Programmable and Customizable Hardware Accelerators for Self-adaptive Virtual Processors in FPGA

by

Ing. Jaroslav Sýkora

A dissertation thesis submitted to the Faculty of Information Technology, Czech Technical University in Prague

Doctoral study programme: Informatics

Prague, December 2013
Supervisor:
Ing. Martin Daněk, Ph.D.
Signal Processing, ÚTIA AV ČR, v.v.i.; and
Computer Systems Group, TU Darmstadt
xdanek@email.cz

Copyright © 2013 Jaroslav Sýkora
http://www.jsykora.info/

This book is available under the terms of the ‘Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported’ license (CC BY-NC-SA 3.0).

Electronic version of the book is available at: http://www.jsykora.info/thesis
This text is the version v15-1-g297c.
Abstract

The roots of all evil are the latencies that are statically unpredictable. Dynamic schedule of operations, constructed on-the-fly in data-driven machines, is needed to overcome them. Microthreading is a unified data-driven and dynamically scheduled model for efficient programming of many-core general-purpose processors. It overcomes unpredictable latencies in off-chip memories (DRAMs) and in on-chip shared interconnect. As silicon chips became power-limited, causing the shift from frequency scaling to many-core scaling, the previous work envisioned large-scale homogeneous many-core chips because it assumed that low-clock frequency silicon is easily scalable in space. However, the contemporary and future power constraints will favor heterogeneous (specialized) rather than homogeneous (general-purpose) many-cores because the thermal design power of a chip could be so low that not all cores may be powered up simultaneously.

Besides the power issues the other negative side-effect of silicon scaling is an increase in latency of interconnect (metal wires) relative to that of gates: new designs are becoming limited by interconnect delays. As the interconnect delays depend on details of physical placement of modules in a chip or in a reconfigurable array they are difficult to predict accurately early on in the design process. Consequently, future hardware will be special-purpose and customized due to the power issues, and it will be data-driven to overcome on-chip interconnect latencies.

This dissertation explores dataflow latency-tolerant techniques with a focus on customized hardware design using reconfigurable hardware arrays. Dataflow is studied at the gate and chip levels: gate-level dataflow overcomes on-chip interconnect delays, and chip-level dataflow allows for the composition of scalable heterogeneous many-cores.

The first contribution is an analysis of a contemporary statically scheduled instruction-driven architecture for customized computing realized in an FPGA. In contrast to the original design bases of the architecture it is shown here that high-frequency instruction issue is needed even in an architecture with batch (vector-based) data processing. The second contribution is a method to achieve the high-frequency instruction issue by using dictionary tables of instruction fragments.

Statically scheduled data-path used to be preferred because all latencies (including interconnect) were assumed to be fully known early in the design time. The third contribution is a new structured and extensible approach for synthesis of hardware controllers from synchronous Petri nets. The fourth contribution is a new technique for dataflow hardware synthesis from Petri nets. The technique is based on augmented synchronous Petri nets with optimal throughput.

The fifth contribution is a technique that combines the data-driven microthreaded procedural computation model with the special-purpose data-driven hardware in structurally programmed reconfigurable arrays. Adaptive transparent migration of microthreads between the general-purpose and special-purpose hardware is demonstrated.

Keywords:


Acknowledgement

This work has been partially supported from projects **Apple-CORE** (project number FP7-ICT-215216 and MSMT 7E08013), **SMECY** (project number Artemis JU 100230 and MSMT 7H10001), and **ADVANCE** (project number FP7-ICT-248828). The work has been carried out during my employment at the Department of Signal Processing (SP) of the Institute of Information Theory and Automation (UTIA) of the ASCR in Prague, Czech Republic.

_I would like to thank my advisor, Martin Daněk, for the chance he gave me to work on projects Apple-CORE and SMECY in UTIA, and also for all the guidance I have received._
# Contents

Abstract iii  
List of Figures x  
List of Tables xv  
Abbreviations xvii  

1 Introduction  
1.1 Reasons for Data-Driven Scheduling 1  
1.1.1 Latencies and Scheduling 1  
1.1.2 Data-Driven vs. Instruction-Driven Machines 2  
1.1.3 Scheduling Granularity 2  
1.2 Convergence of Procedural and Structural Programming 3  
1.3 Objectives and Contributions 4  
1.4 State of the Art 6  
1.5 Organization 9  

2 Technology Aspects  
2.1 Gate Scaling: Power Constraints 11  
2.2 Wire Scaling: Interconnect Routing Delays 13  
2.2.1 Latency-Tolerant Structured Interconnect Instead of Global Wires 14  
2.2.2 Interconnect Delays in Reconfigurable Arrays 14  
2.3 Summary 18  

I Limits of Static Scheduling in Programmable Architectures 19  

3 Programmable Accelerators in FPGA  
3.1 Motivation for Static Scheduling 21  
3.2 Motivation for Software Programmability in FPGA 21  
3.3 Motivation for Vector Processing 23  
3.4 System Integration 24  

4 Application-Specific Vector Processors 25  
4.1 Description of the Architecture 25  
4.1.1 Overview & System Integration 25  
4.1.2 Vector Processing Unit (VPU) 26
## CONTENTS

4.1.3 Embedded Scalar Controller .................................. 28
4.2 Domain Function Unit (DFU) ..................................... 30
  4.2.1 Customizing the Domain Function Unit ..................... 30
  4.2.2 Practical Example ........................................... 31
4.3 Characterization of the Hardware Implementation ................. 32
  4.3.1 Operating Frequency ........................................ 32
  4.3.2 Resource Utilization ....................................... 33
4.4 Performance of Application Kernels ............................ 35
  4.4.1 Matrix Multiplication (MATMUL) ............................ 36
  4.4.2 Mandelbrot Fractal Set (MF-set) ............................ 37
  4.4.3 Image Segmentation (ImgSeg) ............................... 40
4.5 Summary .......................................................... 44

5 Reducing Instruction Issue Overheads ............................ 47
  5.1 Analysis of the Cause ........................................... 47
    5.1.1 Solution 1 (Not Used): Improving Controller Performance .. 48
    5.1.2 Solution 2 (Adopted): Improving the V-Instruction Issue Hardware ... 49
  5.2 Comparison to the VLIW Approach ............................. 49
  5.3 New Dictionary Tables and APIs ................................ 50
  5.4 Evaluation ...................................................... 53
  5.5 Summary ........................................................ 60

II Dataflow at the Gate Level ....................................... 61

6 Controller Synthesis Using Synchronous Petri Nets ............... 63
  6.1 Petri Nets: A Multitude of Variants (Definitions) ............ 63
    6.1.1 Synchronous Petri Nets (SPN) ............................ 65
  6.2 Activation Nets: A New Method for SPN Synthesis ............ 66
    6.2.1 Places .................................................... 66
    6.2.2 Transitions ............................................... 67
    6.2.3 Alternate Choices ....................................... 68
  6.3 Synthesis to Hardware ......................................... 73
    6.3.1 Non-Autonomous Petri Nets ............................... 73
  6.4 Case Study: Transputer Link Controller ....................... 74
    6.4.1 Original Solution (SIP-net) .............................. 75
    6.4.2 New Solution (SPN/AN) ................................... 77
  6.5 Summary ........................................................ 78

7 Dataflow Hardware Synthesis from Petri Nets ..................... 81
  7.1 Petri Nets and the Bubble Wall ................................ 81
  7.2 Flow-Transfer Level: Scheduling Without Bubbles ............ 82
    7.2.1 ‘Cells’ Instead of Places ................................ 83
    7.2.2 Composition in FTL ...................................... 84
  7.3 Attaching Data Paths ........................................... 84
  7.4 Algebra with FTL Nets ......................................... 86
  7.5 Hardware Synthesis ............................................. 89
CONTENTS

7.5.1 Synthesis of Pipes as the Module Interfaces ...................... 89
7.6 Case Study: Mandelbrot Fractal Set ................................ 90
  7.6.1 Conversion to the FTL Net Model .............................. 92
  7.6.2 Implementation in FPGA ........................................ 93
7.7 Summary ........................................................................ 97

III Dataflow at the Chip Level .............................................. 99
8 Hardware Families of Threads .............................................. 101
  8.1 Data-Flow Processing Across Heterogeneous Multicores .......... 101
  8.2 Execution in Threads .................................................. 102
  8.3 Data-Driven Thread Scheduling ...................................... 103
  8.4 Multicore Support ...................................................... 105
  8.5 Heterogeneous Multicores and Hardware Families of Threads ... 106
      8.5.1 Overview .......................................................... 107
      8.5.2 Create Events .................................................... 108
      8.5.3 Family Classification ........................................... 108
      8.5.4 Thread Mapping Table ......................................... 109
  8.6 Complete Model of the HWFAM ...................................... 112
  8.7 Summary ...................................................................... 113

9 Evaluating the Hardware Families of Threads ................................ 115
  9.1 Hardware Implementation Details .................................... 115
  9.2 Controlling the Processor/Accelerator Task Placement ........... 117
      9.2.1 Occupancy-Based Placement ................................ 117
      9.2.2 Magnitude-Based Placement .................................. 123
  9.3 Implementation in FPGA .............................................. 126
  9.4 Tolerance of the TMT Latency Scaling ............................... 126
  9.5 Summary ...................................................................... 127

IV Results and Conclusions .................................................. 129
10 Comparison of the Dataflow Approaches ................................ 131
  10.1 Platforms .................................................................... 131
  10.2 Performance Metrics .................................................. 133
  10.3 Evaluation ................................................................... 134
      10.3.1 Lines of Code ...................................................... 134
      10.3.2 Finite Impulse Response Filter (FIR) ......................... 135
      10.3.3 Discrete Cosine Transform (2D-8x8 DCT) ................. 138
      10.3.4 Mandelbrot Fractal Set (MF-set) ............................ 140
  10.4 Sensitivity to Unpredictable Latencies ............................... 143
      10.4.1 Latency Tolerance in Processors .............................. 146
      10.4.2 Latency Tolerance in Specialized Hardware .............. 146
  10.5 Summary .................................................................... 148
11 Conclusions ................................................................. 153
CONTENTS

F.2 Mapping of the FIR Filter in HWFAM .......................... 210

G HWFAM: Experimental Data ........................................... 213
  G.1 Experimental Set-up ............................................... 213
  G.2 Experimental Configurations ..................................... 214
  G.3 Ad: Occupancy-Based Placement ................................. 218
## List of Figures

1.1 Statically predictable vs. unpredictable latencies. .......................... 1
1.2 The type of latencies encountered determines which machine type is best suitable. 3
1.3 Instruction- and data-driven machines programmed in time and space. 4
1.4 Map of the thesis coverage. .................................................... 5
1.5 Different types of elasticity: a) Inelasticity; b) asynchronous elasticity; c) Synchronous elasticity. Idle periods denoted by shaded boxes. [CCKT09] 7
2.1 Some wires do not scale. ......................................................... 13
2.2 Composition of circuit delay. .................................................. 14
2.3 Illustration of a mapping of a SoC consisting of 8 ASVP cores, MicroBlaze (MB), DRAM memory controller (MPMC), DMA engine. ....................... 15
2.4 Mapping of a SoC consisting of 8 ASVP cores in FPGA ....................... 16
2.5 Timing failure due to difficult placement. Fixed by register insertion in a different place. 17
2.6 Detail of pipelining around BRAMs at the DMA/ASVP boundary. .............. 17
3.1 Architecture-based approach to the design of custom accelerators ............ 22
3.2 Multithreading, SIMD threading, vector processing. ........................ 23
3.3 Vector processor is controlled by a scalar controller. ......................... 23
3.4 Basic shared bus system. ....................................................... 24
3.5 Using DMA engine with separate data bus for bulk data transfers. ............ 24
3.6 HWFAM organization with microthreaded processor. ........................ 24
4.1 Internal organization of the ASVP core. ...................................... 26
4.2 Vector Processing Unit. ......................................................... 27
4.3 The crossbar between local memory banks (left) and the address generators (right). 28
4.4 Crossbar arbiter. ............................................................... 28
4.5 Area breakdown (FF, LUTs and Slices) in Virtex 6 technology .................. 35
4.6 Mapping process. ............................................................... 36
4.7 Matrix Multiplication (MATMUL) execution time in microseconds for different matrix sizes and technology nodes with varying frequency/latency ratios. .... 38
4.8 Using complex addressing modes, one row of matrix multiplication can be computed using only one vector instruction. 38
4.9 Mandelbrot fractal set execution times ....................................... 41
4.10 Updating Gaussian models in the image segmentation algorithm (illustration of the principle). 42
4.11 Data structures in the image segmentation kernel. ............................ 43
4.12 Image Segmentation execution time ......................................... 43
5.1 Structure of a vector instruction in an ASVP with 5xAG. ....................... 47
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>Doubling the controller operating frequency $f_0$ (simulation only results)</td>
<td>48</td>
</tr>
<tr>
<td>5.3</td>
<td>Image Segmentation histogram</td>
<td>49</td>
</tr>
<tr>
<td>5.4</td>
<td>Compact representation of v-instructions using two-level dictionary tables.</td>
<td>51</td>
</tr>
<tr>
<td>5.5</td>
<td>Vector processor is controlled by a scalar controller with a specialized configuration store.</td>
<td>52</td>
</tr>
<tr>
<td>5.6</td>
<td>Controller in ASVP</td>
<td>52</td>
</tr>
<tr>
<td>5.10</td>
<td>Toolchain</td>
<td>55</td>
</tr>
<tr>
<td>5.11</td>
<td>The MF-set kernel: original (Level 0 API only) vs. new (all APIs)</td>
<td>57</td>
</tr>
<tr>
<td>5.12</td>
<td>The ImgSeg kernel</td>
<td>58</td>
</tr>
<tr>
<td>5.13</td>
<td>New configuration scheme: resource costs in FPGA slices compared to the original cores.</td>
<td>59</td>
</tr>
<tr>
<td>6.1</td>
<td>Illustration of a firing (transition) rule.</td>
<td>64</td>
</tr>
<tr>
<td>6.2</td>
<td>Extended Petri net with inhibitor arcs (dashed) representing a producers-consumers system with a priority.</td>
<td>65</td>
</tr>
<tr>
<td>6.3</td>
<td>A single PN place and its activation net.</td>
<td>66</td>
</tr>
<tr>
<td>6.4</td>
<td>Activation net of one place.</td>
<td>67</td>
</tr>
<tr>
<td>6.5</td>
<td>A four-place pipeline in PN and AN.</td>
<td>68</td>
</tr>
<tr>
<td>6.6</td>
<td>General multi-input multi-output transition in SPN ($s_3$) is constructed in AN by using a common wand node.</td>
<td>69</td>
</tr>
<tr>
<td>6.7</td>
<td>Two undesirable situations in synchronous Petri nets [FAP97].</td>
<td>69</td>
</tr>
<tr>
<td>6.8</td>
<td>The ‘switch’ and ‘select’ choice transitions deterministically resolve conflicts.</td>
<td>69</td>
</tr>
<tr>
<td>6.9</td>
<td>Declarative specification of a ‘choice’ with implicit (built-in) priority-based conflict resolution.</td>
<td>70</td>
</tr>
<tr>
<td>6.10</td>
<td>The choice AN fragment constructed from the table on the left.</td>
<td>71</td>
</tr>
<tr>
<td>6.11</td>
<td>Splitting a wand node.</td>
<td>72</td>
</tr>
<tr>
<td>6.12</td>
<td>A simple SPN with the select choice, and its AN.</td>
<td>72</td>
</tr>
<tr>
<td>6.13</td>
<td>Explicitly controlled choices.</td>
<td>73</td>
</tr>
<tr>
<td>6.14</td>
<td>Predicates (has-token?, fires?) and commands (enable!) that provide the interface between PN or AN and the environment.</td>
<td>74</td>
</tr>
<tr>
<td>6.15</td>
<td>Transputer link adapter.</td>
<td>74</td>
</tr>
<tr>
<td>6.16</td>
<td>Timing of the link transmitter.</td>
<td>75</td>
</tr>
<tr>
<td>6.17</td>
<td>Timing of the link receiver.</td>
<td>75</td>
</tr>
<tr>
<td>6.18</td>
<td>Transputer link controller in SPN (SIP-net) using the method in [FAP97].</td>
<td>76</td>
</tr>
<tr>
<td>6.19</td>
<td>Transputer link controller using the SPN/AN approach with alternate choice nodes $S_1$-$S_4$.</td>
<td>76</td>
</tr>
<tr>
<td>6.20</td>
<td>Macro-places used in the transputer link controller.</td>
<td>76</td>
</tr>
<tr>
<td>6.21</td>
<td>In the previous work the dynamic conflict $c_2$ was resolved manually by adding an inhibitor arc from $p_{14}$ to $t_8$.</td>
<td>77</td>
</tr>
<tr>
<td>7.1</td>
<td>Timing of synchronous systems with global clock: RTL circuit and Petri net.</td>
<td>81</td>
</tr>
<tr>
<td>7.2</td>
<td>Petri nets are bubble-limited.</td>
<td>82</td>
</tr>
<tr>
<td>7.3</td>
<td>Synchronous flow of tokens requires global information about the state of Petri net.</td>
<td>83</td>
</tr>
<tr>
<td>7.4</td>
<td>The thought process starts at SPN.</td>
<td>83</td>
</tr>
<tr>
<td>7.5</td>
<td>Definitions of the standard place and the new cell nodes in SPN and FTL.</td>
<td>84</td>
</tr>
<tr>
<td>7.6</td>
<td>A pipeline composed of 4 cells with the flow-through state.</td>
<td>85</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

7.7 A loop of cells causes a cycle in the corresponding AN. Edges of the cycle are marked by red colour. ........................................ 86
7.8 List of FTL components. ......................................................... 87
7.9 List of constants. ................................................................. 88
7.10 Algebra. ............................................................................. 88
7.11 Module interfaces using pipes. $P \in \mathbb{B}$ is the Present signal; $R \in \mathbb{B}$ is the Ready signal; $\vec{D} \in \mathbb{B}^n$ is the Data signal. ......................................................... 90
7.12 Direct composition of FTL modules cannot be done in RTL because of the loop between $s_1$ and $s_2$. ................................................................. 90
7.13 Flowchart of the MF-set algorithm. The main basic blocks are $B_1$ and $B_2$. ................................................................. 91
7.14 Mandelbrot set pipeline. Bracketed text along pipes annotates data types. ................................................................. 92
7.15 Integration of the generated MF-set pipeline (‘manfract’) in MF-ASVP ................................................................. 94
7.16 The FPGA area occupied by the MF-set compute core (depicted in Figure 7.15). ................................................................. 96

8.1 In SVP the threads are composed in a hierarchical tree. Threads are created in families. ................................................................. 102
8.2 A family of threads with its parent thread ................................................................. 103
8.3 The parameter ‘blocksize’ controls resource utilization: the number of physical replicas concurrently executing the thread code. ................................................................. 104
8.4 I-structure operation. The architectured 32-bit registers are extended with the state flags $(P, F, W)$. ................................................................. 105
8.5 An example of thread mapping in a homogeneous microgrid. ................................................................. 106
8.6 Thread mapping in a heterogeneous microgrid. ................................................................. 107
8.7 Flowchart of the create events in the HWFAM subsystem. ................................................................. 108
8.8 Classification of thread families in the HWFAM subsystem. ................................................................. 109
8.9 The flow of the create events between the CPU scheduler (left) and the Thread Mapping Table (TMT, right). ................................................................. 110
8.10 Typical performance scaling in CPU and accelerator (a cartoon only). ................................................................. 111
8.11 Functional model of the CPU–HWFAM system. ................................................................. 112
8.12 Accelerator interface follows a common model. ................................................................. 112
8.13 In accelerators the local memory blocks are segmented for continuous streaming operation. ................................................................. 113
8.14 Interleaving the DMA and computation using continuous streaming in the FIR accelerator. ................................................................. 113

9.1 Structural scheme of the combined CPU and HWFAM system. ................................................................. 116
9.2 The blocksize parameter specifies the maximal number of thread slots (T-SLOTS) allocated for a given family. ................................................................. 117
9.3 Execution in an accelerator (Acc) occupies a thread slot (T-SLOT) of the parent thread. ................................................................. 118
9.4 Benchmark: DCT, 1x. ................................................................. 119
9.5 Benchmark: DCT, 2x. ................................................................. 119
9.6 Benchmark: DCT, 4x. ................................................................. 119
9.7 Execution of two independent DCT kernels in UTOLEON3 and/or HWFAM. ................................................................. 121
9.8 Benchmark: FIR, 1x, larger instances (32x768). ................................................................. 122
9.9 Benchmark: FIR, 4x, larger instances (32x768). Two accelerators (‘Acc 1’, ‘Acc 2’). ................................................................. 122
9.10 FIR kernel execution time. ................................................................. 123
9.11 FIR kernel speed-up of the always-hardware and the hybrid schemes ................................................................. 124
9.12 FIR kernel speed-up with mixed $t$ but static $n_{accmin} = \{48, 40\}$. ................................................................. 125
9.13 Impact of the TMT latency scaling on the processor performance. 127
10.1 Platforms and the benchmarks used. 131
10.2 FIR filter implemented in FTL. 137
10.3 The DCT hardware developed by the manual approach and using the FTL differ slightly in the architecture. 139
10.4 Still images from a video demonstrating the execution speed of the microthreaded UTLEON3 vs. the scalar LEON3. 141
10.5 MF-set: Manually specified static pipeline implementation (HDL) in Xilinx System Generator. 142
10.6 MF-set: LEON3 and UTLEON3 processors, Instructions per Cycle (IPC). 144
10.7 Latencies in the UTLEON3 processor. 145
10.8 Latencies in the ASVP core. 145
10.9 Efficiency (IPC) of the non-pipelined long-latency operations. 147
10.10 Efficiency (IPC) of the pipelined long-latency operations. 147
10.11 FIR filter: LEON3 vs. UTLEON3, sensitivity of IPC to unpredictable latencies. 149
10.12 DCT: LEON3 vs. UTLEON3, sensitivity of IPC to unpredictable latencies. 150
10.13 MF-set: LEON3 vs. UTLEON3, sensitivity of IPC to unpredictable latencies. 150
11.1 Practical composition of the three methods (ASVP, FTL, HWFAM) as discussed in the thesis. 153
11.2 Author’s subjective estimation of the effort needed for using the platforms (horizontal axis). 156
B.1 Timing schemes in digital circuits. 173
B.2 Power breakdown in a high-performance CPU [TSR+98]. 174
B.3 Recursive H-tree pattern to distribute global clock signal uniformly and with equal delay over area. 174
B.4 The SELF protocol. [CCKT09] 175
B.6 A comparison of the instruction-driven (left) and data-driven (right) processors [Vee86]. 177
B.7 Example of the tagged-token dynamic dataflow machine. 178
B.8 VIRAM microarchitecture 181
B.9 CODE architecture 181
B.10 Vector-Thread architecture 182
B.11 VIPERS microarchitecture 182
C.1 Examples of correct and incorrect compositions of transition nodes. 184
C.2 An example of the wand multi-split transform. The wand $s_F$ is split in three wand-pairs for the three choice nodes. 185
C.3 Structure of a standard floating-point compute core. 186
C.4 Integration of a legacy pipelined core in FTL. 186
C.5 FIFO queue. 188
C.6 Expanded activation net of the FIFO queue. Data path in blue. 188
C.7 Representation of ‘alternate choices’ in flowcharts, FTL, and in hardware data path. 189
D.1 UML class diagram for construction of flow netlist. 192
D.2 UML class diagram for activation nets. 192
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.3</td>
<td>Graphical representation.</td>
<td>193</td>
</tr>
<tr>
<td>E.1</td>
<td>Structural diagram of the UTLEON3 processor.</td>
<td>200</td>
</tr>
<tr>
<td>E.2</td>
<td>A functional model of the UTLEON3 processor.</td>
<td>200</td>
</tr>
<tr>
<td>E.3</td>
<td>Latencies in the legacy LEON3 processor pipeline.</td>
<td>201</td>
</tr>
<tr>
<td>E.4</td>
<td>Latencies in the micro-threaded UTLEON3 processor pipeline.</td>
<td>201</td>
</tr>
<tr>
<td>F.1</td>
<td>The FIR algorithm in threads.</td>
<td>207</td>
</tr>
<tr>
<td>F.2</td>
<td>Data-flow in one thread of the ‘fir_inner’ family.</td>
<td>209</td>
</tr>
<tr>
<td>F.3</td>
<td>Execution of the FIR motivation example in HWFAM.</td>
<td>210</td>
</tr>
<tr>
<td>F.4</td>
<td>Internal structure of the FIR accelerator.</td>
<td>211</td>
</tr>
<tr>
<td>G.1</td>
<td>Work-flow of the experimental set-up.</td>
<td>213</td>
</tr>
<tr>
<td>G.2</td>
<td>Benchmark: FIR, 1x, larger instances (32x128). Two accelerators (‘Acc 1’, ‘Acc 2’). Config.: Listing G.2.</td>
<td>218</td>
</tr>
<tr>
<td>G.3</td>
<td>Benchmark: FIR, 1x, larger instances (32x512). Two accelerators (‘Acc 1’, ‘Acc 2’). Config.: Listing G.2.</td>
<td>219</td>
</tr>
<tr>
<td>G.4</td>
<td>Benchmark: FIR, 1x, larger instances (16x1024). Two accelerators (‘Acc 1’, ‘Acc 2’). Config.: Listing G.2.</td>
<td>219</td>
</tr>
<tr>
<td>G.5</td>
<td>Benchmark: FIR, 4x, larger instances (32x128). Two accelerators (‘Acc 1’, ‘Acc 2’). Config.: Listing G.2.</td>
<td>220</td>
</tr>
<tr>
<td>G.6</td>
<td>Benchmark: FIR, 4x, larger instances (32x512). Two accelerators (‘Acc 1’, ‘Acc 2’). Config.: Listing G.2.</td>
<td>220</td>
</tr>
<tr>
<td>G.7</td>
<td>Benchmark: FIR, 4x, larger instances (32x1024). Two accelerators (‘Acc 1’, ‘Acc 2’). Config.: Listing G.2.</td>
<td>221</td>
</tr>
</tbody>
</table>
List of Tables

2.1 MOS scaling rules, \( k > 1 \), from [DGR+74]. .................. 11
2.2 Scaling of the silicon technology in FPGA chips. Smaller feature sizes and lower voltages are better. Source: Xilinx and Altera FPGA data-sheets. .................. 12
2.3 Critical paths data in LEON3 processor implemented in FPGAs. .................. 15
4.1 The primary C API functions used to access (program) the vector unit from the sCPU. 29
4.2 Some of the vector operations implemented in DFU ........................ 32
4.3 The frequency \( f_{VPU} [MHz] \) depends on the implementation technology and the pipeline depth in compute units. \( Ax = \) FP-Adder latency, \( Mx = \) FP-Multiplier latency. 33
4.4 Utilization of FPGA slices. .......................... 34
5.1 PicoBlaze 3 maximum clock frequency according to Xilinx [Xii11]. .......... 48
5.2 The fields encoded in the new configuration table. ........................ 53
5.3 The C API functions for accessing the vector unit. ........................ 53
6.1 State space of the activation net of one place. .......................... 67
6.2 Evolution of the four-place pipeline over 8 clock ticks (\( t_1 \) to \( t_8 \)). At the time \( t_1 \) all places are initialized with a token. .................. 68
6.3 Function table of the \textit{choice} AN fragment. Symbol ‘×’ is a don’t-care value. ...... 71
6.4 All possible markings (states) of the PN in Figure 6.12. .................. 73
7.1 State space of the activation net of one \textit{cell}. (cf. Table 6.1 for \textit{place}) .......... 84
7.2 Evolution of the four-\textit{cell} pipeline over 5 clock ticks (\( t_1 \) to \( t_5 \)). At the time \( t_1 \) all cells are initialized with a token. .................. 85
7.3 Maximal operating frequency in MHz of the data-path as a function of pipeline depth in FP cores. .......................... 95
7.4 Finishing times of the MF-set algorithm in GP-ASVP and in the FTL-net based MF-ASVP. .......................... 98
8.1 The fields of the content addressable memory (CAM) in the Thread Mapping Table (TMT) module. .......................... 110
9.1 HWFAM synthesis results in \textit{Xilinx Virtex 5} FPGA (XC5VLX110T). .......... 126
10.1 The benchmarks and platforms used for the comparison. .................. 132
10.2 Platform characterization data used for the evaluation. .................. 132
10.3 Resource utilization of the LEON3 and UTLEON3 processors in the Virtex 5 FPGA and in TSMC 90nm technology. .................. 133
10.4 Lines-of-Code (LOC) written by the author for implementing the algorithms in the platforms. .................. 134
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.5</td>
<td>FIR filter: Area utilization $a$ in slices of the FTL-net version.</td>
<td>136</td>
</tr>
<tr>
<td>10.6</td>
<td>FIR filter: GP-ASVP and FTL-net.</td>
<td>136</td>
</tr>
<tr>
<td>10.7</td>
<td>FIR filter: LEON3 vs. UTLEON3 processors.</td>
<td>136</td>
</tr>
<tr>
<td>10.8</td>
<td>FIR filter: Manual HDL and FTL-net.</td>
<td>137</td>
</tr>
<tr>
<td>10.9</td>
<td>DCT: Manual static pipelining vs. FTL-net, used resources.</td>
<td>140</td>
</tr>
<tr>
<td>10.10</td>
<td>DCT: Manual static pipelining vs. FTL-net, finishing times.</td>
<td>140</td>
</tr>
<tr>
<td>10.11</td>
<td>DCT: Finishing times in LEON3 vs. UTLEON3.</td>
<td>140</td>
</tr>
<tr>
<td>10.12</td>
<td>MF-set: Resource utilization.</td>
<td>143</td>
</tr>
<tr>
<td>10.13</td>
<td>MF-set: All platforms, finishing times.</td>
<td>143</td>
</tr>
<tr>
<td>10.14</td>
<td>MF-set: LEON3 and UTLEON3 processors, finishing times.</td>
<td>144</td>
</tr>
<tr>
<td>10.15</td>
<td>MF-set: Degradation of the execution time when the access latency to local memory is increased.</td>
<td>151</td>
</tr>
<tr>
<td>B.1</td>
<td>Summary of advantages of asynchronous circuits [JS02] and GALS.</td>
<td>174</td>
</tr>
<tr>
<td>C.1</td>
<td>The meaning of interface signals in a standard FP core.</td>
<td>186</td>
</tr>
<tr>
<td>F.1</td>
<td>Software FIR parameters.</td>
<td>211</td>
</tr>
<tr>
<td>F.2</td>
<td>Configurable parameters in the FIR accelerator core.</td>
<td>211</td>
</tr>
</tbody>
</table>
Abbreviations

Common Mathematical Functions and Operators

◦ An operator, such as addition, multiplication etc.
Ψ◦ Full reduction using operator ◦
ε◦ Neutral element with respect to operator ◦
∀ For all elements/values...
Ξ◦ Prefix reduction using operator ◦
≈ Approximately
O(x) The big O notation
## LIST OF TABLES

### Miscellaneous Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADL</td>
<td>Architecture Description Languages</td>
</tr>
<tr>
<td>AN</td>
<td>Activation net</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>ASIP</td>
<td>Application Specific Instruction Processor</td>
</tr>
<tr>
<td>ASVP</td>
<td>Application Specific Vector Processor</td>
</tr>
<tr>
<td>BCE</td>
<td>Basic Computing Element</td>
</tr>
<tr>
<td>BLAS</td>
<td>Basic Linear Algebra Subprograms package</td>
</tr>
<tr>
<td>CAD</td>
<td>Computer Aided Design</td>
</tr>
<tr>
<td>CAM</td>
<td>Content-Addressable Memory</td>
</tr>
<tr>
<td>DLP</td>
<td>Data-Level Parallelism</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>EB</td>
<td>Elastic Buffer (in SELF protocol)</td>
</tr>
<tr>
<td>EMI</td>
<td>Electromagnetic Interference</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response filter</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Arrays</td>
</tr>
<tr>
<td>FP</td>
<td>Floating Point (number system)</td>
</tr>
<tr>
<td>FPU</td>
<td>Floating Point Unit</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General-purpose computing on graphics processing units</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HWFAM</td>
<td>Hardware Families of Threads</td>
</tr>
<tr>
<td>IPC</td>
<td>Instructions per Cycle</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction-Level Parallelism</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>JCU</td>
<td>Job Control Unit</td>
</tr>
<tr>
<td>LHS</td>
<td>left-hand side</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-Up Table</td>
</tr>
<tr>
<td>MAC</td>
<td>Multiply-Accumulate Unit</td>
</tr>
<tr>
<td>MIPS</td>
<td>Million Instruction Per Second</td>
</tr>
<tr>
<td>MFLOPS</td>
<td>Million Floating-Point Operations Per Second</td>
</tr>
<tr>
<td>NoC</td>
<td>Network on Chip</td>
</tr>
<tr>
<td>PN</td>
<td>Petri net</td>
</tr>
<tr>
<td>RHS</td>
<td>right-hand side</td>
</tr>
<tr>
<td>RTL</td>
<td>Register-Transfer Level</td>
</tr>
<tr>
<td>SELF</td>
<td>Synchronous Elastic Flow protocol</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SIP-net</td>
<td>Synchronous Interpreted Petri net</td>
</tr>
<tr>
<td>SoC</td>
<td>System on Chip</td>
</tr>
<tr>
<td>SVP</td>
<td>Self-adaptive Virtual Processor</td>
</tr>
<tr>
<td>TLP</td>
<td>Thread-Level Parallelism (or Task-Level in single-thread systems)</td>
</tr>
<tr>
<td>TMT</td>
<td>Thread-Mapping Table</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The roots of all evil are the statically-unpredictable latencies.

1.1 Reasons for Data-Driven Scheduling

1.1.1 Latencies and Scheduling

There are two basic metrics for measuring systems’ performance: latency and throughput. Latency (response time), measured in seconds, is the time it takes for a system or a module to process a given request and generate the result. Throughput, measured in requests per second, is the number of requests or results that a system or a module is able to take in and generate per unit of time in steady state.

Latencies in modules impact the total latency and throughput of systems. Of course the best situation happens when individual latencies are negligibly short compared to the required total latency of the system. The latency of the system will be short and the throughput high, as shown in Figure 1.1.(a).

Non-zero latencies are either statically predictable or statically unpredictable, depending if the length of the latency is fully known in advance during system construction. Statically predictable

![Diagram showing latency and throughput](image-url)

**Figure 1.1:** Statically predictable vs. unpredictable latencies.
latencies cause relatively small trouble because the system can be organised such that these latencies are ‘hidden’ (overlapped) and the total throughput will not suffer, although the system response time is affected (Figure 1.1.(b)). Statically unpredictable latencies cause more trouble. Using only pre-computed (static) scheduling the system throughput will be less than optimal due to unnecessary stalls (Figure 1.1.(c)). Run-time (dynamic) scheduling of operations based on real latencies is needed to efficiently utilize the resources.

1.1.2 Data-Driven vs. Instruction-Driven Machines

The execution (actual schedule) of operations in a machine may be driven by a flow of instructions or flow of data. Traditional von Neumann computer architectures are driven by instructions because operations are started in response to the arrival of explicit instructions. In data-driven (or dataflow) architectures, on the other hand, operations are started in response to the availability of input data.

Dynamic scheduling is needed when execution latencies of operations are unpredictable. In computers the usual example is accessing DRAM chips. In DRAMs the practical unpredictability of read/write access latencies stems from periodical refreshing of memory capacitors that blocks the memory for a moment, and from the state-full memory interface (access to a closed row takes longer than to an open one, reversing access direction takes time, etc.).

On the other hand, if all latencies are statically known in advance during compile time (or system construction) then the compiler could statically reorder instructions and construct a perfect schedule (i.e. not worse than the best dataflow). This holds for arbitrary long latencies as long as they can be determined in advance.

