SPARK: A Parallelizing High-Level Synthesis Framework

Sumit Gupta

Rajesh Gupta, Nikil Dutt, Alex Nicolau

Center for Embedded Computer Systems
University of California, Irvine and San Diego
http://www.cecs.uci.edu/~spark

Supported by Semiconductor Research Corporation & Intel Inc
System Level Synthesis

System Level Model

Task Analysis

HW/SW Partitioning

High Level Synthesis

Hardware Behavioral Description

Software Behavioral Description

Software Compiler

ASIC

FPGA

I/O

Memory

Processor Core

Copyright Sumit Gupta 2003
High Level Synthesis

Transform behavioral descriptions to RTL/gate level

From C to CDFG to Architecture

\[
\begin{align*}
  x &= a + b; \\
  c &= a < b; \\
  \text{if } (c) \text{ then} \\
  d &= e - f; \\
  \text{else} \\
  g &= h + i; \\
  j &= d \times g; \\
  l &= e + x;
\end{align*}
\]
High Level Synthesis

Transform behavioral descriptions to RTL/gate level

From C to CDFG to Architecture

Problem # 1: Poor quality of HLS results beyond straight-line behavioral descriptions

Problem # 2: Poor/No controllability of the HLS results
Outline

- Motivation and Background
- Our Approach to Parallelizing High-Level Synthesis
- Code Transformations Techniques for PHLS
  - Parallelizing Transformations
  - Dynamic Transformations
- The PHLS Framework and Experimental Results
  - Multimedia and Image Processing Applications
  - Case Study: Intel Instruction Length Decoder
- Conclusions and Future Work
High-level Synthesis

- Well-researched area: from early 1980’s
  - Renewed interest due to new system level design methodologies
- Large number of synthesis optimizations have been proposed
  - Either operation level: algebraic transformations on DSP codes
  - or logic level: Don’t Care based control optimizations
  - In contrast, compiler transformations operate at both operation level (fine-grain) and source level (coarse-grain)
- Parallelizing Compiler Transformations
  - Different optimization objectives and cost models than HLS
- Our aim: Develop Synthesis and Parallelizing Compiler Transformations that are “useful” for HLS
  - Beyond scheduling results: in Circuit Area and Delay
  - For large designs with complex control flow (nested conditionals/loops)
Our Approach: **Parallelizing HLS (PHLS)**

- Optimizing Compiler and Parallelizing Compiler transformations applied at **Source-level** (Pre-synthesis) and during **Scheduling**
  - Source-level code refinement using **Pre-synthesis** transformations
  - Code Restructuring by **Speculative** Code Motions
  - Operation **replication** to improve concurrency
  - **Dynamic** transformations: exploit new opportunities during scheduling
PHLS Transformations
Organized into Four Groups

1. **Pre-synthesis**: Loop-invariant code motions, Loop unrolling, CSE

2. **Scheduling**: Speculative Code Motions, Multi-cycling, Operation Chaining, Loop Pipelining

3. **Dynamic**: Transformations applied dynamically during scheduling: Dynamic CSE, Dynamic Copy Propagation, Dynamic Branch Balancing

4. **Basic Compiler Transformations**: Copy Propagation, Dead Code Elimination
Speculative Code Motions

Operation Movement to reduce impact of Programming Style on Quality of HLS Results

Across Hierarchical Blocks

Speculation

Early Condition Execution
Evaluates conditions As soon as possible

Conditional Speculation

Reverse Speculation

If Node

Operation Movement to reduce impact of Programming Style on Quality of HLS Results

Copyright Sumit Gupta 2003
Dynamic Transformations

- Called “dynamic” since they are applied during scheduling (versus a pass before/after scheduling)

- Dynamic Branch Balancing
  - Increase the scope of code motions
  - Reduce impact of programming style on HLS results

- Dynamic CSE and Dynamic Copy Propagation
  - Exploit the Operation movement and duplication due to speculative code motions
    - Create new opportunities to apply these transformations
  - Reduce the number of operations
Dynamic Branch Balancing

Resource Allocation

Original Design

Scheduled Design

Unbalanced Conditional

Longest Path

If Node

BB 0

BB 1

BB 2

BB 3

BB 4

T F

+ a

+ b

− c

− d

+ a

− c

+ b

− e

− d

Original Design

Scheduled Design
Insert New Scheduling Step in Shorter Branch

Resource Allocation

If Node

Original Design

Scheduled Design

Copyright Sumit Gupta 2003
Dynamic Branch Balancing inserts new scheduling steps

- Enables Conditional Speculation
- Leads to further code compaction
Dynamic CSE: Going beyond Traditional CSE

