Techniques and Tools for Customising Reconfigurable Processors

Wayne Luk
Department of Computing
Imperial College

Tutorial at Int. Symp. on System-on-Chip
18 November 2003

When customise?

• design time = fabrication time + compile time
• application-specific integrated circuits (ASICs)
  – customise application: fabrication time
• instruction processors
  – customise application: compile time
  – customise execution: run time
• reconfigurable hardware
  – e.g. Field-Programmable Gate Arrays (FPGAs)
  – software flexibility + ASIC speed
  – customise application: compile time
  – customise reconfiguration: run time
Why customise and reconfigure?

• why customise?
  – support domain-specific description/compilation
  – improve re-use of design tools/design components
  – upgrade to new functions/standards/protocols
  – optimise speed/size/power consumption

• why reconfigure?
  – ASICs: design and fabrication cost and time
  – FPGAs: standard parts, reconfigurable SoC
  – share resources by reconfiguration (time mux)
  – adapt to run-time conditions

Customise what?

• domain-specific customisation
• customised data processors
  – design-time customisation
  – run-time customisation
• custom instruction processors
• imperative languages e.g. C: RISC, Java: JVM
• logic languages e.g. Prolog: WAM, VAM
• design tool customisation
• platform-specific customisation
1. Domain-specific customisation

- graphics pipeline
  - non-standard data format and operations
  - source: LightWave3D
- fixed-point/floating-point signal processing
  - customise bit-width reducing size and power
  - source: Matlab Simulink (a)
- firewall processor
  - irregular CAM with rule reduction
  - source: Ponder, a high-level security language (b)

1a. Digital Signal Processing (DSP)

- Simulink: block-diagram for DSP simulation
- System Generator: hardware blockset

Parameterisable, automatically maps to Xilinx circuit libraries

(source: Xilinx)
Xilinx System Generator

- **features**
  - top-down design
  - block diagram system-level simulation
  - model key components: DSP, control logic, analog/mixed-signal
  - integrated with MATLAB
- **benefits**
  - clear, executable block diagram specifications
  - efficient mapping to hardware library blocks
  - verify design, detect flaws early and rapidly
  - reduced time-to-market, fast upgrade

(source: Xilinx)

Upgradable hardware

![Graph showing the number of new users over time for upgradable and non-upgradable products.](source: Algotronix Consulting / Xilinx)

- upgradability: min time-to-market
- max time-in-market

(source: Algotronix Consulting / Xilinx)
1b. Network security: firewall

- an automated design flow for hardware packet filters
- high-level firewall description in network management language Ponder
  - improve efficiency of design
  - increase confidence in correctness
- optimisation based on rule reduction and hardware sharing

High-level firewall description

- separate control needs from IP/port addresses
- domain hierarchy: flexible and easy to manage

```
inst auth- Deny(/any/net1/net2, /any/ip, /any, /any/ip);
inst auth+ Permit(/any, /any/ip/tcp/20, /any, /any/ip/tcp/>1024);
```
High-level firewall description

```
inst auth- Deny(/any/net1/net2, /any/ip, /any, /any/ip);
inst auth- Deny(/any, /any/ip, /any/net1/net3, /any/ip);
```

Low-level firewall rule representation

<table>
<thead>
<tr>
<th>Type</th>
<th>Source IP address</th>
<th>Source port</th>
<th>Destination IP address</th>
<th>Destination port</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip</td>
<td>127.0.0.0/0.255.255.255</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>deny</td>
</tr>
<tr>
<td>ip</td>
<td>10.0.0.0/0.255.255.255</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>deny</td>
</tr>
<tr>
<td>ip</td>
<td>172.16.0.0/0.15.255.255</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>deny</td>
</tr>
<tr>
<td>ip</td>
<td>192.168.0.0/0.255.255.255</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>deny</td>
</tr>
<tr>
<td>ip</td>
<td>*</td>
<td>*</td>
<td>0.0.0.255/255.255.255.0</td>
<td>*</td>
<td>deny</td>
</tr>
<tr>
<td>ip</td>
<td>*</td>
<td>*</td>
<td>0.0.0.0/255.255.255.0</td>
<td>*</td>
<td>deny</td>
</tr>
</tbody>
</table>

