Verification of programs:
- Normal popular programs in C/SystemC
- (Partial) Programmable hardware
- HW/SW combined

Of Multiple Programs Based on Difference Identification

Masahiro Fujita
VLSI Design and Education Center
University of Tokyo

Message: Should verify HS/SW total systems
In the future should include mechanical parts as well
Outline

• Formal analysis with symbolic simulation
  – If control flow is not complicated, this is enough
  – Ex: Pipelined Tsunami simulation with GPU/FPGA
  – HW/SW combined bug remained in interface part

• Equivalence checking with difference identification
  – Robust identification of difference between two descriptions
  – Examine only the different portions
  – Ex: SoC/embedded system incremental designs

• Software controlled hardware (patchable HW)
  – With thin transistors hardware is no longer so reliable, and SW should help
  – Control flow and function are slightly modifiable
  – Ex: Post-design/development verification and analysis

• Formal analysis through functional dependencies
  – QBF formulation for “func1 can be realized with func2 and func3 ?”
  – SAT-based fast solvers
  – Ex: Debugging of hardware as well as software

Easy case is easily verified

Difficult case can be verified with tactics

But hardware is becoming unreliable and so programmability is a must

Debugging both HW/SW is critical
QBF can be solved with repeated SAT
Outline

• Formal analysis with symbolic simulation
  
  Let’s start with simple cases!

• Software controlled hardware (patchable HW)

• Formal analysis through functional dependencies
Equivalence checking with symbolic simulation

- Symbolically simulate two descriptions for all execution paths
- Check whether the results are equivalent or not with SMT/SAT solvers
  - If control flow is complicated, there can be path explosion problem
  - Simple pipelining can be easily verified by analyzing all the data to be computed
- Let's see simple examples
Symbolic simulation with equivalence class

Description 1 (programs for FPGA/GPU)

\[
a = v_1; \\
b = v_2; \\
add_1 = a + b;
\]

Description 2 (original program)

\[
add_2 = v_1 + v_2;
\]

4 EqvClasses are generated from 4 assignments

E1 \((a, v_1)\)  
E2 \((b, v_2)\)  
E3 \((add_1, a+b)\)  
E4 \((add_2, v_1+v_2)\)

E3 and E4 can be merged by substituting \(a\) with \(v_1\), \(b\) with \(v_2\)
Formal analysis with symbolic simulation

• Based on symbolic simulation with maximally shared graph
  • Size grows linearly with numbers of lines

\[ x_0 = a + b; \]
\[ \text{if } x_0 < 0 \text{ then} \]
\[ x_1 = -x_0; \]
\[ y_1 = y_0; \]
\[ \text{else} \]
\[ x_1 = x_0; \]
\[ y_1 = y_0 + x_0; \]
\[ \text{assert}(0 \leq y_1); \]
Example verification: Tsunami simulation

- Based on the values of many earthquake sensors (wired/wireless), compute how Tsunami wave will propagate.
- Goal: Realize supercomputer level performance in Tsunami simulation with FPGA/GPU.

Compute/Predict Tsunami as fast and accurate as possible.

Generate initial wave from sensor data
Propagate wave by numerically solving partial differential equations.
Partial differential equations to be solved

\[ \dot{\eta} = \text{vertical displacement of water above still water} \]

\[ D = \text{Total water depth} = h + \dot{\eta} \]

\[ g = \text{Acceleration due to gravity} \]

\[ A = \text{horizontal eddy viscosity current} \]

\[ \tau = \text{friction along x or y direction} \]

\[ M = \text{water flux discharge along X direction} \]

\[ N = \text{water flux discharge along Y direction} \]

\[ \frac{\partial \eta}{\partial t} + \frac{\partial M}{\partial x} + \frac{\partial N}{\partial y} = 0 \]

\[ \frac{\partial M}{\partial t} + \frac{\partial}{\partial x} \left( \frac{M^2}{D} \right) + \frac{\partial}{\partial y} \left( \frac{MN}{D} \right) + gD \frac{\partial \eta}{\partial x} + \frac{\tau_x}{\rho} = A \left( \frac{\partial^2 M}{\partial x^2} + \frac{\partial^2 M}{\partial y^2} \right) \]

\[ \frac{\partial N}{\partial t} + \frac{\partial}{\partial x} \left( \frac{MN}{D} \right) + \frac{\partial}{\partial y} \left( \frac{N^2}{D} \right) + gD \frac{\partial \eta}{\partial y} + \frac{\tau_y}{\rho} = A \left( \frac{\partial^2 N}{\partial x^2} + \frac{\partial^2 N}{\partial y^2} \right) \]

Reference: Tsunami Modeling Manual by Prof Nobuo Shuto
Finite difference methods

- Solution of mass conservation equation based on finite difference method

\[- Z(i,j,t+1) = Z(i,j,t) - \left( \frac{dt}{dx} \right) (M(i,j,t) - M(i-1,j,t) + N(i,j,t) - N(i,j-1,t)) \]

