Parallelizing High-level Synthesis

Rajesh K. Gupta, Sumit Gupta
University of California, San Diego

mesl.ucsd.edu

Contributors: Nick Savoiu, Sudipta Kundu
Collaborators: Nik Dutt, Alex Nicolau, UC Irvine
**SPARK Vision & Strategy**

- **Vision**
  - Aggressive & global code motions in an attempt to get past the QOR issue in HLS

- **Strategy**
  - Identify *really useful* parallelizing transformations
  - Apply coarse and fine grain HL & compiler optimizations
    - target control flow transformations
    - “Fine grain” loop optimization techniques for multiple and nested loops
    - Mixed IR suitable for fine and coarse grain compiler transformations (similar to other systems such as SUIF)
  - Make it accessible through C, SystemC

- **Advantages**
  - Improve quality and controllability of HLS results.
Our Approach to Parallelizing HLS (PHLS)

- Transformations applied at the source level 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
- Use heuristics to balance parallelism against hardware costs.

Diagram:
- C Input → Original CDFG → Scheduling & Binding → Optimized CDFG → VHDL Output
- Source-Level Compiler Transformations
- Scheduling Compiler & Dynamic Transformations
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
1. Pre-synthesis: Loop Invariant CM

Loop Node

i < n

BB 1

Loop Node

i < n

BB 1

1: a = b + c
2: d = a + c
3: e = e + d
... i = i + 1

BB 2

BB 3

BB 4

Loop Exit

1: a = b + c
2: d = a + c
3: e = e + d
... i = i + 1

BB 2

BB 3

BB 4

Loop Exit
1. Pre-synthesis: Loop Invariant CM

- Reduce number of operations that execute in loops
- Putting code inside loops is a *programming convenience*
- Common situation in media applications
1. Common Sub-Expression Elimination

C Description

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

HTG Representation

```
If Node

BB 0
a = b + c

BB 1
T
BB 2
d = a
F
BB 3
e = g + h

BB 4

After CSE
```
1. Common Sub-Expression Elimination

We use notion of Dominance of Basic Blocks

- A basic block $BB_i$ dominates another basic block $BB_j$ if all control paths from the initial basic block of the design graph leading to $BB_j$ goes through $BB_i$

- We can eliminate an operation $op_j$ in $BB_j$ using common expression in $op_i$ if $BB_i$ dominates $BB_j
2. Scheduling Transformations: Speculative Code Motions

Hierarchical Task Graph Representation

Resource Utilization

Speculative Code Motions

Across Hierarchical Blocks

Speculation

BB0

BB1

BB2

BB3

T

F

If Node

Reverse Speculation

Conditional Speculation

Schedule under Resource Constraints
2. Scheduling Transformations: Speculative Code Motions

Hierarchical Task Graph Representation

Lead to Shorter Schedule Lengths by utilizing resources that are "idle" in earlier cycles
Reduce the impact of programming style (operation placement) on quality of HLS results
Hardware Costs of Speculative Code

Motions: Speculation

c = a < b

d = e + f
g = h + i

BB0
BB1
BB2
BB3

S0
S1

Speculation
If Node

ALU

S1.c
S1.!c
Hardware Costs of Speculative Code

Motions: Speculation

d' = e + f; c = a < b

d = d'
g = h + i

e h f i

If Node

T F

BB1 BB2

BB3

ALU

S0 S1.!c

S0 d'

S1.c

S1.!c

g

d
Hardware Costs of Speculative Code

Motions: **Conditional Speculation**

\[ g = h + i; c = a < b \]
Hardware Costs of Speculative Code

Motions: Conditional Speculation

- If Node
- BB0: \( g = h + i; c = a < b \)
- BB1: \( d = g + f \)
- BB2: \( e = g \)
- BB3: \( d = e + f \)
- BB4: \( h, g, e, i, f \)
- S0: Fewer Cycles but more complex Control and Multiplexing

\( d \) and \( g \) are initial values.

- S1: \( S1.c \) and \( S1.!c \)
- S2: \( S1.c + S2.!c \)
- S3: ALU

Fewer Cycles but more complex Control and Multiplexing
Speculative code motions lead to **two opposite effects** on the control, multiplexing and area costs.

