Reconfigurable Computing

Applications

Chapter 9

Prof. Dr.-Ing. Jürgen Teich
Lehrstuhl für Hardware-Software-Co-Design
Overview

- FPGAs have been used in the past mostly in **Rapid prototyping** and **Non-frequent reconfigurable systems**

- Hardware implementation, sometimes specific for the FPGA architecture:
  - Searching (text, genetic database, etc.)
  - Image processing

- Partially Reconfigurable Systems
  - Image Processing
  - SQL Processing
  - Stress-aware Placement
Searching – pattern matching

- Pattern matching is the basis of search engines
- The purpose is to find and (count) the occurrence of a given pattern in a given text
- Useful in:
  - Dictionaries
  - Document collection indexing
  - Document filtering and classification
  - Spam avoidance
  - Content surveillance
Searching – pattern matching – sliding windows

- Sliding windows
  (Cockscot & Foulk)
  - Keywords are kept in register. One character / Byte
  - A set of comparators are used. One comparator / Byte
  - Hit signal is set whenever the text-segment matches the corresponding word

- Advantage:
  - Easy to replace old patterns

- Drawbacks:
  - Not flexible: Fixed length of registers
  - Redundancy: more comparators than necessary for word with same prefix

Searching – pattern matching - sliding windows

- Avoid redundancy
  - Use only one comparator for common characters in different words

- Data folding (Foulk)
  - Fold the data in the circuit
  - Consider the bit-representation of each character
  - Generate a comparator circuit for each character in the words to be searched for
Searching – pattern matching - FSM-Based

- FSM-Based pattern matcher
  - Each regular grammar can be recognized by an FSM
  - In pattern matching, the target words define the regular grammar
  - The target words are compiled in the automaton
  - Each word defines a unique path from the start state to an end state
  - When scanning a text, the automaton changes its state with the appearance of characters
  - Reaching a final state corresponds to the appearance of a word
  - Redundancy is avoided by implementing common prefix

Searching – pattern matching - FSM-Based

- FSM-Based pattern matcher
  - RAM-implementation
    - One RAM or ROM for storing the state transition table
    - One state register
    - One character register
    - A hit detector
  - The input character and the state register are used to determine the next state
  - The hit detector checks if the current state is equal to a hit state and sets a hit for the corresponding word
- Advantage:
  - Simple to implement
- Drawback:
  - Expensive in terms of flip flops
- FSM-Based pattern matcher
  - One-hot implementation
    - Each state is coded in one flip flop
    - The D-input of the flip flop is obtained by an AND of the output of the previous flip flop with the result of the comparator
  - The comparator is character-specific
  - Only n FF are used to implement a word of length n
  - Advantage:
    - Low cost
    - Reflects the structure of the grammar
  - Drawback:
    - Not easy to build
    - Redundancy in the comparators

Searching – pattern matching - FSM-Based

- FSM-Based pattern matcher
  - Exploiting common prefix
    - For words with common prefix, only one common starting path corresponding to the length of the common prefix is used.
  - Redundancy of comparators can be avoided by implementing only one comparator for each character. The result of the comparison will then be provided to all gates using them.

Words with common prefix and the corresponding FSM.
Searching – pattern matching - FSM-Based

- FSM-Based pattern matcher
  - Optimized architecture
    - Implement the common prefix
    - Redundancy of comparators is removed: Each character in the set is implemented in a position vector: pos(i) = 1 iff character i is detected

Block diagram of the optimal pattern matcher

Detailed structure of the optimal pattern matcher

Pattern matching – use of reconfiguration

- FSM-Based pattern matcher
  - Use of reconfiguration
    - Replace the character comparators
    - Replace the FSM for a set of words

New character comparator

New set of words
Signal processing applications (FFT, Convolution, Filter algorithms) are characterized by MAC-intensive computations.

Signal processing functions are usually implemented on special processors:
- DSPs
- ASICs

FPGAs provide the advantage of reconfigurability, but MAC-intensive applications are expensive.

