Custom Processor Core
Construction from C Code

Jelena Trajkovic and Daniel D. Gajski
Center for Embedded Computer Systems
UC Irvine
Outline

- Introduction
- Initial Data path Extraction
- Data path Optimization
- Experimental results
- Conclusion and Future Directions
Introduction

Application implementation options

1. Chose processor, compile application
   C Cheaper, shorter time-to-market
   C Scalable with code size
   D May not have resources for optimal execution

2. Design HW for the application
   C Optimal components and structures
   D Longer development time
   D Not scalable

Our goal: benefits from 1 & quality of 2
Problem Definition

What is the best data path to execute given C code?

“The best” architecture is one that meets given constraints

Manual design

Not scalable for large application

Solution: Automatically extract data path from C code
Approach (1/2)

p Traditional
n Data path and controller are generated simultaneously
  p Applicable to small code size

p Proposed
n Separate *Data path* and controller generation
n Derive Data path from C code
Approach (2/2)

- Our design flow
  - Extract *Initial Data path (IDp)*
    - For max performance
  - Optimize Data path
    - Iteratively based on designer constraints

- Benefits
  - Automate data path extraction
  - Standard input – C reference code
  - Scalability (~10000 lines of code)
  - Controllability of data path design
  - Quality
Outline

- Introduction
- Initial Data path Extraction
- Data path Optimization
- Experimental results
- Conclusion and Future Directions
Initial Data path Extraction

- **Set of C code properties**
  - Data types, operators, parallelism, loops, dependences

- **Set of HW components and templates**
  - Functional units, storage elements, interconnect
  - Connectivity scheme, pipelining

- **Matching heuristics**
  - Date types and representation → bit-width
  - Parallelism → number of FUs, number of registers in RF, pipelining
Example: C Code Properties → HW

- **Set of operations**
  \{+, -, shr, *, >\}

- **Max concurrent occurrence**
  \( m_+ = 3, m_- = 2, m_{shr} = 1, m_* = 2, m_> = 1 \)

- **Mapping operations → FU**
  + → ALU, - → ALU, shr → ALU, * → mul, > → comp

- **Max concurrent transfers of source and destination operands**
  - Source and destination busses
  - Greedy interconnect

- **Create Initial Data path (IDp)**
Outline

- Introduction
- Initial Data path Extraction
- Data path Optimization
- Experimental results
- Conclusion and Future Directions
Main steps in Data path Optimization

- Compile and profile IDp
- Select *Critical code* to be optimized
- Create *Usage Plot* for components
- Estimate *Timing Overhead* from *Resource constraints*
- Balance out the remaining components of Dp
Critical Code Extraction

- **Critical Code**
  - Set of Basic Blocs (BB)
  - Contributes the most to the execution time

- **Selects based on relative length (l) and frequency (f)**
  - High length
  - High frequency
  - High frequency-length product

- **Designer specifies Critical Code**
  - By selecting parameters $P_l$, $P_f$, $P_{fl}$
Usage Plot and Timing Overhead

- For selected BBs create usage plot
  - Usage per cycle
  - Annotate with execution frequency

- Resource constraint
  - e.g. adder = 2

- Model delayed execution
  - Extra operations “executed” in available cycles
  - Estimate $d_c$ – number of extra cycles

- The smallest $d_c$ becomes *Timing Overhead*
Balancing Unrestricted Components

- Select designs to be evaluated
  - Set number of components with resource constraints
  - Set other resources to values in IDp
  - Select a component type to be varied

- E.g. multiplier: two designs
  - With one or with two multipliers
  - Having one multiplier satisfies Timing Overhead
Outline

- Introduction
- Initial Data path Extraction
- Data path Optimization
- Experimental results
- Conclusion and Future Directions
Experimental Results

<table>
<thead>
<tr>
<th>Bench</th>
<th>(P₁, P₂, P₃) [%]</th>
<th>LoC</th>
<th>Gen. Time [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-pipe</td>
</tr>
<tr>
<td>bdist2</td>
<td>(60, 50, 45)</td>
<td>61</td>
<td>0.2</td>
</tr>
<tr>
<td>Sort</td>
<td>(80, 60, 45)</td>
<td>33</td>
<td>0.1</td>
</tr>
<tr>
<td>dct32</td>
<td>(18, 65, 50)</td>
<td>1006</td>
<td>1.3</td>
</tr>
<tr>
<td>Mp3</td>
<td>(30, 55, 50)</td>
<td>13898</td>
<td>15.6</td>
</tr>
</tbody>
</table>

- Two set of experiments to be shown
  - Interactive design exploration
  - Design refinement quality

- Benchmarks
  - bdist2 (from MPEG2 encoder), Sort (bubble sort), dct32 (from MP3 decoder) and Mp3 (decoder)
Interactive Design Exploration

p  Designer specifies
  n  Partial resource constraint
  n  Parameters for critical code extraction
  n  Design choice (pipelined/non-pipelined)

p  Baseline:
  n  MIPS-style: ALU, multiplier, 128-entry RF2x1 and 2 source and 1 destination bus
  n  A divider for Mp3 baseline implementation
  n  Memory size customized per application
Description of Generated Architectures