The advantage of the instruction-driven paradigm is a small size of machine state information (or context) that must be stored in the machine to keep the computation going. The context is a ‘snapshot’ of the state in a quiescent moment after an operation has just completed and another one is about to begin; it is machine’s architectural state. In a pure instruction-driven processor the context is defined by the content of memory (including architected registers) and the content of the instruction pointer register.

In the data-driven machines the architectural state is typically larger and less centralized than in the instruction-driven systems. The difference is the amount of information that must be kept around for the correct selection of the next operation to be executed. In the instruction-driven processor an \( n \)-bit instruction pointer selects unambiguously one of the \( 2^n \) instructions in the program memory. In the data-driven processor an \( n \)-bit ‘set of enabled arcs’ register can represent the state of only \( n \) arcs in the program graph, if the set is represented by a bit mask vector. (Dataflow computers are discussed in Section B.4.)

Consequently, when operation latencies are statically predictable during the system design (or program compilation) the instruction-driven paradigm is the better choice as hardware is typically simpler. Small perturbations in execution latencies might be tolerated using statistical approaches such as processor caches. Maintaining the execution efficiency under statically unknown latencies requires data-driven dynamic scheduling. Figure 1.2 depicts a decision diagram that, starting from the type of latencies encountered in a system, suggests either instruction-driven or data-driven machine organization.

1.1.3 Scheduling Granularity

In data-flow systems the run-time construction of dynamic schedule costs additional resources that could otherwise be devoted to useful computations. This overhead can be decreased by performing the
1.2. CONVERGENCE OF PROCEDURAL AND STRUCTURAL PROGRAMMING

The classification of computing machines as instruction-driven or data-driven is one possible point of view. An orthogonal classification is whether the system is programmed procedurally in time or structurally in space.

Figure 1.3 shows a matrix that puts the instruction- and data-driven machines in relation with programming in time and space. Classical von Neumann computers are instruction-driven and programmed in time. Dataflow computers with stored program are data-driven and also programmed in time. Single-core out-of-order execution processors are hybrids programmed in time; the processor ISA (and the pipeline front-end) are based on the instruction-driven paradigm, the pipeline back-end and the execution core is data-driven. Many-core in-order execution processors are instruction-driven, with the fine-grain programming in individual cores done in time and the coarse-grain application mapping done in space. Reconfigurable hardware is programmed in space. The application designer can choose if the computation is instruction-driven or data-driven by coding appropriate hardware structures from primitives provided in the architecture.
CHAPTER 1. INTRODUCTION

In the past when frequency scaling in new silicon generations was still a norm the processor hardware was built to maximise sequential performance. Single-core superscalar processors were programmed procedurally in time. When frequency scaling had ended and many-core scaling had begun, the old sequential software programming models were no longer viable for new systems with more than a couple of processor cores. New programming models had to be invented to efficiently program many-core processor hardware with complex structure. One such new model is the Self-adaptive Virtual Processor (SVP, [Jes04, Jes06, Jes08, kP12]). Besides efficient utilization of many-cores the SVP model must also overcome unpredictable latencies in off-chip memories (DRAMs) and in on-chip shared interconnect. Hence, the SVP model is data-driven (dynamically scheduled) and it contains some aspects of the structural programming in space (e.g. delegation of threads to places).

Previous work in SVP envisioned large-scale homogeneous many-core chips because it assumed that low clock-frequency silicon is easily scalable in space [JLZ09]. However, the contemporary and future power constraints will favour heterogeneous many-cores in which parts of the chip could be powered off if the given special function is not used at the moment [EBS11]. The thermal design power of a chip could be so low that not all cores may be powered up simultaneously.

Specialized hardware can deliver orders of magnitude higher performance. Computing in reconfigurable arrays allows to tap the performance of specialized solutions without the cost of manufacturing a single-purpose silicon hardware. A typical approach is to combine general-purpose many-core processor cluster with special-purpose cores to accelerate selected key applications. In this work the use of (embedded) reconfigurable arrays in place of (some of) the special-purpose cores is postulated. The resulting system is programmed procedurally in individual general-purpose cores, and structurally among the general-purpose cores and in the reconfigurable arrays. The system is data-driven to overcome unpredictable latencies in off-chip memories, on-chip shared interconnect, and in interconnect in the reconfigurable arrays.

1.3 Objectives and Contributions

The coverage of this thesis is depicted in Figure 1.4 as a ‘map’ that puts thesis contributions in relations. In the centre there are six main topics that determine the research field: heterogeneity, programming in space, multithreading, customized hardware, reconfigurable arrays, and the ease of programming.

A1. Statically scheduled instruction-driven architecture for custom computing on FPGA, called ASVP and described in [DBK08].
1.3. OBJECTIVES AND CONTRIBUTIONS

A2. Dynamically scheduled data-driven architecture for general-purpose computing using microthreading, described in [Jes04, KP12] and implemented in [X.12].

The main objectives of the thesis are:

O1. To design and develop a new implementation of the architecture A1, to characterise its hardware and software performance. Based on the characterization new features in the architecture A1 are proposed, implemented and evaluated that improve execution efficiency.

O2. To propose and develop a method for dataflow-oriented specification of problem-dependent (specialized) compute pipelines. Using a newly implemented software tool the method is tested by generating specialized compute units that can be used in the architecture A1.

O3. To propose and develop a method for integrating simple statically-scheduled architectures such as A1 into the advanced multi-core data-driven processor architecture A2.

O4. To compare the performance of the architectures A1 (ASVP), A2 (microthreading) with the method developed in O2 and with usual industry-standard approaches (manually coded HDL, scalar processor).

There are five main contributions, labelled (1)–(5) in the picture:

(1) Analysis of a statically scheduled instruction-driven vector processing architecture for customized computing realized in a reconfigurable array.

The particular architecture A1 was disclosed in previous work [DKBK08]. The architecture is
CHAPTER 1. INTRODUCTION

characterised here separately in hardware implementation and in application mapping. It is shown here that (contrary to previous design expectations) a high-frequency issue of vector instructions is needed in spite of using batched processing in the data path.

(2) A method for achieving high-frequency instruction issue using dictionary tables in an architecture with wide horizontal instructions generated on the fly.
A quantitative analysis of the structure of vector instructions and of the issue frequencies are performed. A technique is proposed that reduces overheads in the instruction issue by storing frequently used combinations in a dictionary table.

(3) Structured and extensible approach for synthesis of hardware controllers from synchronous Petri nets.
The concurrency in hardware controllers is limited by a decomposition step that partitions the functional specification into cooperating sequential automatons. Controller specification using Petri nets has the advantage that decomposition for hardware synthesis is not needed.

(4) A new technique for dataflow hardware synthesis from Petri nets.
The idea is that traditional HDLs can describe logic structure (gates) of hardware but neglect the importance of interconnect delays. The new technique based on SPN synthesis (3) gets around the problem by decoupling functionality from timing, hence hardware specifications are ‘elastic’ and adaptable to interconnect latencies in a changing implementation technology.

(5) Bridging the gap between the data-driven microthreaded procedural computation (arch. A2) with the special-purpose data-driven hardware in reconfigurable arrays (arch. A1).
A transparent software/hardware interface that couples a general-purpose microthreaded model and custom accelerators is demonstrated. Concurrency issues are handled in hardware using the SVP model.

1.4 State of the Art

In practice the usability of any reconfigurable architecture is critically connected with the existence of proper design entry tools [Guo06]. In FPGAs, the most prevalent type of fine-grained reconfigurable arrays, the VHDL and Verilog languages are the most often used because the knowledge of these HDLs is transferable to/from ASIC design. Synthesis from higher-level languages, usually C, was attempted many times, with varying success [VPNH10, RFB09, BHLS10, OBDA11, PGS+09, PLS+11]. An overview of reconfigurable computing is in Section B.5.

The first part of this thesis builds on the previous work [DKBK08]. The motivation was twofold: (a) to bridge the gap between algorithm development in higher language and the hardware implementation by using a customized architectural template; (b) to demonstrate fast runtime reconfiguration in FPGA by supporting multiple program contexts. This thesis focuses on the former.

In [Bom02] it is argued in favour of using microprogrammed controllers in FPGAs for the design of complex finite state machines. In [NBK+13] a microprogrammed controller is put to practical use in a custom accelerator in FPGA for a numerical application using the conjugate gradient method. Both these works proposed a tailor made microprogrammed controller, customized for their applications. The disadvantage is a need to deal with custom microassembly language, which only shifts the burden of the low-level programming from hardware to software. In [DKBK08] the use of an industry-standard microcontroller was postulated, taking advantage of existing software toolchain. Here it is demonstrated that the raw performance of the standardized microcontroller is insufficient,
1.4. STATE OF THE ART

Inelasticity; b) asynchronous elasticity; c) Synchronous elasticity. Idle periods denoted by shaded boxes. [CCKT09]

but the controller can be augmented to support the statically-scheduled hardware without sacrificing legacy software.

Gate-level dataflow techniques used to be almost a synonym with the asynchronous-clock design. Asynchronous circuits potentially promise [JS02] low power consumption, less electromagnetic interference (EMI) noise, better composability and modularity, and other—see Table B.1. However, the lack of mature CAD tools, especially for timing analysis and testing, is often cited as the main drawback of the technology for practical use. Another drawback is increased die area because of the more complicated control circuitry. Tangram [vBKR+91] and BALSA [BE00] are CAD system for syntax-directed specification and generation of VLSI asynchronous circuits. More details can be found in Section B.2.

Due to the practical reasons pure clock-synchronous systems are likely to stay the mainstream implementation method in the future. It was proposed, therefore, to use the asynchronous and event-based approaches only for specification of circuits’ behaviour. The implementation of such circuits would be done using standard clock-synchronous technology. This leads to the class of elastic circuits.

Elastic circuits decouple function from the timing of the circuit. The timing amounts to physical clock-cycle time (i.e. the operating frequency in clock-synchronous systems) determined by the critical path, and also to the logical schedule of operations in the circuit. Asynchronous circuits are naturally elastic, while synchronous circuits are typically inelastic. Different types of elasticity are illustrated by examples in Figure 1.5, ranging from inelastic (a), asynchronous elasticity (b), to synchronous elasticity (c). This thesis focuses on the synchronous elasticity.

Elasticity in circuits [CCKT09, HB08] provides tolerance to variations in computation and communication latencies. For example, an elastic adder can have data-dependent latency: it could take one clock-tick for short operands and two ticks for operands requiring long carry-chain propagation. Elasticity can be also exploited for tool-driven architectural optimizations [GOCBK10]. The disadvantage of the previous work is the use of special CMOS technology cells called ‘Elastic Buffers’. The SELF protocol is discussed in detail in Section B.3.

In dataflow systems computation is triggered by the arrival of data. In dynamic dataflow systems the dependency graph evolves in time, possibly depending on actual data values. Example: a for-loop that is dynamically unwound in a processor. In static dataflow the graph is predetermined. In synchronous dataflow (SDF) [LM87] the number of data tokens consumed and produced by each node on each invocation is specified a-priori. A multiplexer is not an SDF component because token rates on its inputs are not constant. The function of a dataflow computer is described in Section B.4.

In Kahn process networks [Kah74] (KPN) each actor (box) runs a sequential code, connections between the actors are statically known and modelled by unbounded FIFO queues. Firing rules in actors
are similar as in Petri nets. Dataflow Process Networks [LP95] (DPN) are special case of KPN that can be directly mapped to hardware because each firing rule can be executed in a cycle. Writes to the FIFO queues are non-blocking, reads are blocking. C~ (C-flow) [WSC+13] is a C-based higher-level hardware language based on the DPN.

The static dataflow model is extensively used in the digital signal processing (DSP) domain to specify the behaviour of both the software and hardware components. StreamIt [TKA01] is a domain-specific language and a compiler for stream-based specification of DSP programs for CPUs and GPUs. CAPH [SBA11] is a domain-specific language for stream applications on FPGA. These systems model each channel as a FIFO queue, and employ domain-specific knowledge in the compiler to optimize the granularity of tasks.

Term rewriting systems (TRS) can be used to describe the operational semantics of hardware circuits (TRAC [HA99], Bluespec [DNA05]) and also for architectural exploration [MR01]. By increasing the abstraction level the designs are easier to comprehend and maintain. A TRS consists of a set of terms and a set of rewriting rules. A rule consists of a pattern, an optional predicate, and an action (a rewrite) that is carried out on the system state if the rule fires. The effects of actions are atomic; several terms may fire simultaneously only if their actions do not conflict. Term rewriting systems are defined as non-deterministic. During hardware synthesis the compiler extracts parallelism and creates a deterministic schedule of actions.

Pipelining techniques are often studied in the context of micro-architectural research [GOCBK10, MR01, NHKLch]. In [NHKLch] the authors describe T-piper, a tool that can synthesise processor pipeline out of a transactional (single-cycle) specification. The tool allows to decouple functional specification and the assignment of modules to stages. It can automatically synthesise interlocks, forwarding paths and speculative execution logic.

Communicating Sequential Processes (CSP) [Hoa04] describe an algebra of processes that engage only in events. Although parallelism in CSP is ubiquitous, it is defined in terms of interleaving of processes; therefore, events in CSP occur in an ordered sequence (as a hypothetical sequential trace). This is cumbersome when the CSP paradigm is used to specify truly concurrent systems, such as the clock-synchronous hardware, where multiple events may happen in the same clock cycle. In Handel-C [Men] the issue is resolved by assigning priorities to channels that are concurrently waited-on [BW02, Law03].

Petri nets (PN) are a mathematical model for the description of concurrent distributed systems. Using PN models for synthesis of hardware controllers is motivated by the desire to increase the concurrency in systems without complexity in specification.

In [CKP96] the authors present an FPGA implementation of synchronous Petri nets. The authors designed and implemented an FPGA macro called cell using 12 CLBs in Xilinx 3000 FPGA. Each cell implements $k$ places and a common transition. A controller specified by the Petri net is transformed into a radix-$k$ Petri net in which each transition has at most $k$ source places. This Petri net is partitioned and mapped to an array of cells. The resulting schematic diagram is implemented in FPGA using vendor tools.

In [FAP97] the authors convert hierarchical synchronous Petri nets into synthesisable VHDL code. Logic expressions including input signals are assigned to transitions (guards). Output signals are assigned to places and transitions, to represent controller actions. All transition firings are synchronized to global clock-ticks. The presented method does not handle the conflict and overflow situations (Figure 6.7); the source Petri net must be free of them, otherwise the generated hardware will not function correctly.

In [SBK+04] the authors present synthesis of asynchronous hardware systems from VHDL high-level specification. The input is a subset of VHDL; the high-level constructs such as ‘if’ conditions
and ‘while’ loops are supported. This specification is refined into Petri nets. Coloured Petri nets (CPN) are used for data path and labelled Petri nets (LPN) for controller. The various PNs are mapped to asynchronous circuit library elements.

In [Kub04, Kub05] the authors present a direct implementation of Petri net based models in FPGA. The input is a model specified in Petri Net Markup Language (PNML). PN places are mapped to counters that count the number of tokens in a place. Transitions are mapped to AND gates. Selection from multiple enabled transitions is random by LFSR.

In [ASG08] the authors present a method of translating standard asynchronous Petri nets in FPGAs for rapid prototyping of complex digital systems. The approach allows places with multiple tokens (non-binary PNs) and timed transitions. They generate structural VHDL code of components representing places and transitions.

In [SAGP12] yet another approach of direct translation of Petri net models to synchronous digital systems is described. Places are represented by counters, and transitions by AND-gates.

Dataflow networks of processes can be implemented by software threads. An example is the S-Net technology [GSS08, X.5]: a coordination language and a runtime system for creating and scheduling streaming networks of actors. The coupling of multithreaded execution in processors with a concurrent execution in custom and reconfigurable hardware introduces the problem of sharing the hardware accelerators between multiple threads [ZKG09]. This requires synchronization between potentially unrelated threads, and/or a context-switch mechanism in the accelerator. In RCC [UMKU06] (the multithreaded extension of the Molen [VWG+04] protocol), only one thread at a time is allowed to execute in the reconfigurable unit. In Chimaera [HFK97] the RFU is always stateless (i.e. it is purely combinatorial) so there is not any context to be saved on a thread switch. The Garp [HW97] RFU architecture provides special instructions for context save and restore on a thread switch, to be used by the operating system.

The use of accelerators in multi-threading or multi-core environments is complicated by the need for locking and synchronization between otherwise unrelated threads, as they compete for hardware resources. In [IS11] an all-software framework for management of hardware accelerators in traditional multi-threaded (single-core) environment was proposed. They use modified pthreads library in embedded Linux (PetaLinux, MicroBlaze). The decision if a given thread should be run in software or in hardware is according to the thread function address. However, in pthreads library threads are created individually, not in groups (families), hence the FUSE framework is not aware of the amount of work the creation events represent. The FUSE scheme is not suitable in situations when both smaller and larger datasets are processed by the same thread functions.

Application software usually works in virtual memory address space while typical hardware sees only physical bus addresses. In [VPI05] the authors propose an integration technique for portable hardware accelerators. It includes a unit for virtual-to-physical address translation for hardware accelerators.

Hthreads [PAA+06] provide a standardized hardware interface between an operating system in the host processor and accelerators in FPGA. Hardware accelerators are recognized in the OS as processes and they may communicate using file IO and standard locks (mutexes).

1.5 Organization

In Chapter 2 the technological aspects of silicon scaling are discussed. The main body of the thesis is organized in three parts: Part I (Chapters 3–5) discuss statically scheduled vector processor with dataflow at the task level. The implementation platform is an extensible and customizable application-
specific vector processor (ASVP) in FPGA. This Part presents the first two thesis contributions. Part II (Chapters 6–7) focuses on fine-grained dataflow at the hardware gate level using Petri nets. This Part presents the third and fourth thesis contributions. Part III (Chapters 8–9) focuses on chip-level dataflow. Interfacing issues between a highly parallel microthreaded multicore-capable processor and specialized function hardware are analysed. This Part presents the fifth thesis contribution.
Chapter 2

Technology Aspects

This chapter reviews some of the issues of the contemporary silicon implementation technology. The gate and wire scaling is discussed and two examples (simple experiments) illustrating the significance of interconnect delays are given.

The well-cited Moore’s law states that you can pack double the number of transistors in the same area every $x$ months. In the beginning in 1965 the parameter $x$ was 12 months. Later it was recalibrated to 24 months in 1975. Modern day physicists and engineers quarrel whether the trend could continue in the future given that the silicon implementation technology will reach atomic dimensions. It is likely that the Moore’s law will just slow down [Iwa09] and $x$ will increase. In the past a circuit architect or designer could count on upgrading to a new silicon process technology every $x$ months and he/she would obtain a predictable improvement in the circuit delay (clock speed), area and power consumption. But as the $x$ increases the need for architecture-level enhancements becomes more acute because each technology generation stays longer.

2.1 Gate Scaling: Power Constraints

The MOSFET scaling rules, first described by Dennard in 1973 [DGR+74], are the primary enabling factor of the Moore’s law. Table 2.1 lists the scaling rules. When a baseline technology is scaled by a factor $1/k$ in both linear dimensions $(x, y)$, the voltage, capacitance and delay are reduced by the factor $1/k$, and the area and power (for the same circuit) reduced by the factor $1/k^2$.

Product designers will hardly put old circuits in new chips. They will typically use the same or larger die size to put in more transistors to implement new algorithms; hence, the total IC power will not decrease with $1/k^2$. Voltage $U$ typically does not scale down perfectly from one technology.

<table>
<thead>
<tr>
<th>Device/Circuit Parameter</th>
<th>Scaling Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device dimensions $L_C, W_C$</td>
<td>$1/k$</td>
</tr>
<tr>
<td>Doping concentration</td>
<td>$k$</td>
</tr>
<tr>
<td>Voltage $U$</td>
<td>$1/k$</td>
</tr>
<tr>
<td>Field</td>
<td>$1$</td>
</tr>
<tr>
<td>Current $I$</td>
<td>$1/k$</td>
</tr>
<tr>
<td>Gate delay</td>
<td>$1/k$</td>
</tr>
<tr>
<td>Power dissipation/device</td>
<td>$1/k^2$</td>
</tr>
<tr>
<td>Power density</td>
<td>$1$</td>
</tr>
<tr>
<td>Speed power product/device</td>
<td>$1/k^3$</td>
</tr>
</tbody>
</table>

Table 2.1: MOS scaling rules, $k > 1$, from [DGR+74].
node to the next because of other problems (transistor threshold voltage). Frequency $f$ was scaled up more aggressively in the past than warranted by Dennard’s rules. The dynamic power is proportional to $U^2$ and $f$, hence the total power consumption of chips increases over time, particularly in the microprocessor segment.

Limits of the maximal power dissipation of IC packages were reached shortly after the year 2000. Due to the power constraints it was no longer feasible to use very high clock frequencies $f$ to make processors go faster. To make single-core processor perform faster the only remaining option was to redesign it with a more clever and complex microarchitecture. However, this turned out to be a very difficult task due to the design and verification costs.

Many new applications are parallel (multithreaded). When the silicon technology is scaled by $1/k$ the existing and verified chip architecture can be replicated $k^2$ times to fill up the same die area. The resulting chip contains multiple identical cores, each operating in parallel to others. This overcomes the need for complete redesigning of chips from scratch.

How many processor cores is it feasible to fit on a die? There are two constraints: parallelism and power. The maximal achievable speed-up with multiple processor cores is effectively limited by the parallel fraction in Amdahl’s law. For a given application the performance gain diminishes with more processing cores. But even if new highly-parallel algorithms (or compiler transformations) are discovered that could utilize many-core architectures the chips themselves are becoming power-limited.

The study [EBSA+11] projects scaling of PARSEC benchmark applications over 5 technology generations from 45 nm to 8 nm. Although the ideal speed-up in the scenario is $32 \times (32 = 2^5)$ the study shows that due to IC power constraints (which limit the number of cores) the maximal speed-up is only $15 \times$. Regardless of chip organization and topology, multicore scaling is power limited. Even

### Table 2.2: Scaling of the silicon technology in FPGA chips. Smaller feature sizes and lower voltages are better. Source: Xilinx and Altera FPGA data-sheets.

<table>
<thead>
<tr>
<th>Width (nm)</th>
<th>0.85V</th>
<th>0.9V</th>
<th>1V</th>
<th>1.1V</th>
<th>1.2V</th>
<th>1.5V</th>
<th>2.5V</th>
</tr>
</thead>
<tbody>
<tr>
<td>220nm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>180nm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>130nm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120nm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90nm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65nm</td>
<td>A-S3</td>
<td></td>
<td></td>
<td></td>
<td>A-C3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60nm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A-C4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45nm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X-S6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40nm</td>
<td>A-A2, A-S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Xilinx devices** (‘X’-): X-S{3E,3A,6} = Xilinx Spartan {3E,3A,6}. X-V{1, 2, 2Pro, 4, 5, 6, 7} = Xilinx Virtex 1–7. X-K7 = Xilinx Kintex 7. X-A7 = Xilinx Artix 7.

**Altera devices** (‘A’-): A-C{1, 2, 3, 4, 5} = Altera Cyclone 1–5. A-A{1, 2, 5} = Altera Aria 1–5. A-S{1, 2, 3, 4, 5} = Altera Stratix 1–5.
at 22 nm, 21% of a fixed-size chip must be powered off, and at 8 nm, this number grows to more than 50%.

Consequently, contemporary and future hardware is special-purpose and customized due to the power issues. Heterogeneous many-cores are already a norm. A good example is the ARM big.LITTLE many-core system that (for instance) combines a quad-core cluster of ‘big’ Cortex-A15 processors and another quad-core cluster of ‘little’ Cortex-A7 processors. The big cores are more powerful but also more energy demanding. Only one of the two clusters is active at any time, the other is powered off. The system has lower energy consumption and lower thermal power envelope.

Computing on graphics cards (GPGPU) is an example of using specialized programmable hardware to efficiently execute data-parallel algorithms. Individual GPU chips are not complete systems and require a general-purpose host processor for task preparation and control. Therefore the whole system is heterogeneous. Since 2010 the GPUs are increasingly being integrated with x86 CPUs in hybrid chips, sometimes called Accelerated Processing Unit (APU).

Xilinx Zynq-7000 is marketed as an ‘all programmable SoC’. The chip contains dual-core ARM Cortex-A9 processor combined with the FPGA reconfigurable array.

In this thesis FPGAs (Field-Programmable Gate Arrays) are used exclusively as an implementation technology. FPGAs are versatile devices used in many applications. The chips are therefore produced in high volumes, bringing the unit cost down. The FPGA has a regular structure: it is an array of look-up tables (LUT), flip-flop registers, interconnect wires, switch boxes, local memories, and other special elements. The regular organization of the chip makes for an easier scaling of the FPGA silicon to new process technologies. Table 2.2 lists FPGA devices from Altera and Xilinx according to the silicon technology used: the table is organized by the feature size in nanometres and the supply voltage.

### 2.2 Wire Scaling: Interconnect Routing Delays

Wires connect transistors in gates to make an electrical circuit. The shortest are local wires that connect transistors within gates. They are situated in the lowest metal layers of the chip. Semiglobal wires on midlevel layers of metal typically connect functional units within a block. Global wires on the top layers of metal route power, ground, and global signals.

When a design is scaled the delays in local wires track delays of gates, or grow slowly [HHM99]. Global wires do not scale since they communicate signals across the chip as shown in Figure 2.1. The average length of a global wire is $W/2$, where $W$ is the chip width. In contemporary deep sub-micron CMOS technologies signal delays on interconnect wires amount to most of the critical path latency (Figure 2.2) [Boh95].

Interconnect wires could be shared among modules (e.g. buses), or dedicated for a point-to-point connections. Shared interconnect could be scheduled statically if all communication patterns are predefined, or dynamically. In dynamically scheduled shared interconnect the exact communication latencies are statically unknown.
CHAPTER 2. TECHNOLOGY ASPECTS

2.2.1 Latency-Tolerant Structured Interconnect Instead of Global Wires

Using structured *networks on chip* (NoC) instead of dedicated global wires was proposed to mitigate the difficulty of achieving the timing closure of the design [DT01]. While dedicated global wires are more flexible, the practical problem is to electrically characterize the wiring. Statistical models of wires used by synthesis tools are conservative and use large CMOS gates to achieve good noise immunity. The models tend to oversize most of the drivers but undersize enough of them to make achieving the timing closure a difficult task. In contrast, on-chip interconnection network structures are precisely defined at the beginning of the design process. The network forms a top-level wiring of the IC, and partitions the design into tiles. The long wires used by the NoC can be isolated from intra-tile wiring, and optimized for signal strength and integrity. In fine-grain reconfigurable arrays the use of networks-on-chip instead of long routes is an area of active research [RFD12].

2.2.2 Interconnect Delays in Reconfigurable Arrays

In reconfigurable arrays the fraction of the critical path spent on delays in the interconnect is considerable and it is getting higher with CMOS technology generations. In FPGAs, the most widely used types of reconfigurable arrays, the placement and routing processes are automated in CAD tools supplied by device vendors. In modern million-LUT devices these processes are increasingly more difficult. Designers are encouraged to floor-plan the FPGA manually to help the placement process.

Example 1: LEON3 in FPGA

The significance of interconnect routing delays on critical path will be demonstrated by the following experiment in Xilinx and Altera FPGAs. As a representative design I selected the 32-bit LEON3 soft-core processor from Aeroflex Gaisler. The processor is available as an open-source code in synthesizable VHDL. The vendor supplies complete demonstrator designs for many development boards with different FPGAs. Selected designs were synthesized by Xilinx ISE 14.4 and Altera Quartus 13.0.


### 2.2. WIRE SCALING: INTERCONNECT ROUTING DELAYS

<table>
<thead>
<tr>
<th>FPGA</th>
<th>Logic levels</th>
<th>Clock [ns]</th>
<th>Logic [ns, %]</th>
<th>Route [ns, %]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclone 1 (130nm)</td>
<td>25</td>
<td>40.4ns</td>
<td>8.4ns / 26.1%</td>
<td>23.8ns / 73.9%</td>
</tr>
<tr>
<td>Spartan 3E (90nm)</td>
<td>15</td>
<td>25.00ns</td>
<td>12.57ns / 51.2%</td>
<td>11.98ns / 48.8%</td>
</tr>
<tr>
<td>Stratix 2 (90nm)</td>
<td>14</td>
<td>12.9ns</td>
<td>3.6ns / 32.4%</td>
<td>7.5ns / 67.6%</td>
</tr>
<tr>
<td>Virtex 4 (90nm)</td>
<td>10</td>
<td>15.38ns</td>
<td>2.72ns / 18.2%</td>
<td>12.21ns / 81.8%</td>
</tr>
<tr>
<td>Spartan 5 (65nm)</td>
<td>8</td>
<td>12.5ns</td>
<td>1.71ns / 13.9%</td>
<td>10.6ns / 86.1%</td>
</tr>
<tr>
<td>Stratix 3 (65nm)</td>
<td>9</td>
<td>6.6ns</td>
<td>2.0ns / 32.0%</td>
<td>4.2ns / 68.0%</td>
</tr>
<tr>
<td>Cyclone 3 (65nm)</td>
<td>25</td>
<td>19.8ns</td>
<td>6.0ns / 33.1%</td>
<td>12.1ns / 66.9%</td>
</tr>
<tr>
<td>Virtex 6 (45nm)</td>
<td>14</td>
<td>16.5ns</td>
<td>4.0ns / 24.8%</td>
<td>12.1ns / 75.2%</td>
</tr>
<tr>
<td>Virtex 5 (40nm)</td>
<td>15</td>
<td>13.3ns</td>
<td>2.1ns / 15.5%</td>
<td>11.2ns / 84.5%</td>
</tr>
<tr>
<td>Average</td>
<td>15</td>
<td>27 %</td>
<td>73 %</td>
<td></td>
</tr>
</tbody>
</table>

**Table 2.3:** Critical paths data in LEON3 processor implemented in FPGAs.

**Figure 2.3:** Illustration of a mapping of a SoC consisting of 8 ASVP cores, MicroBlaze (MB), DRAM memory controller (MPMC), DMA engine.

Tools in appropriate FPGAs, as shown in Table 2.3. Each design is a basic system on chip: it contains a single-core LEON3 processor, a memory controller, and peripheral devices.

In Table 2.3 the FPGAs are ordered according to the implementation silicon technology in nanometers. The other columns report on the critical path in the soft-core processor. The information was extracted from the static timing analysis. Each critical path is characterized by the number of levels of logic gates, by the maximal allowed delay (‘Clock’), and by the fraction of total delay spent in logic gates and in interconnect (‘route’). The minimal observed routing delay fraction is 48.8% in Spartan 3E, the maximum is 86.1% in Virtex 5, the average is 72.5%.

**Example 2: Mapping a Multi-Core in FPGA**

A mapping of a multicore system into Xilinx FPGA is shown in Figure 2.3. The system consists of 8 identical custom accelerator cores (ASVP, Chapter 4), MicroBlaze host processor, a DRAM memory controller (MPMC), and a DMA engine. The memory controller is an FPGA hard-core having a
Figure 2.4: Mapping of a SoC consisting of 8 ASVP cores, MicroBlaze (32-bit RISC CPU), controller for off-chip DDR3 DRAM (MPMC), and DMA engine in Xilinx Spartan 6 S6LX150T FPGA. The ASVP and MicroBlaze cores were manually constrained by UCF to designated areas, the MPMC and DMA cores were wholly placed by automated tools. The MPMC core is attracted to the top-left corner near the hard-core physical memory interface. The DMA engine is spread over the FPGA.
fixed placement in the chip. The ASVP soft-cores are spread over the chip by the placer tool. Each core’s local memory banks are mapped to dual-ported BlockRAMs (FPGA BRAMs). One port of each BRAM is connected to the shared DMA engine that transfers data to/from the external DRAM via the MPMC hard-core. The problem of this particular placement are uneven (often long) routing distances between the DMA engine situated in the centre and the BRAMs located near ASVPs. Ideally, each link between the DMA engine and the BRAM should be individually (automatically) pipelined according to the real distance after the automated placement.

Still, the placement illustrated in Figure 2.3 is rather optimistic. What the author has observed in reality though, is that the BRAMs were attracted to the DMA engine and placed closer to it than to the ASVP crossbars (Figure 2.5). This placement resulted in timing failures at the BRAM/crossbar interface within ASVP cores because of the long-distance routing. Figure 2.6 shows the pipelining around BRAMs at the DMA/ASVP boundary in detail. The BRAM/crossbar interface has been pipelined already, as shown at the right side of the picture. This internal ASVP pipelining is intended for a high clock frequency mode in DFU. In a low clock frequency mode the pipelining unnecessarily increases the start-up latency of DFU operations; it is omitted from the design in that case (controlled by the VHDL generic parameter; cf. Figure 4.5).

Hence, these new timing failures at the BRAM/crossbar interface were annoying. Analysis of the
post-place&route timing report showed that the critical wire from crossbar to BRAM had only 2 levels of logic: only 9% of the critical path was due to the logic, 91% was due to long-distance routing. Bad routing is typically due to bad placement. Placement is driven by timing-oriented heuristics. Bad placement could be a result of tight timing constraints that force two components (DMA and BRAM) close together while the third component (crossbar) suffers. By inserting pipelining registers at the other side–between the DMA engine and BRAM–the timing there is loosened, the placer has more freedom to localize BRAMs around the chip and the routing results improve.

The final situation is shown in Figure 2.4. The ASVP cores and the host processor had been locked manually to specific regions in the FPGA to minimize interference during placement.

The kind of pipeline register insertion shown here is not trivial. An analysis of FPGA post-place&route report together with the knowledge of design details is required. The actual insertion of registers in VHDL source code requires careful analysis of data dependencies. In the presented case, registers were inserted at both the BRAM address and data-out signals (Figure 2.6); consequently, an internal push signal enabling FIFO queue must be delayed by two clock-ticks as well.

2.3 Summary

Frequency scaling in integrated chips has ended once the chips became limited by power dissipation. Instead, many-core scaling has been employed. It is projected that many-cores will also become power-limited, requiring parts of chips being powered-off at any time.

To extract the most from a silicon generation and still satisfy power limits we will see more specialization of cores in the future. As the customization becomes a norm new approaches to the IC design will be required to maintain productivity.

Besides power, latencies in IC interconnects cannot be ignored any more. New structured approaches such as Networks-on-Chip have already appeared. The NoC approach is top-down, requires manual partitioning in modules with NoC interfaces, and works at the top inter-module level only.
I Limits of Static Scheduling in Programmable Architectures
Chapter 3

Programmable Accelerators in FPGA

The first Part of the thesis studies a contemporary statically scheduled vector processor realized on FPGA and suitable as an accelerator for DSP and image processing tasks. The architecture is called ASVP, *Application-Specific Vector Processor*. This short chapter provides motivations that drove development of the architecture.

3.1 Motivation for Static Scheduling

The obvious advantage of static scheduling is the lower utilization of run-time resources needed for control functions. As discussed previously, pre-computed static schedules are optimal when the latencies of all operations in the schedule can be determined off-line at the time the schedule is constructed. In practical applications the latencies of basic compute operations (e.g. arithmetic) are known statically, but the latencies of accesses to DRAM chips are unpredictable.

If the memory addresses of DRAM reads and writes can be predicted long before the actual data are needed for computation, the dynamic scheduling can be performed at a coarser granularity of computational blocks (or ‘tasks’). Within tasks the operations are scheduled statically because all their input/output data are held in dedicated local memory blocks with known latencies. The tasks themselves are scheduled dynamically as complete blocks once their data has been made available in local memories.

The advantage of dynamic scheduling at coarse granularity is the lower frequency at which the scheduling decisions must be made. Therefore the scheduling operations can be performed using sequential hardware or in software. The disadvantage is the need for partitioning the original algorithm into tasks.