However, for MAC computations involving one constant vector, FPGAs present one of the best alternatives to DSPs.
Distributed arithmetic – Basics

Solution of the following equation: \( Z = A \times X = \sum (A_i \times X_i) \)

A constant row vector, \( X \) column vector

With the binary representation for \( X_i \):

\[
X_i = \sum X_{ij} 2^j
\]

\[
Z = A \times X = \sum A_i \times \left( \sum (X_{ij} \times 2^j) \right) = \sum 2^j \times \left( \sum A_i \times X_{ij} \right)
\]

\[ Z = \sum 2^j \times \left( \sum A_i \times X_{ij} \right) \] is the classical form of distributed arithmetic

Because the \( A_i \) are constant, there exist \( 2^n \) possible values for \( \sum A_i \times X_{ij} \)

We can pre-compute the possible values and store them in a LUT (DALUT) and retrieve them on demand at run-time

FPGA Advantage: Computation is memory-based (use of LUTs)
Distributed arithmetic – Basics

To better understand, we spread the DA equation

$$Z = [X_{10}A_1 + X_{20}A_2] + [X_{11}A_1 + X_{21}A_2] + \ldots + [X_{1(n-1)}A_1 + X_{2(n-1)}A_2] \cdot 2^{0}$$

$$\vdots$$

$$+ [X_{1(W-1)}A_1 + X_{2(W-1)}A_2] \cdot 2^{W-1}$$

The bits of the variables will be used to address the memory and retrieve the required values in a bit-serial way.

The DA-datapath implementation is straightforward.
Distributed arithmetic – Datapath

Parallel bit-serial input

\[ X_{1(W-1)} X_{1(W-2)} \quad \ldots \ldots \]

\[ X_{2(W-1)} X_{2(W-2)} \quad \ldots \ldots \]

\[ X_{n(W-1)} X_{n(W-2)} \quad \ldots \ldots \]

DA-LUT Address

\[ X_{11} X_{10} \]

\[ X_{21} X_{20} \]

\[ \ldots \]

\[ \ldots \]

\[ X_{n1} X_{n0} \]

DA-LUT

\[
\begin{array}{c}
0 \\
A_1 \\
A_2 \\
A_1 + A_2 \\
A_3 \\
A_3 + A_1 \\
A_3 + A_2 \\
A_3 + A_2 + A_1 \\
A_4 \\
\vdots
\end{array}
\]

j-shift

\[ Z \]

+/-
Distributed arithmetic – Datapath

k-parallel

\[ X_1 \times X_1 (w - 1) \]
\[ X_2 \times X_2 (w - 1) \]
\[ \ldots \]
\[ X_n \times X_n (w - 1) \]

DA-LUT 1

DA-LUT 2

DA-LUT k

ACC 1

ACC 2

ACC k

Adder tree

Reconfigurable Computing
Distributed arithmetic – Example

Recursive convolution of time domain simulation of optical multimode intra/system interconnects

Recursive formula to be implemented on 3 intervals

\[ y(t_n) = f_0 \cdot y(t_{n-1}) + f_4 \cdot x_0 - f_5 \cdot x_1 + f_{24} \cdot x_2 + f_{53} \cdot x_3 \]

Comparison of different implementations

<table>
<thead>
<tr>
<th>Workstation</th>
<th>1 interval</th>
<th>3 intervals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun Ultra 10</td>
<td>73.8 ms</td>
<td>354.2 ms</td>
</tr>
<tr>
<td>Athlon (1.53 GHZ)</td>
<td>558 ms</td>
<td>1967.4 ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FPGA (time)</th>
<th>1 interval</th>
<th>3 intervals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure dot-product</td>
<td>25.6 ms</td>
<td>76.8 ms</td>
</tr>
<tr>
<td>Sequential DA</td>
<td>19.4 ms</td>
<td>19.4 ms</td>
</tr>
<tr>
<td>3-parallel DA</td>
<td>6.4 ms</td>
<td>6.4 ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FPGA (area)</th>
<th>1 interval</th>
<th>3 intervals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure dot-product</td>
<td>fit</td>
<td>fit</td>
</tr>
<tr>
<td>Sequential DA</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>3-parallel DA</td>
<td>yes (7 %)</td>
<td>yes (14 %)</td>
</tr>
</tbody>
</table>