Where

- \(i, j = x, y\) coordinate
- \(Z(i,j,t) = \) Water Surface level at time \(t\)
- \(H(i,j,t) = \) Still water depth at time \(t\)
- \(dt = \) temporal step
- \(dx = \) spatial step
- \(M(i,j,1) = \) water flux discharge along \(x\)-axis at time \(t\)
- \(N(i,j,1) = \) water flux discharge along \(y\)-axis at time \(t\)
- \(Z(i,j,2) = \) water surface level at time \(t + dt\)
Wave Height Computation

Mass conservation (Wave height update)

M, N, H, Z

M, N, H, Z

1 second

M, N, H, Z

M, N, H, Z

1 second

M, N, H, Z

M, N, H, Z

Time T

Time T+1

Momentum conservation (Wave height update)
C Implementation (base program)

• Mass and Momentum functions are computed alternatively
  – Each function raster-scans the grids
• Since there is no data dependency between the computations at grids, they can be parallelized
Profiling of Computation time of the software: Typical HW/SW co-design/development

Simulation Cycles and
Computation time of TUNAMI

Some control flows

- Start
- Input
- Initial condition
- Main Loop (t=1; t<=T; t++)
  - Mass Conservation
  - Open Boundary Condition
  - Momentum Conservation
- Stop

Chart showing computation time vs. simulation cycles with different components highlighted.

- moment conservation
- open boundary
- mass conservation
- initial

Legend:
- Purple: moment conservation
- Green: open boundary
- Red: mass conservation
- Blue: initial
## Co-execution

<table>
<thead>
<tr>
<th>C on microprocessor</th>
<th>FPGA (pipeline execution)</th>
<th>GPU (parallel execution)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read earthquake information from files</td>
<td>Mass Conservation</td>
<td>Read data from DRAM</td>
</tr>
<tr>
<td>Compute initial wave</td>
<td>Open Boundary Condition</td>
<td>Main loop</td>
</tr>
<tr>
<td>Load initial wave to DRAM memory of FPGA board</td>
<td>Momentum Conservation</td>
<td>Store results to DRAM</td>
</tr>
<tr>
<td>Run FPGA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Idle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Store results into files</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Mass Conservation**
- **Open Boundary Condition**
- **Momentum Conservation**
Introducing streams

- **Steam** = Sequence of data, functions, or combined

---

Formal equivalence checking with symbolic simulation with manipulation of transformation matrix

---

One stream for each region

One stream for each time step

---

Original program code

Data object (can be a single stream)
Target GPGPU Architecture

NVIDIA Tesla C2075 (Fermi architecture)
14 Streaming Multiprocessors
6GB Main Memory
768KB L2 Cache

Register File (32k words)
Shared Memory /L1 Cache (64KB)

Streaming Multiprocessor (SM)
32 Integer & FP cores
Naïve GPGPU Implementation

Mass Conservation (Z update)

Momentum Conservation (M,N update)

Block (16x16 threads)
Threads in a block shares the shared memory

Warp (32 threads)
Threads in a warp are executed in parallel

[16x16 threads]

[Gidra et al., IEEE HPCC 2011]
Eliminating Synchronization

- Global synchronization can be eliminated by merging Mass and Momentum functions
  - However, Mass and Momentum depend on neighboring values of the block
    - Neighboring values are loaded onto the shared memory
    - Neighboring Z values are also computed
    - Duplicated load & computation do not impact on runtime
Some verification results

• CUDA based implementation

• Speed of Tsunami simulation for 7200 iterations (2 hours)
  – Original C implementation: 78.7 sec
  – GPU implementation: 1.96 seconds (40.2X speedup)

• Formally verified to be equivalent with symbolic simulation
  – All the data to be computed have proven to be the same

• Easy verification with symbolic simulation
Transforming latency-based to throughput-based computation for FPGA (HW)

- **Stream** based programming
  - Communication/buffering becomes explicit
  - Easier for formal analysis as well

- **Works for both FPGA and GPU**
  - And also for many-cores
Pipeline processing for higher throughput

- Latency
  - After receiving input, how many cycles are required to generate its output

- Throughput
  - How frequently input data can be processed

Goal: Faster throughput
= Larger numbers of pipeline stages
FPGA implementation and its formal verification

- FPGA design has been synthesized from data flow graph
  - Automatic pipeline insertions
  - Final Implementation has over 1,200 pipeline stages

- Formal verification with symbolic simulation on the correctness of data flow graphs to be computed is straightforward
int a, b, c;

void fct()
{
    a++;
    if (c > 0)
        b = a + c;
    else
        b = a * c;
    c = b;
}

Hardware compiler automatically highly pipelined circuits (over 1,000 pipelines)
Correctness of DFG with respect to the original program codes can be easily verified with symbolic simulation

A bug remained in interface between HW and SW which need over all verification including HW and SW
Performance of the main loop part

![Bar chart showing performance comparison between C, FPGA, and GPU.]

