Generating and Executing Multi-Exit Custom Instructions for an Adaptive Extensible Processor

Hamid Noori†, Farhad Mehdipour†, Kazuaki Murakami†, Koji Inoue†, and Maziar Goudarzi†

† Kyushu University, Japan
‡ AmirKabir University of Technology, Iran
Outline

- Introduction
- Overview of ADEXOR Architecture
- Generating Multi-Exit Custom Instructions (MECIs)
- Proposing an Architecture for the CRFU
- Experimental Results
- Conclusions and Future Work
Introduction

- Motivation
  - Increase in manufacturing and NRE costs
  - Increase in design and verification costs due to more complexity
  - Shorter time-to-market
  - More required flexibility due to evolution of standards, user requirements, supporting multiple applications and etc

- Our proposed approach
  - Generating custom instruction after chip-fabrication
  - Proposing multi-exit custom instructions
  - Proposing a reconfigurable functional unit with conditional execution using a quantitative approach
General Overview of ADEXOR Architecture
General Overview of ADEXOR Architecture

- **Phases**
  - Design phase
  - Configuration phase
  - Normal phase
Generating Multi-Exit Custom Instructions

- Multi-Exit Custom Instructions
  - Include hot directions of branch instructions
  - Include single entry but multiple exits
Generating Multi-Exit Custom Instructions

- Finding the largest sequence of instructions in the CFG
- Checking the anti-dependence and flow-dependence and moving executable instructions to the head and tail(s) of MECIs
- Rewriting the object code where instructions are going to be moved.

Detecting subgraphs in a HIS

Moving instructions after checking data dependency (binary rewriting)

Overwriting the entry node with \textit{mtc1} instr
Generating Multi-Exit Custom Instructions

- Tool Chain

1. Instruction Set Simulator (Simplescalar) → Profiler → Detecting Start Address of HBBs → Reading HBBs from Object code → Linking HBBs using Profiling Information and Generating HISs
2. Integrated Framework (Partitioning and Mapping MECIs) → Updating CDFG and Binary rewriting → Generating Multi-Exit Custom Instructions → Generating CDFG for HISs
CRFU Architecture: A Quantitative Approach

- 22 programs of MiBench were chosen
- Simplescalar toolset was utilized for simulation
- CRFU is a matrix of FUs
  - No of Inputs
  - No of Outputs
  - No of FUs
  - Connections
  - Location of Inputs & Outputs
- Some definitions:
  - Considering frequency and weight in measurement
    - CI Execution Frequency
    - Weight (To equal number of executed instructions)
    - Average = for all CIs ($\sum$Freq*Weight)
  - Rejection rate: Percentage of MECIs that could not be mapped on the CRFU
  - Mapping rate: Percentage of MECIs that could be mapped on the CRFU
Inputs/Outputs

![Graph showing the mapping rate for inputs and outputs with increasing number of inputs/outputs.](image-url)
Functional Units

![Graph showing the mapping rate vs. number of FUs.](image)
Width/Depth

Number of Width and Depth

Mapping Rate

- Width without constraints
- Depth without constraints
- Width with constraints
- Depth with constraints
CRFU Architecture
Supporting Conditional Execution

[Diagram showing a circuit with multiple functional units (FU1, FU2, FU3, FU4) and selector-mux configuration bits.]
Synthesis result

- Synopsys tools
- Hitachi 0.18 μm
- Area: 2.1 mm²
- Configuration bits: 615 bits

### Delay

<table>
<thead>
<tr>
<th>Depth of DFG of MECI</th>
<th>Delay (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.2</td>
</tr>
<tr>
<td>2</td>
<td>4.2</td>
</tr>
<tr>
<td>3</td>
<td>6.1</td>
</tr>
<tr>
<td>4</td>
<td>7.9</td>
</tr>
<tr>
<td>5</td>
<td>9.8</td>
</tr>
</tbody>
</table>
Experiment setup

- 22 applications of Mibench
- Simplescalar

<table>
<thead>
<tr>
<th>Issue</th>
<th>4-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1- I cache</td>
<td>32K, 2 way, 1 cycle latency</td>
</tr>
<tr>
<td>L1- D cache</td>
<td>32K, 4 way, 1 cycle latency</td>
</tr>
<tr>
<td>Unified L2</td>
<td>1M, 6 cycle latency</td>
</tr>
<tr>
<td>Execution units</td>
<td>4 integer, 4 floating point</td>
</tr>
<tr>
<td>RUU size &amp; Fetch queue size</td>
<td>64</td>
</tr>
<tr>
<td>Branch predictor</td>
<td>bimodal</td>
</tr>
<tr>
<td>Branch prediction table size</td>
<td>2048</td>
</tr>
<tr>
<td>Extra branch misprediction latency</td>
<td>3</td>
</tr>
</tbody>
</table>
Speedup CIs & MECIs