Virtex 2000E implementation on the Celoxica RC1000-PP board

Source: Image generated using Xilinx ISE.
Signal processing – Fast Fourier Transform (FFT)

- Fourier Series developed by the French Mathematician Joseph Fourier (1807)
  - Application of the initial idea in the field of heat diffusion

- The advent of digital computation and “discovery” of Fast Fourier Transform (FFT) in the 50's revolutionized the field of signal processing
  - Practical processing and meaningful interpretation of signals with exceptional importance for human and industry
    - Medical monitors and scanners
    - Modern electronic communications
    - Image processing
Signal processing – FFT - Basics

- Fourier transform $F(u)$ of a continuous function $f(x)$:
  \[ F(u) = \int_{-\infty}^{\infty} f(x) e^{-j2\pi ux} \, dx \]

- Given $F(u)$, we can obtain $f(x)$ by means of inverse Fourier transform
  \[ f(x) = \int_{-\infty}^{\infty} F(u) e^{+j2\pi ux} \, du \]

- Extension in 2-D, $F(u,v)$
  \[ F(u, v) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y) e^{-j2\pi (ux+vy)} \, dx \, dy \]

- Corresponding inverse in 2-D
  \[ f(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} F(u, v) e^{+j2\pi (ux+vy)} \, du \, dv \]
Signal processing – FFT - Basics

- Fourier transform of a discrete function of one variable $F(u)$
  \[ F(u) = \frac{1}{M} \sum_{x=0}^{M-1} f(x) e^{-j2\pi ux/M}, \quad x = 0, \ldots, M-1, u = 0, \ldots, M-1 \]

- Given $F(u)$, we can obtain, $f(x)$ by means of inverse Fourier transform
  \[ f(x) = \sum_{u=0}^{M-1} F(u) e^{+j2\pi ux/M}, \quad x = 0, \ldots, M-1, u = 0, \ldots, M-1 \]

- The Brute force computation of the Fourier transform requires $M^2$ multiplications and additions

- The performance can be improved to $(M \log M)$ using the successive doubling method
For notational convenience, we replace the previous equation with:

\[
F(u) = \frac{1}{M} \sum_{x=0}^{M-1} f(x) W_M^{(ux)}, \quad W_M = e^{-j2\pi/M}
\]

We assume \( M \) to be of the form \( M = 2^n = 2K \)

With \( n \) and \( K \) being positive integers each, we have:

\[
F(u) = \frac{1}{2K} \sum_{x=0}^{M-1} f(x) W_{2K}^{(ux)}
\]

\[
F(u) = \frac{1}{2} \left[ \frac{1}{K} \sum_{x=0}^{K-1} f(2x) W_{2K}^{(u(2x))} \right] + \frac{1}{K} \sum_{x=0}^{K-1} f(2x + 1) W_{2K}^{(ux)}
\]

\( F_{\text{even}}(u) \) \quad \text{and} \quad \text{F}_{\text{odd}}(u) \)}
Signal processing – FFT - Basics

\[ F(u) = \frac{1}{2} \left[ F_{\text{even}}(u) + F_{\text{odd}}(u) W_{2K}^u \right] \]

- Since \( W_{M}^{(u+M)} = W_{M}^{u} \) and \( W_{2M}^{(u+M)} = -W_{2M}^{u} \) we have

\[ F(u + K) = \frac{1}{2} \left[ F_{\text{even}}(u) - F_{\text{odd}}(u) W_{2K}^u \right] \]

- The FFT is a fast implementation of the Discrete Fourier Transform (DFT).
  - Based on a divide-and-conquer model,
  - \( M \log_2 M \) computations

Reconfigurable Computing
The N-point DFT computation can be divided into two N/2-point DFT computations. These N/2-point DFT computations can be divided into two N/4-point DFT computations, and so on.