C Description

\[ a = b + c; \]
\[ cd = b < c; \]
\[ if (cd) \]
\[ d = b + c; \]
\[ else \]
\[ e = g + h; \]

HTG Representation

If Node

BB 0

BB 1

BB 2

BB 3

BB 4

After Traditional CSE

BB 0

BB 1

BB 2

BB 3

BB 4

If Node

BB 2

BB 3

BB 4
Dynamic CSE: Going beyond Traditional CSE

We use notion of Dominance of Basic Blocks

- Basic block BBi dominates BBj if all control paths from the initial basic block of the design graph leading to BBj goes through BBi

- We can eliminate an operation opj in BBj using common expression in opi if BBi dominates BBj

```
a = b + c;
cd = b < c;
if (cd)
d = b + c;
else
e = g + h;
```
New Opportunities for “Dynamic” CSE Due to Code Motions

Scheduler decides to Speculate

CSE not possible since BB2 does not dominate BB6

CSE possible now since BB0 does not dominate BB6
New Opportunities for “Dynamic” CSE
Due to Code Motions

Scheduler decides to Speculate

If scheduler moves or duplicates an operation $\text{op}$, apply CSE on remaining operations using $\text{op}$
Scheduler decides to Conditionally Speculate

a = b + c

d = b + c

a' = b + c

a = a'

d = b + c
Condition Speculation & Dynamic CSE

Scheduler decides to Conditionally Speculate

BB 0

BB 1

BB 2

a = b + c

BB 3

BB 4

BB 5

BB 6

BB 7

BB 8

d = b + c

BB 2

BB 3

BB 4

BB 5

BB 6

BB 7

BB 8

da' = b + c

BB 8
Use the notion of dominance by groups of basic blocks

All Control Paths leading up to BB8 come from either BB1 or BB2: => BB1 and BB2 together dominate BB8
Loop Shifting: An Incremental Loop Pipelining Technique

Loop Node

BB 0

BB 1

BB 2

BB 3

BB 4

Loop Exit

Loop Shifting

Copyright Sumit Gupta 2003
Loop Shifting: An Incremental Loop Pipelining Technique

**Loop Node**

- **BB 0**
- **BB 1**
- **BB 2**
- **BB 3**
- **BB 4**

Loop Shifting Compaction

- **BB 0**
- **BB 1**
- **BB 2**
- **BB 3**
- **BB 4**
SPARK High Level Synthesis Framework

C Input

Parser Front End

PreSynthesis Optimizations
- Loop Unrolling
- Loop Fusion
- Loop Invariant Code Motion
- CSE, IVA, Copy Propagation, Inlining, Dead Code Elimination

Scheduling and Allocation

Heuristics
- HTG Scheduling Walker
- Candidate OpWalker
- Get Available Ops
- Loop Pipelining

Transformation Toolbox
- Percolation/Trailblazing
- Speculative Code Motions
- Chaining Across Conditions
- Dynamic CSE & Copy Prop

Resource Binding & Control Synthesis

Operation/Variable Binding

FSM Generation/Optimiz.

Code Generation BackEnd

Synthesizable RTL VHDL, Behavioral VHDL & C
SPARK Parallelizing HLS Framework

- C input and **Synthesizable** RTL VHDL output
- **Tool-box** of Transformations and Heuristics
  - Each of these can be developed independently of the other
- **Script based** control over transformations & heuristics
- Hierarchical Intermediate Representation (HTG's)
  - Retains structural information about design (conditional blocks, loops)
  - Enables efficient and structured application of transformations
- **Complete HLS tool**: Does Binding, Control Synthesis and Backend VHDL generation
  - Interconnect Minimizing Resource Binding
- Enables **Graphical Visualization** of Design description and intermediate results
- 100,000+ lines of C++ code
Synthesizable C

- ANSI-C front end from Edison Design Group (EDG)
- Features of C not supported for synthesis
  - Pointers
    - However, Arrays and passing by reference are supported
  - Recursive Function Calls
  - Gotos
- Features for which support has not been implemented
  - Multi-dimensional arrays
  - Structs
  - Continue, Breaks
- Hardware component generated for each function
  - A called function is instantiated as a hardware component in calling function
Resource Utilization Graph

