Practical Reconfigurable Computing

Michael Babst
DSPlogic, Inc.
www.dsplogic.com
301-977-5970
I am so happy, my C code runs 100x faster without changing one line of code. It runs on everyone’s reconfigurable computer, so I can easily compare them and just buy the fastest one! It even runs on both Xilinx and Altera FPGAs!
RC Fears

- *What* kind of C do I have to learn?
- Can’t you partition my algorithm for me?
- Fixed point? You must be kidding!
- Did you say – VHDL?
Example Design flow (Existing SW Application)

- Benchmark performance, profile execution, I/O
- Partition Algorithm
- Define CPU/FPGA messaging scheme
- Learn VHDL or Verilog
- Design FPGA
  - Code Application I/O (interface w/ custom vendor cores)
  - Verify I/O - Synthesize/Place/Route FPGA
  - Optimize I/O for BW/latency
  - Code Application Core (verify)
  - Synthesize/Place/Route FPGA (App+I/O)
  - Fiddle with I/O, Application until FPGA builds
  - Verify APP+I/O - Synthesize/Place/Route FPGA
  - Done!
  - Make a small change to the application
  - Become an Expert in VHDL/Verilog, HW design, timing diagrams...
  - Redesign I/O, Modularize design, Isolate I/O from application
  - Optimize speed and timing
- Working design!
Example Design flow (Existing SW Application)

- Here comes the Virtex 4!
- Re-partition Algorithm
- Redesign I/O ... 
- Get the idea?

RC can be a challenge, even for a hardware designer
Can I really use RC?

- The role of hardware abstraction
- Implementations
- Model-based design
- Application Examples

Reasonable methods exist today to accelerate your algorithm using RC
Hardware Abstraction
What is a Hardware Abstraction Layer

- According to [www.thefreeedictionary.com](http://www.thefreeedictionary.com)
  - Hardware Abstraction Layer - (HAL) The layer of Microsoft Windows NT where they have isolated their assembly language code.
- An abstract representation of a hardware function or an interface
- Described by an API
  - HLL: Function Calls
  - VHDL: Signal descriptions / “HDL wrapper”
  - Model: Block instantiation
- Purpose
  - To make algorithms portable (hardware independent)
  - To make the lives of programmers easier
  - Foster innovation / competition
FPGA Resource Abstraction

Algorithm Description

What is the best way to describe algorithm?

Automatic partitioning
Interconnect Optimization
Resource Mapping and Instantiation

Standardized abstraction for multipliers? adders? RAM?...

What is the best way to describe algorithm?
Device-level Abstraction

Algorithm Description

Standardized abstraction for interfaces?

Automatic partitioning
Interconnect Optimization
Resource Mapping and Instantiation
Multiple layers of Abstraction

- **APPLICATION**
  - Partition among Chassis
  - Partition among Modules
  - Partition among Devices

- **Chassis/Network Level Abstraction**
  - Chassis 1
  - …
  - Chassis N

- **Module Level Abstraction**
  - FPGA Module(s)
  - Backplane
  - I/O Module
  - GPP Module(s)

- **Device Level Abstraction**
  - FPGA Device 1
  - FPGA Device N
  - Memory Device(s)
  - Device Interconnect(s)

- **FPGA Resource Abstraction**
  - Multiplier
  - BlockRam
  - Wire
  - Slice
  - Flip Flop

Chassis
Practical Abstractions Today

Algorithm Description

Manual Partition

CPU Description (HLL)

API

FPGA Description (HLL* or Model)

API

Code Generation

Compile

RTL/HDL

Synthesize, Map, Place, Route

Processor

DSPlogic, Inc. Proprietary
Benefits of Hardware Abstraction

- Application portability / re-usability
  - Only need to port API
- Application upgradeability
- Speed new application development
- Valuable for ALL algorithm description and implementation methods
  - HDL, C, Model-based
- Simplify the lives of SW vendors
- Support OpenFPGA!