3.2 Motivation for Software Programmability in FPGA

The mainstream design-entry languages (HDLs) for programming of contemporary FPGAs are VHDL and Verilog. Synthesis from higher-level languages, such as C [VPNH10], OpenCL [OBDA11] or CUDA [PGS+09] is difficult for two closely related reasons: First, the FPGA design space is nearly infinite and a program in a high-level language can be implemented in many different ways, with wildly varying resource usage, total cycle count, and cycle time (operating frequency). For compiler it is very difficult to strike the optimal configuration that minimizes the overall program execution latency (the absolute time). Second, even if the optimal configuration is already known, a single (re-)compilation and re-synthesis process after a source code modification may take tens of minutes up to several hours.
This severely impacts turn-around time of the application development, lowering efficiency of human programmers.

The alternative to coding hardware directly in HDL is to use an approach based on a custom programmable architecture. Figure 3.1 depicts the work-flow. Conceptually, an accelerator is described in a higher-level specification that combines hardware feature requirements with software (firmware) that implements the accelerated function using those features. Based on the required feature set the architecture implementation is selected from a library, or newly built. The architecture is programmable by firmware to some pre-defined extent, hence (minor) modifications in the algorithm during development can be quickly tested as they will trigger only recompilation of the firmware.

The practical implementation presented here is called Application-Specific Vector Processor (ASVP). It is loosely based on the previous work [DKBK08] where it used to be called Basic Computing Element (BCE).

The ASVP architecture satisfies the following requirements:

- The architecture is customizable, i.e. it is possible to add or remove features as needed.
- The architecture is programmable by firmware. This facilitates separation of concerns between the algorithm development in software and the hardware feature implementation and tuning.
3.3. MOTIVATION FOR VECTOR PROCESSING

Figure 3.2: Comparing the thread, SIMD, and vector-based execution of a parallel computation. Letters in the coloured boxes symbolise instruction types.

- The architecture supports floating-point (long latency) and integer (short latency) operations for DSP and video applications.
- The architecture abstracts low-level characteristics of the underlying FPGA implementation technology. The firmware programmer is shielded from the impact of adapting the latency/frequency ratio of the hardware units to the target technology.

3.3 Motivation for Vector Processing

Vector processing was selected for ASVP because of its relative implementation simplicity and resource use. Figure 3.2 compares the three basic strategies for instruction issue in a machine with replicated data-path: (a) multithreading, (b) SIMD threading, (c) vector processing. In multithreading each replicated data-path executes different instruction stream, hence multithreading is the most flexible but also the most costly. In SIMD threading independent data-paths share a common controller, hence they must all execute the same instruction simultaneously, subject only to masking. The software programming model recognizes multiple instruction streams, but the hardware supports only a single instruction stream. In vector processing there is a common controller and a single instruction stream. Instructions are replicated by hardware over many data elements, and processed in space in concurrent data-paths and/or in time by pipelining.

The Vector Instruction Set Architecture (V-ISA) in ASVP is built around wide, horizontally en-

Figure 3.3: Vector processor is controlled by a scalar controller.
coded instructions. Vector instructions are not read from a program memory, but rather generated on-the-fly by a scalar controller processor. The conceptual organization is shown in Figure 3.3. The scalar control processor is embedded in the fetch stage (V.FE) of the vector processor.

The main advantage of this scheme is that the format of vector instructions (V-ISA) can be easily extended for controlling any special hardware in the vector data path. Control-flow is handled within the embedded scalar controller. The disadvantage of the scheme is the relatively low issue rate of vector instructions because each v-instruction must be generated in software in the embedded controller (the consequences are discussed in Chapter 5).

3.4 System Integration

To facilitate easy integration in various system configurations the hardware interface of the ASVP core consists solely of dual-ported memory blocks, ubiquitous in the target FPGA technology.

The basic organization of a system with a shared bus, a host CPU, and a DRAM memory controller is shown in Figure 3.4. Without a Direct Memory Access engine (DMA E.) the host CPU transfers all the data for the accelerator in software (using e.g. the memcpy()), making this configuration rather slow. If the DMA engine is included, as shown in Figure 3.5, the host CPU manages but does not carry out the data transfers, and controls the accelerator. Finally, with microthreaded host CPU, Figure 3.6, the HWFAM organization moves also the concurrency management to hardware functions. This last special case will be discussed in Part III.
Chapter 4

Application-Specific Vector Processors

This chapter gives technical description of the ASVP architecture (Section 4.1) and of the originally intended approach to the construction of the Domain Function Unit in ASVP (Section 4.2). Evaluation is given separately for hardware (Section 4.3) and for applications (Section 4.4). The contents of this chapter was published in [X.7, X.9] and in [X.2].

4.1 Description of the Architecture

4.1.1 Overview & System Integration

Internally, each ASVP core is composed of three main blocks: Controller (sCPU), Vector Processing Unit (VPU), and multi-banked local memory (Figure 4.1). All data processing is exclusive within the Vector Processing Unit. Multi-banked local data memory, pipelining and batch processing allows the VPU to sustain the peak throughput 1 word/cc. The controller issues vector instructions to the VPU. Each VPU instruction potentially specifies many elementary operations performed over a sequence of values read from local memories.

Controlling functions are spread over three layers:

1. **Task scheduling** is dedicated to the host CPU (e.g., Xilinx MicroBlaze, LEON3).

2. **Scheduling of vector instructions** for VPU is realized in a simple scalar control processor (sCPU) local in each core. The sCPU forms and issues wide vector instruction words to the VPU.

3. **Data-path control** is realized autonomously in the VPU. The unit handles vector-linear (element-wise) and vector-reduction operations, and schedules accesses to the local memory banks.

Memory hierarchy is exposed on two levels:

1. **Global off-chip shared memory**, typically accessed via a DMA engine. The engine is programmed either by controllers in ASVP cores or by the host CPU. It delivers data directly in and out of the local memory banks.

2. **Local storage** in ASVP cores is realized using multiple memory banks—BlockRAMs in FPGA. The Vector Processing Unit accesses all the banks in parallel.

The on-chip local memory is used as a staging buffer for working sets. A kernel function running in the ASVP accesses data with non-unit strides, and often the same data is reused multiple times in
one computation run (temporal locality). In contrast, off-chip memory (DDR DRAM) has high latency, and it delivers high bandwidth only when unit-stride long data arrays are transferred.

4.1.2 Vector Processing Unit (VPU)

Vector Processing Unit implements the complete data-path in ASVP core. The structure is shown in Figure 4.2. Internally the VPU consists of: a crossbar, address generators (AG), and a domain function unit (DFU). Vector instructions (commands) are delivered to VPU sequentially as individual very long, horizontally encoded v-instruction words. The long instruction words are formed and issued to VPU by the controller (sCPU) that is local in each ASVP core.

The VPU fetches data vectors from the local memory banks, processes them, and stores the result back. The memory banks are dual-ported. In each bank one port is connected in the VPU and the other port is left as interface to outside world, such as for DMA engine. This organization allows for complete overlap of computations and data transfers in banks. In the default configuration each bank is a flat array of 1024 words. Each data word is 32-bits, suitable for holding single-precision floating-point values.

Local Memory Banks

The ASVP architecture does not contain traditional vector register file. Register files are notoriously cumbersome and inefficient to implement in FPGAs. Even the simplest 3-port (2R-1W) register file when implemented using dual-ported BlockRAMs requires duplicate (data-redundant) storage, hence incurring 2x overhead. Instead, the ASVP employs a multi-banked single-ported local memories, and a crossbar. The default configuration has 4 memory banks; this allows to read or write up to 4 values at a time. Further, local memory is not statically partitioned in separate architectural vector registers. Instead, applications treat memory banks as independent linear address spaces and partition the available memory into vector variables as needed. Some use few very long vectors (e.g. FIR, matrix multiplication), other use many short vectors for implementing complex computation (e.g. image segmentation).
4.1. DESCRIPTION OF THE ARCHITECTURE

Vectors are extracted from memory banks by Address Generators (AG). The maximal VPU hardware configuration has two AGs for each of the four operand channels, for a total of 8 AGs. The main AG 0-3 handle basic addressing modes: linear or stridden (increment ≠ 1) accesses (the increment can be negative), with lower and upper wrap-around bounds (overflowing the upper bound resets the pointer to the lower bound, and vice-versa). The second set of AG 4-7 is for indexed accesses. The secondary AGs have similar configuration registers as the main AGs, but the data streams they read from memory banks are passed to main AG. They are treated there as offsets to be summed with local addresses and sent down to the memory bank. Figure 4.2 shows a slightly modified architecture with a single shared indexing AG to save resources.
Figure 4.3: The crossbar between local memory banks (left) and the address generators (right).

**Crossbar**

The crossbar is located between local memory banks and address generators (Figure 4.3). Each vector instruction has several vector operands—inputs and outputs. Crossbar arbiter shown in Figure 4.4 allows for the special case when different vector operands lie in identical memory bank. In that case the crossbar automatically handles multiplexing of accesses from different address generators.

**4.1.3 Embedded Scalar Controller**

There are two distinct instruction sets (ISAs) in the architecture: The control sCPU executes a classical scalar ISA programmed in the C language. The implementation uses the 8-bit Xilinx PicoBlaze™ processor. An optimizing C compiler was developed in the LLVM framework for this target [X.14].
4.1. DESCRIPTION OF THE ARCHITECTURE

<table>
<thead>
<tr>
<th>Level</th>
<th>API Function in C in sCPU</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>void pb2dfu_set_cnt(uint16_t cnt);</td>
<td>Set the (input) vector length.</td>
</tr>
<tr>
<td>0</td>
<td>void pb2dfu_set_repetitions(uint8_t nrep);</td>
<td>Set the number of repetitions of a vector operation (batch length).</td>
</tr>
<tr>
<td>0</td>
<td>void pb2dfu_set_addr(uint8_t ag, uint16_t addr);</td>
<td>Set the base address of the vector operand ag.</td>
</tr>
<tr>
<td>0</td>
<td>void pb2dfu_set_bank(uint8_t ag, uint8_t mbank);</td>
<td>Select the bank (mbank) for the specified operand ag.</td>
</tr>
<tr>
<td>0</td>
<td>void pb2dfu_set_inc(uint8_t ag, int16_t inc);</td>
<td>Set the stride of the vector operand ag.</td>
</tr>
<tr>
<td>0</td>
<td>void pb2dfu_set_agflags(uint8_t ag, uint8_t flags);</td>
<td>Set the operation mode (flags) in the address generator (operand).</td>
</tr>
<tr>
<td>0</td>
<td>void pb2dfu_start_op(uint8_t opcode, uint16_t cnt);</td>
<td>Start the vector operation, with the given vector length.</td>
</tr>
<tr>
<td>0</td>
<td>void pb2dfu_restart_op(uint8_t opcode);</td>
<td>Start the vector operation.</td>
</tr>
<tr>
<td>0</td>
<td>uint8_t pb2dfu_wait4hw();</td>
<td>Wait until the vector op. has finished.</td>
</tr>
</tbody>
</table>

Table 4.1: The primary C API functions used to access (program) the vector unit from the sCPU.

The VPU, on the other hand, executes vector instructions. VPU instructions are prepared by sCPU in a so-called ‘instruction forming buffer’. The forming buffer is an array of registers connected to sCPU as an I/O periphery. Typically, several sCPU I/O instructions are required to set-up a VPU vector instruction in the buffer. Ideally the set-up of the next vector instruction should be finished before the previous vector instruction has completed in VPU, so that useful work in sCPU and VPU is overlapped.

From programmer’s point of view the VPU instruction buffer is accessed via a set of C functions with prefix pb2dfu_. The basic set of these functions (the so-called ‘Level 0’ set) is listed in Table 4.1. (There are also Level 1 and Level 2 pb2dfu_ functions— they will be discussed in Chapter 5). The basic Level 0 functions directly access the vector instruction forming buffer via sCPU I/O facilities. For example, the function pb2dfu_set_cnt(), which sets the vector length for the next VPU operation, is defined in the C library this way:

```c
/** Set the (input) vector length. */
static inline void pb2dfu_set_cnt(uint16_t cnt)
{
  __output_port(value, (port));
}

load s4, -112
output s4, 33 ; 33 = REG_DFUVECL_L
load s4, 1
output s4, 34 ; 34 = REG_DFUVECL_H
```

Inlining not only removes call/return overheads, but (when properly used) it typically improves register allocation in caller functions.
4.2 Domain Function Unit (DFU)

The Domain Function Unit (DFU) contains the actual hardware that performs computations on vector streams. The operation in DFU is controlled by three fields of the vector instruction word (Figure 4.2, see the top right corner):

1. *operation code* (opcode) – specifies what operation to do (Table 4.2),
2. *vector length* – how many elements in input streams constitute a single vector operation, and
3. *number of repetitions* – how many times the whole operation shall be restarted.

The field ‘number of repetitions’ allows autonomous restart of the same operation multiple times to create a ‘batch’ of operations. Obviously, this trick lowers the total number of distinct vector instructions that must be issued by the sCPU.

4.2.1 Customizing the Domain Function Unit

Basic custom vector instructions are composed of a hypothetical control loop with embedded acyclic static data-flow graph. Three types of control loops are supported:

a. *element-wise function mapping* (e.g. vector addition),

b. *full reductions* (e.g. vector summation), and

c. *prefix reductions* (e.g. cumulative summation).

**Full Reduction**

**Full reduction** $\Psi_o$ of vector $A_i$ using operator $\circ$ is defined:

$$ r = A_0 \circ A_1 \circ ... \circ A_{n-1} = \Psi_o A_i $$  \hspace{1cm} (4.1)

(E.g., summation is: $\Psi_o + \equiv \sum_i; r = \sum_{i=0}^{n-1} A_i$).

The operator denoted $\circ$ represents the embedded data-flow function with latency $k_o$. It has to be associative and commutative so that the reduction can be pipelined, and there has to exist a neutral element $\epsilon_o$ (e.g. $\epsilon_o = 0$). Given this the Equation 4.1 can be rewritten (for the case $k_o = 2$):

$$ r = \epsilon_o \circ A_0 \circ A_1 \circ ... \circ A_{n-1} = $$

$$ = (\epsilon_o \circ A_0 \circ A_2 \circ ...) \circ (\epsilon_o \circ A_1 \circ A_3 \circ ...) $$ \hspace{1cm} (4.2)

The last form leads to a hardware sub-circuit model labelled ‘full reduction’ in Figure 4.2 in the DFU block. The reduction computation in the circuit goes through three phases:

1. In the first $k_o$ cycles the operator output $c$ is invalid, thus the input $b$ must be supplied with the value $\epsilon_o$.

2. In the next $n - k_o$ cycles the rest of the vector $A_i$ is consumed, and partial results on $c$ are routed to the input $b$. 

3. In the wind-up phase which lasts \( w \) cycles the partial results circling in the pipeline are gradually reduced down to the final value \( r \).

In the wind-up phase it takes \( k_o \) cycles (one round) to halve the number of values circling in the pipeline, thus \( \log_2 k_o \) rounds are required:

\[
w = (1 + \log_2 k_o)k_o - 1 \tag{4.4}
\]

Therefore, the pipelined (parallel) implementation of the full reduction is much faster than the sequential one:

\[
t_{seq, \Psi_o}(n) = k_o \cdot n \tag{4.5}
\]

\[
t_{par, \Psi_o}(n) = n + w = n + (1 + \log_2 k_o)k_o - 1 \tag{4.6}
\]

The execution times \( t \) are given in clock-cycles here. In the pipelined algorithm the wind-up latency is \( w \) cycles, but there is only \( k_o \) values in the execution pipeline at the beginning. Hence, the operator \( \circ \) is used only in \( k_o - 1 \) cycles, and empty cycles are inserted into the pipelined unit otherwise.

### Prefix Reduction

**Prefix reduction** \( \Xi_o \) of a vector \( A_i \) into vector \( C_j \) is defined:

\[
\forall j = 0, 1, \ldots, n-1 : C_j = \psi_o A_i, \text{ hence: } C = \sum_{i=0}^{n-1} A_i \tag{4.7}
\]

Prefix reductions are much more difficult to implement in pipelined manner. The pipelined algorithm requires \( \log_2 n \) passes over the array, and in each pass \( i \) roughly \( (n - 2^i) \) operations are performed that can be pipelined. Thus:

\[
t_{seq, \Xi_o}(n) = k_o \cdot n \tag{4.8}
\]

\[
t_{par, \Xi_o}(n) \approx \sum_{i=0}^{\log_2 n} (n - 2^i) = n \cdot \log_2 n - n + 1 \tag{4.9}
\]

The parallel algorithm is faster for some combinations of \( k_o \) and \( n \), but it is asymptotically slower (in \( n \)) than the sequential algorithm \( (O(n) < O(n \cdot \log_2 n)) \) when only one instance of the reduction operator \( \circ \) is available. Thus, when the prefix reduction algorithm is needed the sequential implementation is used in hardware.

### 4.2.2 Practical Example

In the **Image Segmentation** (ImgSeg) kernel\(^1\) there is an operation that locates the minimal (maximal) value in a given vector of floating-point numbers. The operation either returns the value (then it is VMIN, VMAX), or the integer index where the value is located (INDEXMIN, INDEXMAX). The integer index can be used in subsequent operations for indexed accesses in address generators.

These operations can be efficiently implemented as **full reductions**. First we define a scalar function \( \text{Min}(a, b) \) that simply returns the lesser of two arguments. The function is commutative \( (\text{Min}(a, b) = \text{Min}(b, a)) \) and associative \( (\text{Min}(a, \text{Min}(b, c)) = \text{Min}(\text{Min}(a, b), c)) \). The neutral element is \( e_{\text{Min}} = \)

---

\(^1\)Application kernels will be described in Section 4.4.
Table 4.2: Some of the vector operations implemented in DFU. Type: M=element-wise function map, FR=full reduction

+∞ because Min(a, +∞) = a. Thus we can place VMIN = ΨMin. The logical composition is shown in Figure 4.2.

Another example of application-specific vector instructions are the VCONVR/VCONVG/VCONVB instructions. The instructions take a 32-bit pixel, extract a given 8-bit colour (R, G, or B) from it, and convert the colour to a floating-point value.

Table 4.2 lists other vector instructions implemented for the ImgSeg application. The DPROD operation is the dot-product that is very useful for implementing matrix multiplications. The VCMPLT (compare-less-than) operation compares two vectors element-wise and returns a vector of boolean values. The VSELECT operation is a vectorized conditional ternary operator from the C language.

4.3 Characterization of the Hardware Implementation

The ASVP architecture was implemented in several generations of the Xilinx FPGA technology: Virtex 5 (65nm), Virtex 6 (40nm), Spartan 6 (45nm), and Kintex 7 (28nm). Virtex-class devices are considered by Xilinx as high-end, Kintex-class (new in the 7-th generation) as middle-end, and Spartan-class is low-end.

4.3.1 Operating Frequency

The two main parts of the ASVP architecture—the sCPU and the VPU—are implemented in separate clock domains:

1. The controller sCPU with the configuration interfaces are clocked at the base frequency $f_0$.
2. The Vector Processing Unit (VPU) is clocked at $f_{VP}$. Typically $f_0 \leq f_{VP}$.

The division line between the two clock domains runs along the vector-instruction forming buffer, see Figure 4.1. The controller sCPU is often part of a wider ecosystem with which it has to communicate (e.g. host CPU, DMA engine configuration registers). Also the PicoBlaze processor in sCPU is optimized for a lower frequency range 50-100MHz (in Spartan 6 and Virtex 5). The VPU is coupled to an environment via command/status links to sCPU, and through the dual-ported local memory banks.

The base operating frequency $f_0$ for controller is usually determined by external factors of the embedded system, such as the System-on-Chip platform and the host CPU. Here we assume $f_0(\text{Spartan6}) = 50\text{MHz}$ and $f_0(\text{Virtex5+6, Kintex7}) = 100\text{MHz}$.
4.3. CHARACTERIZATION OF THE HARDWARE IMPLEMENTATION

<table>
<thead>
<tr>
<th>A3</th>
<th>M2</th>
<th>M3</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>150</td>
<td>166</td>
</tr>
<tr>
<td>A4</td>
<td>A5</td>
<td>A6</td>
</tr>
</tbody>
</table>

(a) Virtex 5 (65nm)

<table>
<thead>
<tr>
<th>A2</th>
<th>M3</th>
<th>M4</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>&lt;125</td>
<td>125</td>
</tr>
<tr>
<td>A3</td>
<td>A4</td>
<td>A5</td>
</tr>
</tbody>
</table>

(b) Virtex 6 (40nm)

<table>
<thead>
<tr>
<th>A3</th>
<th>M2</th>
<th>M3</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td>A5</td>
<td></td>
</tr>
</tbody>
</table>

(c) Spartan 6 (45nm)

<table>
<thead>
<tr>
<th>A3</th>
<th>M2</th>
<th>M3</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(d) Kintex 7 (28nm)

Table 4.3: The frequency $f_{VPU}$ [MHz] depends on the implementation technology and the pipeline depth in compute units. $Ax$ = FP-Adder latency, $Mx$ = FP-Multiplier latency.

The $f_{VPU}$ is determined by the pipelining depth of the data paths. In VPU, address generator units and the DFU are coupled by short FIFO queues and additional registers are inserted between the crossbar switch and memory banks. The largest impact on $f_{VPU}$ has a latency of compute units within the DFU.

For computations in the floating-point (FP) number system the DFU uses pipelined cores from Xilinx. The FP adder and multiplier are needed to support a minimal useful set of operations. The FP cores can be generated with configurable latency: the latency is the number of pipeline stages in the hardware implementation. Higher latency usually enables higher operating frequency. Configurations are expressed in $AxM_y$ notation: $Ax$ specifies that the FP-adder has the latency $x$ clock-cycles, and $M_y$ specifies that the FP-multiplier has the latency $y$.

The frequency scaling characteristics of the ASVP implementation is in Table 4.3. The tables give the maximal achievable $f_{VPU}$ in MHz for a given FPGA technology and the DFU pipeline depth. The DFU pipeline depth are given using the $AxM_y$ notation. For instance, in Virtex 6 FPGA if $f_{VPU} = 150$ MHz is required the DFU in the ASVP core has to be configured with the latency (at least) A4M3. Conversely, if slower $f_{VPU} = 100$ MHz is needed, the DFU can be configured to A3M2, thus lowering the wind-up latency per Equation 4.4.

4.3.2 Resource Utilization

Information about area breakdown in flip-flops (FF), combinatorial resources (LUTs), and slices is given in Figure 4.6 for the Virtex 6 technology; in the other FPGA technologies the results are similar (Table 4.4). The figure compares a single-clock domain (low-speed, $f_0 = f_{VPU}$) with a dual-clock domain (high-speed, $f_0 < f_{VPU}$) configurations. In Virtex 6 FPGA technology one slice represents 4 six-input LUTs and 8 storage (FF) elements.

The sCPU controller itself (PicoBlaze 3) consumes very few resources, about 56 slices. However, its peripheral circuitry (I/O decoders and multiplexers) consumes about $5 \times$ more space than the controller itself (292 slices).

In the dual-clock configuration the registers for crossing the clock-domain from $f_0$ to $f_{VPU}$ constitute about 15% of flip-flops. This relatively high count is caused by registering the long vector instruction word between the sCPU and VPU. The dual-clock configuration is also optimized for higher frequency as additional registers are used in the VPU for deeper pipelining. The crossbar is about $3.8 \times$ larger in flip-flops because registers are instantiated on all its inputs. Address generators consume about $1.8 \times$ more registers because several more FIFOs are instantiated, and the existing ones are deeper.

The high-speed dual-clock configuration requires $1.6 \times$ more registers but roughly an equal number of LUTs. In the Virtex 6 technology one slice represents 4 six-input LUTs and 8 storage elements.
### Table 4.4: Utilization of FPGA slices.

<table>
<thead>
<tr>
<th>FPGA</th>
<th>VCU</th>
<th>Total Ctrl</th>
<th>Ctrl.PB3</th>
<th>VPU</th>
<th>VPU.AG</th>
<th>VPU.Arith</th>
<th>VPU.Mux</th>
<th>VPU.XBAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC5VLX110T 100M</td>
<td>A3M2</td>
<td>100</td>
<td>32</td>
<td>2017</td>
<td>358</td>
<td>65</td>
<td>1659</td>
<td>576</td>
</tr>
<tr>
<td>XC5VLX110T 150M</td>
<td>A4M3</td>
<td>150</td>
<td>37</td>
<td>2148</td>
<td>389</td>
<td>53</td>
<td>1759</td>
<td>596</td>
</tr>
<tr>
<td>XC5VLX110T 166M</td>
<td>A5M3</td>
<td>166</td>
<td>38</td>
<td>2076</td>
<td>361</td>
<td>58</td>
<td>1715</td>
<td>570</td>
</tr>
<tr>
<td>XC6VLX240T 100M</td>
<td>A3M2</td>
<td>100</td>
<td>32</td>
<td>1693</td>
<td>339</td>
<td>56</td>
<td>1354</td>
<td>477</td>
</tr>
<tr>
<td>XC6VLX240T 150M</td>
<td>A4M3</td>
<td>150</td>
<td>36</td>
<td>1685</td>
<td>324</td>
<td>54</td>
<td>1361</td>
<td>469</td>
</tr>
<tr>
<td>XC6VLX240T 200M</td>
<td>A6M3</td>
<td>200</td>
<td>41</td>
<td>1781</td>
<td>366</td>
<td>58</td>
<td>1415</td>
<td>478</td>
</tr>
<tr>
<td>XC6SLX45T 50M</td>
<td>A2M3</td>
<td>50</td>
<td>30</td>
<td>1789</td>
<td>362</td>
<td>52</td>
<td>1427</td>
<td>487</td>
</tr>
<tr>
<td>XC6SLX45T 125M</td>
<td>A3M4</td>
<td>125</td>
<td>35</td>
<td>1840</td>
<td>351</td>
<td>57</td>
<td>1489</td>
<td>482</td>
</tr>
<tr>
<td>XC7K325T 100M</td>
<td>A3M2</td>
<td>100</td>
<td>32</td>
<td>1630</td>
<td>472</td>
<td>182</td>
<td>1158</td>
<td>345</td>
</tr>
<tr>
<td>XC7K325T 200M</td>
<td>A4M3</td>
<td>200</td>
<td>39</td>
<td>1667</td>
<td>490</td>
<td>188</td>
<td>1177</td>
<td>332</td>
</tr>
<tr>
<td>XC7K325T 250M</td>
<td>A5M3</td>
<td>250</td>
<td>35</td>
<td>1635</td>
<td>459</td>
<td>168</td>
<td>1176</td>
<td>338</td>
</tr>
<tr>
<td>XC7K325T 300M</td>
<td>A6M3</td>
<td>300</td>
<td>40</td>
<td>1640</td>
<td>470</td>
<td>194</td>
<td>1178</td>
<td>346</td>
</tr>
<tr>
<td>XC7K325T 400M</td>
<td>A7M3</td>
<td>400</td>
<td>45</td>
<td>1650</td>
<td>478</td>
<td>201</td>
<td>1180</td>
<td>352</td>
</tr>
<tr>
<td>XC8VLX240T 200M</td>
<td>A4M3</td>
<td>200</td>
<td>37</td>
<td>1781</td>
<td>357</td>
<td>58</td>
<td>1415</td>
<td>478</td>
</tr>
<tr>
<td>XC8VLX240T 300M</td>
<td>A6M3</td>
<td>300</td>
<td>42</td>
<td>1795</td>
<td>377</td>
<td>63</td>
<td>1427</td>
<td>487</td>
</tr>
<tr>
<td>XC8VLX240T 400M</td>
<td>A8M3</td>
<td>400</td>
<td>47</td>
<td>1800</td>
<td>391</td>
<td>68</td>
<td>1444</td>
<td>505</td>
</tr>
<tr>
<td>XC9VLX330T 200M</td>
<td>A4M3</td>
<td>200</td>
<td>39</td>
<td>1781</td>
<td>357</td>
<td>58</td>
<td>1415</td>
<td>478</td>
</tr>
<tr>
<td>XC9VLX330T 300M</td>
<td>A6M3</td>
<td>300</td>
<td>42</td>
<td>1795</td>
<td>377</td>
<td>63</td>
<td>1427</td>
<td>487</td>
</tr>
<tr>
<td>XC9VLX330T 400M</td>
<td>A8M3</td>
<td>400</td>
<td>47</td>
<td>1800</td>
<td>391</td>
<td>68</td>
<td>1444</td>
<td>505</td>
</tr>
</tbody>
</table>

Legend:
- VCU: Vector Control Unit
- Total Ctrl: Total Control Elements
- Ctrl.PB3: Control Elements for PB3
- VPU: Vector Processing Unit
- VPU.AG: VPU Allocation
- VPU.Arith: VPU Arithmetic Elements
- VPU.Mux: VPU MUX Elements
- VPU.XBAR: VPU XBAR Elements
- FPGA: Field-Programmable Gate Array
- Total: Total Utilization
- Ctrl: Control Utilization
- VPU: VPU Utilization
- VPU: VPU Allocation
- VPU: VPU Operations
- VPU: VPU Memory
- VPU: VPU XBAR
- VPU: VPU XBAR
- VPU: VPU XBAR
4.4 Performance of Application Kernels

Three application kernels implemented in the ASVP platform are evaluated: matrix multiplication (MATMUL), computation of Mandelbrot fractal set (MF-set), and image segmentation (ImgSeg). All programs work in floating-point (FP) single precision (32-bit) format.

The algorithms were transformed into a vectorised form. The vectorised form gives a control-data flow graph consisting of the operations and the data dependencies (Figure 4.6). From this a graph of vector live ranges (lifetimes) is obtained. The control-data flow graph is used to map individual vector variables into one of the four memory banks (A–D). For example, in the operation $\vec{z} = \text{VADD}(\vec{x}, \vec{y})$ the variables $\vec{x}$, $\vec{y}$, $\vec{z}$ should be mapped to distinct banks so that individual elements of the vectors can be read or written in parallel. Although the hardware allows for operands being mapped to identical banks but it is with reduced execution speed (Subsection 4.1.2, crossbar). Once vector variables are assigned to banks, the graph of live ranges is used to place them to consecutive arrays of elements within the banks. Vectors with overlapping live ranges must not occupy overlapping arrays in banks.

To measure the execution time and to profile the application kernels the cycle-accurate synthesizable VHDL model of the ASVP core is simulated in the ModelSim® simulator. In an HDL simulator

Figure 4.5: Area breakdown (FF, LUTs and Slices) in Virtex 6 technology, in the single-clock/low-speed and dual-clock/high-speed configurations. One slice represents 4 six-input LUTs and 8 flip-flops (FF).

After packing the design into slices, the dual-clock configuration requires roughly $1.2 \times$ more slices.
it is much easier to observe timings of relevant events. Configurable parameters of the model – the latencies of the FP cores and (the interpretation of) the clock-cycle time – are set according to the technology characteristics obtained previously.

Bar plots in Figures 4.7, 4.9, and 4.12 give the total execution times in microseconds for different FPGA technology nodes. The total execution times are split into four segments in the bars:

- **D**: execution on sCPU that was not overlapped with computation in DFU;
- **C**: DFU has full pipeline (useful computation);
- **B**: DFU pipeline bubble due to the reduction wind-up;
- **A**: DFU stall (input data not available).

For comparison, the bar plots are overlaid with line plots showing a hypothetical execution time under the ideal clock frequency scaling: the first column in each group is taken as the baseline and the subsequent data points are simply scaled by frequency difference. Of course, this ideal scaling is highly optimistic because it assumes that the clock-cycle time decreases uniformly in the core, while in reality we keep the sCPU clock frequency \( f_0 \) constant and change only \( f_{VPU} \).

In the two more complex benchmarks (MF-set, ImgSeg) the lower bound of the execution time determined by the fixed sCPU speed \( f_0 \) is presented. The column is called ‘Limit’ in the plots. This execution time is measured in simulation by setting \( f_{VPU} \) to an extremely high value (10000MHz). The latency of the sCPU is fully exposed because the VPU execution takes only a few clock-cycles in the \( f_0 \) domain.

### 4.4.1 Matrix Multiplication (MATMUL)

**Algorithm**

The MATMUL program computes dense matrix multiplication \( C = A \times B \). Source code of the kernel is shown in Listing 4.1, and in Figure 4.8. In square \( n \times n \) matrices the computation requires \( n^2 \)
dot-products, grouped in \( n \) batches. Each batch computes one row (or one column) of the result matrix.

Figure 4.8 shows how the repetitions feature in AG is used to compute one result row in matrix multiplication by a single vector instruction. The AG 1 is set-up to repeatedly scan a row in the matrix A. Back-jumps at the end of the row are implemented using a boundary detection feature. The AG 2 scans matrix B in a column-wise manner. This requires a help from its slave address generator: the AG 6 supplies offsets along a column of matrix, while AG 2’s local address is incremented only when AG 6 overflows and back-jumps to the top of column. The AG 0 supplies addresses where individual results are placed. The DFU sees the AG outputs as two streams of floating-point numbers incoming from local memories. It executes dot-product over each \( p \) elements (\( p \) is the number of columns in A and rows in B), generates a result, then restarts \( n \) times (\( n \) is the number of columns in B and C). The firmware C code that accomplishes the computation is in Listing 4.1.

**Results**

Figure 4.7 shows results measured for square matrices \( n \times n \). In matrix multiplication the main source of inefficiency are bubbles caused by reduction operations in DFU. For instance, in the fastest Virtex 6 node A6M3@200M when multiplying 32x32 matrices the reduction cycles contribute about 35% of the total execution time.

### 4.4.2 Mandelbrot Fractal Set (MF-set)

**Algorithm**

The MF-set program computes the Mandelbrot fractal set. The output is an image of the fractal. The basic sequential algorithm for a single pixel is given in Listing 4.2. Image pixels are mapped in 2D rectangle in a complex space from \((-2.5 - 1j)\) to \((1 + 1j)\). In the algorithm complex number mathematics have been already decomposed to real number operations. The real axis corresponds to the \( x \) pixel dimension and the imaginary axis to the \( y \) pixel dimension. The algorithm iterates using the \texttt{while} condition loop until the given pixel is known to lie outside the fractal set. Pixels that lie in the set would iterate indefinitely because the condition \( x^2 + y^2 < 4 \) is always satisfied for them. The algorithm decides that pixel lies in the set when the number of iterations reaches some arbitrary upper bound, 50 in this case. When the while loop exits the number of iterations is used for colouring the pixel in the output image.

Image pixels are independent in the fractal set, hence the algorithm is vectorized by simultaneously computing blocks of pixels. The vectorised algorithm is given in Listing 4.3. The issue is that distinct pixels in the same block require different number of iterations of the \texttt{while} loop. The solution is to physically compute all 50 iterations of the \texttt{while} loop for all pixels in the block, but to update the state vectors \((x, y, \text{it})\) based on a bit-mask vector of condition outcome. The bit-mask condition vector is computed at line 8, the state vectors \( x, y, \) and \( \text{it} \) (number of iterations) are updated using the \texttt{VSELECT} operation at line 21.

**Results**

The maximal block size in the implementation is 200 pixels. The constrain is given by the size of the local memory in ASVP. Execution times are shown in Figure 4.9. Compared to MMULT, the MF-set kernel does not use reduction operations such as \texttt{VSUM} or \texttt{DPROD}, hence the segment ‘B’ is zero. On the other hand we see higher ratio of stalls caused by starving (segment ‘A’). The kernel scales to higher \( \text{f}_{VPU} \) quite well because of the relatively long 200-element vectors; the sCPU has enough
CHAPTER 4. APPLICATION-SPECIFIC VECTOR PROCESSORS

Figure 4.7: Matrix Multiplication (MATMUL) execution time in microseconds for different matrix sizes and technology nodes with varying frequency/latency ratios.

Figure 4.8: Using complex addressing modes, one row of matrix multiplication can be computed using only one vector instruction.
4.4. PERFORMANCE OF APPLICATION KERNELES

Listing 4.1: Firmware source for matrix multiplication (MATMUL) kernel.
4.4.3 Image Segmentation (ImgSeg)

The ImgSeg program implements real-time video motion detection, sometimes also called foreground/background pixel segmentation. The algorithm is derived from [KB01]; the implementation does not use shadow detection and it has several modifications intended to lower compute complexity. The implementation in ASVP firmware was done by our colleagues and presented in a joint paper [X.9].
### 4.4. Performance of Application Kernels

#### Figure 4.9: Mandelbrot Fractal Set (MF-set) execution times in microseconds for different matrix sizes and technology nodes with varying frequency/latency ratios. The column labelled ‘Limit’ shows the minimal execution time imposed by fixed-frequency sCPU.