Scheduling
Example of Complex HTG

- Example of a real design: MPEG-1 pred2 function
  - Just for demonstration; you are not expected to read the text
- Multiple nested loops and conditionals
Experiments

- Results presented here for
  - Pre-synthesis transformations
  - Speculative Code Motions
  - Dynamic CSE

- We used SPARK to synthesize designs derived from several industrial designs
  - MPEG-1, MPEG-2, GIMP Image Processing software
  - Case Study of Intel Instruction Length Decoder

<table>
<thead>
<tr>
<th>Scheduling Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of States in FSM</td>
</tr>
<tr>
<td>Cycles on Longest Path through Design</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VHD L: Logic Synthesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical Path Length (ns)</td>
</tr>
<tr>
<td>Unit Area</td>
</tr>
</tbody>
</table>
## Target Applications

<table>
<thead>
<tr>
<th>Design</th>
<th># of Ifs</th>
<th># of Loops</th>
<th># Non-Empty Basic Blocks</th>
<th># of Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPEG-1</td>
<td>4</td>
<td>2</td>
<td>17</td>
<td>123</td>
</tr>
<tr>
<td>pred1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPEG-1</td>
<td>11</td>
<td>6</td>
<td>45</td>
<td>287</td>
</tr>
<tr>
<td>pred2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPEG-2</td>
<td>18</td>
<td>4</td>
<td>61</td>
<td>260</td>
</tr>
<tr>
<td>dp_frame</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GIMP tiler</td>
<td>11</td>
<td>2</td>
<td>35</td>
<td>150</td>
</tr>
</tbody>
</table>
Scheduling & Logic Synthesis Results

MPEG-1 Pred1 Function

- Longest Path (cycles): [Bar Graph]
- Critical Path (nanoseconds): [Bar Graph]
- Total Delay (cycles * nanoseconds): [Bar Graph]
- Unit Area: [Bar Graph]

MPEG-1 Pred2 Function

- Longest Path (cycles): [Bar Graph]
- Critical Path (nanoseconds): [Bar Graph]
- Total Delay (cycles * nanoseconds): [Bar Graph]
- Unit Area: [Bar Graph]

Non-speculative CMs: Within BBs & Across Hier Blocks

- + Pre-Synthesis Transforms

+ Speculative Code Motions

+ Dynamic CSE
Overall: 63-66 % improvement in Delay
Almost constant Area
Scheduling & Logic Synthesis Results

### MPEG-2 DpFrame Function
- **Longest Path (l\_cyc)**
- **Critical Path (c\_ns)**
- **Total Delay (c*l)**
- **Unit Area**

### GIMP Tiler Function
- **Longest Path (l\_cyc)**
- **Critical Path (c\_ns)**
- **Total Delay (c*l)**
- **Unit Area**

#### Non-speculative CMs: Within BBs & Across Hier Blocks
- + Speculative Code Motions
- + Pre-Synthesis Transforms
- + Dynamic CSE

Copyright Sumit Gupta 2003
Scheduling & Logic Synthesis Results

**MPEG-2 DpFrame Function**
- Longest Path: 0.4 units
- Critical Path: 0.2 units
- Total Delay: 0.6 units
- Unit Area: 1.1 units

**GIMP Tiler Function**
- Longest Path: 0.5 units
- Critical Path: 0.3 units
- Total Delay: 0.8 units
- Unit Area: 1.3 units

Overall: 48-76% improvement in Delay
Almost constant Area
Case Study: Intel Instruction Length Decoder

Stream of Instructions ➔ Instruction Buffer ➔ Instruction Length Decoder

First Insn  Second Insn  Third Instruction
Example Design: ILD Block from Intel

- Case Study: A design derived from the Instruction Length Decoder of the Intel Pentium® class of processors
  - Decodes length of instructions streaming from memory
    - Has to look at up to 4 bytes at a time
  - Has to execute in one cycle and decode about 64 bytes of instructions

- Characteristics of Microprocessor functional blocks
  - Low Latency: Single or Dual cycle implementation
  - Consist of several small computations
  - Intermix of control and data logic
Basic Instruction Length Decoder: Initial Description

Total Length Of Instruction = Length Contribution 1 + Length Contribution 2 + Length Contribution 3 + Length Contribution 4

Need Byte 4 ?
Need Byte 3 ?
Need Byte 2 ?