- **C**: 1.1 GFLOPS
- **FPGA**: 46.0 GFLOPS
- **GPU**: 41.5 GFLOPS

*7200 (loop only)*
Comparison of Power Consumption

- FPGA is much better in terms of energy consumption.
Outline

• Formal analysis with symbolic simulation
• Equivalence checking with difference identification
• Software controlled hardware (patchable HW)
• Formal analysis through functional dependencies
Equivalence checking with identification of description difference

- Symbolic simulation cannot be applied to a whole large designs
  - Because of the path explosion problem
  - Approach: Localizing the areas that are symbolically simulated utilizing differences

\[
\text{return a;}
\]

\[
\text{return a;}
\]

\[
\text{return a;}
\]

\[
\text{return a;}
\]
Equivalence identification with formal methods

• Comparison between Module A and Module B

- Same description
- Different description and non-equivalent
- Different description but proved to be equivalent
System Dependence Graph

- Sufficient representation since verification methods for net-list are applicable to SDG
- Problems
  - Existing SDGs are too complicated (# nodes/edges are huge)
  - Not directly corresponding to abstract syntax trees
Edges in ExSDG

- Control Flow Edge
- Data Dependence Edge
- Control Dependence Edge
- Declaration Dependence Edge
- Parameter In/Out
- Summary Edge (Interprocedural Dependence Edge)
- Parallel Edge
- Communication Edge
- Port Reference Edge
int x, y;
event e1, e2;
void main(void) {
    y = 0;
    x = 5;
    par{
        func1();
        func2();
    }
}

void func1() {
    y = x;
    notify(e1);
    wait(e2);
    x = y;
}

void func2() {
    wait(e1);
    y++;
    y = y * y;
    notify(e2);
}
Hierarchical Dependence

module M1(int in, int out, event e1){
    void main(void){
        wait(e1);
        out = in * in;
    }
};

module M2(int in, int out, event e1){
    void main(void){
        out = in + in;
        notify(e1);
    }
};

module M3(int inout){
    int w1;
    event e1;
    M1 m1(w1, inout, e1);
    M2 m2(inout, w1, e1);
    void main(void){
        par{
            m1.main();
            m2.main();
        }
    }
};

Port Reference Edge
Extended Syntaxes in ExSDG from C

Hierarchical Structure

Concurrency

Timed Behavior

Bit Vector
Program Slicing for C/SystemC

- The codes to be analyzed (symbolically simulated) are extracted by program slicing
  - This means *only* extracted codes will be processed for verification
- Program slicing can extract the codes that can affect (be affected by) a variable
- Two kinds of slicing: backward slicing and forward slicing
- Based on C program slicer, program slicer for SpecC can be developed
  - Extensions for parallel statements, structural hierarchy, event manipulation statements, and others
Backward Slicing

• Backward slicing for a variable $v$ extracts all codes that affect the variable $v$

Backward slicing

```
a = 2;
b = 3;
c = 5;
a = a + 10;
b = a * c; /start/
c = c + a;
a = a * b;
```

```
a = 2;
b = 3;
c = 5;
a = a + 10;
b = a * c; /start/
c = c + a;
a = a * b;
```
Forward Slicing

- Forward slicing for a variable $v$ extracts all codes that are affected by the variable $v$

Forward slicing

```
a = 2;
b = 3;
c = 5;
a = a + 10;
b = a * c; /start/
c = c + a;
a = a * b;
```

```
a = 2;
b = 3;
c = 5;
a = a + 10;
b = a * c; /start/
c = c + a;
a = a * b;
```
Efficient Equivalence Checking

- Comparison of two similar design descriptions
  - Extract textual differences
  - Dependency analysis by program slicing type techniques
  - Symbolically simulate the real difference and analyze the results

\[
\text{Description1: } \text{return a;}
\]
\[
\text{Description2: } \text{return a;}
\]
Approaches for Difference Computation

Approach 1: Textual Difference Between C Programs
- Sensitive to name changes &
  Difficult to map the difference onto CDFG after optimization

High-Level Description

Instruction Sequence

Proposed Approach
- Compute the difference between “Instruction Sequences”

Approach 2: Maximum Common Subgraph Between CDFGs
- Practically Intractable/
  Need to define subgraphs between CDFGs

High-Level Description

Instruction Sequence

CDFG

Post-ECO High-Level Description

Post-ECO Instruction Sequence

CDFG

Post-ECO CDFG
Difference Computation Between Instruction Sequences

Advantages

• Difference computation between sequence is much easier than that between graphs
• Efficient textual difference computation (UNIX diff) can be utilized
• Data and control flows can be consistently represented

Disadvantage

• Graph-based method may lead to a smaller solution
SSA-Form Representation
1. Each variable is assigned once
2. Every instruction is an assignment
3. Instruction name = variable name

Control Instructions
Control flows are represented by label and branch instructions
Motivational Example (1/3)

**Pre-ECO Design**

```
for(i = 2; i < 16; i++)
  f[i] = f[i-1] + f[i-2];
```

```
n1 = (lbl) label
n2 = (nil) jmp n3
n3 = (lbl) label
n4 = (i32) phi (i32) 2, n1, n14, n7
n5 = (i1) lte n4, (i32) 16
n6 = (nil) br n5, n16, n7
n7 = (lbl) label
n8 = (i32) sub n4, (i32) 1
n9 = (i32) load n8
n10 = (i32) sub n4, (i32) 2
n11 = (i32) load n10
n12 = (i32) add n9, n11
n13 = (nil) store n12, n4
n14 = (i32) add n4, (i32) 1
n15 = (nil) jmp n3
n16 = (lbl) label
n17 = (nil) ret
```

