Codesign for Reconfigurable Systems

Design Automation Lab.
Jurng Byoung-Yil
Introduction

- Speed gap between general purpose processor and custom silicon
  - FPGA is an solution.
  - RPU (Reconfigurable Processing Unit) : XC6000 (Xilinx)
- General Purpose Reconfigurable Co-Processor [1]
  - An accelerator for the main processor
  - Static Reconfigurable System ( using custom FPGA )
- Dynamically Reconfigurable Embedded System [2]
  - Based on Java specifications
  - Dynamic Reconfigurable System ( using RPU )
Reconfigurable Co-Processor Architecture

- Internal Bus
  - 50MHz
  - 32bit Data, Addr
- Can be accessed via bus address
- Sharing the main memory
- 1MByte local cache
  - 2 cycle / local access
  - 3 cycle / cache hit
  - 20-40 cycle when cache miss

Figure 1: The structure of a general purpose co-processor for SUN SBus.
Codesign for the Reconfigurable Co-processor

• Specification language : C
  - the main program on the main processor
  - some user functions on the co-processor

• Flow of codesign
  - Parsing of C program and CDFG generation
  - Hardware module generation for each function
  - Estimation of execution clocks and hardware resources
  - HW/SW partitioning : based on C functions
  - Modify some part of main program implemented in HW
  - Rescheduling of functions for HW/SW parallelism
Evaluation of Performance

• Software simple block
  - disassemble the compiled code and count number of operations
  - assume no pipeline hazard and CPI is 1
• Hardware simple block
  - maximum clocks of the scheduled data flow graph
• If-statement
  - condition is assume to be true with 0.5 probability
• Loop statement
  - unpredictable iteration count : user parameter
• Memory access
  - co-processor is 4 times slower than main processor
Experimental Results

• Two examples

<table>
<thead>
<tr>
<th>Data</th>
<th># clk soft</th>
<th># clk hard</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECC</td>
<td>1329</td>
<td>63</td>
<td>21.1</td>
</tr>
<tr>
<td>GCD</td>
<td>58</td>
<td>28</td>
<td>6.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data</th>
<th># clk soft</th>
<th># clk hard</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECC</td>
<td>1329</td>
<td>46</td>
<td>28.9</td>
</tr>
<tr>
<td>GCD</td>
<td>58</td>
<td>49</td>
<td>1.3</td>
</tr>
</tbody>
</table>

• Lexical Analysis: change 32 C keyword to upper case
  - source: gcc ver.2.5.8 source text
  - SW only: 16.3 sec on SS10/51
  - Using co-processor
    - lexical analysis(HW): 1.26 sec
    - upper case conversion(SW): 0.92 sec
    - 13 times speed-up
Dynamically Reconfigurable Embedded System

- Next generation embedded systems
  network interaction, multiple concurrent applications, and changing operation condition
  => new demands - adaptivity, reconfigurability

- Target architecture
  : a set of processors and reconfigurable FPGA

- This paper presents a semi-automated co-synthesis and co-simulation framework
  1) fast prototyping HW/SW system
  2) specified in JAVA - software oriented strategy
Co-synthesis environment

1) Start from a JAVA specification
2) Identify HW parts (partitioning)
3) Interface generation - partially in software and in hardware
4) Software synthesis
   - bytecode generation
   - executed in JVM
5) Hardware synthesis
   - generate synthesizable VHDL description
   - Synthesize and map to the reconfigurable FPGA
Partitioning

- Granularity is the level of JAVA methods
- Methods chosen for HW can run as software methods, but methods for SW cannot run as hardware methods.
- Java compiler guavac translate Java method into internal tree representation.
- From internal tree representation bytecode or VHDL is generated
Hardware code generation

- High-level synthesis
  1) produce synthesizable VHDL
  2) selected Java methods must comply with the restrictions of the operator library
  3) design is split into a data-path and a controller

- RT-level synthesis
  1) add VHDL frame for interfacing with the software part
  2) use SYNOPSYS DESIGN COMPILER
Hardware code generation