<table>
<thead>
<tr>
<th>Technology Node</th>
<th>Execution Time [µs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtex 5/6</td>
<td>1000</td>
</tr>
<tr>
<td>Spartan 6</td>
<td>900</td>
</tr>
<tr>
<td>Kintex 7</td>
<td>800</td>
</tr>
<tr>
<td>Limit (sCPU@100M)</td>
<td>700</td>
</tr>
</tbody>
</table>

**Algorithm**

The goal of image segmentation is to mark each pixel in an image frame as a part of static background or moving foreground. The decision depends on statistical models and their mixtures. All pixels in image are considered independently. Each pixel is modelled by a mixture of $K$ strongest Gaussian models of background, $K = 4$ in our implementation. Each Gaussian model $k$ is defined by a set of 3 mean values $\mu_{R,k}, \mu_{G,k}, \mu_{B,k}$, corresponding to the three primary colours red, green, blue; by variance $\sigma_k$, and by weight $w_k$. Figure 4.11 shows data structures used in the algorithm. Models represent RGB colours that are considered to be ‘stationary background’ colours of the pixel. As there are $K = 4$ independent Gaussian models kept for each pixel the algorithm allows for situations when the pixel periodically changes between two colours, such as moving escalators or trees in wind–these scenes are classified as stationary. Each model also contains the weight parameter $w_k$ indicating how often that particular model successfully described background in the pixel.

Figure 4.10 shows how the algorithm updates the models; for simplicity the presentation ignores RGB colours and shows only three models. The first picture at the top shows initial situation with three models $M_1= (\mu_1, \sigma_1)$, $M_2= (\mu_2, \sigma_2)$, and $M_3= (\mu_3, \sigma_3)$. Mean values $\mu_i$ position ‘bell’ shaped models on the horizontal colour (greyscale) axis; variances $\sigma_i$ define widths of the ‘bells’; and model weights $w_i$ are represented by the heights of the ‘bells’. When new pixel colour hits in one of the models the model is ‘strengthen’ by slightly increasing its weight, and the pixel colour is classified as
CHAPTER 4. APPLICATION-SPECIFIC VECTOR PROCESSORS

Figure 4.10: Updating Gaussian models in the image segmentation algorithm (illustration of the principle).

a background that is stationary. This situation is shown in the picture in the middle: the colour hits in model M3, the weight $w_3$ is increased. If the hit is not precise the model is also slightly shifted towards the new colour.

However, when new pixel colour does not hit any existing Gaussian model, the colour is classified as a foreground that is moving. The weakest model is erased and replaced by a new model representing the new colour, albeit with small initial weight. This is illustrated in the last picture in Figure 4.10: the weakest model M3 has been replaced by new model.

Results

The algorithm was vectorized to compute 50 independent pixels simultaneously. Vector length is limited by the local memory and the storage requirements of the algorithm. As shown in Figure 4.11, for each RGB pixel the local memory in ASVP core has to keep $4 \cdot 5 = 20$ floats of the contextual (model) data, plus the original R-G-B components converted to floats. This gives 23 floating point
4.4. PERFORMANCE OF APPLICATION KERNELS

Figure 4.11: Data structures in the image segmentation kernel.

Figure 4.12: Image Segmentation (ImgSeg) execution time in microseconds for different implementations (V5/6, S6, K7) and varying frequency/latency ratios. The column labelled 'Limit' shows the minimal execution time imposed by fixed-frequency sCPU.
CHAPTER 4. APPLICATION-SPECIFIC VECTOR PROCESSORS

(32-bit) numbers per each pixel that must be loaded for processing of a single pixel in an image frame.

Conditional branches are vectorized by speculatively computing both possibilities and then VSELECTing the correct value per each element. The operations INDEXMIN and INDEXMAX are used to find indexes of the strongest and weakest models.

Execution times are shown in Figure 4.12. The obvious performance problem is a high fraction of controller overhead (‘D’). For example, in Spartan 6 A4M4@125M implementation the VPU is idle 52% of time. The issue lies in vector instructions with complicated configurations of address generators that must be loaded in the forming buffer by the controller. In the next chapter a method that efficiently solves the performance problem will be described.

4.5 Summary

The chapter has provided the first thesis contribution: (1) Analysis of a statically scheduled instruction-driven vector processing architecture for customized computing realized in a reconfigurable array. The analysis has been carried out separately in practical FPGA hardware synthesis and in application mapping.

Data-path is easy to design, difficult to develop.

In Section 4.2 a simple model of constructing new customized operators in DFU was described. However, the practical implementation in hardware is performed manually in VHDL. The current implementation uses the classical approach with a finite state machine and a data path configured by multiplexers. Author’s practical experience has shown that the code is difficult to maintain, extend with new operations, and optimize. Based on it a new general data-driven design technique will be presented in Part II.

Resource consumption: large non-compute modules (address generators and the sCPU periphery).

The resource utilization of an ASVP core in FPGA is comparable or higher than that of the Xilinx Microblaze processor. For instance, in Virtex 6 the ASVP core in default configuration consumes 1627 slices, the Microblaze in the default area-optimized (lower performance) configuration 1223 slices. Address generators occupy around 27% of the area, followed by DFU multiplexers and XBAR each with 17%, and sCPU periphery with 16%. The sCPU itself is around 3%. The address generators are relatively complex and costly functions. In the horizontally-encoded v-instructions each AG requires 46 bits of configuration data. The other surprisingly large structure is the sCPU peripheral circuitry that includes port address decoders and multiplexers. The reason is a dense occupation of the 8-bit address space that requires full-address decoders.

Efficiency loss due to imperfect vectorization (redundant computations in Mandelbrot).

The Mandelbrot kernel illustrates the vectorization problem present in any data-parallel architecture. The original sequential procedure of the Mandelbrot algorithm was vectorized via the outer pixel-parallel loop. However, pixel-loop iterations are not identical: each pixel takes different number of iterations in the inner ‘while’ loop. Hence, if all data-parallel instances are made iterating together the efficiency will suffer because some operations are superfluous.

Insufficient low v-instruction issue rate caused by relatively slow general-purpose controller.

In ImgSeg the problem is very high VPU waiting time caused by insufficiently low issue rate of vector instructions. This could be side-stepped by increasing the computation vector length but at the cost of local memory space, or by replacing the existing general-purpose sCPU controller.
with a tailor-made design. In Chapter 5 a more clever and scalable solution based on targeted controller specialization is presented.
Chapter 5

Reducing Instruction Issue Overheads

In Chapter 4 it has been shown that overheads in the controller when creating the v-instructions for VPU significantly contribute to the running time of the image segmentation kernel. After an initial analysis (Section 5.1) this chapter describes (Section 5.3) and evaluates (Section 5.4) an extension to the ASVP architecture that significantly improves the performance. The contents of this chapter was published in [X.6].

The ASVP is conceptually a vector processor in which its vector instructions are generated on-the-fly in an auxiliary scalar processor embedded in the fetch stage of the vector core, as depicted in Figure 3.3. The v-instructions are encoded horizontally: specific functions of the vector hardware are directly controlled by dedicated bits in the v-instruction word and not much decoding is needed. This scheme is simple and extensible if new hardware in VPU is to be controlled. The disadvantage of the horizontally encoded v-instructions is their bit width, varying only according to hardware configuration (Figure 5.1).

![Figure 5.1](image)

**Figure 5.1:** Structure of a vector instruction (v-instruction) in an ASVP with $5 \times AG$ (dst, src1, src2, src3, idx). In the default configuration each of the AG requires 46 b of configuration data, the DFU needs another 26 b, hence a VPU with $5 \times AG$ requires 256 b v-instruction word. A program memory this wide is impractical.

5.1 Analysis of the Cause

The controller that issues the v-instructions is built around Xilinx PicoBlaze 3 microcontroller [Cha11]. PicoBlaze 3 is a simple 8-bit RISC-style processor optimized for minimal area in Xilinx FPGA technology. The processor executes one instruction each 2 clock-cycles: it has IPC = 0.5.

In Spartan 6 the PicoBlaze runs at 50MHz; hence, its performance is 25 MIPS. (In Virtex 5,6 and Kintex 7 the processor runs at 100MHz / 50 MIPS.) The peak performance of the VPU in Spartan 6 A4M4@125M is 250 MFLOPS, and it can output 125M-floats/s. The ImgSeg kernel executes 58 vector operations that collectively take 2790 cycles of the DFU execution time (as counted in the $f_0$ domain). Between two consecutive vector operations there are on average only 48 clock-cycles of controller execution that are fully overlapped by useful work in VPU/DFU. However, in these 48 cycles the PicoBlaze can execute only 24 instructions. A common operation that is usually performed during
this time is a loading of 16-bit values into fields of the vector-instruction forming buffer; for instance, starting addresses in AGs are usually modified between two succeeding operations. To perform this operation the PicoBlaze processor has to execute 4 instructions (2x LOAD, 2x OUTPUT). In 24 instructions the controller can do only six simple 16-bit loads. Obviously, the problem is the low performance of the controller when preparing vector instructions or, conversely, a requirement to rewrite large parts of the issue buffer in the time between two vector instructions.

5.1.1 Solution 1 (Not Used): Improving Controller Performance

Performance of the controller could be improved by increasing its operating frequency (shortening the cycle time) or by improving the IPC. Figure 5.2 shows hypothetical execution times of the ImgSeg kernel if the controller performance (throughput) had been doubled. The results were obtained in simulator by clocking the controller at twice the normal speed: 200 MHz in V5, V6 and K7, and 100 MHz in S6. Similar results might be obtained at the original base frequency by doubling the IPC to IPC = 1.0 using pipelining. Controller overhead ('D') is reduced by 3× to 4×, the total execution time is improved on average by 27%.

However, clocking a soft-core controller in FPGA at around 200 MHz could be challenging. Ta-
5.2. COMPARISON TO THE VLIW APPROACH

Figure 5.3: Histogram of controller’s writes in fields of the vector-instruction forming buffer, obtained for the ImgSeg kernel.

Table 5.1 lists the maximum clock frequencies achievable by PicoBlaze 3 controller as reported by Xilinx in [Xil11]. The PicoBlaze line of controllers was designed for small area (tens of FPGA slices), the execution speed was not seen as so important.

5.1.2 Solution 2 (Adopted): Improving the V-Instruction Issue Hardware

Instead of trying to increase raw speed of the controller the organization of the issue buffer could be modified to decrease the required number of interventions from the controller. Figure 5.3 depicts a histogram of writes performed by the controller in the vector-instruction forming buffer when running the ImgSeg kernel. The vertical axis represents probability [%] that a given field of the buffer (horizontal axis) needs to be modified between two consecutive vector instructions. The field AG0.addr, which defines the destination address where the vector result is deposited, is always (100% cases) modified between two consecutive instructions. The fields addr (starting address) and msel (memory bank selection) in address generators AG 0, 1, and 2 are modified most often (probability over 40%). Hence, by making just these operations faster the performance of the ASVP could be improved. The other address generators and the DFU fields veclen and repetitions are not so critical for performance.

5.2 Comparison to the VLIW Approach

The horizontal coding of v-instructions in ASVP is similar to the Very Long Instruction Word (VLIW) approach. In VLIWs the long instructions are divided into several slots; each slot can hold a single independent sub-instruction executed concurrently with the other ones on the hardware associated with
the slot. Often the slots are not equal in the range of supported sub-instruction types; e.g. a branch sub-instruction may be located only in the first slot.

In ASVP the v-instructions are also composed of several slots, dedicated to the address generators and to the DFU. These hardware modules are controlled by the sub-instructions and they work in concert with all the other. For instance, the sub-instruction controlling the DFU has no meaning without other sub-instructions that control address generators. Nor it is possible in ASVP to split a v-instruction into several VLIW instructions and fill unused slots with nops.

Previous works [NPKvL99, RS04, LXW07, CB09] proposed code compression for VLIWs to reduce energy consumption of the fetch stage and to improve the code density because memory in embedded systems is costly. The compression methods use a dictionary table that stores instructions (or parts of thereof) that are often repeated in the code. Instructions in the code are replaced by indices into the table. The hardware performs decompression either when code is loaded in I-Cache (so that the I-Cache stores decompressed code) or during instruction fetch in the processor.

5.3 New Dictionary Tables and APIs

In ASVP wide vector instructions are generated procedurally on-the-fly in an auxiliary scalar controller (8-bit processor). The scalar controller uses an independent ISA encoded vertically to save program memory space. In ASVP the PicoBlaze processor has 18 b s-instruction words, a reduction of $14 \times$ over the vector instructions. However, for this scheme to work efficiently the scalar controller must achieve sufficient generation rate so that new v-instructions are prepared in a forming buffer just before the previous one has ended. Otherwise the vector hardware will stall, waiting for a v-instruction.

A solution proposed and implemented here is to specialize the v-instruction issue hardware according to the prevalent properties of vector instructions used in real applications. The Figure 5.4 in the upper half shows the first six v-instructions of the Mandelbrot kernel whose source code is in Listing 4.3. The hardware contains five address generators configured via the 46b v-instruction slots dst, scr1, src2, src3, and idx. We see that the AG configuration data is highly repetitive. The same AG configuration corresponding to the vector ‘v1’ first appears in vi1 in the dst field and next in vi3 in the src1 field. Hence the AG configuration data can be stored in compact form in one place in a vector dictionary (‘Level 1’), and fetched to the proper field in the v-instruction forming buffer when needed.

The same compression scheme can be repeated once more: typically a v-instruction is just a 3-address vector operation along with an opcode. The other sub-fields in the 26b ‘oper’ field–the vector length and the number of repetitions–change infrequently. The 3-address vector operation is encoded in instruction dictionary table (‘Level 2’). The table stores 3 pointers to the vector dictionary that in turn specify the full AG configuration that must be loaded into the forming buffer.

A crucial insight is that of bandwidth available for different fetch strategies (Figure 5.6). The scalar controller must be able to create any instruction pattern in the forming buffer. The controller should have narrow vertically encoded instructions so that its program memory maps well to on-chip memory blocks. The practical result is a low bandwidth between the C.EX stage (Figure 5.5) and the buffer. In the implementation this bandwidth is 4 bit/cycle in the best case, but typically lower. The new special hardware that performs the pre-fetch from the Level 1/2 dictionary tables will achieve higher bandwidth; in the implementation the bandwidth from D.EX to buffer is 12 bits/cycle.

New Hardware

Based on analysis of the histogram a new hardware shown in Figure 5.6 was designed and implemented. The additional hardware comprises of a new configuration memory 512 x 32-bit and an automaton that
5.3. NEW DICTIONARY TABLES AND APIS

<table>
<thead>
<tr>
<th>Time (program steps)</th>
<th>dst AG</th>
<th>src1 AG</th>
<th>src2 AG</th>
<th>src3 AG</th>
<th>idx AG</th>
<th>oper</th>
</tr>
</thead>
<tbody>
<tr>
<td>vi1</td>
<td>v1 ag-cfg</td>
<td>x ag-cfg</td>
<td>x ag-cfg</td>
<td>ag-cfg</td>
<td>ag-cfg</td>
<td>vmult</td>
</tr>
<tr>
<td>vi2</td>
<td>v2 ag-cfg</td>
<td>y ag-cfg</td>
<td>y ag-cfg</td>
<td>ag-cfg</td>
<td>ag-cfg</td>
<td>vmult</td>
</tr>
<tr>
<td>vi3</td>
<td>v3 ag-cfg</td>
<td>v1 ag-cfg</td>
<td>v2 ag-cfg</td>
<td>ag-cfg</td>
<td>ag-cfg</td>
<td>vadd</td>
</tr>
<tr>
<td>vi4</td>
<td>v4 ag-cfg</td>
<td>v3 ag-cfg</td>
<td>c4 ag-cfg</td>
<td>ag-cfg</td>
<td>ag-cfg</td>
<td>vcmplt</td>
</tr>
<tr>
<td>vi5</td>
<td>v5 ag-cfg</td>
<td>i ag-cfg</td>
<td>maxit ag-cfg</td>
<td>ag-cfg</td>
<td>ag-cfg</td>
<td>vcmplt</td>
</tr>
<tr>
<td>vi6</td>
<td>v6 ag-cfg</td>
<td>v4 ag-cfg</td>
<td>v5 ag-cfg</td>
<td>ag-cfg</td>
<td>ag-cfg</td>
<td>vband</td>
</tr>
</tbody>
</table>

**Figure 5.4:** Compact representation of v-instructions using two-level dictionary tables.

accesses it. The configuration memory is split in two logical segments, also called *levels*. The first 256-word segment is a dictionary describing a layout of arbitrary 256 vectors stored in ASVP data banks. The layout of each vector is specified on 32 bits using the fields *msel* (*memory bank*, 2 bits), *addr* (*starting address within a bank*, 10 bits), and *len* (*vector length*, 10 bits, currently unused).

The second part (the higher 256 words) of the new configuration memory encodes dictionary of vector instructions for DFU. Each 32-bit word in this segment encodes three 8-bit pointers to the vector dictionary stored in the first segment that must be loaded in address generators AG0, AG1, and AG2, respectively. The remaining 8 bits in each word encode a vector operation that will be issued to DFU, such as VADD, VSUM etc. The encoding is summarised in Table 5.2.

**New Software APIs**

The new hardware is supported by two new API functions in the controller, pb2dfu_set_vector() and pb2dfu_start_insn(), listed in Table 5.3. The functions are categorized as *Level 1* and *Level 2* depending if they use information in the first segment or in both segments. For completeness, existing API functions listed in Table 4.1 are classified as *Level 0*. 
The process is initiated in the 8-bit controllers. Finally, it sends the op-code to the DFU and starts vector execution. The process is initiated in PicoBlaze by a single segment and obtains a word that encodes the configuration of AG0, 1, 2, and the DFU op-code. It requires 3x instructions that will execute in 6 cycles.

Figure 5.6: Controller in ASVP. The controller issues vector instructions to the VPU, but with the maximal bandwidth of only 8 bits per 2 clock cycles (8b/2cc). Alternatively, a subset of configurations can be preloaded from a new configuration table having a higher bandwidth of 12 bits per cycle.

Figures 5.7, 5.8, and 5.9 schematically depict the execution flow of the Level 0, 1, and 2 functions, respectively. Level 0 is illustrated by the API function pb2dfu_set_fulladdr(ag, msel, addr). The function directly sets the addr and msel fields in address generator ag. In PicoBlaze the function requires 3x OUTPUT instructions that will execute in 6 cycles.

In Level 1 API, the function pb2dfu_set_vector(ag, vdi) indexes a row vdi in the first segment and transfers the configuration into the address generator ag. The process is initiated in PicoBlaze by a single OUTPUT instruction, and it is carried out by a dedicated automaton.

In Level 2 API, the function pb2dfu_start_insn(insn) indexes a row insn in the second segment and obtains a word that encodes the configuration of AG0, 1, 2, and the DFU op-code. It then fetches the configurations one by one from the first segment and loads them into the address generators. Finally, it sends the op-code to the DFU and starts vector execution. The process is initiated in PicoBlaze by a single OUTPUT instruction, and it is carried out by a dedicated automaton.

The use of a dedicated automaton for carrying out the Level 1 and 2 schemes is crucial. In PicoBlaze...
5.4. EVALUATION

<table>
<thead>
<tr>
<th>Segment (level)</th>
<th>Field</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>msel</td>
<td>2b</td>
<td>memory bank selection</td>
</tr>
<tr>
<td></td>
<td>addr</td>
<td>10b</td>
<td>starting address within a bank</td>
</tr>
<tr>
<td></td>
<td>len</td>
<td>10b</td>
<td>length (unused)</td>
</tr>
<tr>
<td>2</td>
<td>opcode</td>
<td>8b</td>
<td>DFU instruction op-code</td>
</tr>
<tr>
<td></td>
<td>ag0</td>
<td>8b</td>
<td>pointer to vector in segment 1, to load in AG0</td>
</tr>
<tr>
<td></td>
<td>ag1</td>
<td>8b</td>
<td>pointer to vector in segment 1, to load in AG1</td>
</tr>
<tr>
<td></td>
<td>ag2</td>
<td>8b</td>
<td>pointer to vector in segment 1, to load in AG2</td>
</tr>
</tbody>
</table>

Table 5.2: The fields encoded in the new configuration table.

<table>
<thead>
<tr>
<th>Level</th>
<th>API Function in C in sCPU</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>void pb2dfu_set_vector(uint8_t ag, uint8_t vdi);</code></td>
<td>Load the operand ag with the vector defined at the index vdi in the Level 1 table (Figure 5.8)</td>
</tr>
<tr>
<td>2</td>
<td><code>void pb2dfu_start_insn(uint16_t insn);</code></td>
<td>Load AGs 0, 1, and 2 indirectly through the Level 2 and Level 1 tables, and start the operation defined in the Level 2 table (Figure 5.9).</td>
</tr>
</tbody>
</table>

Table 5.3: The C API functions for accessing the vector unit.

Each OUTPUT instruction deposits 8 bits of information in an external register and takes 2 clock cycles: the peak I/O bandwidth is thus 8b/2cc or 4b/cc. Real I/O bandwidth is lower because not all instructions can be OUTPUTs. In the implementation the automaton has bandwidth 12b/cc.

Improved Toolchain

Software programming toolchain was updated to generate the contents of the new configuration table. The new toolchain flow is shown in Figure 5.10. At the beginning the application algorithm is expressed as a data-flow graph, shown on the left-hand side of the picture. The boxes (green) represent vector variables and the ovals (violet) represent vector operations. This data-flow graph is implemented in a textual source code, as shown in the middle part of the picture. The source code comprises three parts (from top to bottom in the picture):

(a) vector layouts (green),
(b) partial vector instructions (violet), and
(c) controller firmware (yellow; for PicoBlaze).

The sources (a) and (b) are processed by a special assembler that generates a binary image of the configuration memory and a header file in the C language that assigns physical addresses to symbolic names used in the source codes. The firmware source (c) is compiled by the C language toolchain.

5.4 Evaluation

The proposed scheme is evaluated using the Mandelbrot fractal set kernel and the image segmentation kernel. The original kernels were modified to take advantage of the new APIs.
CHAPTER 5. REDUCING INSTRUCTION ISSUE OVERHEADS

Figure 5.7: Level 0: Direct access to DFU/AG configuration registers from C API.

Figure 5.8: Level 1: Indirect pre-load of a vector definition into an address generator (AG).

Figure 5.9: Level 2: Execution of a DFU instruction which preloads AGs 0-2 and the op-code field.
Figure 5.10: Toolchain supporting the new configuration table. The configuration memory (on the right) is 512 x 32bits with two sections. The first 256-word segment contains vector layouts for the Level 1 APIs. The second segment contains partial vector instructions for Level 2 APIs. The contents is specified in a text source code and assembled.
CHAPTER 5. REDUCING INSTRUCTION ISSUE OVERHEADS

Performance of the Mandelbrot Fractal Set Kernel (MF-set)

Figure 5.11 compares performance of the original Mandelbrot fractal kernel using only Level 0 API, and the improved version that takes advantage of the new configuration memory by using the Level 1, 2 APIs. (The ‘Level 0’ results are copies from Figure 4.9.) The bold numbers above columns show the reduction of the execution time in percentages (e.g. ‘-6%’), and the reduction of controller overheads (e.g. ‘9×’).

The new approach reduces controller overheads by 9× and the total execution time by 9% on average (speed-up 1.09×).

Performance of the Image Segmentation Kernel (ImgSeg)

Figure 5.12 compares performance of the image segmentation kernel:

(a) original version using only Level 0 API (a copy of Figure 4.12),
(b) doubling the controller performance (2× f0) (a copy of Figure 5.2),
(c) and using the new configuration table and APIs.

As shown in the previous chapter, using only the original Level 0 API the kernel execution time is largely determined by the controller. For example, in Spartan 6 A4M4@125M technology the VPU is idle 52% of time. The proposed configuration APIs improve the absolute execution times tremendously: by 30% on average, and almost by 2× (48%) in Spartan 6 A4M4@125M technology. In the latter case the sCPU overhead itself is reduced by 13× from 52% of the absolute execution time (60.34μs of 116.1μs) down to 8% (4.58μs of 60.7μs).

The proposed scheme is better than the alternative scheme–doubling controller’s raw performance. It reduces controller overhead by 2× to 3× relative to that alternative.

A positive side-effect of the new APIs is kernel code size reduction. The original ImgSeg kernel had to be coded and optimized manually in assembly language because compiler-generated code was simply too large for the controller program memory (1024 words in PicoBlaze 3), and even then only 58 words of the program memory remained free. The version using new APIs is written entirely in the C language (and still 236 words of the program memory remain free).

Resource Utilization

Hardware resource costs of the proposed scheme is shown in Figure 5.13 for Spartan 6, Virtex 6 and Kintex 7 FPGAs. The first bar in both plots represents area in FPGA slices of the original ASVP core, the second bar represents the area of the extended core. The largest increase in resource count (+25%) is in the controller peripheral circuit (marked ‘B’ in the plot) where the new configuration automaton is located. Overall, in Spartan 6 the core area increased by 5%, in Virtex 6 by 7%, and in Kintex 7 by 3%.

Efficiency

An efficiency of the additional hardware is a ratio of the speed-up over the newly occupied chip area. The worst speed-up is 1.09×, the worst area increase is 1.07×, hence the efficiency is 1.09/1.07 = 1.02. As this efficiency is higher than 1.0× the new hardware function delivers a super-linear improvement.
Figure 5.11: The MF-set kernel: original (Level 0 API only) vs. new (all APIs). The bold numbers above columns show the reduction of the execution time in percentages (e.g. '-6%'), and the reduction of controller overheads (e.g. '9x').
Figure 5.12: The ImgSeg kernel: (a) original (Level 0 API only); (b) doubling $f_0$; (c) new APIs. The bold numbers next to columns show the reduction of the execution time in percentages (e.g. -15%), and the reduction of controller overheads (e.g. '9x') relative to the original version (a).
Figure 5.13: New configuration scheme: resource costs in FPGA slices compared to the original cores.
5.5 Summary

The chapter has provided the second thesis contribution: (2) A method for achieving high-frequency instruction issue using controller specialization in an architecture with wide horizontal instructions generated on the fly. The ease of firmware development is the reason why an off-the-shelf industry-standard controller PicoBlaze 3, with existing toolchain support, was used instead of a new tailor-made microprogrammed controller. One of the conclusions in Section 4.5 was that the standardised controller unfortunately does not provide for the required performance. After further analysis the solution was proposed in this chapter to partially specialize the v-instruction issue hardware. The long, horizontally encoded, vector instructions are stored in a compact, two-level form in a new hardware memory. The v-instruction issue is still fully under the control of the general-purpose controller.

Improvement in v-instruction issue rate.
Experiments show that the controller overhead, i.e. the time not overlapped by a useful computation in VPU, is reduced typically $9 \times$ (max. $13 \times$). The absolute execution time is reduced by 10% (min 6%, max 15%, speed-up $1.1 \times$) for the Mandelbrot application, and 30% (min 16%, max 48%, speed-up $1.3 \times$) for the ImgSeg application.

Small additional hardware resource cost.
The resource cost of the additional hardware is around 100 slices in Virtex 6 and Spartan 6, and 55 slices in Kintex 7 (Figure 5.13). This is around 7% and 3.3%, respectively, of the total area of the core in FPGA. The new hardware is speed-up/area efficient.

Firmware code size reduction.
Controller firmware size is reduced because long linear code sequences that push data into the v-instruction forming buffer are replaced by a short sequence (one or two instructions) that start the fetching process from dictionary. Depending on the total occupancy of controller program space the code size reduction may be crucial for fitting a kernel in the firmware memory.
II Dataflow at the Gate Level
This chapter defines (Section 6.2, Section 6.3) a new method for hardware synthesis of Synchronous Petri nets (SPN). The first section (Section 6.1) gives basic definitions. Evaluation is given in Section 6.4 in a case study of a transputer serial-link controller.

The summary of Chapter 4 has noted two conceptual issues:

1. Data-path is easy to design, but difficult to develop in practice.

2. Efficiency may be lost due to imperfect vectorization of algorithms (e.g. the redundant computations in Mandelbrot) when only basic vector operations are used.

The first issue suggests a need for a higher-level specification technique—a language or a model—for the design and the automated implementation of custom data-paths. The second issue suggests that scalar-oriented data paths would deliver better performance if the problems associated with fine-grained control of many independent elementary operations could be surmounted.

Application domains considered in this thesis are mainly DSP and image processing. In these domains data is processed in large quantities, individual functions are applied over streams of data. Therefore the specification technique should be data-driven. Furthermore, in Chapter 2 we have seen that latencies in interconnect at the silicon or FPGA fabric level are difficult to predict in the design time, hence any new specification technique shall abstract away from exact cycle-accurate timing.

Synchronous Petri nets (SPN) were selected as the base model for the data-driven specification. Controllers and finite state machines (FSM) are typically specified using state graphs with transitions. The disadvantage of the approach is a limited support for concurrency. A controller is decomposed into cooperating FSMs, but the individual FSMs are sequential. The concurrency in the controller is thus limited by the decomposition step. Controller specification using Petri nets has the advantage that decomposition is either not needed, or it could be performed automatically by a tool.

6.1 Petri Nets: A Multitude of Variants (Definitions)

The definitions in this section are based on [GV03] and [Kub05]. Petri nets are built around the following five principles: duality, locality, concurrency, graphical representation, algebraic representation.

1 **Duality**: There are two disjoint sets of elements: *P-elements* (‘places’, states), and *T-elements* (‘transitions’). Real-world entities interpreted as passive elements are represented by *P-elements*,...
e.g. conditions, places, states. Real-world entities interpreted as active elements are represented by T-elements, e.g. events, transitions, actions.

II **Locality**: The behaviour of transitions exclusively depends on its locality. Locality includes all input and output places (pre- and post-conditions) and the transition itself.

III **Concurrency**: Transitions having disjoint locality occur independently (concurrently).

IV **Graphical representation**: P-elements (places) are represented by rounded graphical shapes (circles, ellipses); T-elements (transitions) are represented by edged graphical shapes (bars, rectangles). Arcs (edges) connect each T-element with its locality, i.e. a set of P-elements.

V **Algebraic representation**: For each graphical representation there is an algebraic representation containing the same information.

Figure 6.1 shows an example of the most commonly used type of Petri net: a place/transition net (P/T-net) with weighted arcs and indistinguishable tokens. The net models a well-known chemical reaction \(2H_2 + O_2 \rightarrow 2H_2O\). The picture illustrates the firing rule: two tokens (●) in each input place in (a) show that there are two units of \(H_2\) and \(O_2\) available. The transition \(s\) is enabled because token counts in pre-places satisfy the weight guards on both input arcs. After firing \(s\) the marking changes to (b), and the transition is no longer enabled. Output weight at \(s\) has caused the generation of two tokens in the \(H_2O\) place. Tokens are indistinguishable (anonymous), and only their count at a place matters.

A PN is said to be **finite** if the sets of places and transitions are finite. A PN is said to be **\(k\)-bounded** if in all reachable markings no place has more than \(k\) tokens. A 1-bounded PN is called a **safe PN**.

**Prioritised Petri nets** add priorities to transitions. A transition cannot fire if a higher-priority transition is enabled (i.e. can fire).

In **Petri nets with place capacities** some or all places have capacities assigned. Transition relation (firing rule) is the usual one restricted to the markings in which each place with a capacity has at most that many tokens.

A more powerful type of Petri nets are **Coloured Petri nets (CPN)**. In CPN tokens are distinguishable by colour (e.g. \(\circ, \varnothing, \Theta, \ldots\)). Colours can be thought of as data types. All places, transitions and arcs are marked by colour sets (colour domains). Colour domains in arcs specify that only tokens of given colour(s) can move over the arcs.

A common extension of Petri nets is the addition of **inhibitor arcs**. An inhibitor arc connects a place and a transition and it is represented by a dashed line with a small circle at the transition. The net in Figure 6.2 shows an example of a producers-consumers system. The system A has a higher priority over the system B: the consumer A can proceed whenever the buffer A has tokens, but the consumer B can consume tokens only when buffer A is empty and buffer B holds tokens. The arc from \(p_3\) to \(t_7\) is the inhibitor arc. Inhibitor arcs disable the transition when the input place has a token and enable the
transition when the input place is empty. This adds the ability to test ‘zero’, i.e. the absence of tokens in a particular place. No tokens are moved over inhibitor arcs when transitions fire.

6.1.1 Synchronous Petri Nets (SPN)

In common types of Petri nets the firing of transitions is asynchronous. Each firing is an individual event. The events are interleaved in the discret time, even if some of them could be executed in true concurrency.

Synchronous Petri nets (SPN) have a notion of global clock in the environment (SynPN in [HL93], SIP-net in [FAP97]). All firing events are synchronised to the global clock ticks. For an external observer firing events that occur during the same clock-tick are truly simultaneous.

In synchronous Petri nets the environment is considered as synchronous and the internal actions have null time duration. The models evolve according to the pace of the synchronous environment, i.e. according to the global clock. The system internal time is a sequence of discret instants. Outside these instants, system inputs are not enabled. SPNs are fully synchronised because every transition firing is associated with one clock-tick event.

The synchronous interpreter guarantees a complete and coherent evolution of the system. For each basic clock-tick a cycle of transition firing is started, bringing the net from an initial stable state to a new stable one compatible with the external events (input signal states) that are present at the same instant. The duration of the firing cycle is considered as null. All the transitions fired in one cycle are
assumed to be perfectly synchronized, although they are, in the interpreter, being fired in sequential order [HL93].

Synchronous Petri nets are a suitable formalism for modelling of parallel controllers and their synthesis into hardware [FAP97]. Interface between a model in Petri net and the hardware is realised by signals. Output signals are attached to places and/or transitions to represent actions that the PN carries out. Input signals are parts of logic expressions assigned to transitions as guards.

### 6.2 Activation Nets: A New Method for SPN Synthesis

*Activation nets* (AN; also ‘activation graphs’) are a new tool I have devised to enable the translation of the SPN specification into synchronous hardware. The AN computes which *transitions* in the corresponding PN fire in a clock-tick. The net is a finite static graph\(^1\) of logic operators (and: \(\land\) or: \(\lor\), not: \(\neg\)) in the standard Boolean set \(\mathbb{B} = \{0, 1\}\). There is also a special *flip-flop* (ff) operator that represents a clock-synchronous register. Formal definitions are provided separately in Section C.1.

![Diagram](image.png)

**Figure 6.3:** A single PN *place* and its activation net.

#### 6.2.1 Places

Activation nets are constructed by expansion of PN components (i.e., places, transitions) using pre-defined AN fragments (patterns). Figure 6.3.a shows a *place* in an SPN net, with one input \((s_1)\) and one output transition \((s_2)\). The symbol ‘\(v\)’, \(v \in \mathbb{B}\), inside the *place* symbolizes dynamic marking of the place. When \(v = 0\) the place is empty, when \(v = 1\) the place holds a token. *Transitions* \((s_1, s_2)\) are mapped to *wired-and nodes* which are depicted as ‘\(\land\)’. Besides wired-ands there are also *or nodes* (\(\lor\)), *negation nodes* (\(\neg\)), and *clock-synchronous flip-flops* ff. (In flip-flops graphical symbols the longer sides are used for data input/output, the shorter side is used for clock enable input. The common global clock-ticks input is not shown.) *Wired-and nodes*, *wands* in short, are ubiquitous in the approach.