**Post-ECO Design**

```
for(i = 3; i < 16; i++)
  f[i] = f[i-3] + f[i-2] + f[i-1];
```

```
n1 = (lbl) label
n2 = (nil) jmp n3
n3 = (lbl) label
n4 = (i32) phi (i32) 3, n1, n17, n7
n5 = (i1) lte n4, (i32) 16
n6 = (nil) br n5, n19, n7
n7 = (lbl) label
n8 = (i32) sub n4, (i32) 3
n9 = (i32) load n8
n10 = (i32) sub n4, (i32) 2
n11 = (i32) load n10
n12 = (i32) add n9, n11
n13 = (i32) load n12
n14 = (i32) add n4, (i32) 1
n15 = (nil) jmp n3
n16 = (lbl) label
n17 = (nil) ret
n18 = (nil) jmp n3
n19 = (lbl) label
n20 = (nil) ret
```
A straightforward application of Longest Common Subsequence (LCS; a basis of UNIX diff command) does not work because many instructions do not match simply due to their names.
Motivational Example (3/3)

| n1 = (lbl) label |
| n2 = (nil) jmp n3 |
| n3 = (lbl) label |
| n4 = (i32) phi (i32) 2, n1, n14, n7 |
| n5 = (i1) lte n4, (i32) 16 |
| n6 = (nil) br n5, n16, n7 |
| n7 = (lbl) label |
| n8 = (i32) sub n4, (i32) 1 |
| n9 = (i32) load n8 |
| n10 = (i32) sub n4, (i32) 2 |
| n11 = (i32) load n10 |
| n12 = (i32) add n9, n11 |
| n13 = (nil) store n12, n4 |
| n14 = (i32) add n4, (i32) 1 |
| n15 = (nil) jmp n3 |
| n16 = (lbl) label |
| n17 = (nil) ret |

| n1 = (lbl) label |
| n2 = (nil) jmp n3 |
| n3 = (lbl) label |
| n4 = (i32) phi (i32) 3, n1, n14, n7 |
| n5 = (i1) lte n4, (i32) 16 |
| n6 = (nil) br n5, n16, n7 |
| n7 = (lbl) label |
| n8 = (i32) sub n4, (i32) 3 |
| n9 = (i32) load n8 |
| n10 = (i32) sub n4, (i32) 2 |
| n11 = (i32) load n10 |
| n18 = (i32) sub n4, (i32) 1 |
| n19 = (i32) load n18 |
| n20 = (i32) add n9, n11 |
| n12 = (i32) add n20, n19 |
| n13 = (nil) store n12, n4 |
| n14 = (i32) add n4, (i32) 1 |
| n15 = (nil) jmp n3 |
| n16 = (lbl) label |
| n17 = (nil) ret |

However, renaming the instructions can increase the matches of the instructions!
Problem Formulation

Problem: Given two instruction sequences, assign the names of the instructions such that the number of the instruction matches are maximized, where:

- Two instructions are said to be matched if the textual representations of the instructions are equivalent.
- Every two instructions in the same program must have different names.
- Two instructions in the different programs can have the same name.
- The order of an instruction sequence is fixed.
  - Note: Swapping the instructions may increase the matches further, but this is not taken into account in this work.
Overall Flow

Step 1: Conversion to Instruction Sequence

Estimate the different regions (which may include non-different instructions)

Step 2: Pessimistic Method Based on Textual Difference

\[
\begin{align*}
\text{Maximize} & \quad I_{p,q} \quad \text{subject to:} \\
& \quad (p,q) \in S \\
& \quad I_{p,q} + N_{p,q} - (p,q) \in O_{p,q}, (p,q) \in S \\
& \quad N_{p,q} + N_{r,s} - 1 \in (p,q),(r,s) \in T \\
\end{align*}
\]

Step 3: Exact Method Based on 0-1 Linear Programming

Minimum Difference

Narrow down the differences from the regions found in Step 2
### Pessimistic Method (Step 2-1): Name Normalization

First, by normalizing the instruction names to ""$<type>$", an optimistic LCS is obtained.
Pessimistic Method (Step 2-2): Renaming