- There are $\log_2(N)$ stages

- After the division and the DFT computation, a merging process is performed, in which the transforms are reassembled.
Signal processing – FFT – Algorithm – butterfly unit

- The butterfly unit
  - The reassembling is done by using the complex elements say, g and h from the previous stage.
  - Current element = \( g - h^*W_N^k \) and \( g + h^*W_N^k \)

- The twiddle factor
  - The twiddle factor terms (of the \( W_N^k = \exp(j \, 2\pi k/N) \)) must be available
  - The real and complex parts of these factors are stored in a ROM.
  - The factors correspond to a sine (imaginary part) and cosine (real part) functions, in a set of \( N/2 \) equally-spaced angles in an interval from 0 to \( (N-2)\pi/N \). Therefore, only a \( N/2 \)-position memory is needed.
Signal processing – FFT – FPGA implementation

- 16 points FFT
- Pipelined READ, EXECUTE and OUTPUT stage
  - Read one complex input (32-bits) in every cycle
    - 16 point input is read in 16 cycles
  - Output one complex result (32-bits) in every cycle
  - EXECUTE stage also takes 16 cycles
- Performance
  - Latency: 48 cycles
  - Throughput: 1 transform per 16 cycles
Signal processing – FFT – FPGA implementation

- **Resource requirements:**
  - 1432 out of 21504 flip-flops (6%)
  - 1037 out of 21504 LUTs (4%)
  - 65 out of 624 I/O pins (10%)

- **Timing**
  - Minimum clock period: 5.899ns
    (Maximum frequency: 169.520MHz)

- **Power estimation** 440 mW

Source: Image generated using Xilinx ISE.
Image processing – Basic operators

- Image processing algorithms usually process an image (set of points with a given characteristics such as color, gray level, luminance, etc.) point by point.
- The resulting pixels depend only on the pixels in the original picture.
- A sequential processor needs quadratic run-time to process a complete image. By using parallelism, each pixel can be computed independently.
- Many image processing system are based on the following operators:
  - Median filtering
  - Basic Morphological operations
  - Convolution
  - Edge detection
- Algorithms are often based on the moving window operator
The moving or sliding window algorithm usually processes one pixel of the image at a time.

The value of the pixel is changed by a function of a local pixel region covered by the window.

The operator moves over the image to cover all pixels.

For a pipelined implementation, all the pixel of the windows must be accessed at the same time for each clock.

FPGA implementation uses FIFO buffers.

FIFO Implementation
- FIFOs are implemented using circular buffers constructed from Multi-ported RAMs (available in, e.g., Virtex FPGA)
- Indexes keep track of the front and tail items in the buffer
- BLOCK RAMs are readable and writable in one clock cycle. This allows a throughput of one pixel per cycle.

3x3 windows
- 2 buffers of size W-3 (W = image width) are used
- The two FIFO buffers must be full to access all the window pixels in one cycle
- In each clock cycle, a pixel is read from the memory and placed into the bottom left corner
- The content of the window is shifted to the right with the right most member being added to the tail of the FIFO
- The top right pixel is disposed after computation, since it is not used in the future computation
Image processing – Median filtering

- **Basics**
  - An impulse noise (or salt and pepper noise) in an image has a gray level with higher low different from the neighbor point.
  - Linear filters have no ability to remove this type of noise.
  - Median filters share remarkable advantages on removing this type of noise.
  - Often used in digital signal and image/video applications.

- **Implementation**
  - Use a sliding window of odd size (e.g., 3x3) over an image.
  - At each window position, the median of the sample values is taken to replace the value at the center of the window.
  - High computational cost $O(N \log N)$ even using most efficient sorting algorithms.
  - General purpose processors are not a good solutions for real time implementation. This justifies the use of FPGAs.
Image processing – Median filtering

- **Sequential implementation** (pseudo code)

  ```
  For x=1 to # rows
    For y = 1 to # cols
      Build Windows array
      pixel(x,y) = Median(window array)
    End
  End
  ```

- **Complexity** $O(#\text{rows} \times #\text{cols} \times N \log N)$
  
  $(N=3)$
Image processing – Median filtering - result

Original image

Filtered image

Source: https://en.wikipedia.org/wiki/Lenna
Morphology in image processing studies the appearance of objects.