Maximize value of investment in application cores/libraries by making them more portable
Reconfigurable Computing I/O (RCIO) API
Coprocessor Interfaces

Loose Coupling  High-Overhead  Low-BW
Tight Coupling  Low-Overhead  High-BW

Network
Board-board
Today’s RC Platforms
Chip-chip
On-chip

DSPlogic, Inc. Proprietary
RCIO API Goals

- Speed new application development
  - Simplified communications interface
  - Eliminates communications driver development
    - FPGA and software
- Application portability
  - Only need to port API
- High-bandwidth / low-latency
- Very Low overhead
- Multiple CPU/FPGA support
- Separate control and data paths
- Support processing of data blocks smaller than message size
- Re-use existing standards where possible
Reconfigurable Computing I/O API

- Multiple processor / multiple FPGA support

- Transparent, Portable Application Interface!

Multiple processor / multiple FPGA support

Platform Specific Connection Fabric

CPU #1
- User Application
- RCIO SW library
- FPGA #1
- RCIO FPGA Core
- User Application

CPU #2
- User Application
- RCIO SW library
- FPGA #2
- RCIO FPGA Core
- User Application

CPU #N
- User Application
- RCIO SW library
- FPGA #N
- RCIO FPGA Core
- User Application

Transparent, Portable Application Interface!
Reconfigurable Computing I/O API

- A Simple, future-proof CPU/FPGA messaging interface

- Dramatically reduces FPGA development time
- Quickly achieve optimum latency and bandwidth
- Portable RC application development
- Future-proof: easy migration to newer, higher-performance FPGAs
- Separate data and control message paths
- Low-overhead
Software API - Data Interface

- `rcio_send()`  
  - Send single message to FPGA
- `rcio_receive()`  
  - Receive single message from FPGA
- `rcio_stream()`  
  - Blocking function
    - Break dataset into messages
    - Send all messages to FPGA
    - Receive all result messages from FPGA
  - Bandwidth-optimized
Software API - Control Interface

- Messaging control functions
  - rcio_config()
    - Initialize / configure CPU/FPGA communications link
  - rcio_status()
    - Return status of communications link
      - fpgaStatus
      - nMsgReceived, nMsgReturned
      - Fifo levels / over/underflow
  - rcio_close()
    - Close CPU/FPGA communications link

- User application control and status commands
  - rcio_appcfg()
    - Write to user-definable application control register
  - rcio_appstat()
    - Read user-definable application control register
RCIO Hardware Abstraction Layer (HAL)

DSPlogic™ RCIO Core

- Input Data Message FIFO
- Output Data Message FIFO
- Control I/O

User FPGA Application

- in_ready
- in_data
- in_write
- in_start
- in_length
- out_ready
- out_data
- out_write
- out_start
- out_length
- ctrl_reg(0-7)
- stat_reg(0-7)
- ib_depth, ob_depth

Data Processor

Platform Specific RAM / Peripherals

clk
rst
Data Message Structure (64-bit)

Dataset = M words

Dataset

\[ F_0 \ldots F_{M/K-1} \]

Message

K words / message

\[ D_0 \ldots D_{K-1} \]

User Data

64 bits/word

\[ \text{din}[63:0] \]

User-definable format

- real32
- real32
- real64
- int16
- int16
- int16
- int16
- custom

User data memory

User definable message format

Optional message header

I/O often limits performance
- Use care with CPU/FPGA algorithm partitioning
- Consider smaller data types
RCIO Hardware Abstraction Layer (HAL) API

- FPGA design tool-independent
- Supports wrappers/APIs in all design environments
  - High-level design tools
    - Xilinx System Generator, etc.
  - Custom VHDL / Verilog
  - High-level C
    - System C, Handel-C, System Verilog, Mitrion-C, etc.
RCIO FPGA Core / SW Library

Implementation for the Cray XD1 Supercomputer
Seamless Cray XD1™ CPU-FPGA Messaging

DSPlogic™ RCIO SW Library
rcio_send(fpga_id, *datap, txMsgLen)

Directly Link CPU and FPGA Applications!

Total Hardware Abstraction!