<table>
<thead>
<tr>
<th>n1 = (lbl) label</th>
<th>n1 = (lbl) label</th>
</tr>
</thead>
<tbody>
<tr>
<td>n2 = (nil) jmp n3</td>
<td>n2 = (nil) jmp n3</td>
</tr>
<tr>
<td>n3 = (lbl) label</td>
<td>n3 = (lbl) label</td>
</tr>
<tr>
<td>n4 = (i32) phi (i32) 2, n1, n19, n9</td>
<td>n6 = (i32) phi (i32) 3, n1, n25, n9</td>
</tr>
<tr>
<td>n5 = (i1) lte n4, (i32) 16</td>
<td>n7 = (i1) lte n6, (i32) 16</td>
</tr>
<tr>
<td>n8 = (nil) br n5, n27, n9</td>
<td>n8 = (nil) br n7, n27, n9</td>
</tr>
<tr>
<td>n9 = (lbl) label</td>
<td>n9 = (lbl) label</td>
</tr>
<tr>
<td>n10 = (i32) sub n4, (i32) 1</td>
<td>n13 = (i32) sub n6, (i32) 3</td>
</tr>
<tr>
<td>n11 = (i32) load n10</td>
<td>n14 = (i32) load n13</td>
</tr>
<tr>
<td>n12 = (i32) sub n4, (i32) 2</td>
<td>n15 = (i32) sub n6, (i32) 2</td>
</tr>
<tr>
<td>n16 = (i32) load n12</td>
<td>n16 = (i32) load n15</td>
</tr>
<tr>
<td>n17 = (i32) add n11, n16</td>
<td>n20 = (i32) sub n6, (i32) 1</td>
</tr>
<tr>
<td>n18 = (nil) store n17, n4</td>
<td>n21 = (i32) load n20</td>
</tr>
<tr>
<td>n19 = (i32) add n4, (i32) 1</td>
<td>n22 = (i32) add n14, n16</td>
</tr>
<tr>
<td>n20 = (i32) store n17, n4</td>
<td>n23 = (i32) add n22, n21</td>
</tr>
<tr>
<td>n21 = (i32) store n17, n4</td>
<td>n24 = (nil) store n23, n6</td>
</tr>
<tr>
<td>n22 = (i32) add n4, (i32) 1</td>
<td>n25 = (i32) add n6, (i32) 1</td>
</tr>
<tr>
<td>n23 = (i32) store n17, n4</td>
<td>n26 = (nil) jmp n3</td>
</tr>
<tr>
<td>n24 = (nil) store n17, n4</td>
<td>n27 = (lbl) label</td>
</tr>
<tr>
<td>n25 = (i32) add n6, (i32) 1</td>
<td>n28 = (nil) ret</td>
</tr>
<tr>
<td>n26 = (nil) jmp n3</td>
<td>n28 = (nil) ret</td>
</tr>
</tbody>
</table>

For each matched instructions, assign the same name. If differences are small, most matches can be identified here.
Exact Method: ZOLP Formulation (1/5)

- Variables (all binary)
  - $N_{p,q}$: Name match
    - 1 if and only if two instructions $p$ and $q$ have the same name
  - $I_{p,q}$: Instruction match
    - 1 if and only if two instructions $p$ and $q$ match, i.e. the following conditions are all satisfied
      - $p$ and $q$ have the same name
      - Operation types of $p$ and $q$ are equivalent
      - Every operands of $p$ and $q$ are equivalent
Cost function

- Maximize the number of instruction matches, \textit{that is}:

$$\text{Maximize } \sum_{(p,q) \in S} I_{p,q}$$

where $S$ is a set of potential instruction matches

- $(p,q) \in S$ if and only if $p$ and $q$ have the same operation type
Exact Method: ZOLP Formulation (4/5)

• Constraint 1 (Instruction match)
  – An instruction match \((p,q)\) is 1 if and only if:
    • Their names match
      \[ I_{p,q} \quad N_{p,q} \quad (p,q) \quad S \]
    • And every pair of operands match
      \[ I_{p,q} \quad N_{r,s} \quad (r,s) \quad O_{p,q}, \quad (p,q) \quad S \]
      where \(O_{p,q}\) is a set of operand pairs
      – \((r,s)\) \(O_{p,q}\) if and only if \(r\) and \(s\) are the corresponding operands of \(p\) and \(q\)
Exact Method: ZOLP Formulation (5/5)

• Constraint 2 (Match conflict)
  – Two name matches \((p,q)\) and \((r,s)\) cannot be 1 at the same time if the matches conflict

\[
N_{p,q} + N_{r,s} \leq 1 \quad \text{for} \quad (p,q),(r,s) \in T
\]

– Match conflicts are categorized into two types (explained in the following slides):
  • Match uniqueness
  • Order violation
Match Conflict 1: Match Uniqueness

Since each instruction can match at most one instruction, the blue and red matches conflict.
Match Conflict 2: Order Violation

Since matched instructions must follow the sequence order, the blue and red matches conflict.
By solving the ILP problem, the final solution is obtained
By solving the ILP problem, the final solution is obtained
Preliminary Experimental Result

- We have two different implementations:
  - In our high-level synthesis framework Cyneum
    - For analyzing small differences before and after ECOs
  - As a LLVM command llvm-smart-diff
    - For larger-scale, practical software

- Example: ADPCM from CHStone Benchmark Suite
  - C program: 901 lines with 14 functions
  - Instruction sequence size: 1076 (after all functions are inlined)
  - We artificially applied ECOs: loop reverse, swapping variables
  - Completed in ~1 second
    - Initial difference: 156 matches, 920 differences
    - Pessimistic difference: 1049 matches, 27 differences
    - Final difference: 1059 matches, 17 differences
Outline