Useful for example in:
- Skeletonization
- Edge detection
- Restoration

Processing
- The image is processed pixel-by-pixel using a structuring element (the sliding windows)
- The window may fit or not to the image

Most basic building blocks:
- Erosion (shrinks or erodes an object in the image)
- Dilation (grows the image)
- Operations like opening and closing of an image can be derived by performing erosion and dilation in different order
Image processing – Basic Morphological Operators

- **Erosion**
  - Replaces the center pixel in the sliding window by the smallest pixel value in the window array
  - The bright area of the image shrinks, or erodes

- **Dilation**
  - Replaces the center pixel in the sliding window by the greatest pixel value in the window array
  - The bright area of the image grows

- **Algorithm**
  - Same as the median
  - Instead of selecting the median element, the minimum is selected for erosion and the maximum is selected for dilation
Image processing – Median filtering - result

Original image  Erosion  Dilation

Source: https://en.wikipedia.org/wiki/Lenna
Image processing – Convolution - Basics

- Convolution multiplies two arrays of numbers with different sizes and produces a third array of numbers.
- In image processing, convolution implements operators whose output pixels are computed as linear combinations of certain input pixels values.

1-D Convolution
- Formally, convolution takes two input functions $f(x)$ and $g(x)$ and generates $h(x) = f(x) * g(x)$ where $g(x)$ is referred to as the filter:

$$h(x) = \int_{-\infty}^{\infty} f(\tau) g(x-\tau) d\tau$$

2-D Convolution
- Most important in modern image processing
- A finite size window (convolution mask) is scanned over the image
- The output pixel value is the weighted sum of the input pixels within the window
- The weight is the value of the filter assigned to each pixel in the window
2-D Convolution

- Mathematically represented by the following equation:

\[ y(m, n) = \sum_{i=1}^{\text{img-height}} \sum_{j=1}^{\text{img-width}} h(i, j) x(m - i, n - j) \]

where \( x \) is the input image, \( h \) is the filter and \( y \) is the output image.

- Supports a virtual infinite variety of masks, each with its own feature.

- 3x3 convolutions are most commonly used and operate only on a pixel and its directly adjacent neighbours.
Image processing – Convolution – Gaussian filters

- Gaussian convolution filters
  - 1-D: \( G(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{\frac{-x^2}{2\sigma^2}} \)
  - 2-D: \( G(x,y) = \frac{1}{2\pi\sigma} e^{\frac{-x^2+y^2}{2\sigma^2}} \)

- The idea is to use the 2-D distribution as a point spread function. This is achieved by convolution.
- A discrete approximation of the Gauss function is required to perform the convolution.
- In theory, Gauss distribution is zero anywhere. Therefore an infinite large convolution kernel may be required.
- But in practice, the convolution kernel is truncated as shown in the pictures.

\[
\begin{array}{ccc}
 1 & 21 & 31 \\
 31 & 48 & 31 \\
 21 & 31 & 21 \\
\end{array}
\]

3x3 Gaussian smooth filter

\[
\begin{array}{cccccc}
 2 & 4 & 5 & 4 & 2 \\
 4 & 9 & 12 & 9 & 4 \\
 5 & 12 & 15 & 12 & 5 \\
 4 & 9 & 12 & 9 & 4 \\
 2 & 4 & 5 & 4 & 2 \\
\end{array}
\]

5x5 Gaussian smooth filter

\( \sigma = 1.4 \)
Image processing – Convolution – Gaussian filters

Original image

Convolution

Source: https://en.wikipedia.org/wiki/Lenna
**Image processing – Edge detection - Basics**

- **Edges**
  - Placed in image with strong intensity contrast
  - Often occurs at image location representing boundaries

- **Edge detection**
  - Extensively used in image segmentation, i.e., dividing an image into areas corresponding to different objects
  - Representing an image by its edges significantly reduces the amount of data
  - Since edges correspond to strong illumination gradients, the derivatives of the image are used to compute the edges
  - Operators often used are
    - Laplace operator
    - Sobel operator
    - Canny edge detection algorithm
Image processing – Edge detection - operators

- **Laplace**
  - Gradient operator
  - Intensity difference are enhanced. The edges are more pronounced
  - For each pixel, the gray value of its four neighbours (top, left, bottom, right) pixel value are subtracted from its own value