The activation net in Figure 6.3.b is acyclic, hence it can be directly used as a logic circuit. To clarify the function of the net the Figure 6.1 lists all possible states of the logic. The transitions \(s_1, s_2\) are driven by auxiliary signals \(s'_1, s'_2\) representing other parts of the net; the value 0 means that the transition is inhibited. In the table the value 1 in \(s_1, s_2\) marks states when the transition is going to fire at the next clock tick. The column \(v_{next}\) is the new marking after the next clock-tick: bubble (\(\bigcirc\) or token (●), also represented by 0 and 1 in logic.

Of particular interest is the case when the place holds a token \((v = 1)\) and both transitions are enabled \(s_1 = s_2 = 1\) (the last row in the table). The AN logic disables the input transition \(s_1 = 0,\)

---

\(^1\)The word ‘arc’ is reserved for PN graphs, and ‘edge’ for AN graphs.
6.2. ACTIVATION NETS: A NEW METHOD FOR SPN SYNTHESIS

![Figure 6.4: Activation net of one place.]

fires the output transition \( s_2 = 1 \) (at the next clock-tick) and enables the flip-flop \( CE = 1 \). At the next clock-tick the flip-flop becomes empty, but that is correct because the output transition was enabled and the token has moved to the next place.

### 6.2.2 Transitions

*Transition nodes* in Petri nets may have multiple input and multiple output arcs connected to *places*. All the input and output arcs must be enabled simultaneously for the transition to fire at the next clock-tick. Input arcs are enabled when the input place holds a token (\( \bullet \)). In capacity=1 PN the output arcs are enabled when the output place is empty (\( \bigcirc \)). In addition there may be special *inhibitor* arcs that disable the transition if the source place of the inhibitor arc contains a token.

SPN transitions are mapped to the wired-and (\( \wedge \) *wand*) nodes in activation nets. A single SPN transition node must be represented by a single *wand* node in AN. The AN fragment for a place in Figure 6.3.a has two transitions \( s_1, s_2 \) embedded in it. Connecting two places via a common transition node is done by *overlapping* (unifying) these *wand* nodes in the activation net.

The technique is illustrated in Figure 6.5. The picture shows a four-place SPN pipeline and the corresponding activation net. The place \( v_1 \) has no ingoing transition in SPN, hence the *wand* \( s_1 \) is driven to constant 0 in AN. The transition \( s_5 \) has no outgoing arc, it acts as a *drain*. In the AN the *wand* \( s_5 \) is driven by 1, although this has by itself no effect because \( x \wedge 1 = x \). By extension, a *wand* without any input edge is assumed to have the constant value 1.

The evolution of the four-place pipeline in time is shown in Figure 6.2. At clock-tick \( t_1 \) all places are initialized with a token (\( \bullet \)). The table lists values of signals in the AN over 8 clock ticks (\( t_1 \) to \( t_8 \)). In transitions \( s_1 - s_5 \) the value 1 indicates that the transition is enabled. The actual ‘firing’ takes place at the next clock tick (active edge of the clock signal in synchronous hardware). Hence the AN emulates behaviour of the synchronous Petri net shown in Figure 7.1.b.

The technique of overlapping *wands* allows easy construction of ANs for SPN transitions with any number of input and output arcs. Figure 6.6 illustrates this for a transition \( s_3 \) that has 2 input and 2 output arcs. The transition will fire *if and only if* the upper two places contain a token (\( v_1 = v_2 = 1 \)) and the lower two places are empty (\( v_3 = v_4 = 0 \)). In that case \( s_3 = 1 \) and the transition fires at the next clock tick, otherwise \( s_3 = 0 \). The AN was constructed by composition from the pre-defined fragment of an individual place.

<table>
<thead>
<tr>
<th>( v )</th>
<th>( s'_1 )</th>
<th>( s'_2 )</th>
<th>( s_1 )</th>
<th>( s_2 )</th>
<th>CE</th>
<th>( v_{next} )</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ( )</td>
<td>0 (any)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1 (( \bigcirc ))</td>
<td>no change</td>
<td></td>
</tr>
<tr>
<td>0 ( )</td>
<td>1 (any)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1 (( \bullet ))</td>
<td>loading</td>
<td></td>
</tr>
<tr>
<td>1 (( \bullet )) (any)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1 (( \bigcirc ))</td>
<td>unloading</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: State space of the activation net of one place.
6.2.3 Alternate Choices

In all the examples given so far there was at most one input and one output arc in each place in PN. Figure 6.7 shows two examples of Petri nets that violate this assumption: in picture (a) there is an alternate choice between $s_1$ and $s_2$ firing, in (b) there is similar choice between $s_3$ and $s_4$. In the standard, asynchronous Petri nets transition firings are individual events, each separate from all other. When several transitions happen to be enabled at the same time they will not fire simultaneously. Rather, the firings are non-deterministically serialised, perhaps based on probabilities. In Figure 6.7 one of the transitions $s_1$ or $s_2$ (or $s_3/s_4$) would fire earlier than the other, thus resolving the indeterminism.

In synchronous Petri nets the situation is more severe. All transitions that are enabled always fire at the next clock tick; if the transition should not fire it shall not be enabled in the first place! The two situations in Figure 6.7 called ‘conflict’ and ‘overflow’ should be resolved by deterministically disabling one of the transitions, perhaps via a priority scheme. In previous work [FAP97] the authors dismiss networks that may evolve conflicts or overflows as incorrectly formed.

The solution is to define a completely new kind of a transition node called a ‘choice’. There are two types of the choice transition, as depicted in Figure 6.8:

1. the ‘switch’ choice has a single common input arc and multiple alternate outputs (think: switch to mult. places), and

2. the ‘select’ choice has a single common output arc and multiple alternate inputs (think: select from mult. places).
6.2. ACTIVATION NETS: A NEW METHOD FOR SPN SYNTHESIS

Figure 6.6: General multi-input multi-output transition in SPN \( s_3 \) is constructed in AN by using a common \textit{wand} node.

Figure 6.7: Two undesirable situations in synchronous Petri nets [FAP97].

Figure 6.8: The ‘switch’ and ‘select’ choice transitions deterministically resolve conflicts.

The function of choice transitions is conceptually carried out in null time (it is a combinatorial function in hardware). The switch (select) choice transition guarantees that at most one output (input) alternate arc is enabled at any time, iff the single common input (output) arc is enabled as well. This is in contrast to the normal PN transition node that guarantees all its arcs being disabled or enabled simultaneously.

In SPN with all places having the implicit capacity for 1 token only there may be at most one incoming and/or one outgoing alternate arc firing for each place at a clock-tick. At most one outgoing alternate arc can fire because a full place holds only 1 token, and at most one incoming alternate arc can fire because an empty place has capacity only for 1 token. Choice transitions conveniently centralize the decision which of the alternate arcs should fire next. If a place would have more than one incoming and/or outgoing arc the select/switch choice node is used in the PN instead of routing the alternate arcs directly to the place node in the graph. A great advantage of the approach is that by using choice transitions all places in the net will have (at most) one input and one output arc. This simplifies binding of data paths to Petri net, which is a matter discussed in Chapter 7.

Declarative Specification of Choices

Choice transitions can be controlled either explicitly or implicitly. Explicit control simply means that one of the alternate arcs is preselected by an independent external condition and all the other are inhibited. The condition signal could be for example a presence of a token in some other \textit{place} or a specific data value in the associated data path.
If different conditions are assigned to alternate arcs in a choice node only one of them may be active at any time. This too is rather easy to force: the conditions are ordered by static priority, activation of a higher priority condition automatically inhibits all the lower priority ones.

The implicit control is more interesting. Alternate arcs in a choice node are ordered by priority. If several alternate arcs could fire concurrently the choice node enables the highest priority one and inhibits all the other. Figure 6.9 schematically shows the switch and select choices with implicit, priority-driven control. The numbers 1, 2 indicate priorities: lower number (1) means higher priority. In AN the arcs connected to the choice node are represented by wands \( s_a, s_b, s_c \). (The convention is to assign the letter \( a \) or the number 0 to the common arc, then in the order of decreasing priorities.)

The picture does not show a proper AN fragment for the SPN choice node yet. The table in the picture hints that there are three distinct states of the wands \( (s_a, s_b, s_c) \) that the choice node allows to occur, and the other 5 are forbidden. The default state \((0, 0, 0)\) happens any time when the choice transition cannot fire. In the state \((1, 1, 0)\) a token is transported along the \( s_a - s_b \) arcs, in \((1, 0, 1)\) it is transported along the \( s_a - s_c \) arcs.

The switch and select choices behave identically with respect to enabling or inhibiting their transition arcs (wands \( s_x \) in AN). The table in Figure 6.9 is valid for both types of choices. In PN the switch and select nodes differ in the direction of their arcs, but in AN the fragment is identical.

![Figure 6.9: Declarative specification of a ‘choice’ with implicit (built-in) priority-based conflict resolution.](image)

**Computing the Implicit Choice by the AN**

The table in Figure 6.9 at the bottom declares which states of the choice node are allowed and which are not, but it does not say how to select one of them. The numbers 1, 2 indicate priorities: lower number (1) means higher priority.

In the end the AN fragment corresponding to the new PN choice transition node must be defined. The AN fragment is a combinatorial boolean function that outputs the three-tuple \((s_a, s_b, s_c) \in \mathbb{B}^3\), which eventually evaluates to one of the three allowed states listed in the table. But what are the inputs to the looked-for AN fragment if \((s_a, s_b, s_c)\) are the outputs? We will postpone the full answer a little bit, and just symbolically identify the inputs as \((s'_a, s'_b, s'_c)\) for now. (For simplicity here only
6.2. ACTIVATION NETS: A NEW METHOD FOR SPN SYNTHESIS

The AN fragment for the PN choice transition node is a function from \((s'_a, s'_b, s'_c) \rightarrow (s_a, s_b, s_c)\). The input three-tuple \((s'_a, s'_b, s'_c)\) represents an ‘intention’ of the surrounding AN about which transition arcs could fire if there was no choice node involved. Based on the information given so far about the intended behaviour of the choice node the function can be constructed as a logic table—see Figure 6.3. As there are 3 input boolean variables the table should contain 8 rows, but there is a couple of important don’t-cares, marked ‘\(\times\)’ in the table.

- Row 1: When the common wand \(s'_a\) is inhibited (e.g. no token is available at a switch input) all the alternate wands must be disabled \((s_b = s_c = 0)\).
- Row 2: When all alternate wands \(s'_b, s'_c\) are inhibited (e.g. all inputs of a select have no token) the common wand must be disabled \((s_a = 0)\).
- Row 3: When the common wand \(s'_a\) together with the lowest priority alternate wand \(s'_c\) are enabled and all the other higher priority wands are disabled \((s'_b = 0)\), this configuration is allowed with no change.
- Row 4: When the common wand \(s'_a\) together with some (one or more) alternate wands(s) are enabled, all the alternate wands with the priority lower than that of the highest enabled one \((s'_b = 1)\) are inhibited \((s_c = 0)\).

The description above allows easy extension to any number of alternate arcs in the SPN choices. Figure 6.10 shows the AN fragment directly constructed out of the table discussed above. The same AN fragment is used for both the switch and select choices, even though that in SPN the directions of transition arcs are reversed.

### Splitting the wands

The AN of the place node has only a single wand per each SPN arc (Figure 6.3): one wand corresponding to SPN input arc, the other wand on the other side to SPN output arc. The AN of the choice node has two wands per each SPN arc (e.g. \(s'_a\) and \(s_x\)).

Any wand can be split in two by the transformation in Figure 6.11. A new wand \(s'_x\) is created and all the original incoming edges are redirected to it: \(s'_x = a \land c\). The original wand \(s_x\) is rewritten to depend on the new wand: \(s_x = s'_x\). After wand splitting there is no change in the function of the AN, the transformation is reversible. However, the two wands \(s'_x\) and \(s_x\) are separate now. New in-edges may be later introduced to \(s_x\) that will make it logically different from \(s'_x\).

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(s'_a)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>×</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.3: Function table of the choice AN fragment. Symbol ‘\(\times\)’ is a don’t-care value.

Figure 6.10: The choice AN fragment constructed from the table on the left.
The splitting transformation is used to create two wands out of one, exactly what is needed for the choice AN. The whole process is best illustrated by an example. Figure 6.12 shows a simple SPN with 3 places and the select choice node. It explicitly shows the transitions $s_a$, $s_b$ and $s_c$ for clarity. The AN on the right is constructed as follows: First, the AN fragments for the three places are instantiated. Second, the relevant boundary wands of the fragments are split. Third, the AN fragment for the choice transition is instantiated, using the newly split wands.

The behaviour of the SPN and the AN is shown in Table 6.4 by listing all eight possible markings of the three places. At rows 2 and 3 we see a transfer of a token from the place $v_2$ and $v_1$, respectively, to the place $v_0$. At row 4 both places $v_1$ and $v_2$ hold a token, but the former place has a higher priority in the choice, hence the competing transfer from $v_2$ is implicitly inhibited ($s_c = 0$). At rows 5–8 the output place $v_0$ is full, hence no new token can be admitted in it.

### Explicit Control of Choices

Explicitly (externally) controlled choices have an additional input signal: a condition vector $\vec{c} \in \mathbb{B}^n$, where $n$ is the number of alternate arcs, $n \geq 2$. The individual boolean bits in the condition vector $\vec{c} = (c_1, c_2, \ldots, c_n)$ correspond to alternate arcs $(s_b, s_c, \ldots)$. When a bit $c_i$ is 0 the particular alternate arc is inhibited. To ensure that at most one alternate arc is selected in a choice node at a time the bits in the condition vector $\vec{c}$ are prioritized from left to right: lower-indexed $c_i$ has the higher priority, $p(c_i) > p(c_{i+1})$ where $p()$ is a priority predicate.
6.3. SYNTHESIS TO HARDWARE

Using *Activation Nets* a direct synthesis from SPN specification to RTL hardware is a straightforward matter. Input SPN diagram is transformed into the activation net by substituting the appropriate AN fragments for SPN components. Care must be given to correctly overlap (unify) the boundary *wands* when connecting up the AN fragments, and to correct *wand* splitting. The resulting AN graph shall be acyclic. The graph represents a spatial digital circuit composed of the *and/or/not* gates and flip-flops. Technically, the digital circuit is a Mealy-style automata with synchronous clock and with one flip-flop per each PN *place*. It is printed in a technology-independent VHDL code which is a source for standard synthesis CAD tools.

### 6.3.1 Non-Autonomous Petri Nets

Basic Petri nets that have no connection to environment are *autonomous*. Once set in motion their evolution depends only on the initial marking and on the internal firing rules (that may incorporate
indeterminism). For controller synthesis a non-autonomous Petri net that interacts with its environment is needed. The interaction is mediated via I/O signals that tap in the net. Figure 6.14 shows that transitions can be controlled via an ‘enable!’ command and tested via a ‘fires?’ predicate, and places can be tested via a ‘has-token?’ predicate. In AN (Figure 6.14.b) these signals connect to the wands that represent the transitions, and to the flip-flops in the place, respectively.

The ‘enable!’ and ‘fires?’ signals in transitions initially both connect to the identical wand, as depicted below in Figure 6.14 in $s_1$. If that wand is later split (by the transformation shown in Figure 6.11) the signals will refer to distinct wands.

![Figure 6.14: Predicates (has-token?, fires?) and commands (enable!) that provide the interface between PN or AN and the environment.](image)

### 6.4 Case Study: Transputer Link Controller

To clarify the concepts the hardware synthesis of a transputer serial-link controller using the synchronous Petri net is demonstrated. This example was selected because it is relatively simple, yet non-trivial. The same example was used in [FAP97]: here it allows to contrast the proposed approach to the previous work.

![Figure 6.15: Transputer link adapter.](image)

The transputer serial link adapter is depicted in Figure 6.15. Transputer links (LinkOut, LinkIn) are clock-synchronous bit-serial wires. Two wires are needed for full-duplex communication. Each data packet is transmitted as two start bits, 8 data bits, and one stop bit. Receiver acknowledges a packet by sending a start bit followed by a stop bit. The acknowledgement can be sent as soon as the packet header is received. Data packets and acknowledgement packets can be interleaved on the wires, hence continuous full-duplex communication is possible.
The adapter performs a serial-to-parallel (receiver) and parallel-to-serial (transmitter) conversions. Example timing diagrams are shown in Figure 6.16 and Figure 6.17. The transmitting parallel interface consists of the wires \( \text{IData}(0-7) \) (in), \( \text{IValid} \) (in), and \( \text{IAck} \) (out). The receiving parallel interface consists of the wires \( \text{QData}(0-7) \) (out), \( \text{QValid} \) (out), and \( \text{QAck} \) (in).

### 6.4.1 Original Solution (SIP-net)

The solution originally presented in [FAP97] is shown in Figure 6.18. The link controller is specified using synchronous Petri net (‘SIP-net’). The Petri net uses two macroplaces labelled \( \text{SerPar} \) and \( \text{ParSer} \) that realize the serial-to-parallel and parallel-to-serial conversions, respectively. The controller interacts with the environment using the input signals \( \text{IValid}, \text{QAck}, \text{LinkIn}, \text{IData} \), and the output signals \( \text{IAck}, \text{QValid}, \text{LinkOut}, \text{ShiftEnable} \). The input and output signals connect to the PN transitions: an input signal in 0 inhibits the transition, an output signal is 1 iff the transition is firing.

The PN can be divided in four parts:

1. Places \( p_1, p_2 \), and macroplace \( \text{SerPar} \) perform serial-to-parallel conversion (receiver) and acknowledge detection.

2. Places \( p_{28}, p_{11}, p_{12} \), and macroplace \( \text{ParSer} \) perform parallel-to-serial conversion (transmitter) and acknowledge sending.

3. Places \( p_{13} - p_{16} \) and \( p_{20} \) represent the second part of the receiver: generating the \( \text{QValid} \) and waiting for the \( \text{QAck} \) signals on the parallel interface.

4. Place \( p_{17} \) controls the sharing of the \( \text{LinkOut} \) signal for acknowledge generation.

### The Issues with the Original Solution

The PN synthesis method described in the previous work [FAP97] is limited to synchronous Petri nets that do not suffer from the conflict and overflow conditions (see Figure 6.7). Absence of these situations...
Figure 6.18: Transputer link controller in SPN (SIP-net) using the method in [FAP97]. Potential conflicts \((c_1, c_2)\) and overflows \((o_1, o_2)\) are marked by yellow filled areas.

Figure 6.19: Transputer link controller using the SPN/AN approach with alternate choice nodes \(S_1-S_4\).

Figure 6.20: Macro-places used in the transputer link controller.
must be checked separately by analysis or simulation tools, otherwise the generated hardware model could be incorrect.

In the transputer link controller there are two potential conflicts (\(c_1 = (p_2, t_{10}, t_2)\)), and two potential overflows (\(o_1 = (p_1, t_{10}, t_2)\), and \(o_2 = (p_{17}, t_9, t_6)\)), marked by the yellow-filled areas in Figure 6.18. Overflow happens when the Petri net is not ‘safe’ (= 1-bounded): if there is a reachable marking that puts more than one token in a place. In \(o_1\) this would be if \(t_{10}\) and \(t_2\) fire simultaneously, and in \(o_2\) if \(t_6\) and \(t_9\) fire simultaneously. Conflicts happen if more than one outgoing arcs and transitions in a place are enabled simultaneously.

The dynamic conditions for which the conflicts or overflows would happen are:

\[
o_1 = \neg p_1 \land p_2 \land p_{28} \land \neg \text{LinkIn} \land \neg p_{11} \land p_{29} \land p_{10} \land \neg p_{13} \quad (6.2)
\]

\[
o_2 = \neg p_{17} \land \neg p_{28} \land p_{27} \land p_{15} \land \neg p_{16} \quad (6.3)
\]

\[
c_1 = p_2 \land \neg p_1 \land \neg p_{11} \land p_{28} \land \neg \text{LinkIn} \land \text{LinkIn} \land \neg p_3 \quad (6.4)
\]

\[
c_2 = p_{17} \land p_{12} \land \text{IValid} \land \neg p_{18} \land p_{14} \land \text{QAck} \land \neg p_{15} \quad (6.5)
\]

In \(c_1\) in Equation 6.4 we easily see that the potential conflict will never occur because of the term \(\ldots \land \neg \text{LinkIn} \land \text{LinkIn} \land \ldots\), hence \(c_1 = 0\). In the remaining cases this may not be proved so easily. The authors in [FAP97] used a properties analyser that creates a reachability graph of the Petri net, and showed that overflows \(o_1\) and \(o_2\) will also in fact never occur. The conflict \(c_2\) is real. To prevent it they had to manually modify the Petri net of the link controller by adding an inhibitor arc from \(p_{14}\) to \(t_8\) as shown in Figure 6.21.

### 6.4.2 New Solution (SPN/AN)

The model of the controller using the SPN/AN approach is shown in Figure 6.19. The approach requires that all SPN places have at most one input and one output arc. To split or merge arcs in PN the alternate choice transition node must be used (Subsection 6.2.3). In the link controller there are four of them, labelled \(S_1\)-\(S_4\).

The potential conflict area \(c_1\) has been replaced by the the switch \(S_2\). The controller input signal \(\text{LinkIn}\) directly controls the choice in \(S_2\) which way a token from the place \(p_2\) will go. This representation of explicit choice is better readable that the use of inverted predicates in formally separate transitions (‘\(\text{LinkIn}\)’ in \(t_2\) and \(\text{LinkIn}\) in \(t_{10}\) in the original solution).

The area of the conflict \(c_2\) has been replaced by the priority switch choice \(S_3\). The higher priority, which belongs to the output labelled ‘1’, is given towards the transition \(t_5\). The behaviour is the same as previously but no manual fix-up using an inhibitor arc is needed. The CAD tool (Appendix D) generates the following guards for transitions \(t_5\) and \(t_8\):

\[
t_5 = p_{17} \land p_{14} \land \neg p_{15} \quad (6.6)
\]

\[
t_8 = p_{12} \land \text{IValid} \land \neg p_{18} \land p_{17} \land \neg (p_{14} \land \neg p_{15}) \quad (6.7)
\]
CHAPTER 6. CONTROLLER SYNTHESIS USING SYNCHRONOUS PETRI NETS

Since \( t_5 \land t_8 = 0 \), there could be no conflict between the two transitions.

An illustrative extract from the tool-generated VHDL code is shown in Listing 6.1. The code defines a new entity `tslink_ct` and its architecture. The architecture is a Mealy-style finite state machine: the process `all_ffs` at the end defines all flip-flop registers, the signal assignments above it are the next-state combinatorial logic. The signal assignments shown in the listing correspond to the Equation 6.6 and 6.7 that define the firing guards for transitions \( t_5 \) and \( t_8 \).

The controller circuit was translated to VHDL and synthesised using Xilinx XST 14.4 in Spartan 6 generation FPGA. The tool reported the utilization of 29 flip-flops (one per each place) and 68 LUTs.

6.5 Summary

The chapter has provided the third thesis contribution: (3) Structured and extensible approach for hardware synthesis of synchronous Petri nets.

Activation nets and the translation to hardware-synthesisable code.

A translation process from 1-bounded SPN with enforced capacity=1 using back-pressure to activation nets (AN) has been described. The AN is a graph that evaluates firing rules in SPN and computes which transitions will fire at next clock-tick. The AN is used for direct synthesis into clock-synchronous hardware. The approach is extensible because it is based on composition of AN fragments.

New alternate-choice transition nodes.

The synchronous Petri net (SPN) formalism has been extended with the alternate choice transition nodes. The alternate choice nodes allow convenient specification of priorities among multiple alternate arcs that, if enabled simultaneously, could cause dynamic conflict or overflow in the net. This is an improvement over the previous work which forbid SPNs with dynamic conflict or overflow. The use of `switch` and `select choices` instead of unstructured ad-hoc boolean guards on transition nodes improves readability of the graphical representation because the priorities among potentially competing arcs are explicitly declared.
entity tslink_ct is
  port (  
    clk : in std_ulogic;
    rst : in std_ulogic;
    LinkIn : in std_ulogic;
    IValid : in std_ulogic;
    QAck : in std_ulogic;
    IData : in std_logic_vector(7 downto 0);
    IAck : out std_ulogic;
    ShiftEnable : out std_ulogic;
    LinkOut : out std_ulogic;
    QValid : out std_ulogic);
end entity;

architecture rtl of tslink_ct is
  -- Undefined attributes (assuming inputs):
  signal S_LinkIn : std_ulogic;
  signal S_IValid : std_ulogic;
  signal S_IData : std_logic_vector(7 downto 0);
  signal S_QAck : std_ulogic;

  -- Defined attributes:
  signal FP121 : std_ulogic;
  signal FP141 : std_ulogic;
  signal FS40 : std_ulogic;
  signal N_ow_IAck_0 : std_ulogic;
  signal N_FT10J0_SSW_34 : std_ulogic;
  signal PP1st0 : std_ulogic;
  signal PP2st0 : std_ulogic;
  signal PP11st0 : std_ulogic;
  signal PP12st0 : std_ulogic;
  -- ...

begin
  -- I/O assignments:
  S_LinkIn <= LinkIn;
  S_IValid <= IValid;
  S_QAck <= QAck;
  S_IData <= IData;
  IAck <= N_ow_IAck_0;
  ShiftEnable <= N_ow_ShiftEnable_1;
  LinkOut <= N_ow_LinkOut_2;
  QValid <= N_ow_QValid_3;

  -- ... [cut]

  all_ffs: process (clk) is
    begin
      if rising_edge(clk) then
        if to_bool(rst) then -- initial marking:
          PP1st0 <= '1';
          PP2st0 <= '0';
          PP11st0 <= '0';
          PP12st0 <= '1';
          PP25.C0st0 <= '0';
          PP25.C1st0 <= '0';
          -- ...
        else
          if to_bool((FP21 or not PP2st0)) then
            PP2st0 <= FP121;
          end if;
          if to_bool((FP121 or not PP12st0)) then
            PP12st0 <= FT111;
          end if;
          if to_bool((FT111 or not PP11st0)) then
            PP11st0 <= FT10J0;
          end if;
          if to_bool((FP21 or not PP2st0)) then
            PP2st0 <= FT11;
          end if;
          if to_bool((FT11 or not PP1st0)) then
            PP1st0 <= FS10;
          end if;
          -- ...
        end if;
      end if;
    end process;
end architecture;
Chapter 7

Dataflow Hardware Synthesis from Petri Nets

This chapter extends the SPN synthesis method (described in the previous chapter) for dataflow systems. The first section (Section 7.1) analyses the problem called the bubble wall. In Section 7.2 the main idea is described and the Flow Transfer Level method is defined. Section 7.3 describes how the data paths are attached and Section 7.4 notes on some interesting algebraic properties of the method. Hardware synthesis is detailed in Section 7.5 and a case study—Mandelbrot fractal set—is presented in Section 7.6. Parts of this chapter were published in [X.4].

Figure 7.1: Timing of synchronous systems with global clock: RTL circuit (left), and 1-bounded Petri net (right). The time flows from top to bottom in the picture. The throughput is lower in the Petri net because bubbles are essential for token propagation.

7.1 Petri Nets and the Bubble Wall

There is one fundamental difference between normal or synchronous Petri nets and the standard clock-synchronous design as used in hardware, and that is the problem called ‘bubble wall’. Figure 7.1 com-
pares a timing of pipelines implemented in clock-synchronous RTL circuit (left) and in synchronous Petri net (right). In RTL pipeline all data tokens (\(\bullet\)) are shifted simultaneously to the right on each clock tick. In SPN pipeline each data token moves to the right when the next place is empty. All moves are synchronized to the common global clock. In RTL it takes 4 clock-ticks to move the token \(\bullet\) out of the pipeline, but in Petri net it takes 7 clock-ticks.

The reason for the lower throughput is that Petri nets are bubble-limited. Throughput is an average number of tokens transported over a link (wire, edge) per clock-tick. Figure 7.2 further illustrates this ‘bubble wall’ using a circular pipeline composed of six clock-synchronous registers (a) or six places (c). Using registers the throughput is determined only by the amount of tokens in the loop. Higher number of tokens means higher throughput. However, tokens in Petri net can jump to the next place in the loop only when the place is unoccupied; hence, empty places (bubbles) are equally important for throughput as tokens. In the example the circular Petri net achieves maximum throughput 0.5 tokens/tick with 3 tokens (and 3 bubbles). With 0 or 6 tokens in the loop the PN pipeline deadlocks.

The ‘bubble wall’ applies to Petri nets, both asynchronous (standard) and synchronous PN, and to any asynchronous token-based system. The precondition is that places have capacity for 1 token only.

### 7.2 Flow-Transfer Level: Scheduling Without Bubbles

Activation nets, presented in Section 6.2, allow direct translation of synchronous Petri nets (SPN) to the clock-synchronous hardware. Timing behaviour of the hardware generated by the SPN/AN synthesis is accurate with respect to the clock-tick events in the original SPN. This means that the synchronous hardware in fact emulates the ‘bubble wall’ of the Petri net.

The ‘bubble wall’ is caused by the locality principle, one of the five principles of Petri nets mentioned at the beginning of Section 6.1. The locality principle says that the behaviour of transitions depends exclusively on all input and output places and on the transition itself. Each transition ‘sees’ the state only of the places it directly connects to. If capacity=1 places are used a transition fires if the source place has a token and the destination place has a bubble. After that the destination place will hold a token and the source place a bubble. Another clock-tick is needed to bring a new token in the source place and to remove the token from the destination place before the transition could fire again. This limits the firing rate of each transition to 0.5 firings per clock-tick in synchronous PN.

To overcome the ‘bubble wall’ Petri nets must be taught to ‘think globally’. The critical situation just happens when two places connected by a transition both hold a token. The token in the place \(p_1\) in
7.2. FLOW-TRANSFER LEVEL: SCHEDULING WITHOUT BUBBLES

At the next clock-tick, \( p_1 \) could send its token to \( p_2 \) if \( p_2 \) will send its token to \( p_3 \) if \( p_3 \) will...

\[ \text{Figure 7.3: Synchronous flow of tokens requires global information about the state of Petri net.} \]

A small modification in AN fragments...

\[ \text{Contrib. (3): synthesis of synchronous Petri nets} \]

\[ \text{Contrib. (4): synthesis of dataflow hardware} \]

\[ \text{Figure 7.4: The thought process starts at SPN.} \]

Figure 7.3 could be transported to the place \( p_2 \) at the next clock-tick \textit{if and only if} the token in \( p_2 \) is simultaneously transported to \( p_3 \) which in turn requires that its token is moved to \( p_4 \), and so on.

Processing of global information to enable synchronous token flows can be realized in activation nets. Figure 7.4 shows the conceptual program. In Chapter 6 I started with the existing \textit{synchronous Petri nets} (SPN) semantics. Based on that the \textit{activation nets} (AN) were defined, a graph that computes when transitions fire and tokens move in SPN. It turns out now that a small modification in the AN fragment of the \textit{place} node allows to overcome the bubble wall in SPN. As this is a radical departure from Petri nets towards the RTL hardware the new approach is called the \textit{Flow Transfer Level} (FTL).

\subsection{7.2.1 ‘Cells’ Instead of Places}

The new component in FTL that replaces a \textit{place} is called \textit{cell}. Figure 7.5 depicts a temporary\(^1\) schematic symbol, the activation net fragment and the table of allowed arc states for the \textit{place} and \textit{cell} nodes.

Compared to the standard \textit{place} component the \textit{cell} allows for an additional transition state called \textit{flow-through}. In the flow-through state the \textit{cell} holds a token and both the input and output transitions are enabled; at the next clock-tick the existing token will be unloaded and a new token loaded simultaneously. In Petri net establishing a flow-through state in a node requires global information about the markings of many \textit{places} and it depends on the topology of the net. However, in activation nets the flow-through state is computed locally by the AN fragment shown in Figure 7.5.b. Table 7.1 lists the whole state space of the new AN fragment.

The behaviour of a pipeline composed of multiple \textit{cells} depicted in Figure 7.6 is shown in Table 7.2. Compared to a similar net of four \textit{places} in Figure 6.5 the FTL version requires half the clock-ticks to

\(^{1}\text{A new set of graphical shapes for all components is presented below.} \)
output all the tokens.

7.2.2 Composition in FTL

The current limitation of the approach is the necessity that activation nets must be cycle-free. A cycle in activation net is any directed closed path in the graph when the flip-flop nodes are not considered (Subsection C.1.2).

Composition in the domain of the synchronous Petri nets as discussed in the Chapter 6 is safe. The standard place nodes are safe because there is no directed path in their AN fragments between the input and output transitions. The composition of transitions and alternate choice nodes is defined such that AN cycles do not appear (Subsection C.1.1).

However, in FTL the cell nodes are not safe because there is a directed path from \( s_2 \) to \( s_1 \) (Figure 7.5). Presently any FTL net whose AN contains a cycle must be considered as not well-formed.

A canonical example of an FTL net that is not well-formed is a loop of cells depicted in Figure 7.7. Intuitively if both cells hold a token \( (v_1 = v_2 = 1) \) they could exchange their tokens at each clock-tick. However, there is a dependency cycle in the AN between the \( s_1 \) and \( s_2 \) wands, marked in red in the picture. Therefore the FTL net is considered ill-formed. (Note that a similar loop composed of two normal places in SPN is correct, although it experiences deadlock when both places hold a token.)

### 7.3 Attaching Data Paths

Any Petri net (SPN) can be directly used to model a controller (Chapter 6). Controller outputs are typically associated with places (an output signal is active when the place holds a token) or with transitions (an output signal is active when transition is enabled). Controller input signals are associated with transitions (inactive input signal inhibits a transition). The data path could be specified using standard abstractions such as RTL and then glued to the controller.
A more interesting approach, perhaps, is to bind pre-specified pieces of data path to the Petri net components such as places, transitions, and choices. A net constructed out of these complex components is simultaneously a data path and a distributed controller. The presence of a token in a place is interpreted as a valid data value, a bubble as an invalid (missing) data value in the associated data path. Therefore, the complex Petri net implements a data-driven processing.

The data path could be attached to the standard synchronous Petri net (SPN). However, the resulting ‘Petri-dataflow net’ will be inefficient in terms of throughput. The ‘bubble wall’ limits throughput of any arc to only 0.5 tokens/tick. By weakening the locality principle the FTL overcomes the ‘bubble wall’ because it does not rely on bubbles when propagating tokens through the net.

Components and Graphical Symbols

All the FTL net components are listed in Figure 7.8. These FTL components are glued from a corresponding SPN component and a data path (DP). A new set of graphical shapes is used for FTL nets so that it looks similar to standard symbols used in RTL digital circuit drawings.

The FTL components are:

1. **Pipes** connect components. In Petri nets a pipe is a transition arc, from the hardware point of view it is a unidirectional point-to-point connection.

2. A **cell** component is a place extended with the flow-through state and with an additional data flip-flop register to store the data vector $\vec{d} \in \mathbb{B}^n$, where $n$ is the data bit-width. The clock-enable
Figure 7.7: A loop of cells causes a cycle in the corresponding AN. Edges of the cycle are marked by red colour.

signal (CE) for the data register is identical to the one used in the AN. The throughput of the cell is 1 token/clock-tick.

3. An i-cell is the place with an additional data flip-flop register to store the data vector $\vec{d}$. The throughput of the i-cell is 0.5 tokens/clock-tick. I-cell stands for isolated cell as there is no combinatorial path between its ports.

4. A fork is a transition with a single input arc and multiple output arcs. The associated data path is wire branching.

5. A join is a transition with multiple input arcs and a single output arc. The data path is realized by a user-defined function $h()$ that combines the multiple input data wires into the output bus wire.

6. A switch is an alternate choice transition with a single input arc and multiple output arcs. The associated data path is wire branching.

7. A select is an alternate choice transition with multiple input arcs and a single output arc. The associated data path is a multiplexer controlled by the AN.

8. A let component represents any instantaneous (combinatorial) computation in the data path.

### 7.4 Algebra with FTL Nets

Algebraic rules allow reasoning about properties of FTL nets based on the structure. Graphical symbols displayed in Figure 7.8 were selected to make this intuitive. These rules directly follow from the decomposition into boolean activation nets.