• Formal analysis with symbolic simulation
• Equivalence checking with difference identification
• Software controlled hardware (patchable HW)
• Formal analysis through functional dependencies
Next-Generation SoCs

• Embed numerous application-specific accelerators
  – Achieve both high performance and high efficiency
    • **Efficiency** is the main differentiator against the others
  – Each PE may be high-level synthesized as accelerators

![Processing Engines in Consumer Portable SoCs]

[Source: ITRS 2007]

**Consumer Portable SoC Template [Source: ITRS 2007]**

**#Processing Engines in Consumer Portable SoCs**

[Source: ITRS 2007]
Conventional SoC Design Flow

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Silicon RTL Verification</td>
<td>Bug Fix</td>
</tr>
<tr>
<td>Machine-Generated RTL</td>
<td>Bug Localization</td>
</tr>
<tr>
<td>Logic Synthesis Place &amp; Route</td>
<td>Need to Understand RTL</td>
</tr>
<tr>
<td>High-Level Synthesis</td>
<td>Error Detection</td>
</tr>
<tr>
<td>High-Level Description</td>
<td>Post-Silicon RTL Validation</td>
</tr>
<tr>
<td>Design</td>
<td>Respin</td>
</tr>
</tbody>
</table>

75% of the whole development time [Source: Intel 2007]
Proposed Patchable SoC Design Flow

- Bug Fix
- Bug Localization
- Verification/Simulation
- Pre-Silicon High-Level Verification
- High-Level Description
- High-Level Synthesis of Patchable HW
- Logic Synthesis
- Place & Route
- Patchable SoC Design
- High-Level ECO
- Post-Silicon
- Error Detection
- Bug Localization
- Bug Fix
- Patch Compilation
- No Respin Needed!

No Respin Needed!
Fixed-Function Accelerator

- Achieves high energy efficiency by customization:
  - Hardwired controller → **No reprogrammability**
  - Highly-customized datapath → **Low flexibility**

![Diagram of Fixed-Function Accelerator](diagram.png)
Proposed Patchable Accelerator

- Behavioral reprogrammability by control patching
- Increased flexibility by adding register file via data bus
Another big reason why we should have (partial) programmability in HW

- Lots of variability in performance due to manufacturing process variations for thinner transistors
  - Speed of gates in a chip can vary more than 20%
  - Aggressive design sometimes works, and sometimes does not work
- Semiconductor chips are aging just like you and myself
  - When young (<10 years) fast, but when older, significantly slower
- Better to have some programmability in HW to accommodate correctly such variations
  - Now becoming common that hardware has periodical “health check” various (analog) sensors (can identify how
Patch Logic

Program Counter

= ?

PC1

PC2

= ?

PC1′

PC2′

> PC_{patch}

= ?

Hardwired Controller

Control Signal Memory

Control Signal Patch

Program Counter Patch

Patch Memory
### Patching Example (1/2)

**Scheduling Result of Initial Design**

<table>
<thead>
<tr>
<th>PC</th>
<th>ALU1</th>
<th>ALU2</th>
<th>MUL1</th>
<th>Next PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>✗</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>✫</td>
<td>✗</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>✫</td>
<td></td>
<td>✗</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dataflow graph for Initial Design:

- Hardwired logic
- Patch logic
# Patching Example (2/2)

## Scheduling Result After Engineering Change

<table>
<thead>
<tr>
<th>PC</th>
<th>ALU1</th>
<th>ALU2</th>
<th>MUL1</th>
<th>NextPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>2</td>
<td>x</td>
<td>2 4</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
<td>-</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>+</td>
<td></td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td></td>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Dataflow graph After EC**

**Hardwired logic**

**Patch logic**
Post-Silicon EC Flow by Patching

C Program

EC-Aware High-Level Synthesis (Proposed)

Modified C Program

Incremental Patch Compilation (Proposed)

Patch

Post-Silicon EC (Spec. Changes・Bug Fixes)

Efficeum

Patching
Energy Efficiency Comparison

8x8 IDCT (FreePDK 45nm technology)

Offers a tradeoff between efficiency and programmability
Area & Performance Comparison

- Up to 40% Increase
- 5X Smaller

Operating Frequency [MHz]

- Fixed
- 5-step
- 10-step
- 20-step
- Fully

Area [um^2]

- Fixed
- 5-step
- 10-step
- 20-step
- Fully

5% 20%
Outline

• Formal analysis with symbolic simulation
• Equivalence checking with difference identification
• Software controlled hardware (patchable HW)
• Formal analysis through functional dependencies
Analysis on functional dependency

• Question: Some function can be realized with a set of other functions?
  – Here only deal with Boolean functions
  – func1 can be realized with func2 and func3 only

\[ \text{func1} = x \lor y, \quad \text{func2} = x \land y, \quad \text{func3} = x \]