- Technology mapping, P&R
  1) result of RTL synthesis is EDIF
  2) if fails in P&R, the method must run as software part
  3) a single hardware method can be implemented in a single FPGA device.
Interfacing HW and SW

- The software part
  1) extend the GNU Interpreter KAFFE
     a. read in partitioning information
     b. synchronize and communicate with HW methods
     c. profiling
  2) Only one thread at a time is allowed to access the HW

- The hardware part
  1) argument reg and result reg
  2) control block (interact with RTS)
Run-time system

- Read partitioning information and schedule methods on host PC or FPGA hardware
- rely on a database which contains methods executable as HW or SW
- at the beginning none of the FPGAs is configured, RTS determines the available FPGA and download bit-stream
- execution flow of combined system is dominated by SW
- calling thread is suspended but any other thread can be executed at the same time
Experiment

- use host workstation and dedicated FPGA board connected to host over PCI bus

- Features of XC6216
  1) integrated microprocessor interface
  2) high speed reconfiguration: 18 - 300 ms
  3) after optimization of reconfigure process: 4.5 - 29 ms

- bottleneck: PCI interface of XC6200DS board
Reduce Reconfiguration Overhead

- Hauck’s work
  - main processor and reconfigurable co-processor
  - assume single context
  - HW jobs and SW jobs do not run concurrently
  - Hide reconfiguration overhead by overlapping computation of main processor and reconfiguration job
  - insert prefetch instruction into the code of main processor
Our approach

• Single main processor
• Multiple processes in the reconfigurable FPGA
  - One SW job and several HW jobs can be executed in parallel
• No conflict between computation and reconfiguration of FPGA.
• Single reconfiguration controller.
Motivation

(a) Do not use configuration prefetch

(c) Overlapping the reconfiguration jobs and other computation

(d) Splitting reconfiguration jobs and their corresponding computation
Resource optimize

- In the same latency constraint, (b) needs less resource than (a)
Define problem

- Input: task graph G(V,E)
  - V: set of nodes
  - E: set of edges
  - \( v_i \): node \( i \)
  - (\( v_i \), \( v_j \)): edge from \( v_i \) to \( v_j \)

- Map each node into HW or SW and scheduling

- Constraints
  - \( \lambda \): latency constraint => resource optimize.
  - A: hardware resource constraint => performance optimize.
Experiments

- Formulated this problem in ILP (Integer Linear Programming)
- Applied ILP formulation to a set of synthetic task graphs and a real example (JPEG encoder)
- Solved ILP using CPLEX
- Synthetic Examples
  - Task graph generation: TGFF
  - Solve problem for 8 node task graph
Experiments (cont’d)

- The performance gain of EPR (Early Partial Reconfiguration) compared to SR (Simple Reconfiguration)
Experiments (cont’d)

- The performance gain of EPR compared to no reconfiguration case (configurations are static)
Experiments (cont’d)

- HW resource usage (EPR/SR) of latency-constrained HW-SW partitioning

<table>
<thead>
<tr>
<th>R</th>
<th>15/-</th>
<th>16/17</th>
<th>19/20</th>
<th>21/23</th>
<th>25/29</th>
<th>30/35</th>
<th>40/42</th>
<th>50/50</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>100/-</td>
<td>89/89</td>
<td>78/78</td>
<td>46/46</td>
<td>39/39</td>
<td>32/32</td>
<td>21/21</td>
<td></td>
</tr>
<tr>
<td>0.5</td>
<td>125/-</td>
<td>89/89</td>
<td>86/86</td>
<td>46/46</td>
<td>39/39</td>
<td>32/32</td>
<td>21/21</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>78/-</td>
<td>78/86</td>
<td>46/46</td>
<td>39/39</td>
<td>32/32</td>
<td>21/21</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>67/-</td>
<td>53/53</td>
<td>42/42</td>
<td>32/32</td>
<td>21/21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>67/-</td>
<td>32/-</td>
<td>32/53</td>
<td>21/21</td>
</tr>
</tbody>
</table>
Conclusion

- Using the reconfigurability
  => flexibility and performance gain
- Dynamic reconfiguration
  => more effective use of silicon resource
- Reconfiguration overhead can be reduced by prefetch of partial configuration
Reference