DSPlcog™ RCIO FPGA Core
- ready
- data
- write
- start
- length

Application Processing

FPGA

RAP
RCIO FPGA core and SW Library – Cray XD1™

- **User CPU Application**
- **User Data Memory**
- **Cray AA API**
- **Result Buffer (2 MB)**
- **DSPlgic™ RCIO Library**
- **Cray RT Core**
- **DSPlogic™ RCIO Core**
- **Input Data FIFO**
- **Output Data FIFO**
- **Control**
- **Process Data**
- **Application Accelerator**
- **User FPGA Application**
- **Opteron**
- **Cray QDRII Core**
- **8 MB RAM**

**Transparent interface**

- **1.42 GB/s (Max)**
- **1.1 GB/s (Typ)**

**DSPlogic, Inc. Proprietary**
Cray XD1 Performance

- Demonstrated performance and usability on multiple applications in multiple design environments
- **Extremely Modular - Multiple applications at full speed (200 MHz)**

---

**Achievable Data Rates (Mbytes/sec)** *(Including dataset sizes > 2 MB)*

<table>
<thead>
<tr>
<th></th>
<th>Send</th>
<th>Rcv</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theoretical Max (not achievable)</td>
<td>1422</td>
<td>1422</td>
<td>3022</td>
</tr>
<tr>
<td>Typical Send Only Application</td>
<td>1100</td>
<td>N/A</td>
<td>1100</td>
</tr>
<tr>
<td>Typical Receive Only Application</td>
<td>N/A</td>
<td>1100</td>
<td>1100</td>
</tr>
<tr>
<td>Typical Send/Receive Application</td>
<td>&lt;1100</td>
<td>&lt;1100</td>
<td><strong>1100</strong></td>
</tr>
<tr>
<td><strong>DSPlogic RCIO Send/Receive</strong> *</td>
<td><strong>825</strong></td>
<td><strong>825</strong></td>
<td>&gt;1650</td>
</tr>
</tbody>
</table>

* Using rcio_stream()

---

**Fastest CPU/FPGA Interface Available for the Cray XD1!**

**Combined (Send/Recv) Throughput** *(Symmetric Send/Recv rates)*

Applications immediately benefit from API enhancements
Rapid RC Development Kit (Cray XD1 Version)
A model is an abstract representation of a process
- Sound familiar?

Multiple views
- Multiple behavioral descriptions (Schematic/HLL/Hybrid)
- Multiple implementations/optimizations

Original uses
- Large system design and simulation
- Interface specification

Recent uses
- Code Generation
- Rapid prototyping
- Cycle-accurate simulation
Rapid Reconfigurable Computing Development Kit

Industry standard system design, algorithm modeling

- Implement Algorithm
- Call RCIO API Functions

Algorithm

CPU/FPGA Partition, Specify Dataflow

Simulink

Familiar Matlab/Simulink Environment

- Draw Data Processor Diagram
- Verify Data Processor Output
- RCIO FPGA Builder

Fully Integrated, Verified, Seamless Application

RCIO API
Transparent interface

DSPlogic, Inc. Proprietary
Model-based FPGA Design

- Advantages
  - VHDL not required (but possible)
  - Integrated algorithm verification
  - Industry standard, familiar environment
  - Easy integration of IP cores
  - Clear view of algorithm architecture
  - Optimized core libraries
    - Highly efficient use of FPGA resources
  - Ease of FPGA design verification
  - Automatic design documentation
  - Integrated bitstream generation

- Compatible with high-level languages
  - HLL Behavioral descriptions possible
Direct Algorithm To Platform FPGA!
**Fixed vs. Floating Point Considerations**

- **Fixed point and Floating Point support**
  - Floating Point
    - 32-bit Single Precision (23-bit mantissa, 8-bit exponent, 1 sign bit)
    - 64-bit Double Precision (52-bit mantissa, 11-bit exponent, 1 sign bit)
    - Excellent dynamic range
    - Large FPGA resource utilization
    - **Free Xilinx FP cores now available (Coregen / System Generator)**
  - Fixed Point
    - Mantissa only (Supports Integer or fixed decimal point)
    - No exponent - limited dynamic range
    - Variable mantissa for programmable accuracy
      - Potential for very high precision (greater than double-precision)
    - Well-developed tools for fixed-point design and analysis (Simulink)
    - Highly efficient utilization of FPGA resources