![Graph showing speedup for various benchmarks comparing CIs and MECIs. The x-axis represents different benchmarks such as basicmath, qsort, susan, jpeg, lame, dijkstra, patricia, stringsearch, blowfish, rijndael, sha, adpcm, crc, ft, gsm, and Average. The y-axis represents speedup ranging from 1 to 3.]
Effect of clock frequency of speedup

The diagram illustrates the speedup achieved at various clock frequencies (200 MHz, 250 MHz, 300 MHz, 350 MHz, 400 MHz) for a range of applications. Each bar represents the speedup at a specific frequency compared to a baseline. The applications include bitcnts, basicmath, qsort, susan, jpeg, lAME, dijkstra, patricia, stringsearch, blowfish, rijndael, sha, adpcm, crc, fft, gsm, and the average. The x-axis represents the applications, while the y-axis shows the speedup.
Conclusion

- Our experimental results show that by extending custom instructions over multiple HBBs the average speedup increases by 46% compared to the custom instructions which are limited to only one HBB. This is achieved in return for 83% more hardware and 20% more configuration bits. Utilizing connections with different length are helpful for supporting larger custom instructions with the available number of FUs.
Future work

- Energy evaluation of the ADEXOR
- Exploring the design space of CRFU architecture (To study the effect of number of inputs, outputs, FU on the speedup, area and power)
Thank you for your attention
Introduction (1/2)

- Efficiency and flexibility in embedded system design
  - Both are critical
  - Both are conflicting design goals
  - Custom hardwired feature for more efficiency (performance and energy)
  - Programmable feature for more flexibility

- Design challenges for future embedded systems
  - More required efficiency (performance and energy) for future embedded systems
  - Higher manufacturing and NRE cost of new nanometer-scale technologies
  - Higher cost and risk in development
  - Shorter time-to-market
Introduction (2/2)

- Custom instructions
  - Effective technique for improving efficiency
  - Custom functional units are required (manufacturing, NRE and design cost)

- Our proposed approach
  - Generating custom instruction after chip-fabrication
  - Proposing multi-exit custom instructions
  - Replacing custom functional units with a reconfigurable functional unit with conditional execution
Generating Multi-Exit Custom Instructions

- Motivating example
  - \textit{adpcm} loop
Generating Multi-Exit Custom Instructions

Generating Hot Instruction Sequence
1. MECIs should not cross loop boundaries
2. Sorting loop from innermost to outermost
3. Reading HBBs and linking them to generate Hot Instruction Sequence
4. Sort other remaining HBBs in ascending order considering their start address

Function MAKE_HIS (objfile, HIS, start_addr)
1 if (HBB with start_addr is not included in previous MECIs) then
   read_add_HBB2HIS (objfile, HBB(start_addr), HIS) else return;
2 switch last_instruction(HBB)
3 case (indirect jump, return or call):
   return;
4 case (direct jump): MAKE_HIS(objfile, HIS, target address of jump);
5 case (branch):
   5-1 if (it is hot backward) then return;
   5-2 elsif (not-taken direction is hot)
       then MAKE_HIS(objfile, HIS, target address of not-taken direction) else return;
   5-3 if (taken direction is hot) then
       MAKE_HIS(objfile, HIS, target address of taken direction) else return;
6 default: return;
Generating Multi-Exit Custom Instructions

- MECIs include fixed point instructions except *multiply*, *divide* and *load*. At most on *store* and five *branches*.
- A MECI can have at most four exit points
  - branch with only one hot direction
  - *indirect jump* and *return*
  - *call*
  - hot backward branch
  - an instruction where its next instruction is *non-executable*.
- If both directions of a branch are hot, both corresponding HBBs are added.
Executing MECIs on the CRFU
Proposing an Architecture for the CRFU

Design Methodology

First Phase
- Application
- Generating HIS
- Custom Instructions (Cls)
- Mapping (MapToolP1) and AnalyzingCls for Determining the CRFU Preliminary Architecture
- CRFU Primary Architecture