There are four ‘constant’ components, called source, drain, drought, and stopper, shown in Figure 7.9.

1. The source component is a transition without an input arc: it produces a token whenever the token could be consumed. The data path consists of a user-specified (constant) function.

2. The drain component is a transition without an output arc: it always consumes an incoming token.

3. The drought component is a transition without an input arc that is never enabled: in never produces a token.
Figure 7.8: List of FTL components.


<table>
<thead>
<tr>
<th>FTL</th>
<th>SPN</th>
<th>+</th>
<th>DP</th>
<th>AN + DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td><img src="source.png" alt="Symbol" /></td>
<td><img src="source.png" alt="Symbol" /></td>
<td><img src="source.png" alt="Symbol" /></td>
<td><img src="source.png" alt="Symbol" /></td>
</tr>
<tr>
<td>drain</td>
<td><img src="drain.png" alt="Symbol" /></td>
<td><img src="drain.png" alt="Symbol" /></td>
<td><img src="drain.png" alt="Symbol" /></td>
<td><img src="drain.png" alt="Symbol" /></td>
</tr>
<tr>
<td>drought</td>
<td><img src="drought.png" alt="Symbol" /></td>
<td><img src="drought.png" alt="Symbol" /></td>
<td><img src="drought.png" alt="Symbol" /></td>
<td><img src="drought.png" alt="Symbol" /></td>
</tr>
<tr>
<td>stopper</td>
<td><img src="stopper.png" alt="Symbol" /></td>
<td><img src="stopper.png" alt="Symbol" /></td>
<td><img src="stopper.png" alt="Symbol" /></td>
<td><img src="stopper.png" alt="Symbol" /></td>
</tr>
</tbody>
</table>

**Figure 7.9:** List of constants.

**Figure 7.10:** Algebra.
4. The **stopper** component is a *transition* without an output arc that is never enabled: in never consumes a token.

The **source** and **drain** constants are complementary to the **drought** and **stopper**.

Formally, the fork/join operators and the constant components form a *commutative monoid*. Commutative monoid is an algebraic structure that is associative, commutative and has an identity element. The properties are usually spelled out in textual form as:

\[
\begin{align*}
\text{associativity:} & \quad (a \circ b) \circ c = a \circ (b \circ c) \\
\text{commutativity:} & \quad a \circ b = b \circ a \\
\text{existence of an identity:} & \quad a \circ e = a
\end{align*}
\]

(7.1)  
(7.2)  
(7.3)

In FTL the properties are represented graphically, as shown in Figure 7.10. In (a) the associativity allows to change the grouping in which A, B, and C are joined into D (or D is forked to A, B and C). In (b) the commutativity allows to reorder the pipes that are joined or forked. In (c) the identity element ‘consumes’ the fork/join operator. The identity (neutral) element is the source/drain component.

These diagrams have no direction specified by arrows; they can be read from left to right or right to left as needed. The picture (d) shows existence of a ‘zero’ with respect to the fork/join component. The pictures (e) and (f) show handling of the priority choice component. The equalities are easily proved by substituting ANs of the components and performing logic minimization.

## 7.5 Hardware Synthesis

Synthesis from FTL nets to RTL hardware is based on activation nets with data path. Input FTL net diagram is decomposed into the activation net with data path using the table of fragments in Figure 7.8. Care must be given to correctly overlap (unify) boundary wands in the AN fragments. The nets may be combined with legacy hardware cores using the method in Section C.2. The resulting AN+DP graph shall be acyclic. The graph represents a spatial digital circuit which is printed in synthesisable technology-independent VHDL file.

### 7.5.1 Synthesis of Pipes as the Module Interfaces

During controller synthesis from synchronous Petri nets (Subsection 6.3.1) the hardware I/O signals are attached to transitions and places. In FTL the other possibility is to synthesise pipes as the module interface in RTL.

For each pipe in the FTL net there is a *wand* in the AN whose runtime boolean value indicates if the associated data value in the pipe’s data path is valid or not. For a pipe that is a part of the module’s interface the value of the *wand* must be made *visible and controllable* from the outside world. A single *wand* may be shared among multiple pipes.

A suitable technique is shown in Figure 7.11. A *pipe* is represented in the AN by the *wand* $s$ and the data wire $\vec{D}$. The data wire is brought through the module interface directly as a port. The *wand* $s$ is extended with two edges $P$, $R$, representing new $P$, $R$ hardware wires for the module interface. The $P$ wire signals *presence* of a valid data on the data vector wires $\vec{D}$; the $P$ wire is driven by the producer of the data. The $R$ wire signals *readiness* of the consumer to take the data. The data is transferred over the interface (in either direction) iff both $P$ and $R$ are active at the end of a clock-tick.

This technique can generate the RTL interface for any FTL net, even if the net is just a single component. However, late composition of generated modules in RTL will not work and Figure 7.12
CHAPTER 7. DATAFLOW HARDWARE SYNTHESIS FROM PETRI NETS

Figure 7.11: Module interfaces using pipes. \( P \in \mathbb{B} \) is the Present signal; \( R \in \mathbb{B} \) is the Ready signal; \( \vec{D} \in \mathbb{B}^n \) is the Data signal.

Figure 7.12: Direct composition of FTL modules cannot be done in RTL because of the loop between \( s_1 \) and \( s_2 \). shows why. In (a) the two modules are first compiled to RTL and the interface wires \( P_1/R_1 \) and \( P_2/R_2 \) are generated. Then the modules are naively composed together in RTL, for example by instantiation in VHDL. The resulting system contains a combinatorial loop between \( s_1 \) and \( s_2 \) and further logic synthesis will fail. The correct solution is shown in (b). Modules and components must be connected within the FTL framework by unifying (overlapping) their boundary wands, and then translated to RTL together.

7.6 Case Study: Mandelbrot Fractal Set

The presented concepts will be clarified in a design of a single-purpose domain function unit (DFU) for the ASVP core. The new DFU implements only a single operation: a pipelined computation of a pixel in the Mandelbrot fractal set (MF-set).

The sequential pseudo-code for computing the MF-set is shown in Listing 7.1; the code is the same as in Subsection 4.4.2. The input to the algorithm is a point \((x_0, y_0)\) in the 2D plane, \((x_0, y_0) \in \mathbb{R}^2\), ranged \( x_0 \in (-2.5, 1); y_0 \in (-1, 1) \). The algorithm iterates in while loop at line 5 until the given point is known to lie outside the fractal set. The points that lie in the set would iterate indefinitely because the condition \( x^2 + y^2 < 4 \) is always satisfied for them. In practice the algorithm decides that a point lies in the set when the number of iterations reaches some arbitrary upper bound \( \text{max}_\text{it} \). Here we use \( \text{max}_\text{it} = 50 \) as it produces good results. When the while loop exits the number of iterations is usually used for colouring the pixels in the output image. Typically the points inside the fractal set that reached the maximal iteration count are coloured black, the points outside the set are coloured progressively depending on the iteration count.

The original implementation presented in Subsection 4.4.2 uses a general-purpose DFU in the ASVP and schedules the compute operations sequentially. Local variables in the algorithm are scalar;
7.6. CASE STUDY: MANDELBROT FRACTAL SET

Listing 7.1: Mandelbrot fractal set algorithm, sequential pseudo-code.

```
input (x0, y0)  # a point; x0 in (-2.5,1), y0 in (-1,1)
x = 0.0; y = 0.0; it = 0
max_it = 50

while (x*x + y*y < 4.0 and it < max_it):
    xtemp = x*x - y*y + x0
    y = 2*x*y + y0
    x = xtemp;
    it = it + 1

output (x0, y0, it)  # pixel color = number of iterations
```

Figure 7.13: Flowchart of the MF-set algorithm. The main basic blocks are $B_1$ and $B_2$.

to take advantage of the vector hardware a block of points (pixels) is computed simultaneously as a data-parallel vector. Compute operations are issued as the v-instructions.

The implementation using FTL nets, presented here, is mapped in space and exploits pipelining. The input and output to the specialized MF-set DFU are vectors (streams) of points, the dataflow processing inside is scalar and pipelined. Compute operations are instantiated as physical compute cores in hardware and data dependencies between the operations are mapped to physical links between the cores. As usual, the whole computation is carried out in floating-point numbers.

For clarity, the original ASVP is called here the ‘GP-ASVP’ because it implements a ‘general-purpose’ DFU, and the new ASVP is called the ‘MF-ASVP’ because its DFU is highly specialized for the MF-set algorithm.
7.6.1 Conversion to the FTL Net Model

Graphical flowchart of the MF-set algorithm is depicted in Figure 7.13. There are two main basic blocks, marked $B_1$ and $B_2$. The loop escape condition ‘it < maxit’ is evaluated at the end of the loop. The condition $x^2 + y^2 < 4$ in the block $B_1$ in while-loop is computed using two multiplications, one addition and one comparison (subtraction). In $B_2$ the new values of $x$ and $y$ are computed using two multiplications and three additions, split in two independent sub-blocks $B_{21}$ and $B_{22}$.

This flowchart was manually transformed to the FTL net model depicted in Figure 7.14. The input is in the pipe $A$, the output is from the pipe $Z$. Input tokens are records with three fields: ($p$, $x_0$, $y_0$). The values ($x_0$, $y_0$) are the position of the point in the 2D plane (floating-point numbers). The value $p$ is an integer that uniquely identifies the token. Independent inputs may be processed in parallel in distinct pipeline stages because each token represents an independent context of computation.

Since the number of loop iterations is different for each point in the 2D plane, the tokens at the output are generated out of order relative to their ordering at the input. The context of an output token is identified by attaching a context ID $p$:$i16$. The $p$ is an arbitrary integer number carried through the pipeline along with the other context values ($x_0$, $y_0$, $x$, $y$, etc.), but it is not used for any computation.
7.6. CASE STUDY: MANDELBROT FRACTAL SET

-- I/O data type for the MF-set pipeline:

type ipxy_t is record
   it : iter_t;
   pos : pixpos_t;
   x0 : float_t;
   y0 : float_t;
   x : float_t;
   y : float_t;
end record;

-- The implementation is generated in a tool:
component manfract is
   port (    clk : in std_ulogic; -- clock
             rst : in std_ulogic; -- reset
             A_P : in std_ulogic; -- input pipe A
             A_R : out std_ulogic; -- input pipe A
             A_D : in ipxy_t; -- input pipe A
             Z_P : out std_ulogic; -- output pipe Z
             Z_R : in std_ulogic; -- output pipe Z
             Z_D : out ipxy_t -- output pipe Z
   );
end component;

Listing 7.2: Interface of the MF-set hardware pipeline in VHDL.

in the MF-set pipeline. The \( p \) carries the index of the pixel so that pixels at the output can be put in the correct place.

In Figure 7.14 the text in brackets along the pipes specifies the data types of tokens in the pipe. The types are records; field names are in italic, primitive types in the typewriter font. The context fields were determined by analysis of live ranges of variables.

7.6.2 Implementation in FPGA

Code Generation

The FTL net model in Figure 7.14 is translated by my tool (Appendix D) into a hardware compute pipeline. Compute operations, such as \( x^2, x^2 + y^2, \ldots \), are implemented in the generated VHDL code by instantiating the appropriate Xilinx FP cores using the method described in Section C.2. The I/O interface to the MF-set pipeline is the input pipe \( A \) and the output pipe \( Z \). The input tokens have the record data type \( (p : i16, (x0, y0) : float32[2]) \), where \( p \) is a 16-bit context ID (a pixel index, in fact), and \( (x0, y0) \) is the point position. The output tokens have the record data type \( (p : i16, it : i8, (x, y) : float32[2]) \), where \( it \) is the number of iterations the point has reached in the pipeline, and \( (x, y) \) are the final X-Y values. The number of the iterations is used for pixel colouring.

The tool generates the implementation as a VHDL entity manfract. Listing 7.2 shows the component declaration for the entity. The record type \( ipxy_t \) is the data type of the I/O tokens. The input pipe \( A \) was decomposed using the method described in Subsection 7.5.1 into the input VHDL signals
A. P and A. D and the output VHDL signal A. R. The output pipe Z was decomposed into the input VHDL signal Z.R and the outputs Z.P and Z.D.

**Integration in MF-ASVP**

The hardware component manfract is integrated into a new ASVP core as a special domain function unit (DFU), as shown in Figure 7.15. The new ASVP is called ‘MF-ASVP’. The standard memory blocks B and C store vectors of \((x_0, y_0)\) pairs that are to be fed in the MF-set pipeline. The address generators \(AG_1\) and \(AG_2\) are configured by sCPU to read these input arrays. The configuration is done via the v-instructions.

Each element \((x_0, y_0)\) obtained from the input arrays is complemented by a unique index pos created in a counter. Complete tokens \((pos, x_0, y_0, x = 0, y = 0)\) are fed in the pipeline via the pipe A interface. The output pipe Z produces a sequence of tokens \((pos, it, x, y)\). These are stored in the result array in the bank A. Only the field Z.it is stored in A; the field Z.pos is used as an index in the address calculations in \(AG_0\) so that the output tokens are placed in the correct place in A. It does not matter how tokens are reordered in the manfract pipeline, they will always be stored in the output array in the correct order.

**FIFO Queues**

The FTL-net model depicted in Figure 7.14 is a cyclic pipeline because the MF-set algorithm is iterative. The current limitation of the FTL→VHDL translation process, described in Section 7.5, is the need for acyclic activation nets. Cyclic activation net would translate into combinatorial cycles in the generated RTL circuit, which is forbidden. (The CAD tool would detect the problem during the translation process.)

The workaround is to use a FIFO queue in the pipeline loop to break the cycle. A queue can be constructed in FTL that has no direct path between its input and output pipes (Section C.3). The model in Figure 7.14 contains three FIFO queues, denoted ‘Fifo1-3’ in the picture. The queue Fifo1 is always present so that the potential combinatorial cycle is broken. The other two queues (‘Fifo2’, ‘Fifo3’) are optional and serve for performance optimization.
7.6. CASE STUDY: MANDELBROT FRACTAL SET

Table 7.3: Maximal operating frequency in MHz of the data-path as a function of pipeline depth in FP cores \((A_x=adder latency, M_y=multiplier latency)\). Cells coloured when the optional FIFO queues were used (Fifo2, Fifo3).

FPGA Synthesis Results

Table 7.3 lists the maximal operating frequencies achieved in Virtex 5, Virtex 6, Spartan 6 and Kintex 7 FPGAs. The nomenclature \(A_x, M_y\) refers to the pipelining depth (latency) of the adders and multipliers, respectively, used in the design. Higher latency allows for higher operating frequency but the pipeline is longer. The queue ‘Fifo1’ is used in all designs, the queues ‘Fifo2’ and ‘Fifo3’ are used in configurations whose cells are coloured in the table (A5M6 in Virtex 5, A6M3 in Virtex 6, A5M4 and A6M6 in Spartan 6, A4M4 in Kintex 7).

The utilization of FPGA slices is plotted in Figure 7.16. The numbers are for the whole MF-ASVP core that includes the new specialized MF-set DFU, address generators, and the controller sCPU, as depicted in Figure 7.15. The average resource utilization is 1762 slices. 36% is used for the nine FP cores supplied by Xilinx FPGA tools. Another 36% is used for the pipeline control, the data-path interconnect between the FP cores and the FIFO queues. This part is the one generated by the FTL→VHDL translation tool. 22% is occupied by the sCPU controller that configures address generators and communicates with the host processor. Only 7% of slices is used for the three address generators.

Despite the completely different DFU structure the resource requirements of the original GP-ASVP is the same as in the new MF-ASVP. On average the MF-ASVP consumes 1762 slices, the original GP-ASVP 1715 slices (Figure 5.13).

Run-Time Performance

Execution times are compared between the original GP-ASVP that runs the Mandel kernel, and the new MF-ASVP that uses the specialized MF-set DFU generated from the FTL net model.

The drawback of the original Mandelbrot implementation (Subsection 4.4.2) is the loss of efficiency due to imperfect vectorization. Pixels are computed in data-parallel blocks (vectors). The number of iterations that a pixel should take in the inner while-loop is different, but the vector DFU has no way to skip the pixels that are finished. All operations are executed on all pixels and unneeded results are discarded later. Therefore, the total execution time depends on the vector length (which is constant)
and the hardware configuration.

In the new MF-ASVP the pipeline is scalar, not vector. Pixels proceed through the pipeline as independent tokens. When a pixel is finished it is ejected from the pipeline to the output pipe. This creates a bubble that moves through the pipeline until it reaches the place where new input tokens (fresh pixels) are injected. The hardware pipeline is utilized efficiently and no redundant computation is performed. Therefore, the total execution time depends not only on the hardware configuration but also on the portion of the Mandelbrot fractal set that is being computed. The highest workload is generated by pixels that are in the set, e.g. pixels corresponding to the points in the range $x_0 \in (-0.1, 0.1)$ and $y_0 \in (-0.1, 0.1)$. These pixels iterate $\max_{it}$ times in the pipeline loop.

The execution times in milliseconds for computing a grid of $120 \times 60 = 7200$ points (pixels) are shown in Table 7.4. The subtables a–d list results for different coverages of the grid with pixels in the set. In (a) the $120 \times 60$ grid is mapped to a small rectangle centred around the point $x_0 = -2$, $y_0 = 0.7$ that contains a void. Pixels from this grid iterate only once or twice through the pipeline loop in the FTL-net DFU. In the original GP-ASVP the execution time is always constant irrespective of the grid position. Therefore the speed-ups are the highest.

In subtable (d) the coverage of the grid with pixels in the set is $100\%$, hence the computational load is the highest. In this case the Mandelbrot implementation in the original GP-ASVP performs no

Figure 7.16: The FPGA area occupied by the MF-set compute core (depicted in Figure 7.15).
unneeded computations. Still, the average speedup of the FTL-net hardware is $22 \times$ over the original GP-ASVP. The reason is sequential utilization of a small number of compute cores in the original GP-ASVP.

In the original GP-ASVP the loop body is implemented using 15 vector instructions (Listing 4.3). The vector instructions are scheduled sequentially. Each vector operation computes 1 floating-point or integer operation per clock-tick; the throughput is 1 op./tick. In the FTL-net pipeline there are independent compute cores for all operations, hence the throughput is 15 ops./tick. Further, in GP-ASVP the DFU experiences stalls up to 20% of the execution time (Figure 5.11), hence the real throughput is 0.8 ops./tick. The stalls are caused by bank conflicts and by hardware pipeline start-up times that are experienced in each vector operation. These two effects suffice to explain the $22 \times$ speed-up obtained in MF-ASVP.

Since the FPGA slice utilizations of both the GP-ASVP and MF-ASVP cores are practically the same the performance-per-area has been improved by $22 \times$. The price paid is the deep specialization of the DFU which precludes reconfiguration via firmware upload.

### 7.7 Summary

The chapter has provided the fourth thesis contribution: **A new data-driven dynamically scheduled technique for the design specification of custom hardware.**

The ‘bubble wall’ is caused by the principle of locality.

The principle of locality requires that the behaviour of places and transitions depends exclusively on the local neighbourhood. To move a token over the arc the target place must be empty (in 1-bounded safe PN). The ‘bubble wall’ says that the throughput of tokens over any arc is limited to 0.5 tokens/tick. Classical synchronous Petri nets (SPN) are unsuitable for the modelling of dataflow systems because of the limited throughput.

**Activation nets allow for non-local behaviour in Petri nets.**

Activation net computes the firing conditions for the corresponding SPN. The AN fragments for standard PN components implement the locality principle, but it is not a requirement in the AN.

**New ‘flow-through’ state in ‘cells’ overcomes the ‘bubble wall’.**

The new cell component is a place extended with a ‘flow-through’ state. In flow-through state the cell unloads and loads tokens concurrently in the same firing event (clock-tick). The flow-through state effectively overcomes the ‘bubble wall’ because it eliminates the intermediate state when a place is empty.

**Data path by composition.**

Fragments of pre-specified data paths are attached to the AN fragments of the SPN components. The new components are: cell, i-cell (place), fork, join, switch, select, let.

**The fork/join and constants in FTL form a commutative monoid.**

The algebraic properties follow from the decomposition of the Petri net into the activation net. Activation nets can be minimized as boolean functions.

**Synthesis to hardware via the construction of Activation Nets.**

All FTL components are synthesisable via the deconstruction into their AN fragments.
### Table 7.4: Finishing times of the MF-set algorithm in GP-ASVP and in the FTL-net based MF-ASVP.
Computed for a grid of $120 \times 60 = 7200$ points. The tables (a)–(d) are for different grid coverages of points that iterate $\text{maxIter}$-times in the loop or pipeline.

<table>
<thead>
<tr>
<th>FPGA</th>
<th>Conf.</th>
<th>Freq.</th>
<th>Time</th>
<th>Conf.</th>
<th>Freq.</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>V5</td>
<td>A5M3</td>
<td>166MHz</td>
<td>47.0ms</td>
<td>A5M6</td>
<td>150MHz</td>
<td>0.069ms</td>
<td>681×</td>
</tr>
<tr>
<td>V6</td>
<td>A6M3</td>
<td>200MHz</td>
<td>39.2ms</td>
<td>A6M6</td>
<td>200MHz</td>
<td>0.057ms</td>
<td>688×</td>
</tr>
<tr>
<td>S6</td>
<td>A4M4</td>
<td>125MHz</td>
<td>62.7ms</td>
<td>A6M6</td>
<td>150MHz</td>
<td>0.089ms</td>
<td>705×</td>
</tr>
<tr>
<td>K7</td>
<td>A5M3</td>
<td>250MHz</td>
<td>31.3ms</td>
<td>A4M4</td>
<td>250MHz</td>
<td>0.018ms</td>
<td>1739×</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Average:</td>
<td></td>
<td></td>
<td>953×</td>
</tr>
</tbody>
</table>

(a) MaxIter Pixels: 0%  

(b) MaxIter Pixels: 23%  

(c) MaxIter Pixels: 66%  

(d) MaxIter Pixels: 100%
III Dataflow at the Chip Level
Chapter 8

Hardware Families of Threads

This chapter approaches dataflow scheduling at the level of instructions and threads in processors. The first section (Section 8.1) gives the motivations. The following sections 8.2, 8.3, 8.4 shortly describe the Self-adaptive Virtual Processor (SVP) scheme (a previous work). Section 8.5 describes the newly proposed Hardware Families of Threads (HWFAM) scheme that extends SVP to heterogeneous systems. A functional model of HWFAM is specified in Section 8.6. The content of this and the next chapters was published in [X.10] and in [X.1]. Related work was published in [X.3, X.11, X.12, X.13].

8.1 Data-Flow Processing Across Heterogeneous Multicores

In the previous, second Part of the thesis the specification of data-driven computation has been discussed at the level of individual circuits and gates. The circuit level deals with such issues as pipelining, achieving the synthesis timing closure, and interfacing with legacy hardware IP cores. However, the circuits do not exist in vacuum; data blocks must be communicated with off-chip DRAMs, application tasks must be scheduled on the compute fabric. These higher-level scheduling issues will hardly be implemented in fully customized hardware. The common higher-level control functions are implemented partly in software and partly in a static (non-customized) hardware.

The scheduling and coordination granularity in data-driven systems should be adapted to the granularity and interdependence of operations at which the unpredictable (long) latencies happen. For instance in processors, cache misses are the long-delay unpredictable latencies that happen at the granularity of load/store instructions. The dynamic scheduling is therefore performed at the instruction level.

Silicon scaling allows to put more transistor in a die, but the transistors cannot be more tightly connected than they already are. A machine is better constructed by partitioning its functions into cores that are loosely connected by on-chip networks or other latency-tolerant infrastructure. Chip power constraints may forbid all the transistors to be active simultaneously, hence the cores will be specialized for different aspects (functions) of the machine. The dynamic data-driven scheduling can no longer be limited to processing in individual cores.

Dynamic scheduling at the fine-grained instruction level in von Neumann computers is not scalable as it is limited by the complexity of reorder buffers. Attempts are made to perform scheduling at coarse granularities but without sacrificing performance. For instance, in GPGPU computing the scheduling is at the thread-block level (‘thread warps’), because it is reasoned that there are always many other independent thread-blocks available in a queue and that the behaviour of threads within a block is correlated (when one misses the other are also likely to miss). In microthreading, discussed below,
the unpredictable latencies also happen at load/store instructions but the scheduling is performed at the microthread level. Individual microthreads are not correlated and the programming model (SVP) encourages creating many of them.

The Self-adaptive Virtual Processor (SVP) [Jes04, Jes06, Jes08, kP12] is a general-purpose concurrency model used to design and program multithreaded multicore systems. The SVP is in a good position to express data-driven concurrency across a chip. Within general-purpose processor cores the scheduling can be implemented at the microthread level, with latency detection/recovery at the level of instructions. At the inter-core level the scheduling and synchronization will be implemented at a coarser granularity of thread families (see below). This coarser granularity is welcomed because it allows to better amortize processing latencies.

8.2 Execution in Threads

The SVP expresses fine-grained concurrency by composition of microthreads (Figure 8.1). A microthread1 comprises only a few processor instructions which typically implement a body of a loop and share only a small portion of the processor (ISA) register file. A family of threads is an ordered set of threads, all created by one processor instruction, called create. The ordering is defined by a sequence of integer index values specified during create event as a \{start, step, limit\} triple. These values are provided by the program via the setstart, setstep, and setlimit instructions, respectively. Microthreads can execute in parallel by default. Explicit unidirectional dependencies between successive threads within one family, and between a parent thread and its child family, are allowed only. A family of threads is akin to a parallel for-loop.

By creating families of potentially many fine-grained threads in one event the SVP allows implementations to amortize overheads associated with individual thread management. A family of threads represents a batch of coarse-grained work to be scheduled in the multicore system. Individual threads are executed independently and no restriction is placed on conditional branches.

Threads exist in ordered indexed groups called `families of threads`. Each thread belongs to one family. Each family—except the special root family—has a parent thread. Threads are ordered using a sequence of integer indexes within the family (Figure 8.2). The sequence of indexes is specified by a start index, a step value (an increment between two sibling threads), and a limit index. The family further specifies a thread program starting address: all threads in the family begin at the same instruction, although they may individually diverge via conditional branches later on. Threads receive their unique index value in the register %TL0 upon their start.

All threads in the system are implicitly parallel. The model allows an explicit specification of unidirectional data dependencies from thread $i + step$ to the previous thread $i$ (the data will be transferred

---

1The terms `thread` and `microthread` are used interchangeably throughout the text.
in the opposite direction, i.e. from the thread \( i \) to \( i + \text{step} \). Moreover, there may be an explicit data dependency from the first thread in a family to the parent thread, and from the parent thread to the last thread in the family.

The acyclic dependency graph enables processor implementations a free choice of the number of concurrent threads they will schedule. This parameter is called blocksize. It controls the number of concurrent physical replicas that execute the thread code (Figure 8.3), hence it affects a ratio of performance to resource utilization.

### 8.3 Data-Driven Thread Scheduling

The SVP model combines the instruction-driven and data-driven techniques. Individual operations (arithmetic ops., jumps, memory accesses...) are issued based on the execution of classical instructions in the program order, but they are completed based on the availability of data. When an instruction is stalled (cannot proceed) the processor deschedules the thread.

Conceptually there are two kinds of memory storages: (a) standard main memory, and (b) synchronized memory. The main memory is organized in a globally shared address space. Cache-only memory architecture (COMA) with directory-based coherency protocol and distributed memory modules to improve the throughput were implemented in the microgrid simulator.

The synchronized memory is much smaller in size than the main memory, in fact it is only implemented by the register files in the processor cores. The synchronization is based on I-structures [AToTLfCS80, ANP89], originally invented for dataflow machines and later adapted for SVP (see Figure 8.4). A single I-structure is a memory location that stores a data value and a state flag. The states are pending (P), waiting (W) and full (F). In the beginning the I-structure must be initialized to the pending state (P). A write by the thread A will bring it into the full state (F). All subsequent reads in the full state will simply return the stored value. However, it is forbidden to execute another write into a full I-structure. When the value in the I-structure is no longer needed the structure may be re-initialized to the pending state again.

The behaviour when a read of the pending location is attempted is more interesting. In this case the reading thread B is suspended: conceptually, the thread’s continuation is stored in the I-structure and the state is updated to waiting (W). When the writer thread A later stores value in the waiting I-structure the suspended thread is woken up and may continue.

The I-structure thus implements a light-weight synchronization between two threads (the writer
Figure 8.3: The parameter ‘blocksize’ controls resource utilization: the number of physical replicas concurrently executing the thread code.

and the reader). The limitations are:

1. to avoid races only a single write is allowed;

2. to avoid non-deterministic program behaviour the reader cannot check (peek) the state of the I-structure without being descheduled if the state is not full;

3. only a single reader is allowed unless there is an additional provision to store the suspended reading threads in a linked list.

In SVP all registers in the CPU register files are implemented as I-structures. The reader-writer synchronization is used in the following cases:

1. Communication between sibling threads in thread families.
   Sibling threads may communicate only from the predecessor thread (the writer) to the immediate successor thread (the reader) using shared architectural registers.

2. Signalization of family completion to the parent thread.
   The parent thread specifies one of its registers to serve as the receiver for the family’s final
8.4. MULTICORE SUPPORT

Figure 8.4: I-structure operation. The architected 32-bit registers are extended with the state flags (P, F, W).

termination signal. When the family finishes the hardware automatically issues a write into the associated termination register in the parent thread; the value being written is the family exit code.

3. **Split memory loads in the case of D-Cache miss.**
   When the cache hits, the accessed data is immediately returned and the execution proceeds as usual. When the cache misses, the destination register of the load instruction is initialized to the pending state, while the thread is allowed to continue and the D-Cache starts loading the requested cacheline from the main memory. The processor speculates that the program will not use the requested data immediately. When the cacheline finishes loading the cache controller will automatically deliver the missing data directly to the register.

8.4 Multicore Support

The SVP model is oblivious to the number of processing cores in the system. A family of threads can be dynamically scheduled entirely on a single processor or distributed across multiple cores as shown in Figure 8.5. In the latter case each thread executes in one given core, but different threads can be scheduled in different cores. Threads are never moved between cores once they begin their execution.

Each family of threads is associated to a compute cluster, called the ‘local place’, composed of several tightly coupled cores. Threads of the family are scheduled concurrently within their local cluster. To delegate a family of threads to another cluster the programmer can use the `setplace` instruction.

The distribution of threads is enabled by the I-structure synchronization. Register files of the individual processor cores are connected and organized into a global address space\(^2\). A writer thread executing on one processor core may transparently send data to a register (I-structure) in a different core. This is used both for the communication of sibling threads scheduled in different cores as well as for sending of the family termination signals if the parent thread executes in a remote place.

The multicore SVP requires:

\(^2\)The register address space is separate from the conventional main memory address space
1. An ability to distribute threads among cores.
   Threads may be distributed among cores only before they start executing; already running threads are never moved from core to core. In a simple implementation the whole family of threads is delegated to a remote core; in an advanced implementation the individual threads of the family are distributed over remote cores.

2. An ability to communicate and synchronize between running threads.
   All synchronization between running threads is transparently handled via the I-structure-based register files.

3. An ability to share data in the conventional main unsynchronized memory.
   The SVP specifies a suitable cache coherency protocol for the main memory. Two accesses to the same memory location may return different results if the originating threads are not synchronized. Synchronization means any communication via the I-structures.

8.5 Heterogeneous Multicores and Hardware Families of Threads

Previous work on SVP assumed that cores within clusters are homogeneous (identical) in their ISA to enable the transparent distribution of threads among cores. In this work the view is extended to heterogeneous clusters composed of both the general-purpose and specialized processors (= accelerators), as depicted in Figure 8.6.

Here a new interface between general-purpose microthreaded SVP processors and hardware accelerators is proposed. The scheme is called Hardware Families of Threads (HWFAM), or the microthreaded coupling scheme. The main feature of the scheme is the transparent migration of thread families created in processors to accelerators. The accelerators can be thus viewed as just different kinds of processor cores in the heterogeneous microgrid. Thread migration is not performed individually due to the efficiency reasons, but in whole families.
8.5. HETEROGENEOUS MULTICORES AND HARDWARE FAMILIES OF THREADS

8.5.1 Overview

Each family of threads has clearly defined boundaries: input parameters can be found in the thread global registers, input data-sets can be autonomously loaded from the main memory and/or the on-chip caches, and output data is put in the main memory or in the on-chip caches. The completion signal generated in the accelerator is translated to the standard SVP thread-family termination signal: a value is written to the designated termination register in the parent thread.

Activation and control of the accelerators is handled entirely within a dedicated subsystem, not in the microthreaded processors. This allows for a complete transparency of the accelerated execution relative to the general-purpose processors. The activation of accelerators is based on SVP’s creation events, which commence execution of a new family of threads. The type of accelerator that could be used to accelerate a given family of threads is recognized based on the thread function address that was set using the setthread instruction. The instruction sequence executed in the parent thread is independent of the type and placement of the child family.

The original software (microthreaded) implementation of the new family is still present in the program code. It can be used in cases when the accelerator of the suitable kind is temporarily or permanently unavailable, or when the estimated execution time in the accelerator – that is, including overheads (e.g. data movements) – is higher than the estimated time in the general-purpose processor.

An important advantage is the ability to continue independent execution in the general-purpose processor without interference even after the child family has been placed in the accelerator. The parent thread is not blocked when it creates a new child family as the operations of creating the new family and waiting for its termination signal are separate. Independent tasks can be executed in the parent thread in between (see Listing F.2 at line 19).
8.5.2 Create Events

In SVP a create event represents a family of threads that shall be scheduled in the system; it is an order for execution of a batch of work. Create events are formed in thread scheduler in consequence of executing the create instruction within a would-be parent thread. The data structure (a message) associated with each create event contains run-time parameters required to start the family execution. It carries the following data fields:

(a) **Family ID** (fid) – uniquely identifies the family instance in the grid.

(b) **Thread function address** in the instruction memory – uniquely identifies the function of the family of threads.

(c) **Start/step/limit values** defining the index space of the family of threads – roughly correspond to the amount of work in the family.

(d) Base pointers to the family global and sibling-shared registers in the processor register file – specifies the register locations of the additional family parameters.

(e) Pointer to the synchronization register in the register file – for signalization of the child family termination event back to the parent thread.

![Flowchart of the create events in the HWFAM subsystem.](image)

Circulation of the create events in a simple system consisting of one general-purpose processor and an accelerator is shown in Figure 8.7. The key component of the proposed HWFAM interface is Thread Mapping Table (TMT), located in the middle of the picture. Create events, originating in the microthreaded processor on the left, are routed to the TMT unit. The TMT unit decides which of them should be passed on to accelerators. According to the decision, create events are either handed over to HWFAM’s resource manager which orchestrates the accelerated execution, or they are sent back to GP processor to continue regular thread initialization. Execution in attached accelerators is scheduled and controlled by the HWFAM resource manager. The resource manager translates SVP create events in command sequences for accelerators. Once execution in the accelerator has finished the resource manager issues completion message to the GP processor. The completion message releases the family ID (FID) in the thread scheduler and signals termination to family’s parent thread.

8.5.3 Family Classification

Migration from GP processors to accelerators is performed at the granularity of complete families. Thread families are classified into three groups: software families, virtual families, and hardware families, which form an inclusive set:

\[ \text{HW families} \subseteq \text{Virtual families} \subseteq \text{SW families} \]
The classification is further detailed in Figure 8.8. All thread families belong to the *software families* superset class because there is always a regular software (microthreaded) implementation of the family in the program memory. The *virtual* class are the families that are potentially transferable to hardware accelerators. Create events that target virtual families are routed to the TMT unit to make the final decision where the family shall be placed. The *hardware families* class are those that end up being executed in accelerators.

The term ‘virtual family’ is borrowed from the term ‘virtual function’ in software object-oriented programming (e.g. the C++ language). In object-oriented programming a virtual function can be overridden in derived classes by a function with the same signature to implement a dynamic dispatch. Similarly, in the HWFAM interface the base software implementation may be dynamically overridden by an implementation in accelerated hardware.