– Ex: If \( \text{func1} = \text{funx2} \quad \text{func3} \) then

\[ \text{func1} = x \lor y \]

\[ \text{func2} = x \land y \]

\[ \text{func3} = x \]
Many applications of functional dependence analysis

- Software (hardware as well) debugging
  - Bounded executions are translated into Boolean formulae including arithmetic
  - Can be analyzed by SMT/SAT solvers

- Bugs in conditions can be rectified with function dependency (LUT) based formulation
  - Even if \( (x \neq 1) \), \( (z) \) are wrong, corrections can be found with functional dependence analysis by checking if the conditions can be corrected by the specified set of variables

```c
x = x + y;
if (x != 1) {
    x = 2;
    if (z) x++;
}
```

```c
x1 = x0 + y0;
if (x1 != 1) {
    x2 = 2;
    if (z0) x3 = x2 + 1;
}
```

```c
x1 = x0 + y0
\land x2 = \text{ite}(x1 \neq 1, 2, x1)
\land x3 = \text{ite}(x1 \neq 1 \land z0, x2 + 1, x2)
```
Specification debugging

• Given property, try to make it satisfied by changing their assumptions/conclusions
  – Adding some constraints on the property
  – E.g.: Wrong (initial) \((x \lor (y \land z)) \rightarrow (Xu \land XXu)\)
    Correct \((x \land (y \lor z)) \rightarrow (Xu \land XXu)\)
Formulation with look up table (LUT) for Boolean function

Store truth table in memory

Actual implementation is based on multiplexers and Boolean variables (flipflops)
Example look up

<table>
<thead>
<tr>
<th>ABC</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>011</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
</tr>
</tbody>
</table>

- Can check if functional dependency exists or not
- Need $2^n$ variables to represent truth table for $n$-input
  - If $n$ is small, it works!
Problems that we like to solve

Targeting combinational circuits

Bug found and like to correct it
Specification has changed
Like to make fault-tolerant for some faults

Pick up sets of gates/regions for LUT replacements by some heuristics (Or LUT locations are predetermined by designers)

Design:
  Given one with LUT
  Spec:
  Correct functionality

Design:
  Given one with LUT
  Spec:
  New specification

Design:
  Faulty one with LUT
  Spec:
  Correct functionality

Generate QBF formulae to be solved
Formulation with Quantified Boolean Function (QBF)

• Given circuits, replace subsets of gates with fixed-input LUTs using some heuristics
  – We are still developing good heuristics
    • E.g., replacing last levels of gates in combinational circuits
  – We may connect redundant wires to LUTs for future rectification, if we like

• The question:
  – By appropriately (re-)programming the LUTs, can the circuits have target functionality?
    \[ \exists v. \forall x. f(v, x) = \text{TargetSPEC}(x) \]
    \(V\): Program variables for LUT, \(X\): Inputs

• Here we only deal with combinational circuits
  – Sequential circuits must be time-frame expended
What is QBF

• Quantified Boolean Formula (QBF):
  – Quantification + Propositional formula (SAT)
  – “there exists a value of x in {True, False}”, represented by ∃x
  – “for all values of y in {True, False}”, represented by ∀y
  – Example:
    \[ ∃x ∀y ∃z (x ∨ y ∨ ¬z) ∧ (¬x ∨ z) ∧ (¬y ∨ z) \]
  – Applications: Formal Verification and Model Checking, Planning, Scheduling, Games, …

• Problem:
  – Is the given formula satisfiable or not?
  – PSPACE-Complete (SAT is ‘only’ NP-Complete)

• Some QBF solvers are publicly available
  – Significantly slower than SAT solvers for same numbers of vars
Algorithms for QBF

• DPLL
  – E.g., Qube
• Expansion-based
  – E.g., Quantor and Nenofex
  – Shannon expansion of variable from inner quantifier block
• Skolemizing
  – E.g., Skizzo
• Portfolio
  – E.g., AQME
• Counter example-Guided Abstraction Refinement (CEGAR): Used in our method
CEGAR (Counter Example Guided Abstraction Refinement) based QBF solver

- Consider $\exists X. \forall Y. \phi$ where $\phi$ has no quantifiers
- If $Y$ has only a few variables, we can fully expand $\forall Y. \phi$ (i.e., take the conjunction over all assignments)
  - E.g., $(\exists x. \forall y. \phi \land (x \lor y)) \iff (x \lor \text{False}) \land (x \lor \text{True})$
- Full expansion can be handed to a fast SAT solver
- But what if there are many variables in $Y$?
- Conjecture: For many practical instances, we only need to consider a small number of assignments
  - Experimental results strongly confirm this
- RAReQS: Recursive Abstraction Renement QBF Solver (M. Janota, W. Klieber*, J. Marques-Silva, E.M. Clarke)
Our proposed method

- We use the same approach based on CEGAR
  - Here the problem is 2-QBF
- Two applications of SAT solvers are repeated until:
  - Found a solution for the problem
  - Prove the non-existence of solutions
  - Time up

- First SAT: Small number of assignments to
  - Can also use BDD if we like to have sets of solutions
- Second SAT: Checking if the rectification succeeds
  - Essentially equivalence checking on two combinational circuits and so very efficient algorithms exist
Generate CNF for the conjunction of some value combinations of for-all variables

SAT?