1. **Shorter Schedule lengths**
   - Leads to **Smaller Controllers** (fewer states in State Machine)
   - This leads to smaller area

2. **More multiplexing and control costs** to steer the data
   - Particularly when operations are conditionally speculated
   - Leads to longer critical paths
3. 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
3. Dynamic Branch Balancing

If Node $TF_\text{BB 0}$

BB 1

$+a$

$+b$

$-c$

BB 2

BB 3

$+a$

$-c$

$b = \_d$

BB 4

Unbalanced Conditional

Original Design

Scheduled Design

Resource Allocation

Longest Path

S0

S1

S2

S3
Insert New Scheduling Step in Shorter Branch

Resource Allocation

Original Design

Scheduled Design

If Node

BB 0

BB 1

BB 2

BB 3

BB 4

T

F

Resource Allocation

Original Design Scheduled Design
Insert New Scheduling Step in Shorter Branch

Dynamic Branch Balancing is done
1. While Traversing the design
2. And if it enables Conditional Speculation
3. Dynamic CSE: Going beyond Traditional CSE

C Description

\[
\begin{align*}
    a &= b + c \\
    c &= b < c; \\
    \text{if (c)} \\
    d &= b + c; \\
    \text{else} \\
    e &= g + h;
\end{align*}
\]

HTG Representation

If Node

\[
\begin{align*}
    a &= b + c \\
    d &= a \\
    e &= g + h
\end{align*}
\]

After CSE

\[
\begin{align*}
    a &= b + c \\
    d &= b + c \\
    e &= g + h
\end{align*}
\]
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 dominate BB6
Scheduler decides to Speculate

If scheduler moves or duplicates an operation \( op \), apply CSE on remaining operations using \( op \)
Scheduler decides to Conditionally Speculate
Condition Speculation & Dynamic CSE

- 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

Scheduler decides to Conditionally Speculate
Architecture of the PHLS Scheduler

**Scheduler**
- **IR Walker**
  - Traverses Design to find next basic block to schedule

**Candidate Fetcher**
- Traverses Design to find Candidate Operations to schedule

**Candidate Chooser**
- Calculates Cost of Operations and chooses Operation with lowest cost for scheduling

**Candidate Mover**
- Moves, duplicates and schedules chosen Operation

**Dynamic Transforms**
- Dynamically apply transformations such as CSE on remaining Candidate Operations using scheduled operation
Integrating transformations into Scheduler

- Scheduler
  - IR Walker
  - Candidate Fetcher
  - Candidate Chooser
  - Candidate Mover
  - Dynamic Transforms

Available Operations
- Candidate Walker
- Candidate Validater

- Branch Balancing During Traversal
  - Branch Balancing During Code Motion
  - Apply Speculative Code Motions
  - Determine Code Motions Required to schedule op
The Intermediate Representation

- Hierarchical Task Graph (HTG) is main structure in the intermediate representation (IR)

- Maintains information on:
  - Code structure (IFs, LOOPs)
  - Loop bounds, type (FOR, WHILE)
  - Array accesses are not lowered to address calculation followed by memory access

- Is complete
  - Can regenerate input C code
Use HTG for Hierarchical Code Motions

- Can move operations across large pieces of code without visiting each node in between
SPARK Parallelizing HLS Framework

- C input and **Synthesizable** RTL VHDL output
- **Range of** compiler, parallelizing compiler and HLS **transformations** applied during Pre-synthesis and Scheduling phases
- **Tool-box of** Transformations and Heuristics
  - Each of these can be developed independently of the other
- **Complete HLS tool**: Does Binding, Control Synthesis and Backend VHDL generation
  - Interconnect Minimizing Resource Binding
- Enables **Graphical Visualization** of Design description and intermediate results
- About 100,000 + lines of C++ code
Task Graphs (HTGs) + Data Flow Graphs

C Input

Parser Front End

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

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
High Level Synthesis Framework