- **Sobel operator**
  - Combination of two 1-D operators
    - One for detecting horizontal edges
    - One for detecting vertical edges
Image processing – Convolution – Gaussian filters

Original image

Edge detection

Source: https://en.wikipedia.org/wiki/Lenna
Image processing – Use of reconfiguration

- Intelligent image processing system
  - According to input image and other conditions,
    - Some operations are done to improve the image
      - Filtering (the correct filter is chosen)
      - Smoothing
      - Segmentation (Edge detection)
      - Skeletonization
  - Some adjustments are done on the image input hardware
    - Calibration
    - Focussing
  - Everything is done while the system keeps running
    - Fixed parts of the system will run continuously
    - Reconfigurable must be replaced at run-time
Controller task is to influence the dynamic behavior of a plant.

Input values for the plant depend on plant's outputs (Feedbacks).

A plant is modeled as a linear time invariant (LTI)-System.

Controller is modeled as LTI-System.

Time discretization:
- Scaling to fix-point
- $k, k+1, k+2$: sample points
- $T$: sample period
- $t_c$: calculation time of controller.
Mechanical Control – Basics

\[ \begin{align*}
\mathbf{x}_{k+1} &= A \mathbf{x}_k + B \mathbf{u}_k \\
\mathbf{y}_k &= C \mathbf{x}_k + D \mathbf{u}_k
\end{align*} \]

*\( u \): Controller input, *\( x \): State, *\( y \): Output
*\( A, B, C, D \): Constant coefficient matrices

Combine \( A, B, C, D \) to \( M \), represent computation as a set of scalar products of each row of \( M \) with \( \mathbf{v} \):

\[ \mathbf{z} = M \mathbf{v} = \begin{bmatrix} M_{1\text{row}} \mathbf{v} \\ M_{2\text{row}} \mathbf{v} \\ \vdots \end{bmatrix} \]
Scalar product: const. vector $\vec{a}$ and var. vector $\vec{x}$

\begin{align*}
(1) \quad z &= \vec{a} \cdot \vec{x} = \sum_{i=1}^{n} x_i a_i \\
(2) \quad x_i &= \sum_{j=0}^{w-1} x_{ij} 2^{-j} \\
(3) \quad z &= \sum_{i=1}^{n} a_i \sum_{j=0}^{w-1} x_{ij} 2^{-j} \\
&= \sum_{j=0}^{w-1} 2^{-j} \sum_{i=1}^{n} x_{ij} a_i
\end{align*}

$x_i$ as $w$-bit fix-point (here $x$ just unsigned in $[0,2]$)

Replace (2) in (1), swap the sums since $x_{ij}$ is in $\{0,1\}$

Right sum can have just $2^n$ values

Pre-compute it and store it in a $2^n \times w$ ROM as LUT
Mechanical Control – DA-Implementation (Parallel)

\[ z = \sum_{j=0}^{w-1} 2^{-j} \text{DALUT}(x_{[1..n]j}) \]

c-Bit at a time Architecture

Reconfigurable Computing
Mechanical Control – Multi controller system

- Many controller modules optimized for different operating regimes
- Controllers have different structures not only different coefficients
- Supervisor observes plant and determines best controller module
- Multiplexer switches controller outputs
Mechanical Control – Multi controller system

- Periodic execution of task graph
- Conditional branching to controller modules (CM)
- CMs implement various area/time trade-offs possible

Diagram:

- Periodic execution (every sample period T)
- Graph showing CMs connected in a network.
Multi controller system - Use of reconfiguration

- One slot solution

Diagram:
- SV
- CM1
- CMi
- CMn
- periodic execution (every sample period T)
- FPGA area
- time
- one slot
- CM3
- CM3
- CM3
- CM2
- CM2
- CM2
- CM1
- CM1
- CM1
- CM1
Multi controller system - Use of reconfiguration

- Two slots solution
Multi controller system - Use of reconfiguration

**Execution time:**

\[ t_{ex}^{CM} = f(A^{CM}) \]