### 8.5.4 Thread Mapping Table

The *Thread Mapping Table* (TMT) sits in between the microthreaded processor and accelerators, deciding where new families should be executed. The decisions are made based on thread program starting (function) address – the address of the first instruction of the family’s threads, set by the parent thread using the `setthread` instruction. The TMT unit contains relatively small *content-addressable memory* (CAM). The CAM is (dynamically) preloaded by system software with function addresses of *virtual*-class threads which can be potentially placed in accelerators. The TMT unit continuously compares thread addresses in incoming create events to those stored in the CAM.

The flow of create events between the processor and the TMT unit is detailed in Figure 8.9. The left half of the picture is occupied by the *thread scheduler* in microthreaded processor. The scheduler internally maintains a create queue of new families. The queue is filled by committing `create` instructions in the processor pipeline.

**Virtual Families**

Before a create event is passed to the TMT the thread scheduler checks if the new family belongs to the class of *virtual families*. The purpose is to relieve some of the load on TMT. This check should be very light-weight as it is performed for each new family; it should be evaluated in constant time and with minimal hardware resources.

The membership in the *virtual families* class is indicated by setting the *least significant bit* (LSB) of the thread function address. The main advantage of this choice is that no new instructions nor storage are needed to pass the classification information around. In processors with fixed-size 32-bit
CHAPTER 8. HARDWARE FAMILIES OF THREADS

aligned instructions the two LSBs in instruction addresses are always zero, hence they are ignored in instruction fetch. In program code the symbolic thread function address is simply ‘or-ed’ with 1 before being passed to the setthread instruction, as shown in Listing 8.1 below.

```
14  set fir_outer | 1, %t16 /* ... thread fun. addr. */
15  setthread %t15, %t16
```

Listing 8.1: The virtual family class is indicated in program code by setting bit 0 of the thread address to 1.

The classification based on thread function addresses is not necessarily compile-time only. System software could use relocation information normally kept for dynamic linker to find all places where a given thread function address is being referenced in the program code (presumably by setthread instructions) and update all the places if needed.

Lookup in the Content-Addressable Memory in TMT

The content-addressable memory (CAM) in the TMT unit stores a list of virtual-class families that could be placed in accelerators. The lookup key in the CAM is the thread function address—‘taddr’ (32-bits). The key (‘taddr’) is mapped to an accelerator job type (‘jobtp’), and the minimum family size (‘minsz’). The structure of each record in the CAM is in Table 8.1.

Once the processor scheduler identifies that a family belongs to the virtual class the creation event

```
Field:  Data Type:  Function:
valid   1 b      the record is valid
taddr   32 b     thread function address of the hardware family
jobtp   8 b      identification of the family to the job controller
minsz   32 b     minimum hardware family size
maxsz   32 b     maximum hardware family size
```

Table 8.1: The fields of the content addressable memory (CAM) in the Thread Mapping Table (TMT) module.
Checking the Minimal/Maximal Family Size

The final test performed in TMT before a family creation message is handed over to accelerators is to check the family size against the minimum and maximum sizes specified in the CAM. The reason is that accelerators typically do not perform well on small working sets due to various fixed costs associated with each activation. Other accelerators are unsuitable for large data sets due to hard limits in local storage (Figure 8.10).

The working set size is measured in terms of the number of threads. This can be computed from family parameters in creation message:

\[
threads = \frac{limit - start + 1}{step}
\]  

(8.1)

The equation requires an integer division, which is a costly function in hardware. To alleviate the cost it is assumed that in families considered for accelerated execution the \(step\) parameter will be compile-time known constant. Hence the \(step\) term may be moved to the other side of the equation:

\[
threads \cdot step = limit - start + 1
\]  

(8.2)

In any case the family size computed from parameters of the top-level hardware family is only a heuristic hint. The real workload size will depend, among other things, on the number of threads created in the lower levels of hierarchy.
CHAPTER 8. HARDWARE FAMILIES OF THREADS

8.6 Complete Model of the HWFAM

The functional block diagram of CPU–HWFAM system is shown in Figure 8.11. The microthreaded CPU is UTLEON3 which is described in detail in Appendix E. In the previous section it was described how thread-family creation messages are generated in scheduler and then processed in the Thread Mapping Table (TMT) unit. The creation messages that succeed in all tests in TMT are handed over to a resource manager in HWFAM. Its task is to schedule execution of thread families in attached accelerators.

**Accelerators (ACC)**

Execution in accelerators (ACC) is controlled by the resource manager (RM). The practical model assumes that accelerators have separate control interface for parameters and commands, and a set of independent local memories that must be loaded with working data by an external agent prior to accelerator activation. It is not required that accelerators could fetch or store the working data autonomously from/to the shared main memory (MEM). The subsystem includes a shared Direct Memory Access Engine (DMA-E), controlled by the RM to transfer working data to/from accelerators.

**Figure 8.11:** Functional model of the CPU–HWFAM system. CPU = UTLEON3, see Appendix E. TMT = Thread Mapping Table. RM = Resource Manager. DMA-E = DMA Engine. ACC = Accelerators.

**Figure 8.12:** Accelerator interface follows a common model.
8.7. SUMMARY

**Hardware Resource Manager (RM)**

The resource manager (RM) handles sequencing of control functions within the HWFAM subsystem. Each thread-family creation message received from the TMT unit forms an independent *job*. Each job is composed of a sequence of *tasks*. These tasks include:

- communication with TMT;
- fetching thread-family parameters from the CPU *register file* (RF);
- setting-up data transfer commands in the DMA engine;
- configuring and activating the accelerators;
- checking on the state of accelerators;
- informing the CPU scheduler about the completion of thread families.

**Direct Memory Access Engine (DMA-E)**

The *Direct Memory Access Engine* (DMA-E) handles data transfers between local memory blocks in accelerators and the shared main memory (denoted MEM in Figure 8.11). The interface between RM and DMA-E is designed such as to enable *continuous streaming* of data in and out of local memory blocks while accelerators are in the background running. The goal is to overlap communication and computation. Local memory blocks are conceptually split in several equal-sized segments (typically 2 to 4), such as in Figure 8.13. The RM controls accelerators and the DMA engine so that while accelerator is working on block $k$ (computing $A_k \rightarrow C_k$) the DMA engine stores block $C_{k-1}$ and loads block $A_{k+1}$. An example is depicted in Figure 8.14.

![Figure 8.13](image)

**Figure 8.13:** In accelerators the local memory blocks are segmented for continuous streaming operation.

![Figure 8.14](image)

**Figure 8.14:** Interleaving the DMA and computation using continuous streaming in the FIR accelerator.

**8.7 Summary**

The chapter has provided the description of the fifth thesis contribution: *(5) Bridging the gap between the data-driven microthreaded procedural computation with the special-purpose data-driven hardware in reconfigurable arrays.* The evaluation will be given in the next chapter.
Concurrence of execution.
In previous work (e.g., in the MOLEN protocol [VWG+04]) the main system processor was utilized for both the normal software execution and the control of accelerators because the processor is typically waiting (blocks) for completion of the accelerator runs anyway. In the HWFAM scheme an additional simple processor in the resource manager is used to handle the control of accelerators. It is expected that the main microthreaded processor continues with the execution of unrelated thread families while accelerators run.

Unified model of HW/SW thread synchronization.
In multi-threading and multi-core environments the synchronization between actors is often implemented by ad-hoc primitives. For instance, there are different locking schemes used if the interaction happens between software threads, or between a software thread and a hardware function. The SVP has unified the synchronization between software threads running at the same or distinct cores. The HWFAM scheme extends the SVP to work also in the pure hardware domain, hence it unifies the synchronization between software and hardware functions.

Access to the processor register file.
In modern processors the register file is a performance-critical and area-critical resource. In the HWFAM scheme the resource manager autonomously accesses the CPU register file to obtain the parameters of thread families. Conceptually, a port to the CPU register file for the HWFAM use is required. However, as the HWFAM accesses happen at the coarse grain of thread families they are not frequent and the HWFAM logical RF port can be physically shared (multiplexed) with other functions. The HWFAM shares an RF port with D-Cache.

Portable accelerators.
The HWFAM framework places very few requirements on the organization of custom accelerators. The accelerators are portable between classical single-threaded processor systems and the microthreaded environment.

Portable software.
The HWFAM decouples hardware and software application programming and the toolchain flows. Software source code and binary is not extensively modified to support hardware-accelerated execution because the low-level machine interface for thread-family creation is identical irrespective if the thread function is executed in processor or in accelerator.
Chapter 9

Evaluating the Hardware Families of Threads

This chapter gives the experimental evaluation of the HWFAM system proposed previously. In Section 9.1 the practical implementation details are summarised. In Section 9.2 the strategies for task placement are evaluated. In Section 9.3 the results from the FPGA synthesis are given. In Section 9.4 the latency tolerance of the HWFAM is evaluated.

9.1 Hardware Implementation Details

The diagram of the implemented system is shown in Figure 9.1. It closely follows the functional scheme depicted in Figure 8.11. Besides the Thread Mapping Table (TMT) the subsystem contains the Resource Manager (RM), DMA Engine, Configuration and Data Buses, Host Bridge, and accelerators (ACC). The CPU is the microthreaded UTLEON3 processor, described in Appendix E and in [X.12, X.11].

Accelerators: FIR, DCT

Two types of soft-core accelerators are demonstrated in the evaluation:

1. Finite Impulse Response filter (FIR) – described in Appendix F.

2. Discrete Cosine Transform (DCT) – 2-D forward DCT on 8x8 pixel blocks, as defined in JPEG standard. The original JPEG function in sequential C was transformed manually to microthreaded assembly for UTLEON3. Based on static data-flow graph a hardware soft-core was implemented in VHDL. The external interface of the DCT soft-core accelerator follows the general scheme in Figure 8.12; only instead of 3 local memory blocks there is only a single local memory $A$ holding the input/output 64-element (8x8) pixel matrix.

Direct Memory Access Engine (DMA-E) and the Memory Coherency

The DMA engine implements 16 ‘channels’ that can be pre-configured and launched independently by the resource manager. Channels are convenient abstraction for multi-tasking within the DMA engine. In the implementation the DMA throughput is limited by the AMBA bus and the DDR DRAM controller. The theoretical peak throughput is 1 word (32 bits) per cycle.

The raw compute speed in accelerators is almost always faster than in general-purpose processors. However, the data transfer times are also crucial for good performance. The DMA engine can load data from the global memory at the same speed as the processor D-Cache. Engine’s memory operations
should trigger the system coherency protocol: dirty cachelines must be flushed to the global memory before the DMA engine is allowed to read them back.

With respect to data needed in an accelerator, three general situations may occur:

(a) **cold caches** – input data is located in the shared main memory, and caches are empty;

(b) **warm clean caches** – input data is located in the shared main memory and also cached in processor;

(c) **warm dirty caches** – input data is located in processor cache only and the copy in the main memory is stale.

For processor performance the situations (b) and (c) are nearly identical (the data is cached locally); the situation (a) is sub-optimal because a cache-miss will occur. For hardware accelerator the configurations (a) and (b) are identical (the data can be immediately loaded from the main memory); the situation (c) will cause a delay because a cache-line flush is required.

The original LEON3 processor by Gaisler supports coherency via simple bus snooping and write-through caches. All processor writes are propagated immediately to the bus (no dirty cachelines); when a remote write transaction is detected on the shared bus the local copy is invalidated. Microthreading requires write-back caches for good performance. While the write-back D-Cache was implemented in UTLEON3, the system coherency protocol was not upgraded due to the lack of resources in project Apple-CORE. This means that currently a multicore configuration is possible only with severe limitations in the programming model.

Explicit software `flush` instructions at appropriate places are employed in the current work to cope with this implementation constraint so that correct execution is maintained. The `flush` instructions clear caches completely, not only those cache-lines that would be flushed by a coherency protocol;
9.2. CONTROLLING THE PROCESSOR/ACCELERATOR TASK PLACEMENT

Family: ‘Exec n-times Task X’ with blocksize=2

Initial state:

Some time later:

Figure 9.2: The blocksize parameter specifies the maximal number of thread slots (T-SLOTs) allocated for a given family.

hence, the total execution time is affected negatively. The upside is that the presented evaluation is conservative. Results obtained in a system with proper coherency solution would be better.

9.2 Controlling the Processor/Accelerator Task Placement

9.2.1 Occupancy-Based Placement

The sizes of the processor register file and thread table are bounded. Threads occupy portions of the register file and thread table during their whole lifetime. When a thread is blocked (e.g., on a cache-miss) it is switched out of the pipeline but it continues to occupy the other resources. The microthreaded processor issues at most one instruction per cycle, hence the peak throughput is $IPC_{peak} = 1$. To achieve good IPC it suffices to have at least one ready thread in a queue that can be switched in the pipeline when the current thread is blocked by a long-latency operation (e.g., a cache-miss, a floating-point division).

Suppose there are several independent families of threads executing in the processor. The threads from both families time-share the processor pipeline and space-share the register file and thread table. The pipeline time-sharing is dynamically controlled by hardware. The space-sharing is controlled in software using a parameter called blocksize. The parameter is specified individually for each family by an instruction called setblock. It gives the maximal number of slots in the thread table that could be allocated for family’s threads at any time. Hence, by extension, it also controls the occupancy of the register file. Figure 9.2 gives an example of a family with blocksize=2: the processor will allocate resources corresponding to (at most) two family’s threads at any time.

In [X.11] I showed that an optimal blocksize depends on a class of the long-latency operations that family’s threads issue most often. The analysis shows that non-pipelined long-latency operations saturate even with small blocksize, while pipelined long-latency operations can take advantage of larger blocksize to utilize the processor.

Here it is shown how the blocksize parameter controls the utilization of accelerators in HWFAM and how it can be used to optimize distribution of threads (tasks) among processors and accelerators.

A task implemented by a family of threads physically executed in an accelerator will not occupy processor thread table and register file. However, the parent thread of the task is present in the processor data structures. Hence, by constraining the resources allocated to the parent threads the program can control utilization of accelerators.

Figure 9.3 shows how this is done. The task is to execute $n$-times an operation X, for which a hardware accelerator exists in HWFAM. The picture shows the situation for blocksize=2. The
CHAPTER 9. EVALUATING THE HARDWARE FAMILIES OF THREADS

Family: ‘Exec \( n \)-times Task X’ with blocksize=2

Initial assignment:

Thread #1 done:

Thread #3 done:

Figure 9.3: Execution in an accelerator (Acc) occupies a thread slot (T-SLOT) of the parent thread.

software creates a family of \( n \) logical threads, although the hardware allocates and schedules at most 2 threads from the family at any time due to the limited blocksize. Each thread will run an instance of the operation X by creating a family of threads. The creation events for X are independently processed in HWFAM. If at this time an accelerator for X is idle the creation event is handed over to the resource manager and the operation is executed in accelerator. Otherwise, the creation event goes back immediately to the processor and the operation X is executed as a regular family of threads. Creation events are not queued in the HWFAM subsystem when the suitable accelerator is occupied by a different task.

In Figure 9.3 the operation X in the logical thread #1 is placed in the accelerator, but the operation issued in the logical thread #2 is executed in the processor because the accelerator has been busy running the first instance when the second creation event has come. When the thread #1 finishes its thread slot (T-SLOT) in the thread table is reused for the logical thread #3; at this time the operation X is placed in the accelerator again as it is idle at this time. This scheme repeats until all threads are finished.

The behaviour can be generalised as follows: given \( A \) accelerators, the blocksize \( B \) and the scenario as above, the first \( A \) physical threads will run the operation X in accelerators and the rest in the processor. If \( A \geq B \), all physical threads can claim an accelerator, hence all logical threads will be executed in accelerator. If \( A < B \), only \( A/B \) logical threads will be (on average) executed in accelerators, the rest in the processor.

Experiments With a Single Accelerator

Execution with a single instance of an accelerator in HWFAM will be shown using the DCT benchmark. The baseline execution time of the DCT kernel is shown in Figure 9.4. The graph compares the execution times (in clock cycles) in software with cold (a) and warm (b) caches, and the execution in accelerator (c). The total time is broken down to component times: ‘CPU’, ‘Acc’, ‘DMA’, and ‘Other’. The ‘CPU’ component is the portion of cycles when the processor pipeline commits an instruction; similarly the ‘Acc’ and ‘DMA’ represent cycles when the accelerator is computing or DMA is transferring data, respectively. The ‘Other’ component is simply the rest to the total execution time: it represents overheads. The total execution time and the component times (except ‘Other’) are measured in cycle-accurate VHDL simulation of the RTL design. The CPU, accelerators and the DMA engine all may run concurrently to each other, hence it is not strictly correct to stack the component times on top of each other as done in the figures. Here it is done for brevity.

The raw compute work-load of the DCT kernel is 1615 instructions in processor or 348 cycles in accelerator: the accelerator has a raw speed-up \( 4.6 \times \). The transfer of I/O data to/from the accelerator
9.2. CONTROLLING THE PROCESSOR/ACCELERATOR TASK PLACEMENT

Figure 9.4: Benchmark: DCT, 1x. Speed-up when an accelerator is used. Config.: Listing G.1.

Figure 9.5: Benchmark: DCT, 2x. Sequential execution of two DCTs in accelerators is faster than a parallel execution of two DCTs in the CPU and accelerator. Config.: Listing G.1.

Figure 9.6: Benchmark: DCT, 4x. With four DCTs it is best to execute one instance in the CPU and the other (sequentially) in accelerator. Config.: Listing G.1.

takes 583 cycles and these are in sequence to the execution in the accelerator. The final speed-up, including all overheads, is $1.5 \times$.

When there are multiple kernels to be run it is possible to combine concurrent execution in accelerator and processor. Figure 9.5 and Figure 9.6 show execution of 2 and 4 independent DCT kernels, respectively. In cases when accelerator is used its utilization is controlled by the blocksize parameter $B$. When $B = 1$ (Figure 9.5.d, Figure 9.6.e) all computation is carried out in an accelerator and the processor is idle (see the ‘CPU’ time). When $B = 2$ (Figure 9.5.c, Figure 9.6.d) then effectively one
thread slot is occupied by the accelerator execution and one by the processor execution. When $B = 4$ (which makes sense only for the 4x DCT in Figure 9.6.c) there are four thread slots allocated, one for each instance of the kernel. The first instance occupies the accelerator and the other three are run in the processor.

In the 4x DCT the shortest execution time is with $B = 2$. In that case both the accelerator and the processor are used concurrently to compute independent tasks.

Figure 9.7 shows execution profile of two independent DCT kernels run: (a) in UTLEON3 only, and (b) in UTLEON3 with HWFAM. Time flows from top to bottom in the pictures. The vertical bar at the left hand side of each picture is the utilization of the processor pipeline: green is working, red is a stall. To the right of it the pipeline profile is analysed for different physical threads. In the case (b) the parallel execution of one of the DCT kernels in HWFAM is marked by the black strip next to the pipeline bar. The second DCT is executed in the processor concurrently to the accelerator. Stalls in its execution are caused by memory bus congestion when the HWFAM loads/stores the data buffers while UTLEON3 waits for a cacheline fetch.

**Experiments With Two Accelerators**

Execution with two instances of identical accelerators will be shown using the FIR benchmark. Figure 9.8 shows a plot of the execution time of one instance of the FIR kernel. The instance is of the size 32x768 ($t = 32, n = 737$) elements. The column ‘CPU’ indicates occupancy of the CPU, i.e. the number of committed instructions. The difference between the ‘CPU’ and ‘Total’ are stalls in the processor. The columns (a) and (b) are CPU-only executions, with warm and cold D-Cache, respectively. Since the working sets do not fit in the cache there is practically no difference in execution speed between cold and warm caches. Execution in the accelerator (c) is $6 \times$ faster.

Figure 9.9 shows a plot of the execution time of four instances of the FIR kernel run in parallel. The first group (a) depicts software-only execution. The plots (b)-(d) in Figure 9.9 show execution with hardware acceleration enabled, with different blocksizes in the top thread family. In the case (b) when blocksize $B = 1$ the kernels are executed sequentially since there is only one thread slot allowed for the top family. Logically, the sequential execution of kernels means that only one accelerator is utilized at any time. In the case (c) when blocksize $B = 2$ there are two slots for the top family, hence two FIR kernels execute concurrently in two accelerators. This case yields the minimal total execution time and the best speed-up $11 \times$. In the case (d) when blocksize $B = 4$ there are four slots for the top family, hence all four FIR kernels are created simultaneously. Since there are two hardware accelerators the first two FIR kernels use them, the other two are run in the CPU. The total execution time is dominated by the two instances run in CPU, and the speed-up over the case (a) is only $2 \times$.

**Results**

The final placement of a family of threads is decided at run-time in TMT when the creation event data-structure is available. The basic decision policy considers the momentary availability of hardware accelerators. There is no queueing of creation events waiting for accelerators. Instead, the placement is indirectly controlled via the blocksize parameter, a standard SVP interface for performance throttling.

Experimentally it was shown that the appropriate blocksize value for HWFAM-accelerated threads is equal to the number of accelerators available in hardware (blocksize = numberOfAccelerators). This is a necessary condition so that all threads of the family are executed in HWFAM. Good results are also achieved for the value ‘blocksize = numberOfAccelerators + 1’. In that case a thread is concurrently executed also in the processor.
9.2. CONTROLLING THE PROCESSOR/ACCELERATOR TASK PLACEMENT

Figure 9.7: Execution of two independent DCT kernels in UTLEON3 and/or HWFAM.
Figure 9.8: Benchmark: FIR, 1x, larger instances (32x768). Accelerator speed-up is $6 \times$. Config.: Listing G.2.

Figure 9.9: Benchmark: FIR, 4x, larger instances (32x768). Two accelerators ('Acc 1', 'Acc 2'). In BLOCKSIZE=1 all four FIR instances execute sequentially in one accelerator core. In BLOCKSIZE=2 all four FIR instances execute using both accelerators (the best finishing time). In BLOCKSIZE=4 two FIR instances execute in two accelerators and two FIRs execute in the CPU. Config.: Listing G.2.
Higher-than-the-optimal blocksize values cause over-utilization of the processor and, consequently, underutilization of accelerators because threads are allocated in the processor. Lower-than-the-optimal blocksize values cause underutilization of both the processor and accelerators.

9.2.2 Magnitude-Based Placement

The FIR accelerator computes one Multiply-Accumulate operation (MAC) per clock cycle, while the equivalent software implementation requires at least 4 cycles to perform the same computation.
(2x LOAD, 1x MUL, 1x ADD). However, for small problem sizes the total execution time in hardware accelerator may easily become dominated by data transfers and other control overheads. Figure 9.10 compares the FIR kernel execution times for the smaller problem sizes \( t \in \{4, 8, 16, 24\} \) and \( n \in \{24...128\} \). The line ‘CPU’ shows the baseline software-only execution in the microthreaded processor; the line ‘Acc’ shows execution that is always performed in dedicated accelerator in HW-FAM (though the computation is initialized from the processor). In each of the graphs there is an area of the smallest problem sizes for which the CPU execution is faster than in the accelerator. The split \( n_{\text{accmin}} \) is marked by vertical dashed line in the pictures.

The FIR kernel is implemented such that the number of threads in family corresponds to \( n - t \).

**Figure 9.11**: FIR kernel speed-up of the always-hardware (‘Acc...’) and the hybrid schemes compared to the always-software execution (speedup=1.0). For \( t \in \{4, 8, 16, 24\} \) and \( n \in \{24...128\} \). Config.: Listing G.3, G.4, G.5.
9.2. CONTROLLING THE PROCESSOR/ACCELERATOR TASK PLACEMENT

Using profiling for a fixed $t$ the value of $n_{\text{accmin}}$ when hardware execution is faster than in CPU can be determined. From this it is possible to compute the minimal family size (in terms of number of threads) that is suitable for execution in HWFAM. This minimal family size value is loaded into Thread Mapping Table (TMT) during set-up. As described in Subsection 8.5.4, during run-time the TMT compares the actual family size in creation events to that specified for a given family of threads. It refuses to place a family of threads in accelerator if the actual number of threads is smaller than it is given in TMT.

This dynamic magnitude-based placement strategy is marked ‘Hybrid’ in Figure 9.10 and Figure 9.11. For $n \geq n_{\text{accmin}}$ the family of threads is placed in the accelerator, otherwise the TMT forces execution in the processor. Figure 9.11 shows speed-up over software-only execution compared to the always-in-accelerator policy and the hybrid policy. As expected, the hybrid policy improves performance in the $n < n_{\text{accmin}}$ region. There is a small performance penalty due to latency in TMT, but this practically shows up only in the smallest FIR case $t = 4$.

The magnitude-based placement strategy is limited by the fact that decisions are performed only based on the number of threads in the top-level family. In the FIR kernel this means that the value of $t$, which is normally passed from processor to accelerator via one of the thread global registers, cannot influence any TMT decisions. Figure 9.12 shows performance profiles of FIR kernels with mixed $t$ but static $n_{\text{accmin}}$ in TMT. In the first plot the TMT treats all FIR thread families as if $t = 8$, in the second plot as if $t = 16$. In the first case some of the potential performance in larger instances is lost as they are needlessly kept in the CPU and not allowed in HWFAM. Conversely, in the second case the performance in smaller instances is lost.
CHAPTER 9. EVALUATING THE HARDWARE FAMILIES OF THREADS

<table>
<thead>
<tr>
<th>Component</th>
<th>BRAM</th>
<th>FFs</th>
<th>LUTs</th>
<th>DSP48E</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU UTILEON3</td>
<td>86</td>
<td>5405</td>
<td>10874</td>
<td>4</td>
</tr>
<tr>
<td>DMA + Bridge + Internal Bus</td>
<td>15</td>
<td>495</td>
<td>549</td>
<td>0</td>
</tr>
<tr>
<td>Job Control Unit (JCU)</td>
<td>4</td>
<td>126</td>
<td>409</td>
<td>0</td>
</tr>
<tr>
<td>FIR Accelerator</td>
<td>6</td>
<td>216</td>
<td>114</td>
<td>3</td>
</tr>
<tr>
<td>DCT Accelerator</td>
<td>2</td>
<td>542</td>
<td>451</td>
<td>4</td>
</tr>
</tbody>
</table>

Thread Mapping Table (TMT):
- \( r = 16 \) rows: 9 rows, 915 FFs, 644 LUTs, 0 DSP48E
- \( r = 8 \) rows: 8 rows, 665 FFs, 529 LUTs, 0 DSP48E
- \( r = 4 \) rows: 8 rows, 539 FFs, 495 LUTs, 0 DSP48E
- \( r = 2 \) rows: 8 rows, 475 FFs, 455 LUTs, 0 DSP48E

Table 9.1: HWFAM synthesis results in Xilinx Virtex 5 FPGA (XC5VLX110T). \( r \) = Number of rows in the Thread Mapping Table.

9.3 Implementation in FPGA

The system depicted in Figure 9.1 was synthesized using Synopsis Synplify D-2010.03 and Xilinx ISE 12.3 in the Xilinx Virtex 5 XC5VLX110T FPGA. Silicon area occupied by individual components in HWFAM is reported in Table 9.1.

Thread Mapping Table was implemented as an ideal fully associative content addressable memory (CAM). Lookups in the ideal TMT take only a single cycle, plus a constant overhead for other processing that could be pipelined. However, as can be seen in Table 9.1, the fully associative CAM implementation does not scale well in the FPGA technology. The number of rows \( r \) in TMT governs the number of classes of families of threads (i.e. distinct functions) that may be mapped to hardware accelerators. For example, the experimental set-up has 2x FIR and 1x DCT accelerator cores in HWFAM, hence it requires 2 rows in TMT to support the two distinct functions (FIR and DCT).

As usual there is a trade-off between the latency of CAM implementation and its silicon area. One extreme are CAMs with fully parallel lookup over all rows (time: \( O(1) \), space: \( O(r) \)), the other extreme is a CAM with a sequential lookup (time: \( O(r) \), space: \( O(r) \)). Higher CAM latency causes that creation events take longer time to process. The microthreaded processor was built with the intention to be able to tolerate various dynamic latencies in program execution.

9.4 Tolerance of the TMT Latency Scaling

In the previous section it was shown that the single-cycle lookup latency in the CAM within the TMT cannot be realistically maintained when TMT is scaled to higher number of entries because of the area resource consumption. The obvious way around this is to use a more conservative CAM implementation (area-wise), although that CAM will have a higher latency in lookups. As family creation events are routed through the TMT the question is the impact on the execution speed in the microthreaded processor.

To estimate the impact the TMT was augmented to emulate additional latency when processing creation events. The latency is configurable to an arbitrary number of clock cycles. The emulated latency is non-pipelined, i.e. creation events are processed strictly sequentially.

Figure 9.13.a shows the slow-down of all-software execution when scaling the TMT latency. (The

\[\text{space is } O(r) \text{ but with much lower constant than in the parallel CAM. Storage could be implemented by standard single-port RAM.}\]
9.5 SUMMARY

The chapter has provided the experimental evaluation of the fifth thesis contribution: (5) Bridging the gap between the data-driven microthreaded procedural computation with the special-purpose data-driven hardware in reconfigurable arrays.
CHAPTER 9. EVALUATING THE HARDWARE FAMILIES OF THREADS

Placement of thread families controlled by SVP blocksize.

The placement of thread families into a processor or accelerator is decided at run-time when the family is created. The decision in TMT considers the current resource occupancy—the availability of accelerators. The HWFAM does not queue creation events for later processing. The mapping of threads to processor or accelerators is controlled by the blocksize parameter, a standard SVP mechanism for performance throttling. The experiments have shown that the values ‘blocksize = numberOfAccelerators’ or ‘blocksize = numberOfAccelerators+1’ are usually good choices.

Placement of thread families controlled by workload size (number of threads).

If the work-load size is correlated with the number of threads the HWFAM scheme can switch creation events between accelerators and processor in run-time based on it. Small-sized thread families will not be transferred to an accelerator even if one is available because the fixed-costs associated with the move would outweigh benefits.

Latency tolerance.

Thread-family creation events are routed through the TMT in HWFAM. The microthreaded processor well tolerates new latencies that arise in TMT. The load on TMT processing can be lowered by marking a subset of families as virtual; the processor then sends only these ‘virtual’ families into TMT. Virtual families are marked by setting the LSB of their thread function addresses, hence it does not require new ISA instructions or registers.
IV Results and Conclusions
Chapter 10
Comparison of the Dataflow Approaches

This chapter compares the three dataflow approaches discussed in the previous chapters: (I.) vector processing in ASVP, (II.) pipelining in FTL, and (III.) microthreading in UTLEON3 using three benchmark kernels: Finite Impulse Response filter (FIR), Discrete Cosine Transform (DCT), and Mandelbrot fractal set (MF-set).

<table>
<thead>
<tr>
<th>Programming in TIME</th>
<th>Programming in SPACE</th>
<th>Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA-driven machine</td>
<td>UTLEON3 (microthread)</td>
<td>FIR filter</td>
</tr>
<tr>
<td>INSTRUCTION-driven machine</td>
<td>FTL-nets (pipelining)</td>
<td>DCT</td>
</tr>
<tr>
<td>LEON3 (scalar), GP-ASVP (vector)</td>
<td>Manual HDL</td>
<td>MF-set</td>
</tr>
</tbody>
</table>

Figure 10.1: Platforms and the benchmarks used.

10.1 Platforms

- Static scheduling is represented by three approaches:

1. Vector processing in GP-ASVP: the schedule of vector operations is static (specified in sCPU firmware). The DFU is implemented manually, the pipeline is controlled by a finite state machine.

2. Manual HDL: the FIR filter and the DCT benchmarks were developed in VHDL, the MF-set benchmark was developed in Xilinx System Generator. Both approaches lead to a specialized statically scheduled pipeline.

3. Scalar implementation in the LEON3 processor: LEON3 is a scalar RISC-style 32-bit processor implementing the SPARC ISA. It has an in-order statically scheduled pipeline (the peak IPC is 1). The D-Cache implements a write-through policy (all writes are immediately carried through
to the main memory). Any long-latency event (cache miss, integer multiply, FPU operation) causes the pipeline to stall.

**Dynamic scheduling** is represented by two approaches:

5. **Pipelining using FTL**: control-dataflow graph of an algorithm is represented in an FTL-net, the net is transformed into a pipelined hardware implementation. Operations are executed along the flows of tokens.

6. **Microthreaded implementation in the UTLEON3 processor**: UTLEON3 is LEON3 extended with the support for microthreading and hardware families of threads. The peak IPC remains 1 because there is only a single processing pipeline. Performance gains come from dynamic scheduling of microthreads in the pipeline so that unpredictable long-latency stalls are overcome by useful work in other threads.

<table>
<thead>
<tr>
<th>Platforms</th>
<th>Sched?</th>
<th>Hw?</th>
<th>FIR</th>
<th>DCT</th>
<th>MF-set</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP-ASVP, vector processing</td>
<td>SS</td>
<td>PI</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Manual HDL, static pipelining</td>
<td>SS</td>
<td>PD</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FTL, pipelining</td>
<td>DS</td>
<td>PD</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LEON3, scalar processor</td>
<td>SS</td>
<td>PI</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>UTLEON3, microthreading processor</td>
<td>DS</td>
<td>PI</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Table 10.2**: Platform characterization data used for the evaluation.

<table>
<thead>
<tr>
<th>Platforms</th>
<th>Area:</th>
<th>Frequency:</th>
<th>Techno.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP-ASVP</td>
<td>1781 slices</td>
<td>200MHz</td>
<td>Virtex 6 (xc6vlx240t)</td>
</tr>
<tr>
<td>Manual HDL</td>
<td>(Table 10.5)</td>
<td>150MHz</td>
<td>Virtex 6 (xc6vlx240t)</td>
</tr>
<tr>
<td>FTL</td>
<td>231 sl.</td>
<td>150MHz</td>
<td>Virtex 6 (xc6vlx240t)</td>
</tr>
<tr>
<td>LEON3</td>
<td>1927 slices</td>
<td>80MHz</td>
<td>Virtex 5 (xc5vlx110t-1)</td>
</tr>
<tr>
<td>UTLEON3</td>
<td>7988 slices</td>
<td>40MHz</td>
<td>Virtex 5 (xc5vlx110t-1)</td>
</tr>
</tbody>
</table>

The classification of platforms is summarised in Figure 10.1. The ASVP, LEON3, and UTLEON3 are programmed **procedurally in time**, by programs stored in memory. The FTL and static pipelining approaches represent **structural programming in space**, in FPGAs by bitstreams stored in configuration registers. Platforms’ characterization data (resource usage and operating frequency) that is used in comparisons below is summarised in Table 10.2.

Resource occupancy of the LEON3 and UTLEON3 processors in Virtex 5 FPGA (xc5vlx110t-1) and in the 90nm TSMC technology is in Table 10.3. In the FPGA the particular implementation of the UTLEON3 processor is $4.2 \times$ larger (slices) and $2.0 \times$ slower (critical path delay) than LEON3. The ASIC synthesis results are informative only; details can be found in [X.1].
10.2. PERFORMANCE METRICS

### Table 10.3: Resource utilization of the LEON3 and UTLEON3 processors in the Virtex 5 FPGA and in TSMC 90nm technology. For ASIC details see Appendix E in [X.1]. Configuration: ICache 1kB, DCache 1kB, associativity 4, RF 1024, FTT 32, TT 256.