Constraints & Resource Library

Parser Front End

PreSynthesis Optimizations

Scheduling and Allocation

Resource Binding & Control Synthesis

Code Generation BackEnd

SPARK IR
- Hierarchical Task Graphs (HTGs)
- Data Flow Graphs

Operation/Variable Binding

Resource Binding & Control Synthesis

Code Generation BackEnd
<table>
<thead>
<tr>
<th>Release</th>
<th>Ver</th>
<th>Download</th>
<th>Change Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>05-20-05</td>
<td>1.3</td>
<td>Linux (Built on Gentoo)</td>
<td>Version 1.2 to 1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Windows (built on XP)</td>
<td></td>
</tr>
<tr>
<td>2-5-04</td>
<td>1.2</td>
<td>Linux (Redhat 9.0)</td>
<td>Version 1.1 to 1.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Solaris 5.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Windows (built on XP)</td>
<td></td>
</tr>
<tr>
<td>1-12-04</td>
<td>1.1</td>
<td>Linux (Redhat 9.0)</td>
<td>Version 1.0 to 1.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Solaris 5.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Windows XP</td>
<td></td>
</tr>
<tr>
<td>12-29-03</td>
<td>1.0</td>
<td>Linux (Redhat 9.0)</td>
<td>Version 0.9 to 1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Solaris 5.7</td>
<td></td>
</tr>
<tr>
<td>11-10-03</td>
<td>0.9</td>
<td>Linux (Redhat 9.0)</td>
<td>Version 0.8 to 0.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Solaris 5.7</td>
<td></td>
</tr>
<tr>
<td>09-19-03</td>
<td>0.8</td>
<td>Linux (Redhat 9.0)</td>
<td>None</td>
</tr>
<tr>
<td>10-09-03</td>
<td></td>
<td>Solaris 5.7</td>
<td>Fixed some file name problems</td>
</tr>
</tbody>
</table>
Experiments

- Experiments for several transformations
  - Pre-synthesis transformations: loop invariant code motions, CSE
  - Speculative Code Motions
  - Dynamic CSE

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

<table>
<thead>
<tr>
<th>Scheduling Results</th>
<th>VHDL: Logic Synthesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of States in FSM</td>
<td>Critical Path Length (ns)</td>
</tr>
<tr>
<td>Cycles on Longest Path through Design</td>
<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 pred1</td>
<td>4</td>
<td>2</td>
<td>17</td>
<td>123</td>
</tr>
<tr>
<td>MPEG-1 pred2</td>
<td>11</td>
<td>6</td>
<td>45</td>
<td>287</td>
</tr>
<tr>
<td>MPEG-2 dp_frame</td>
<td>18</td>
<td>4</td>
<td>61</td>
<td>260</td>
</tr>
<tr>
<td>GIMP tiler</td>
<td>11</td>
<td>2</td>
<td>35</td>
<td>150</td>
</tr>
</tbody>
</table>
Code Motions: Logic Synthesis Results

MPEG Pred2 Function

Normalized Values

Speculative Code Motions

50% reduction in delay with 20% Area increase

Within Basic Blocks & Across Hierar. Blocks
+ Speculation

Reverse Speculation & Early Condition Execution
Condition Speculation
CSE/Dynamic CSE Results

MPEG Pred2 Function

Critical Path (c ns)
Total Delay (c*l ns)
Unit Area

Normalized Values

MPEG Pred1 Function

Speculative Code Motions + Dynamic CSE
75 % reduction in delay with No Area increase

30 % reduction in delay, 25 % reduction in Area
SPARK is readily used here. Rather all the applications we test/demonstrate, invariably needs SPARK. Following is the description of how it fits into with *** Tool chain. *** Tools partition the hot spots of the application and place them in a generated hardware accelerator unit. The hardware accelerator unit has components like a slave bus system, DMA controllers, Controller Units etc, but it was missing the actual compute unit. The designers here were manually writing it. Now SPARK is being used for this task. We feed the C hot loops into the SPARK and get VHDL and Verilog RTL for the compute unit. SPARK has to be changed quite a bit to use it the way we want. To give you an estimate: One engineer modified it to make it work and use-able for us. One engineer worked to get Verilog code out of it. And I am generating the hardware interface. You can say that 3 resources are working on it since last 8 months at least. --------