No → No solution

Yes → Generate CNF for the negation with the solution as constraints

SAT?

No → Solution found

Yes → Go back to the top with the solution as constraints
# Experimental results where there is solution (Rectification succeeds)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines</th>
<th>LUT replaced</th>
<th>Proposed Solved (in 3,600 sec)</th>
<th>Proposed Average time (sec)</th>
<th>Proposed Average iterations</th>
<th>sKizzo Solved (in 3,600 sec)</th>
<th>sKizzo Average time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ex1</td>
<td>400</td>
<td>10</td>
<td>20/20</td>
<td>3.2</td>
<td>15.3</td>
<td>18/20</td>
<td>594.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>20/20</td>
<td>7.4</td>
<td>28.3</td>
<td>13/20</td>
<td>41.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>20/20</td>
<td>27.9</td>
<td>65.5</td>
<td>17/20</td>
<td>154.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>20/20</td>
<td>99.4</td>
<td>123.0</td>
<td>17/20</td>
<td>183.1</td>
</tr>
<tr>
<td>Ex2</td>
<td>1,200</td>
<td>10</td>
<td>20/20</td>
<td>5.5</td>
<td>11.0</td>
<td>20/20</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>20/20</td>
<td>14.7</td>
<td>21.0</td>
<td>14/20</td>
<td>25.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>20/20</td>
<td>65.2</td>
<td>48.6</td>
<td>14/20</td>
<td>835.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>20/20</td>
<td>207.6</td>
<td>87.4</td>
<td>10/20</td>
<td>1899.2</td>
</tr>
<tr>
<td>Ex3</td>
<td>2,400</td>
<td>10</td>
<td>20/20</td>
<td>6.2</td>
<td>6.1</td>
<td>0/20</td>
<td>Time out</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>20/20</td>
<td>15.5</td>
<td>10.6</td>
<td>0/20</td>
<td>Time out</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>18/20</td>
<td>589.8</td>
<td>24.4</td>
<td>0/20</td>
<td>Time out</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>15/20</td>
<td>586.1</td>
<td>41.4</td>
<td>0/20</td>
<td>Time out</td>
</tr>
</tbody>
</table>
## Experimental results when there is no solution (Proof of no solution)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines</th>
<th>LUT replaced</th>
<th>Solved (in 3,600 sec)</th>
<th>Average time (sec)</th>
<th>Average iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ex1</td>
<td>400</td>
<td>10</td>
<td>5/5</td>
<td>1.4</td>
<td>8.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>5/5</td>
<td>1.6</td>
<td>6.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>5/5</td>
<td>0.8</td>
<td>3.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>5/5</td>
<td>4.8</td>
<td>16.0</td>
</tr>
<tr>
<td>Ex2</td>
<td>1,200</td>
<td>10</td>
<td>5/5</td>
<td>3.8</td>
<td>7.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>5/5</td>
<td>7.2</td>
<td>11.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>5/5</td>
<td>8.8</td>
<td>11.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>5/5</td>
<td>15.8</td>
<td>14.2</td>
</tr>
<tr>
<td>Ex3</td>
<td>2,400</td>
<td>10</td>
<td>5/5</td>
<td>0.6</td>
<td>1.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>5/5</td>
<td>1.2</td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>5/5</td>
<td>4.8</td>
<td>8.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>5/5</td>
<td>12.0</td>
<td>14.0</td>
</tr>
</tbody>
</table>
Specification mining from designs

- Try to identify what are satisfied in terms of Boolean formulae among internal/external variables

- Simulation-based approach
  - Check what relationships are held on simulation traces
  - Can deal with large programs/designs, but results can be false

- Based on static analysis
  - Essentially SAT-based analysis
  - Some variables are functions of other variables?
  - Some assertions are satisfied among variables?

What is F?
For sequential circuits

- Time-frame expansion
- LUTs are inserted to capture assertions
  - Where to insert is based on heuristics or previous designs
  - Can be used for debugging assertions
Future issues

• Anyway, hardware will become more and more “unreliable” with thinner transistors
  – Delays much depend on temperatures
  – Gate delays can suddenly increase at some point (aging)

• Now hardware has self-heath checking mechanisms to monitor its age
  – How to utilize in SW level is still a big question

• Hardware people are discussing about “approximate computation”
  – $5 \times 3$ may be any value between 14 and 16!
  – For some application such as image processing, this is OK, as long as control flow of computations do not change a lot
  – How SW can utilize this?
My fundamental question

- As I said, hardware is becoming more and more unreliable
  - Somehow very difficult to avoid and we must accept
  - Some VLSI chips are manufactured for 20 years of use
- Now hardware has self-heath checking mechanisms to monitor its age
- Also hardware has (partial) programmability
  - By re-programming hardware, some disorder of hardware can be avoided
  - This is just another software development
- With such software, total system including hardware and software can be more reliable?
  - Or, due to software bugs, things will just become worse?
  - Could be a good problem for state-of-the-art model checker as its model could be small but complicated models