Feedback-guided selection (avg: 17.7%)
- Opportunistic selection (avg: 16.6%)
- GCC (avg: -2.08%)
Baseline: plain 32-bit code using -O3.

Feedback-guided selection (avg: 17.7%)

Opportunistic selection (avg: 16.6%)

GCC (avg: -2.08%)
Baseline: plain 32-bit code using -O3.

<table>
<thead>
<tr>
<th></th>
<th>Feedback-guided selection</th>
<th>Opportunistic selection</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(avg: 17.7%)</td>
<td>(avg: 16.6%)</td>
<td>(avg: -2.08%)</td>
</tr>
<tr>
<td>automotive</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>consumer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>net</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>office</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>telecom</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>average</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Improvement in Cycle Count (in %)
Evaluation - Performance Improvements

Improvement in Cycle Count (in %)

-O3 -fmixed-feedback
-O3 -fmixed-opportunistic
-O3 -fmixed

Baseline: plain 32-bit code using -O3.

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

Improvement in Cycle Count (in %)

- **-O3 -fmixed-feedback**
- **-O3 -fmixed-opportunistic**
- **-O3 -fmixed**

Baseline: plain 32-bit code using -O3.
- Feedback-guided selection (avg: 17.7%)
- Opportunistic selection (avg: 16.6%)
- GCC (avg: -2.08%)

[Diagram showing performance improvements across different categories: automotive, consumer, net, office, and telecom.]
Conclusions
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!

Questions?

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

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