**Reconfiguration time:**

\[ t_{reconfig}^{CM} = g(A^{CM}) \approx r \times A^{CM} \]

\[
A^{(2)} = 2A^{CM} = 2f^{-1}(t_{ex}^{CM}) \\
T_{min}^{(2)} = t_{ex}^{CM} = f(A^{(2)} / 2)
\]

\[
A^{(1)} = A^{CM} = f^{-1}(t_{ex}) \\
T_{min}^{(1)} = t_{ex}^{CM} + t_{rec}^{CM} = f(A^{(1)}) + g(A^{(1)})
\]
FPGA Implementation - inverse pendulum

- Host
  - Simulates Plant
  - Host configurations

- RAPTOR 2000
  - Comm. Resources
  - Configuration Manager

- FPGA
  - Controller Module
  - Supervisor
  - Communication

- Controller characteristics:
  - Dimensions: $p=3$, $n=2$, $q=3$
  - Word-width: Input: 16 Bit, Intern: 32 Bit

- Synthesis results of controller:
  - FPGA: Virtex 800
  - Area: 1003 slices (ca. 10%)
  - Clock Rate: > 70 MHz
  - Computation Time: $t_c < 1 \mu s$

The Raptor 2000 Board

Source: www.raptor2000.de
Partially Reconfigurable SoC

- FPGA platform for rapid prototyping of partially reconfigurable video applications (FPL’10)
- System-on-Chip consists of
  - CPU subsystem
  - Partially reconfigurable regions (PRRs)
- Implemented on XUP V2P-Board

Source: Digilent, Inc. Xilinx’s XUP Virtex-II Pro Development System Board

Partially Reconfigurable SoC

- **CPU subsystem:**
  - PowerPC
  - External interfaces
  - Memory controller
  - PLB bus

- **Partially reconfigurable regions:**
  - Video in/out
  - Streaming bar
  - ReCoBus

**PR Module Communication**

- **I/O-bar:**
  - Streams the *Video signal* through the PRRs
  - Can be used and modified by *hardware modules*

- **ReCoBus:**
  - Communication between the static and the partial regions
  - *DMA access* from hardware modules to the DDR memory

Reconfiguration Flow

Partially Reconfigurable SoC

Advantages Dynamically Reconfigurable SoC

- Modules can be “loaded” and “unloaded” at runtime
- New modules can be developed without touching the static design and integrated on the fly
- Modules can be freely placed in the reconfigurable region
  - The data-flow processing sequence of the modules can be easily exchanged
  - A module can be instantiated multiple times without need to modify the partial bitfile
  - The placement of a module may be determined at runtime
- Summary advantages wrt. statically developed systems:
  - Better resource utilization
  - Faster reconfiguration times
  - Cost-effective implementation, i.e., of Multi-Mode Systems (CODES+ISSS’11), (FPT’11)
## References

<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
<th>Authors</th>
<th>Conference/Proceedings</th>
<th>Pages/Publication Details</th>
</tr>
</thead>
</table>
SQL Query Processing

- Modern database systems must handle tons of data
- Many queries have to be answered as fast as possible
- Usage of FPGAs to accelerate SQL query execution
SQL Query Processing

- **Static approach**: Build one accelerator per query
  - Not applicable, if queries are not known in advance

- **Dynamical approach**:
  - Map single SQL operations (SELECT, WHERE, …) to single modules
  - Assemble a data path at run-time out of the modules to execute a query
  - Use partial reconfiguration to switch fast from one query to the next one
Idea: Translate each SQL query into an FPGA-based accelerator circuit through run-time assembly of dynamically reconfigurable hardware modules (FCCM’12)
SQL Query Processing

- Analyze queries and transform them into appropriate structures
- SELECT * FROM t
WHERE (((a * b) - ((c - d) * e)) + f) < (g + (h - i))
SQL Query Processing

• After building the pipeline, put the table data in and execute the query

<table>
<thead>
<tr>
<th>id</th>
<th>price</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1200</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>450</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>200</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>3000</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>2500</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>550</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>5000</td>
<td>0</td>
</tr>
</tbody>
</table>