Second Phase
- Mapping Process (MapToolP2)
- Mappable Cls (MapCIP2)
- Mapping is successful: Yes
- Rejected Cls

Third Phase
- Integrated Temporal Partitioning & Mapping (IntegFrameP3)
- Mappable Cls (MapCIP3)
- Fixing Final CRFU Architecture
- Final CRFU Architecture

Fourth Phase
- Modifying Mapping Process (MapToolP4) & IntegFrameP4
Mapping Tool

A Custom Instruction

1: SUBU  R3, R0, R3
2: ADDU  R10, R0, R0
3: SRA   R8, R10, 0x3
4: SLT   R2, R3, R8
5: BNE   R0,400488, R2
Integrated Framework (1/2)

- Integrated Framework
  - Performs an integrated temporal partitioning and mapping process
  - Takes rejected CIs as input
  - Partitions them to appropriate mappable CIs
  - Adds nodes to the current partition while architectural constraints are satisfied
  - The ASAP level of nodes represents their order to execute according to their dependencies
Integrated Framework (2/2)

- Incremental HTP
  - The node with the highest ASAP level is selected and moved to the subsequent partition.

- Nodes selection and moving order: 15, 13, 11, 9, 14, 12, 10, 8, 3 and 7.
Supporting Conditional Execution
## Execution time of configuration phase

<table>
<thead>
<tr>
<th>Application</th>
<th>Exec. time (Seconds)</th>
<th>Application</th>
<th>Exec. time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>225</td>
<td>gsm</td>
<td>461</td>
</tr>
<tr>
<td>bitcounts</td>
<td>331</td>
<td>lame</td>
<td>526</td>
</tr>
<tr>
<td>blowfish</td>
<td>94</td>
<td>patricia</td>
<td>84</td>
</tr>
<tr>
<td>basicmath</td>
<td>34</td>
<td>qsort</td>
<td>233</td>
</tr>
<tr>
<td>cjjpeg</td>
<td>75</td>
<td>rijndael</td>
<td>68</td>
</tr>
<tr>
<td>crc</td>
<td>132</td>
<td>sha</td>
<td>29</td>
</tr>
<tr>
<td>dijkstra</td>
<td>101</td>
<td>stringsearch</td>
<td>3</td>
</tr>
<tr>
<td>djpeg</td>
<td>9</td>
<td>susan</td>
<td>122</td>
</tr>
<tr>
<td>fft</td>
<td>36</td>
<td>Average</td>
<td>150.8</td>
</tr>
</tbody>
</table>
Effect of connections on mapping rate

- By deleting the connections with length more than one, 24.2% of MECIs can not be mapped.
General Overview of ADEXOR Architecture

- **Main components**
  - **Base processor**
    - 4-issue in-order RISC processor
  - **Reconfigurable functional units (CRFU)**
    - Coarse grain
    - Based on matrix of functional units (FUs)
    - Multi-cycle
    - Parallel with other functional units of the base processor
    - Read/write from/to register file
    - Functions and connections are controlled by configuration bits
  - **Configuration memory**
    - To keep the configuration data of CRFU for multi-exit custom instructions (MECIs)
  - **Counters**
    - Control read/write ports of register file and select between CRFU and processor functional units
Speedup CIs & MECIs

- The number of inputs, outputs and FUs are the same
- simpler connections and FUs and does not support conditional execution.
- Area: 1.15 mm²
- Delay for a CI with a critical length of five is 7.66 ns.
- Each CI configuration needs 512 bits.
- The average number of instructions included in CIs (one HBB) is 6.39 instructions and for MECIs is 7.85 instructions.
MECIs vs. CI

![Graph showing % speedup enhancement for various programs]

- Basicmath
- Qsort
- Susan
- Djpeg
- Djpeg
- Lame
- Patricia
- Stringsearch
- Blowfish
- Rijndael
- Sha
- Adpcm
- Crc
- Fft
- Gsm
- Average
## Distribution of functions

<table>
<thead>
<tr>
<th></th>
<th>row1</th>
<th>row2</th>
<th>ro3</th>
<th>row4</th>
<th>row5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>and</strong></td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>Or</strong></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Xor</strong></td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>Nor</strong></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Add/sub</strong></td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td><strong>Shift</strong></td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td><strong>Compare</strong></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Control Bits & Immediate Data

- **375** bits are needed as Control Bits for
  - Multiplexers
  - Functional Units
- **240** bits are needed for Immediates
- Each CI configuration needs (308+204 = 615 bits)