Design flow

1. Authorisation policy
2. Design capture
3. Rule reduction
4. Hardware design generation
5. Device specific place and route
6. Hardware configuration bitstreams
Area optimisation: field-level

- rule sets: a mixture of incoming and outgoing traffic controls
- reduce hardware usage
  - 67-80% from regular CAM
  - 24-63% from ‘don’t care’ optimised CAM
- result: data dependent
- improvement: run-time monitoring and reconfiguration

2a. Customising data processor: compile-time

- declarative descriptions: Ruby
  - data-flow parallelism, simple control
  - regular, largely data-dependent designs
  - parametric pipelining, serialisation
- imperative descriptions i: Handel-C
  - user-controlled parallelism and operator sizing
  - simple timing model
- imperative descriptions ii: pipeline vectorisation
  - control-flow parallelism, complex control
  - dependence analysis: pipeline automatically
  - loop transformation e.g. loop tiling
Declarative datapath: description

- parametric description of block diagrams
- describe block with relations e.g. add, fork
- compose pattern with functions, e.g. $R^n$
- enables pipelining via math. transformations
  - $R^{kn} = (R^k ; D)^n ; D^{-n}$ (given $R = D^{-1} ; R ; D$)
  - $kn = m$, fully-pipelined: $k = 1, n = m$
  - non-pipelined: $k = m, n = 1$

Declarative datapath example: convolver with counter-flowing data

$C_{b1} = \text{row } C_{bcell1}$

$C_{bcell1} = \text{mac}'' ; \pi^{-1} \circ \text{fork} ; \text{snd} (\pi_1 ; D)$
Design Cbb2

Imperial Ruby tools
Imperative datapath i: Handel-C

  - based on ANSI-C: high-level, familiar tools
  - variable operator sizing and parallelism: efficient
  - simple timing: assignment/communication 1 cycle, rest 0
  - CSP theory, normal-form algebra: math. basis

- general-purpose platforms
  - 3D graphics, augmented reality, data compression
  - network: adaptive firewall, Quality-of-Service, AES
  - instruction processor: customisable instructions (see later)

High-level design flow

(source: Celoxica)
Imperative datapath ii: pipeline vect.

Pipelined vect.: hardware/software
2b. Customising data processor: run-time reconfiguration (RTR)

- JBits: symbolic access to bitstreams, low-level
- RTPebble: declarative VHDL subset for RTR

![Diagram showing the customising data processor flow for RTR]

(source: based on Xilinx)
3. Customise instruction processor

- application-specific instruction processors
- now practical to implement an instruction processor in FPGA as a 'soft processor'
- a soft processor will have a lower clock rate than a conventional ASIC processor, but...
  - System-on-a-Chip (SoC): can integrate various system components onto a single FPGA chip
  - often location-independent, minimise connections
  - reconfigurable: can customise application at compile time or run time

Custom instructions

- customise instruction set e.g. Intel MMX
  - improve speed, reduce store size/power
- not yet supported in most soft processors e.g. Xilinx MicroBlaze, fpgacpu.org
- supported in some ASIC CPU design tools and Altera Nios, but
  - must fit a restrictive template
  - custom instruction hardware must be designed manually, using e.g. VHDL or Verilog
Soft processors for Spartan IIe

- 8-Bit Microcontroller
  - KCPSM by Xilinx
  - V8-uRISC by VAutomation
  - Flip805x-PR (8051) Core by Dolphin Integration
  - C165X, CZ80CPU, R8051, C8051, D80530, compact D80530, C2901 by CAST
  - W65C02S by Western Design Center
  - Mini-RISC by OpenCores.org
  - Free-RISC8 by The Free-IP Project

- 16-Bit Processor
  - AX1610 RISC by Loarant Corporation
  - x16 by Gray Research

- 32-Bit Processor
  - MicroBlaze Soft RISC by Xilinx
  - Lightfoot Java processor core by Digital Communication Technologies
  - LAVACORE Configurable Java Processor Core by Derivation Systems
  - C68000 16/32-Bit Microprocessor by CAST
  - Configurable RISC processor by ARC Cores
  - OpenRISC 1000 by OpenCores.org
  - LX4189 by Lexra