Byte 1 Byte 2 Byte 3 Byte 4
Instruction Length Decoder: Decoding 2\textsuperscript{nd} Instruction

After decoding the length of an instruction:
- Start looking from next byte
- Again examine up to 4 bytes to determine length of next instruction
Instruction Length Decoder: Parallelized Description

- Need Byte 4?
- Need Byte 3?
- Need Byte 2?
- Need Byte 1?

Total Length Of Instruction

Speculatively calculate the length contribution of all 4 bytes at a time
Determine actual total length of instruction based on this data
ILD: Extracting Further Parallelism

- Speculatively calculate length of instructions assuming a new instruction starts at each byte.
- Do this calculation for all bytes in parallel.
- Traverse from 1st byte to last.
- Determine length of instructions starting from the 1st till the last.
- Discard unused calculations.
### Initial: Multi-Cycle Sequential Architecture

```plaintext
ResetArray(Mark);
NextStartByte = 0;
for (i=0; i < n; i++) {
    if (i == NextStartByte) {
        lc1 = LengthContribution_1(i);
        if (Need_2nd_Byte(i)) {
            lc2 = LengthContribution_2(i+1);
            if (Need_3rd_Byte(i+1)) {
                lc3 = LengthContribution_3(i+2);
                if (Need_4th_Byte(i+2)) {
                    lc4 = LengthContribution(i+3);
                    Length = lc1 + lc2 + lc3 + lc4;
                } else
                    Length = lc1 + lc2 + lc3;
            } else
                Length = lc1 + lc2;
        } else
            Length = lc1;
    } /* if (i == NextStartByte) */
len[i] = Length;
NextStartByte += len[i];
Mark[i] = 1;
} /* end of for i loop */
```
ResetArray(Mark);
NextStartByte = 0;
for (i=0; i < n; i++) {
    if (i == NextStartByte) {
        lc1 = LengthContribution_1(i);
        if (Need_2nd_Byte(i)) {
            lc2 = LengthContribution_2(i+1);
            if (Need_3rd_Byte(i+1)) {
                lc3 = LengthContribution_3(i+2);
                if (Need_4th_Byte(i+2)) {
                    lc4 = LengthContribution(i+3);
                    Length = lc1 + lc2 + lc3 + lc4;
                } else
                    Length = lc1 + lc2 + lc3;
            } else
                Length = lc1 + lc2;
        } else
            Length = lc1;
    } /* if (i == NextStartByte) */

    len[i] = Length;
    NextStartByte += len[i];
    Mark[i] = 1;
} /* end of for i loop */

Speculate Operations, Fully Unroll Loop, Eliminate Loop Index Variable

Results(0) = DataCalculation(0,1,2,3);
Results(1) = DataCalculation(1,2,3,4);
...
Length(0) = ControlLogic(Results(0));
Length(1) = ControlLogic(Results(1));
...

if (0 == NextStartByte) {
    NextStartByte += length[0];
    Mark[0] = 1;
}
if (1 == NextStartByte) {
    NextStartByte += length[1];
    Mark[1] = 1;
}
...
Our toolbox approach enables us to develop a script to synthesize applications from different domains.

Final design looks close to the actual implementation done by Intel.
Conclusions

- Parallelizing code transformations enable a new range of HLS transformations
  - Provide the needed improvement in quality of HLS results
    - Possible to be competitive against manually designed circuits.
  - Can enable productivity improvements in microelectronic design
- Built a synthesis system with a range of code transformations
  - Platform for applying Coarse and Fine-grain Optimizations
  - Tool-box approach where transformations and heuristics can be developed
    - Enables the designer to find the right synthesis script for different application domains
  - Performance improvements of 60-70% across a number of designs
  - We have shown its effectiveness on an Intel design
Acknowledgements

- Advisors
  - Professors Rajesh Gupta, Nikil Dutt, Alex Nicolau

- Contributors to SPARK framework
  - Nick Savoiu, Mehrdad Reshadi, Sunwoo Kim

- Intel Strategic CAD Labs (SCL)
  - Timothy Kam, Mike Kishinevsky

- Supported by Semiconductor Research Corporation and Intel SCL
Thank You