- Difference from baseline

<table>
<thead>
<tr>
<th>Bench</th>
<th>Pipe</th>
<th>ALU1</th>
<th>ALU2</th>
<th>ALU3</th>
<th>RF2x1</th>
<th>RF4x2</th>
<th>RF6x3</th>
<th>RF8x4</th>
<th>RF16x8</th>
<th>IDp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bdist2</td>
<td>N</td>
<td></td>
<td>#R=64</td>
<td>#R=64</td>
<td>#R=64</td>
<td>#R=64</td>
<td>#R=64</td>
<td>#R=64</td>
<td>#R=64</td>
<td>Rf 16x8, 3 Alu, 2 Mul</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td></td>
<td>#R=32</td>
<td>#R=32</td>
<td>#R=32</td>
<td>#R=32</td>
<td>#R=32</td>
<td>#R=32</td>
<td>#R=32</td>
<td></td>
</tr>
<tr>
<td>Sort</td>
<td>N</td>
<td>#R=16</td>
<td>#R=16</td>
<td>#R=16</td>
<td>#R=16</td>
<td>#R=16</td>
<td>#R=16</td>
<td>#R=16</td>
<td>#R=16</td>
<td>Rf 16x8, 2 Alu</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>#R=16</td>
<td>#R=16</td>
<td>#R=16</td>
<td>#R=16</td>
<td>#R=16</td>
<td>#R=16</td>
<td>#R=16</td>
<td>#R=16</td>
<td></td>
</tr>
<tr>
<td>dct32</td>
<td>N</td>
<td>Rf4x2</td>
<td></td>
<td>Rf6x3</td>
<td>Rf8x4</td>
<td>-</td>
<td>2 Alu</td>
<td>3 Alu</td>
<td>3 Alu</td>
<td>Rf 16x8, 3 Alu, 2 Mul</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Rf4x2</td>
<td></td>
<td>Rf4x2</td>
<td>Rf6x3</td>
<td>-</td>
<td>2 Alu</td>
<td>2 Alu</td>
<td>3 Alu</td>
<td></td>
</tr>
<tr>
<td>Mp3</td>
<td>N</td>
<td></td>
<td>Rf6x3</td>
<td>Rf8x4</td>
<td>-</td>
<td>2 Alu</td>
<td>3 Alu</td>
<td>3 Alu</td>
<td>3 Alu</td>
<td>Rf 16x8, 3 Alu, 2 Mul</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Rf4x2</td>
<td></td>
<td>Rf6x3</td>
<td>Rf6x3</td>
<td>-</td>
<td>2 Alu</td>
<td>2 Alu</td>
<td>2 Alu</td>
<td></td>
</tr>
</tbody>
</table>
Example: Mp3

Baseline

MIPS-style: ALU, multiplier, 128-entry RF2x1 and 2 source and 1 destination bus, divider

<table>
<thead>
<tr>
<th>Bench</th>
<th>Pipe</th>
<th>ALU1</th>
<th>ALU2</th>
<th>ALU3</th>
<th>RF2x1</th>
<th>RF4x2</th>
<th>RF6x3</th>
<th>RF8x4</th>
<th>RF16x8</th>
<th>IDp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mp3</td>
<td>N</td>
<td>-</td>
<td>Rf6x3</td>
<td>Rf8x4</td>
<td></td>
<td>2 Alu</td>
<td>3 Alu</td>
<td>3 Alu</td>
<td>3 Alu</td>
<td>Rf 16x8, 3 Alu, 2 Mul</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Rf4x2</td>
<td>Rf6x3</td>
<td>Rf6x3</td>
<td>-</td>
<td>2 Alu</td>
<td>2 Alu</td>
<td>2 Alu</td>
<td>2 Alu</td>
<td></td>
</tr>
</tbody>
</table>
Number of Execution Cycles

- **Non-pipelined**
  - bdist2 - biggest saving for IDp
  - Sort – smallest improvement
  - dct32 – ALU3
  - Mp3 – ALU2 optimum

- **Pipelined**
  - less overall saving
Total Execution Time

- Post-synthesis
  - bdist2
    - pipelined RF4x2
  - Sort
    - Non-pipelined ALU1
      - Reduced resources
    - Non-pipelined ALU2
      - Large Tclk
  - dct32
    - Non-pipelined ALU2
      - Large Tclk
  - Mp3
    - Non-pipelined ALU2
      - Large Tclk
Design Refinement Quality

- Relative to manual design
  - dct32

- Generated vs. manual
  - No.cycles: 23% - 80% extra
  - Tclk: 29% speedup – 20% slowdown
  - Best Texe: RF4x2: 23% extra

- Generated vs. academic HLS
  - Generated has better performance
  - HLS has better area

- Design time: Generated → 2.3 sec vs. Manual → 3-man week
Conclusion

Contributions
- Algorithm for basic block analysis and core generation
- Iterative algorithm for optimizing resource utilization
- Demonstration on large C examples (>13K LOC)

Benefits
- Automatic C input to core datapath construction
- Scalable to any size of C code
- Datapath construction controllable using designer constraints
- Generated core quality comparable to manual design

Future work
- Automatic pipeline configuration from C code
- Forwarding based on static C code analysis
Acknowledgments

- The authors wish to thank Pramod Chandraiah for providing the Mp3 source code.

- Many thanks to Pramod Chandraiah, Weiwei Chen, Gunar Schirner, Lin Yang and Bin Zhang for manual designs and Roger Ang, Hansu Cho and Dongwan Shin for manual and HLS designs for dct32.

- We would also like to thank Mehrdad Reshadi and Bita Gorjiara for compiler support and Verilog generator.
Thank You

Questions?