Compiling to custom instructions

- automatic generation of custom instructions
  - no need for experience of hardware design
  - concise description less error prone
  - scope for optimisation
- such automatic generation is difficult
  - needs architecture, compiler and tools to evaluate
- our approach: custom instructions for
  - imperative languages e.g. C: RISC, Java: JVM
  - logic languages e.g. Prolog: WAM
3a. Custom instruction: C programs

- C source
- Compiler
- Custom Instructions
- Simulator
- Behaviour and performance
- Instructions
- Custom datapaths
- Software
- Custom processor

Compiler structure

- Custom inst.
- C Code
- Code synthesis
- CFG Analysis
- CFG graph
- Software
- Scheduling
- Hardware out
- Custom datapaths
- Scheduling graph
- Custom processor
3b. WAM: Warren’s Abstract Machine

- high-level instruction set based on Prolog
- instructions for unification and backtracking
- fine-grained parallelisation
- concurrent assignments
  - group register assignments
  - make assignments as early as possible
- speculative assignments
  - side effect-free register assignment moved out of conditional branches

M-WAM

- multiple WAM processors for example testing
- controller dispatches queries to WAM processors
Mutagenesis benchmark:
deduce if compound can modify DNA

B: (12000 facts)
atm(d1,d1_1,c,22,0.117).
bond(d1,d1_2,d1_3,7).
E:
active(d18).
:- active(d84).
H:
active(A) :- atm(A,B,o,40,C), atm(A,D,n,32,C).

Benchmark speedup

- max speedup with 188 processors
- max speedup factor: 26
- assumes enough memory
4. Design tool customisation

- scripting language: e.g. Python
  - configure tools at compile time and run time
  - turn interpreted components into compiled ones
- customisable compiler framework
  - metalanguage: customise compiler optimisation
  - interface imperative and declarative blocks
- multilayer framework: customise hierarchically
  - develop and distribute hardware and software components
  - hardware plug-ins for Premiere and DirectShow

---

**IGOL architecture**

![IGOL Architecture Diagram]

- API
- IGOL Interfaces
- Class Library + Runtime
- Handel-C Framework
- Handel-C Implementation
- Software Implementation
Customisation: image processing

- Premiere
- DirectShow
- Photoshop
- IGOL Plugin
- IGOL Filter
- IGOL Plugin

Customisation: cryptography

- Win32 CryptoAPI
- javax.crypto
- IIS
- Crypto Provider
- JNI
- Automation

- Hammond
- RC1000-PP
- C++
- Java

Hammond

Module implementation

Class Library + Runtime

Class Library + Runtime
5. Platform customisation

- **SONIC**: customised for professional video
- multiple PIPEs (Plug In Processing Elements)
- each PIPE: programmable or fixed resources
  - FPGAs and memory
  - dedicated function core or ASIC
- modular interface, dedicated video buses
- multiple implementation targets
  - SONIC-1 : PCI 32-bit/32 MHz, FLEX10K
  - UltraSONIC : PCI 64-bit/66 MHz, XCV1000E
  - Sonic-on-Chip: XC2V, XC2VPro
SONIC architecture

PIPE = Plug In Processing Element

SONIC PIPE
PE: compute
PR: commun.
Multi-pipe design

SONIC-1 and UltraSONIC

(Source: Sony)
SONIC-1 and UltraSONIC

(Source: Sony)

Sonic-on-chip

[Diagram showing the layout and components of a chip, including RAM modules, bus wiring, and other components.]
Summary

- domain-specific customisation
- customised data processors
  - design-time customisation
  - run-time customisation
- custom instruction processors
- imperative languages e.g. C: RISC, Java: JVM
- logic languages e.g. Prolog: WAM
- design tool customisation
- platform-specific customisation

Challenge: theory+practice, extend+compose

Vision

C/Matlab/Premiere/Progol

Compiler

Machine code
Run-time interface
Configuration information

Fixed processor
Custom processor

Custom computing system
References

- overview

- domain-specific customisation
  - System Generator: www.xilinx.com

- customised data processors
  - design-time customisation
  - run-time customisation

References (cont.)

- custom instruction processors
  - imperative languages e.g. C: RISC, Java: JVM
  - logic languages e.g. Prolog: WAM

- design tool customisation

- platform-specific customisation
  - Haynes et al, IEEE Computer, April, 2000, 50-57