SELECT * FROM product WHERE price < 1000 AND quantity > 0
SQL Query Processing

Speedups for simple and complex queries could be achieved in comparison to a standard MySQL installation (INMEMORY tables were used)

Q3: SELECT * FROM test_table WHERE (str < 'abcdefghabcdefgh') OR (g > 500);

Q5: SELECT * FROM test_table WHERE (((a+b)-(c-d))<((e+f)+(g-a))) AND (((b+c)-(d-e))<((f+g)+(a-b))) XOR (((c+d)-e)<(f+g)) OR ((a-b)>(c+d));

<table>
<thead>
<tr>
<th>Execution time for a 256 MB workload</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Query</strong></td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>Q3</td>
</tr>
<tr>
<td>Q5</td>
</tr>
</tbody>
</table>

Reconfigurable Computing
References

[1] On-the-fly Composition of FPGA-Based SQL Query Accelerators Using A Partially Reconfigurable Module Library
   C. Dennl, D. Ziener and J. Teich
   In Proceedings of the IEEE International Field-Programmable Custom Computing Machines Symposium (FCCM)
   Toronto, Canada, April 29th - May 1st, 2012.

   C. Dennl, D. Ziener and J. Teich
   In Proceedings of the IEEE International Field-Programmable Custom Computing Machines Symposium (FCCM)
   Seattle, USA, Apr. 28 - 30, 2013

[3] Energy-Aware SQL Query Acceleration through FPGA-Based Dynamic Partial Reconfiguration
   A. Becher, F. Bauer, D. Ziener and J. Teich
   In Proceedings of the Conference on Field-Programmable Logic and Applications (FPL)
   Munich, Germany, Sep 2-4, 2014

[4] FPGA-Based Dynamically Reconfigurable SQL Query Processing
Is it possible to improve the reliability of an FPGA by using partial dynamic reconfiguration?

Some ideas

- Dynamic placement of Replicas + Voters
- „Scrubbing“ against SEE
- Take degeneration effects into account and place new modules accordingly
Motivation:

- Degeneration effects on digital circuits may lead to performance impact and system failures
- Some areas of the FPGA may degenerate faster than others (e.g., depending on the temperature and)

Idea: Increasing the reliability of a reconfigurable system by wear-leveling of the FPGA (FPL’11)
New Aging Model: keep track of degradation on FPGA at runtime

- Estimate the delay increase due to a module $m^k$ due to NBTI in each (relative) LUT of module (see previous slide)
- Store and update values in degradation table $F(t)$ each time a module $m^k$ finishes its execution at time $t_k$:

$$F(t) = \begin{pmatrix}
  f_{1,1}(t) & \ldots & f_{\text{width},1}(t) \\
  \ldots & \ddots & \ldots \\
  f_{1,\text{height}}(t) & \ldots & f_{\text{width},\text{height}}(t)
\end{pmatrix}$$

$$f_{i,j}(t_k) = \Delta d(m^k_{i-x,j-y}, \Delta t_k + \Delta t')$$

$$= c_1 \cdot \exp \left( \frac{c_2}{m^k_{i-x,j-y}} \right) \cdot (\Delta t_k + \Delta t')^n$$

- Create temperature profile for each module offline
- Use the Arrhenius equation to estimate degradation delay
- Use online measurements for re-calibration
Stress-aware Module Placement

- **Stress-Aware Placement (SAP) algorithm**: Degradation delay depends on execution time and temperature profile of corresponding module
  - Keep track of degradation for each LUT using a degradation table $F$
  - Place new module to minimize maximal degradation in $F$
  - Update degradation table $F$ when a module finishes
Problem:

- **Aging effects** (e.g., HCI, TDDB, NBTI) may lead to permanent system degeneration and reduced lifetime
  - Transistor miniaturization increases susceptibility

- **Radiation effects** can lead to transient faults (SEUs) and system breakdown
  - Reliability issue for SRAM-based FPGAs: dominated by memory cells

How to **protect dynamically reconfigurable systems** against both threats?
Stress-aware Module Placement

\[ R_m(\tau) \]

modules

replicas

allocation

placement

FPGA

(FPT’11)

Reconfigurable Computing
References