- **Accuracy Measurement**
  - Precision / Accuracy $\sim 1/2^{[#\text{mantissa bits}]}$
  - Normalized metric, independent of vector length
    
    $L_n = \left\| x \right\|_n = \left( \sum_{i=0}^{len-1} |x_i|^n \right)^{\frac{1}{n}}$
    
    $\text{compare}(a, b) = \frac{\left\| a - b \right\|_n}{\left\| b \right\|_n}$
Application Examples
FFT Performance Improvement

- ~10x improvement possible today!
  - Complex FFT, FPGA vs. FFTW on AMD 246
- Performance depends on data types

<table>
<thead>
<tr>
<th>Im_1(15:0)</th>
<th>Re_1(15:0)</th>
<th>Im_0[15:0]</th>
<th>Re_0[15:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Im_1(15:0)</td>
<td>Re_1(15:0)</td>
<td>Im_0[15:0]</td>
<td>Re_0[15:0]</td>
</tr>
</tbody>
</table>

in unused Re_0(15:0) unused Re_0[15:0]
out Im[31:0] Re[31:0]

Additional speed at expense of considering scaling / dynamic range effects

Accuracy similar to single-precision floating point algorithms
struct s_point pointprojection(struct s_plane plane, struct s_point p1)
{
    //
    // Get the projection of point p1 on the plane
    //
    v = p1 - plane.p; result = v - (v*plane.n)plane.n + plane.p
    //
    double temp;
    struct s_point vec1, proj;
    //
    // Get the vector (vec1) from a point in the plane to p1
    //
    vec1.x = p1.x-plane.p.x;
    vec1.y = p1.y-plane.p.y;
    vec1.z = p1.z-plane.p.z;
    //
    // temp = vec1 dot plane.n
    //
    temp = plane.n.x*vec1.x + plane.n.y*vec1.y + plane.n.z*vec1.z;
    //
    // Get the global coordinates for p1's projection
    //
    proj.x = vec1.x - temp*plane.n.x + plane.p.x;
    proj.y = vec1.y - temp*plane.n.y + plane.p.y;
    proj.z = vec1.z - temp*plane.n.z + plane.p.z;
    return proj;
}

Courtesy David Raila / Youssef Hashash
struct s_point
pointprojection(struct s_plane plane,
    struct s_point p1)
vec1.x = p1.x-plane.p.x;
vec1.y = p1.y-plane.p.y;
vec1.z = p1.z-plane.p.z;
temp = plane.n.x*vec1.x +
    plane.n.y*vec1.y +
    plane.n.z*vec1.z;
proj.x =
    vec1.x - temp*plane.n.x + plane.p.x;
proj.y =
    vec1.y - temp*plane.n.y + plane.p.y;
proj.z =
    vec1.z - temp*plane.n.z + plane.p.z;
return proj;
}

Performance results

- Computation Rate
  - (conservative)

  ![Graph showing millions of projections per second against message size.]

- Simple message format
  - Multiple projections/message to enhance BW

- Input data type precision
  - \( \sim 10^{-5} \)

- Computation Accuracy
  - Full-precision
  - L2Norm Error = 0

- 4x additional speed improvement (60M projections/sec) with data packing

- Next steps
  - Move more functionality into FPGA
  - Partition algorithm for lower I/O bandwidth
Summary
Questions to ask

- What level of abstraction do tools supported?
  - Efficiency/Flexibility of Mapping
  - New coding styles/constructs required
- What clock rate can I reasonably expect to achieve?
- Is I/O BW throughput dependent on FPGA clock frequency?
- Is I/O bandwidth symmetric / simultaneous?
- Will my application upgrade to next generation platform?
- Will my application be portable?
You can do it!

- No need to be a hardware design expert
- Will need FPGA awareness
- Tools are constantly improving
- Industry working together on standardization
- Reasonable paths to significant algorithm acceleration exist today
Contact Information

Michael Babst
www.dsplogic.com
301-977-5970 x705
mike.babst@dsplclogic.com