<table>
<thead>
<tr>
<th>Processor</th>
<th>FF</th>
<th>LUT</th>
<th>BRAM</th>
<th>Slices</th>
<th>Delay</th>
<th>Area [mm$^2$]</th>
<th>Delay [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEON3</td>
<td>1525</td>
<td>4442</td>
<td>15</td>
<td>1927</td>
<td>12.5ns</td>
<td>0.53 mm$^2$</td>
<td>2.075 ns</td>
</tr>
<tr>
<td>UTLEON3</td>
<td>6822</td>
<td>16338</td>
<td>47</td>
<td>7988</td>
<td>25.0ns</td>
<td>4.40 mm$^2$</td>
<td>3.708 ns</td>
</tr>
</tbody>
</table>

### 10.2 Performance Metrics

**Execution Time Speed-up: $S$**

The speed-up $S$ compares raw execution times (in seconds) of two implementations:

$$ S = \frac{t_1}{t_2} \quad (10.1) $$

The GP-ASVP is taken as the baseline: $S_{\text{GPASVP}} = 1 \times$, except in the DCT case where the equivalent GP-ASVP implementation is not available. The GP-ASVP was selected as the baseline because: (a) it is the main theme of Part I of the thesis, and (b) it is a problem-independent and optimized hardware with a proven track record (e.g. it was used for the FG/BG image segmentation application).

The processors LEON3 and UTLEON3 have the speed-up less than $1 \times$ compared to the specialized hardware because they trade performance for flexibility and ease of programming.

**Time-Area Product: $ta$**

The performance metric that relates latency and resource usage is the product ‘*time* × *area*’, or $ta$ in short. The *time* is the task execution latency in seconds; the *area* is the occupied area measured in FPGA slices. For example, an implementation that occupies 2× more slices and is 2× faster has the same *time* × *area* performance as the original one. Logically, lower $ta$ values are better. The $ta$ values presented below are given in milliseconds × Slices.

The concept was defined in [WH97] as the functional density:

$$ D = \frac{1}{t \cdot a} \quad (10.2) $$

Higher functional density is better.

**Time-Area Speed-up: $F$**

The speed-up ratio $F$ compares the *time* × *area* products (or functional densities) of two implementations 1, 2:

$$ F_{2/1} = \frac{ta_1}{ta_2} = \frac{D_2}{D_1} \quad (10.3) $$

The implementation 1 may be considered as the baseline with which the implementation 2 is compared.

When $F_{2/1} = 1$ the performance scaling of the two implementations is identical. In such a case, if the second implementation is 3× faster it is also 3× larger in area. When $F_{2/1} > 1$ an FPGA slice in the implementation 2 is ‘better used’ (more productive) than in the implementation 1.
TABLE 10.4: Lines-of-Code (LOC) written by the author for implementing the algorithms in the platforms.

<table>
<thead>
<tr>
<th>Platform</th>
<th>FIR</th>
<th>DCT</th>
<th>MF-set</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP-ASVP</td>
<td>LOC firmware in C</td>
<td>120</td>
<td>388</td>
</tr>
<tr>
<td>Manual HDL</td>
<td>LOC VHDL</td>
<td>653</td>
<td>1002</td>
</tr>
<tr>
<td>FTL</td>
<td>LOC netlist in Python</td>
<td>194</td>
<td>226</td>
</tr>
<tr>
<td></td>
<td>LOC nested VHDL (types)</td>
<td>180</td>
<td>152</td>
</tr>
<tr>
<td></td>
<td>LOC wrapper VHDL</td>
<td>267</td>
<td>445</td>
</tr>
<tr>
<td></td>
<td>Generated LOC VHDL</td>
<td>(3331 to 28475)</td>
<td>(1979)</td>
</tr>
<tr>
<td>LEON3</td>
<td>LOC assembler</td>
<td>85</td>
<td>233</td>
</tr>
<tr>
<td>UTLEON3</td>
<td>LOC assembler</td>
<td>92</td>
<td>282</td>
</tr>
</tbody>
</table>

Example: Implementation No.1 occupies 100 slices and it has the latency 10ms. Implementation No.2 occupies 200 slices and it has the latency 5ms. The \( \text{time} \times \text{area} \) products (or functional densities) are identical: \( ta_1 = ta_2 = 1000 \text{ ms-slices} \).

If the task is sufficiently parallel and the finishing time 5ms per subtask is required then the implementation No.1 could be replicated twice in the chip so that two instances of the problem can be solved in parallel. This composite implementation No.3 has the \( \text{time} \times \text{area} \) product: \( ta_3 = (t_1/2) \times (a_1 \cdot 2) = ta_1 \). The \( F \)-speed-up is: \( F_{3/1} = ta_1/ta_3 = 1 \times \). It means the impl. No.3 is a linearly scaled version of the impl. No.1.

Now consider an implementation No.4 that occupies 300 slices and it has the latency 3ms. The \( \text{time} \times \text{area} \) product is: \( ta_4 = 900 \text{ ms-slices} \), which is better than No.1 and No.2. The \( F \)-speed-up is: \( F_{4/1} = ta_1/ta_4 = 1.11 \times \). It means the impl. No.4 uses the FPGA slices more efficiently than the impl. No.1. However, it does not mean that the impl. No.1 or No.2 are worthless; for instance, if the finishing time 2.5ms per subtask is required then it is more efficient to use two parallel instances of the impl. No.2 (or four of No.1) with the total resource requirement 400 slices instead of two instances of the No.3 that would occupy 600 slices (but deliver a better latency 1.5ms).

Instructions Per Cycle: IPC

The processors LEON3 and UTLEON3 are compared using the CPU-specific metric IPC. The IPC is the number of instructions executed per clock-tick. Microthreading increases IPC because stalls are overlapped with instruction execution in different threads.

10.3 Evaluation

10.3.1 Lines of Code

The basic metric to compare the human effort in a programming discipline is the number of lines of code (LOC). The LOCs for our algorithms and platforms are reported in Table 10.4. In LEON3 and UTLEON3 the LOC is reported for assembler source codes. In GP-ASVP the LOC is reported for the firmware source code written in the C language.

The Manual HDL is in LOC of VHDL code, except in the MF-set implementation which is specified in the Xilinx System Generator 14.4 tool. Although System Generator is a graphical tool the abstractions used are very near to actual hardware models.
10.3. EVALUATION

In FTL the code base is composed of three parts: (1) an FTL netlist instantiated in a Python code, (2) a nested VHDL code, and (3) a wrapper VHDL code. The FTL code (1) is written in Python as a series of object instantiations. The nested VHDL code (2) defines data types and primitive functions that are ‘called’ from the netlist. In the future this code should be lifted into the higher-level FTL specification. The wrapper VHDL code (3) is an adapter that interfaces tool-generated entities to the environment, such as to the ASVP template. Finally, the number of VHDL code lines generated by the FTL tool is shown in parentheses.

(Although System Generator is a graphical tool the abstractions used are very near to actual hardware models.)

10.3.2 Finite Impulse Response Filter (FIR)

The Finite Impulse Response (FIR) filter is one of the basic algorithms in the Digital Signal Processing (DSP) field. It is widely used in many applications because of the inherent stability of the filter due to the lack of feedback, and a simple data-parallel implementation. A detailed analysis can be found in Section F.1.

Among the three benchmark algorithms used here the FIR filter is the most simple and regular. The filter has one parameter \( \vec{t} \) (weights or taps) which is typically constant throughout a long-term use of the filter. The filter inputs and outputs are vectors or streams of samples (floating-point values).

Implementations (FIR)

- **Vector Implementation in GP-ASVP:** In the GP-ASVP core the filter is implemented in firmware by repeatedly calling the dot-product (DPROD) vector operation. Each DPROD computes one result word; the latency is \( |\vec{t}| + (1 + \log_2 k) k - 1 \) clock-ticks, where \( k \) is the latency of the floating-point adder in DFU.

- **Manual Implementation in VHDL** unrolls the FIR computation in space: each tap is implemented by a separate instance of a multiplier core and the summation is implemented by an adder tree. The pipeline can input and output one word per clock-tick.

- **Pipelined Implementation in FTL:** The pipelined implementation in FTL is depicted in Figure 10.2. The filter is unrolled in space: each tap is implemented by a separate instance of a multiplier core and the summation is implemented by an adder tree. The pipeline can input and output one word per clock-tick.

- **Scalar Implementation in the LEON3 Processor:** In the scalar LEON3 processor the filter is implemented by a for loop that computes the dot-product, as shown in Listing F.1. One step of the dot-product requires 4 instructions (2 loads, a multiplication, an addition), plus an overhead of the loop control. If the load instruction causes a cache-miss the program is stalled until data is fetched from the main memory. The simple in-order issue and completion of instructions in LEON3 means that any long-latency instruction such as a floating-point operation causes a stall.

- **Microthreaded Implementation in the UTLEON3 Processor:** In the microthreaded processor UTLEON3 the loop control is conveniently handled in hardware by a family of threads, hence there is no overhead for executing conditional branches. Cache misses and other long-latency operations are handled by switching the active thread in the pipeline. The latency of a thread switch is between 0 and 3 clock-ticks (Figure E.4). If all latencies are hidden by a perfect dynamic schedule the cost of computing one result word of the FIR filter is \( 4 \cdot |\vec{t}| \) instructions.
ASVP is Table 10.6: FIR filter: GP-ASVP and FTL-net. The FPGA is Virtex 6 XC6VLX240T. The area of GP-FIR filter: Area utilization $a$ in slices of the FTL-net version, for different FPGAs and lengths of the weight vector $i$. The cases '(x)' are overmapped on DSP48 blocks. $f_0 = 100MHz$, $f_{VP} = 150MHz$.

<table>
<thead>
<tr>
<th>Chip</th>
<th>Manual HDL</th>
<th>FTL (FIR-ASVP)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$t = 4$</td>
<td>$t = 8$</td>
</tr>
<tr>
<td>Virtex 5 xc5vlx110t</td>
<td>1142</td>
<td>1797</td>
</tr>
<tr>
<td>Virtex 6 xc6vlx240t</td>
<td>1017</td>
<td>1588</td>
</tr>
<tr>
<td>Spartan 6 xc6slx45t</td>
<td>1205</td>
<td>1950</td>
</tr>
<tr>
<td>Kintex 7 xc7k325t</td>
<td>1029</td>
<td>1609</td>
</tr>
<tr>
<td></td>
<td>AVG Resource increase:</td>
<td>1.39×</td>
</tr>
</tbody>
</table>

Table 10.5: FIR filter: GP-ASVP and FTL-net. The FPGA is Virtex 6 XC6VLX240T. The area of GP-ASVP is $a = 1781$ slices, it has $f_0 = 100MHz$, $f_{VP} = 200MHz$ (see Section 4.3). The FTL-net implementation is assumed to run at 150MHz. The $t_a$ product is given in $ms \times Slices$ (smaller is better). $S$ is the time-only speed-up, $F$ is the time $\times$ area speed-up. The GP-ASVP is taken as the baseline, hence $F_{GPASVP} = 1$, $S_{GPASVP} = 1$.

| $|t|$ | $|n|$ | $t_a$ [µs] | $t_a$ SGPASVP | $F_{GPASVP}$ | $t_a$ FTL | $S_{FTL}$ | $F_{FTL}$ |
|---|---|---|---|---|---|---|---|
| 4  | 128 | 24.09µs | 42.9 | 1.0× | 1465 | 4.15µs | 6.1 | 5.8× | 7.1× |
|    | 256 | 44.55µs | 79.3 | 2.0× | 2482 | 4.21µs | 10.4 | 5.9× | 4.2× |
|    | 768 | 127.50µs | 22.7 | 2.0× | 4561 | 4.27µs | 19.5 | 6.6× | 2.6× |
| 8  | 128 | 28.53µs | 50.5 | 1.0× | 8837 | 4.43µs | 39.3 | 7.3× | 1.5× |
|    | 256 | 55.89µs | 99.5 | 2.0× |     | 5.17µs | 23.6 | 10.8× | 4.2× |
|    | 768 | 167.13µs | 297.7 | 2.0× |     | 8.59µs | 39.1 | 19.5× | 7.6× |
| 16 | 128 | 32.67µs | 58.2 | 1.0× |     | 5.29µs | 46.7 | 13.3× | 2.7× |
|    | 256 | 70.41µs | 125.4 | 2.0× |     | 8.71µs | 77.0 | 25.6× | 5.2× |
| 32 | 768 | 222.63µs | 396.5 | 2.0× |     |     |     |     |     |

Table 10.6: LEON3 vs. UTLEON3 processors. IPC=Instructions per Cycle. IC=Instruction Count. $S$ is the time-only speed-up, $F$ is the time $\times$ area speed-up. Area: $a(LEON3) = 1927$ slices, $a(UTLEON3) = 7998$ slices. Delay: $T(LEON3) = 12.5ns$, $T(UTLEON3) = 25ns$.

| $|t|$ | $|n|$ | $IPC$ | $ta$ | $S_{L3}$ | $F_{L3}$ | $IPC$ | $ta$ | $S_{UL3}$ | $F_{UL3}$ |
|---|---|---|---|---|---|---|---|---|---|
| 4  | 128 | 76.1µs | 0.56 | 147 | 0.32× | 0.29× | 149.5µs | 0.68 | 1194 | 0.16× | 0.036× |
|    | 256 | 152.9µs | 0.56 | 295 | 0.29× | 0.27× | 293.5µs | 0.69 | 2344 | 0.15× | 0.034× |
|    | 768 | 460.1µs | 0.56 | 887 | 0.28× | 0.26× | 958.8µs | 0.64 | 7659 | 0.13× | 0.030× |
| 8  | 128 | 128.5µs | 0.55 | 248 | 0.19× | 0.18× | 226.3µs | 0.67 | 1807 | 0.11× | 0.024× |
|    | 256 | 262.9µs | 0.56 | 507 | 0.18× | 0.16× | 456.8µs | 0.68 | 3649 | 0.10× | 0.023× |
|    | 768 | 800.5µs | 0.56 | 1543 | 0.17× | 0.16× | 1428.4µs | 0.67 | 11410 | 0.09× | 0.021× |
| 16 | 128 | 222.4µs | 0.55 | 429 | 0.13× | 0.12× | 366.5µs | 0.67 | 2927 | 0.08× | 0.017× |
|    | 256 | 472.0µs | 0.56 | 909 | 0.12× | 0.11× | 769.5µs | 0.68 | 6147 | 0.07× | 0.016× |
|    | 768 | 1470.4µs | 0.56 | 2833 | 0.11× | 0.11× | 2428.2µs | 0.67 | 19396 | 0.07× | 0.015× |
| 32 | 128 | 366.9µs | 0.55 | 707 | 0.09× | 0.08× | 583.7µs | 0.66 | 4662 | 0.06× | 0.012× |
|    | 256 | 846.9µs | 0.56 | 1632 | 0.08× | 0.08× | 1340.4µs | 0.66 | 10707 | 0.05× | 0.012× |
|    | 768 | 2766.9µs | 0.56 | 5332 | 0.08× | 0.07× | 3438.0µs | 0.66 | 35019 | 0.05× | 0.011× |
### Results (FIR)

The **FPGA areas** occupied by the Manual HDL and the FTL-net implementations are reported in Table 10.5. Depending on the FPGA technology used the number of slices varies only slightly. Resource utilization depends mainly on the number of **weights** (taps, $|\vec{t}|$) of the FIR filter that is supported in hardware. A filter with $|\vec{t}|$ taps requires $|\vec{t}|$ multipliers and $|\vec{t}| - 1$ adders.

The resource utilization (Slices) of the FTL-net implementations is about $1.5 \times$ higher than in the Manual HDL approaches. It is caused by the use of FIFO queues at the outputs of Floating-Point cores in the FTL-net version. The basic method for embedding standard (legacy) FP cores in FTL-net designs (Section C.2) cannot perform dataflow scheduling inside the cores’ pipelines. This makes the design not fully elastic and the dataflow scheduling is not perfect. A work-around is to add FIFO queues at the outputs at the cost of increased resource utilization.

**Execution latencies** in GP-ASVP and in FTL-net implementations are reported in Table 10.6. The following vector lengths were selected: $|\vec{n}| \in \{4, 8, 16, 32\}$ and $|\vec{t}| \in \{128, 256, 768\}$. The reported numbers are for the Virtex 6 implementation technology. In GP-ASVP the execution latency is proportional to the product $|\vec{n}| \cdot |\vec{t}|$ which is the number of multiplication operations. The FPGA area...
is constant. In FTL-net implementation the execution latency is proportional to $|\vec{n}|$ (number of input samples), but it does not depend on $|\vec{t}|$ (the number of weights). The FPGA area is proportional to $|\vec{t}|$.

In Tables 10.6 and 10.7 the time × area product (abbrev. ta) is reported in ms × Slices. In GP-ASVP, LEON3 and UTLEON3 the areas (resource utilizations) are constant because the hardware is problem-independent. For the $F$-Speed-ups the GP-ASVP’s ta-products are taken as the baseline, hence $F_{GPASVP} = 1$ by definition.

In the FTL-net version (‘FIR-ASVP’) the highest $F$-speed-up was observed for $|\vec{t}| = 4$, $|\vec{n}| = 768$, i.e. for the shortest weights vector and the longest samples vector. The improvement $F = 18.4\times$ means that an FPGA slice used in the FTL-net version of the FIR filter is 18.4 times more productive (‘better used’) than when used for GP-ASVP. This ‘improvement’ factor is the highest for short $\vec{t}$ and long $\vec{n}$, and lowest for the long $\vec{t}$ and short $\vec{n}$. But even in the worst case observed: $|\vec{t}| = 32$, $|\vec{n}| = 128$, the comparative advantage of the FTL-net implementation is $1.5\times$.

The peak throughput of the GP-ASVP implementation is 1 result per $|\vec{t}|$ clock-ticks, a consequence of having only one multiplier and adder cores in the GP-ASVP. The FTL-net implementation has the throughput 1 result per clock-tick.

Table 10.8 compares performance of the Manual HDL and FTL-net implementations (the latter numbers are repeated from Table 10.6 for clarity). The $S$-speed-ups (without considering resource utilization) are identical between the two versions ($S_{Man} = S_{FTL}$). The $F$-speed-ups are slightly higher in the Manual HDL implementations ($F_{Man} > F_{FTL}$) because of the lower resource utilization (see above).

Comparison of the scalar and microthreaded implementations is in Table 10.7. The scalar LEON3 implementation suffers mainly from the latency in the multiplier which causes stalls for 4 clock-ticks, and to a smaller extent from D-Cache misses. Latency tolerance improves the IPC from 0.56 in LEON3 to 0.69 in UTLEON3. The $F$-speed-ups of both processors are below $1\times$ hence using the processors for single-purpose computations is a worse choice than the GP-ASVP and FTL-net approaches. The $F$-speed-ups of both processors have a decreasing tendency with longer input vectors or, conversely, the specialized hardware realized by GP-ASVP and FTL-nets is more productive for larger input data.

The $F$-speed-up of UTLEON3 compared to LEON3 is roughly $7\times$ smaller. This is caused mainly by the $4.2\times$ larger resource occupancy and $2\times$ lower clock frequency. The improved pipeline efficiency (IPC) and latency tolerance in UTLEON3 cannot make up for the larger size and longer clock-cycle.

10.3.3 Discrete Cosine Transform (2D-8x8 DCT)

The second benchmark kernel is a forward 2D discrete cosine transform used in the JPEG compression algorithm. The DCT is applied on fixed-sized blocks of $8 \times 8$ pixels. The input is a matrix of 64 words (pixels), the output is 64 words of frequency components. The particular implementation used here is optimized to minimize the number of multiplicative operations required [LLM89]. Compared to FIR the DCT algorithm operates in $8 \times 8$ blocks rather than in streams of input samples, and it is more complex. The running time of the DCT algorithm is constant irrespective of the real input data.

The DCT consists of two phases: row phase and column phase. In the row phase each row (8 pixels) is independently read, transformed, and written back in the same place. The column phase starts once all eight rows are processed. In the column phase each column (8 pixels) is independently read, transformed, and written back in the same place. The transformation applied to each row or column consists of 12 integer multiplications and 32 additions or subtractions.
Implementations (DCT)

▷ **Manually Pipelined VHDL Implementation:** An input 64-pixel matrix is stored in a local memory accessible via a single port. A row or a column of 8 pixels is read in 8 clock-ticks. The compute pipeline has 5 stages (Figure 10.3.a); at the front it is fed by the 8-pixel vector, at the back a single output pixel is generated. The same 8-pixel vector must be fed in the pipeline repeatedly 8 times to compute new values of all the elements in row or column. Scheduling is static: each compute operation is assigned to one of the 5 stages of the pipeline.

▷ **Pipelined Implementation in FTL:** The FTL-net implementation processes the whole 8-pixel row or column and generates all eight new results in a single pipeline (Figure 10.3.b). Reading and writing of the pixels must be serialized because of single-port memories. An intermediate buffer for storing an $8 \times 8$ temporary matrix is used since the pipeline has higher throughput. Dynamic schedule of operations within the data-dependency graph depends on the availability of input data and by the back-pressure of output buffers if the writing channel lags behind.

▷ **Scalar Implementation in the LEON3 Processor:** The implementation used in the scalar processor is a C code published by the Independent JPEG Group\(^1\) as distributed in the version jpeg-6a. The code contains two for loops: the first processes matrix rows, the second matrix columns.

▷ **Microthreaded Implementation in the UTLEON3 Processor:** The original C code contains two for loops: one operating over the rows of the input matrix, the second over the columns of the matrix. Both loops have been transformed into families of threads; the bodies of the loops become thread code. Speed-up over the LEON3 implementation is achieved by overcoming stalls caused by long-latency multiply operations by thread switch.

Results (DCT)

The performances of the statically pipelined and the FTL-net versions are reported in Table 10.9 and Table 10.10. The statically pipelined version is smaller but also slower because some internal resources are shared. Conversely, the FTL-net version is $1.9 \times$ larger in slices and $3.9 \times$ larger in DSP48E blocks because it implements a complete 8-point 1D-DCT in hardware. Nevertheless, the time × area products are approximately the same\(^2\), hence $F \approx 1 \times$. It means that the additional hardware in the

\(^{1}\)http://www.ijg.org/  
\(^{2}\)For simplicity not counting the DSP48E blocks.
10.3.4 Mandelbrot Fractal Set (MF-set)

The algorithm for the Mandelbrot fractal set has already been described in Subsection 4.4.2 and Section 7.6. Compared to FIR and DCT the running time of the MF-set algorithm is variable: the finishing time depends on the number of required iterations of the basic MF-set loop which in turn depends on the pixel position.

Implementations (MF-set)

- Vector Implementation in GP-ASVP has been described in Subsection 4.4.2.

- Manual HDL Implementation in System Generator: Figure 10.5 depicts the model of the MF-set pipeline in Xilinx System Generator 14.4. (Although System Generator is a graphical tool the abstractions used are very near to actual hardware models.) The model is conceptually similar to the FTL-net implementation described in Section 7.6. The difference is that the pipeline is specified

<table>
<thead>
<tr>
<th>Chip</th>
<th>Manual VHDL Slices</th>
<th>DSP48E</th>
<th>FTL Slices</th>
<th>DSP48E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtex 5</td>
<td>294</td>
<td>6</td>
<td>558</td>
<td>22</td>
</tr>
<tr>
<td>Virtex 6</td>
<td>231</td>
<td>5</td>
<td>516</td>
<td>20</td>
</tr>
<tr>
<td>Spartan 6</td>
<td>251</td>
<td>5</td>
<td>473</td>
<td>20</td>
</tr>
<tr>
<td>Kintex 7</td>
<td>259</td>
<td>5</td>
<td>448</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 10.9: DCT: Manual static pipelining vs. FTL-net, used resources.

<table>
<thead>
<tr>
<th>Chip</th>
<th>Manual VHDL time [µs]</th>
<th>ta</th>
<th>$S_{Manual}$</th>
<th>$F_{Manual}$</th>
<th>FTL time [µs]</th>
<th>ta</th>
<th>$S_{FTL}$</th>
<th>$F_{FTL}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtex 5</td>
<td>2.35µs</td>
<td>0.690</td>
<td>1.0×</td>
<td>1.0×</td>
<td>1.08µs</td>
<td>0.603</td>
<td>1.1×</td>
<td></td>
</tr>
<tr>
<td>Virtex 6</td>
<td>0.542</td>
<td>0.589</td>
<td>1.0×</td>
<td>1.0×</td>
<td>0.557</td>
<td>0.511</td>
<td>1.0×</td>
<td>1.2×</td>
</tr>
<tr>
<td>Spartan 6</td>
<td>0.680</td>
<td></td>
<td>1.0×</td>
<td>1.0×</td>
<td>0.484</td>
<td></td>
<td>1.3×</td>
<td></td>
</tr>
</tbody>
</table>

Table 10.10: DCT: Manual static pipelining vs. FTL-net, finishing times. The $ta$ product is given in $ms \times Slices$. The clock frequency is assumed 150MHz. $F_{Manual} = 1 \times$ is the baseline.

Table 10.11: DCT: Finishing times in LEON3 vs. UTLEON3.

FTL-net version that handles the data-driven control of the circuit is negligible. The small $F$-speed-up of the FTL-net version relative to the manual approach can be attributed to differences in the circuit architecture and measurement uncertainty. (Any significant $F$-speed-up relative to manually created design is improbable.)

Comparison of the scalar and microthreaded implementations is in Table 10.11. The UTLEON3 processor improves the IPC from 0.48 to 0.76, however the $F$-speed-ups are far below 1× in both cases.
10.3. EVALUATION

Figure 10.4: Still images from a video demonstrating the execution speed of the microthreaded UTLEON3 vs. the scalar LEON3. Each processor computes half of the image of the Mandelbrot fractal set. The microthreaded UTLEON3 (top half of the screen) is roughly $2 \times$ faster than LEON3 (bottom half of the screen) that runs on the same frequency. See https://www.youtube.com/watch?v=frQ7713zBDU.

manually and that it is scheduled statically. Any stall on the input or the output will stall the whole pipeline.

▷ Pipelined Implementation in FTL has been described in Section 7.6.

▷ Scalar Implementation in the LEON3 Processor: The scalar processor implementation uses 32-bit fixed-point (integer) arithmetic coded in LEON3/SPARC assembly language. The code uses two nested loops: the outer for loop iterates over all pixels, the inner while loop determines the number of iterations until the pixel escapes the MF-set.

▷ Microthreaded Implementation in the UTLEON3 Processor: The microthreaded implementation concurrently executes computations of different pixels, but it does not overlap computation inside the pixel code. The outer for loop is transformed into a family of threads, but the inner while loop is coded in the legacy style using standard conditional branches. Speed-up over the LEON3 implementation comes mainly from masking stalls caused by long-latency multiply operations by thread switch.

Results (MF-set)

Resource utilization of the Manual HDL and FTL-net implementations is given in Table 10.12. The FTL-net version is $1.16 \times$ larger in FPGA slices and uses $1.75 \times$ more DSP blocks. The Manual HDL version uses FP cores with 4-cycle latency, the FTL-net version uses FP cores with 6-cycle latency; hence the higher utilization of the DSP blocks.

Execution times of the MF-set in GP-ASVP, FTL-net, LEON3 and UTLEON3 are given in Table 10.13. Results are reported for a block of 100 pixels with the ratio of the in-set pixels either 0% or 100%. (In-set pixels iterate maxiter-times in the loop or pipeline, hence the actual compute load is different.) The $ta$-product of the GP-ASVP is taken as the baseline, hence $F_{GPASVP} = 1 \times$ by definition.

As discussed in Section 7.6 the FTL-net version (MF-ASVP) is orders of magnitude faster than GP-ASVP. In the 0%-in-set case the GP-ASVP is inefficient by $100.5 \times$ as it has to perform many redundant computations. In the 100%-in-set case the FTL-net version beats GP-ASVP by $13.8 \times$ in the $time \times area$ product because it has a comparable resource utilization (Figure 7.16) but much higher throughput.

The $S$-speed-ups (without considering the resource utilization) of the Manual HDL and FTL-net versions are practically identical. The $F$-speed-ups are slightly better for the Manual HDL because of
Figure 10.5: MF-set: Manually specified static pipeline implementation (HDL) in Xilinx System Generator.
10.4 Sensitivity to Unpredictable Latencies

The previous evaluations have shown that microthreading as currently implemented in the UTLEON3 processor does not bring the expected speed-ups compared to the LEON3 processor. The main cause is that the FPGA implementation of UTLEON3 is 4.2× larger in resources and 2× slower in clock frequency due to longer critical path. This itself gives the LEON3 processor an 8.4× advantage over UTLEON3 in the F-speed-ups.

Table 10.14 compares finishing times of the MF-set algorithm in LEON3 and UTLEON3 in clock-cycles without considering different clock-cycle periods. The UTLEON3 improves IPC (pipeline efficiency) from 0.53 to 0.84 (on average), and it has the speed-up 1.59×. The IPC depends on the number of threads (pixels) computed in the MF-set benchmark: more threads allow for better overlap of stalls. This tendency is plotted in Table 10.6 which shows the IPC (vertical axis) relative to the total amount of work in the benchmark (horizontal axis) represented by the Instruction Count (IC) or the number of pixels (= threads).

Each latency source could be either pipelined or non-pipelined. Pipelined latency allows multiple
<table>
<thead>
<tr>
<th># pixels (threads)</th>
<th>LEON3 (scalar)</th>
<th>UTLEON3 (microthread)</th>
<th>Speed-up (cc)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IC time [cc]</td>
<td>IPC</td>
<td>IC time [cc]</td>
</tr>
<tr>
<td>3</td>
<td>1023 2479 0.41</td>
<td>1028 1883 0.55</td>
<td>1.32x</td>
</tr>
<tr>
<td>7</td>
<td>3035 6525 0.47</td>
<td>3020 3902 0.77</td>
<td>1.67x</td>
</tr>
<tr>
<td>11</td>
<td>5221 10825 0.48</td>
<td>5186 6081 0.85</td>
<td>1.78x</td>
</tr>
<tr>
<td>15</td>
<td>8016 15623 0.53</td>
<td>7961 9083 0.88</td>
<td>1.68x</td>
</tr>
<tr>
<td>21</td>
<td>15732 28283 0.56</td>
<td>15647 18117 0.86</td>
<td>1.56x</td>
</tr>
<tr>
<td>31</td>
<td>25982 45624 0.57</td>
<td>25847 28649 0.90</td>
<td>1.59x</td>
</tr>
<tr>
<td>41</td>
<td>36754 64194 0.57</td>
<td>36569 40488 0.90</td>
<td>1.59x</td>
</tr>
<tr>
<td>51</td>
<td>44568 77390 0.58</td>
<td>44333 48838 0.91</td>
<td>1.58x</td>
</tr>
<tr>
<td>71</td>
<td>69277 119163 0.58</td>
<td>68942 76166 0.91</td>
<td>1.56x</td>
</tr>
<tr>
<td>91</td>
<td>84122 143400 0.59</td>
<td>83687 91947 0.91</td>
<td>1.56x</td>
</tr>
<tr>
<td>Average:</td>
<td>0.53</td>
<td>0.84</td>
<td>1.59x</td>
</tr>
</tbody>
</table>

Table 10.14: MF-set: LEON3 and UTLEON3 processors, finishing times. The speed-up compares execution times in clock-cycles.

Figure 10.6: MF-set: LEON3 and UTLEON3 processors, Instructions per Cycle (IPC). The horizontal axis is the number of pixels (threads) computed. In UTLEON3 the threads may run concurrently, more threads improve pipeline efficiency.
requests to be present in a module in different states of completion. New requests can be accepted and worked on while others have not been completed still. Non-pipelined latency allows only a single request to be processed at a time from beginning to end. New requests are queued.

\[
\text{Pipelined module: } \text{initiation\_interval} < \text{latency} \\
\text{Non-pipelined module: } \text{initiation\_interval} \geq \text{latency}
\]  

Figure 10.7 schematically shows how latencies cumulate in a (microthreaded) processor system. A typical source of a latency is a cache miss during a load instruction. The cache miss is resolved by fetching a particular cache-line from the main memory (DRAM) or a higher-level cache. Cache misses happen for individual words but they are resolved by pulling in complete cache-lines to take advantage of the space locality in data.

In a classical scalar processor without any data-driven processing, such as LEON3, the processor pipeline (IU) stalls until a cache miss is resolved. In a data-driven processor, such as UTLEON3, the processor pipeline continues to execute other independent instructions. In microthreading the independent instructions are obtained by switching to another thread.

Figure 10.8 shows how latencies cumulate in the ASVP core along the processing pipeline. There is no implicit hardware synchronization between the compute units inside the core (in the picture from the ‘Local mems.’ to the right) with external data transfer engines. The synchronization is realized in firmware using programmed communication.
10.4.1 Latency Tolerance in Processors

In the UTLEON3 processor a pipelined latency source is an integer multiplier (default latency 4 cc) and a floating-point unit. A non-pipelined latency source is a cache miss because the system bus (AMBA AHB) and the processor do not support multiple outstanding memory requests on the bus. My analysis published in [X.11] shows that microthreading can overcome unpredictable pipelined long-latency stalls given enough independent workload, but it cannot easily overcome non-pipelined stalls that cause congestions.

Figure 10.9 plots the execution efficiency (IPC) relative to a non-pipelined latency incurred in an artificial benchmark. The benchmark is implemented as a for-loop (or family of threads); the body of the loop loads two values from two arrays (with configurable stride), multiplies them (the pipelined long-latency operation) and stores the result in another array. The benchmark (Listing 10.1) is configurable with the amount of independent instructions inserted between the effective ones to simulate large workloads.

Once the memory bus is congested the IPC drops irrespective of the parallelism level, which is controlled by the parameter blocksize. An altogether different situation is depicted in Figure 10.10. Pipelined long-latency stalls are overcome by rescheduling to independent parallel workloads. The parallelism is regulated by the blocksize parameter. Generally, higher blocksize means more parallelism at the cost of increased dynamic resource usage (chiefly by allocations in the register file and the thread table).

The conclusion of [X.11] is that microthreading requires a high-performance memory bus that can pipeline many outstanding requests to higher-level caches and to the main memory. Otherwise the benefits of microthreading may be wasted.

Figures 10.11, 10.12 and 10.13 compare the IPC of the LEON3 and UTLEON3 processors on the FIR, DCT and MF-set benchmarks, respectively. The horizontal axis is an artificial pipelined latency introduced in the multiplier unit in processors, from 0 (no additional latency) to 16 clock-cycles. In LEON3 any latency that stalls instructions in the pipeline causes the drop in IPC. In UTLEON3 the stalls are overcome by thread switches.

10.4.2 Latency Tolerance in Specialized Hardware

Similarly to processors, specialized hardware is vulnerable to memory access latencies too. In the Example 2 in Subsection 2.2.2 it was shown that for sufficiently large design in FPGA the automatic placement and routing tools may fail in achieving the timing closure because some FPGA resources (BlockRAMs) used in the design may be mapped too far from the connecting logic. If manual placement is not desirable a possible alternative solution involves using several empty pipeline stages just for accessing the memory. This solution does not lower average throughput, but it affects the latency.

Potential consequences are demonstrated in the following experiment. The ASVP core and the simulation environment was modified to incur a latency on all accesses to the local data memory banks. The latency is pipelined, hence it does not compromise the peak theoretical throughput. Table 10.15 shows the degradation of the execution times when running the MF-set benchmark in GP-ASVP and MF-ASVP (FTL-net) cores. In the latter case the results are reported separately for 0% and 100% pixels iterating MaxIter-times in the loop.

In GP-ASVP the performance degrades because the memory access latency contributes to start-up latencies of all vector instructions. In MF-ASVP (FTL) the local memory is accessed only once per pixel for reading and once for writing, all other computation happens locally within the specialized pipeline.
10.4. SENSITIVITY TO UNPREDICTABLE LATENCIES

Figure 10.9: Efficiency (IPC) of the non-pipelined long-latency operations. Once the memory bus is congested increasing the blocksize value does not improve the execution efficiency. More details in [X.11].

Figure 10.10: Efficiency (IPC) of the pipelined long-latency operations. For short latencies the efficiency is optimal even with small blocksize values (the processor is saturated) and the effect of the ‘swch’ modifier is most profound. Longer latencies require higher blocksize values to stay in the saturation. More details in [X.11].
Listing 10.1: Assembly source code of the kernel of the artificial benchmark for latency measurements in UTLEON3. Details in [X.11].

10.5 Summary

Vector processing in GP-ASVP: The ASVP approach is costly when the generic DFU is used (the GP-ASVP). Evaluation using the FIR and MF-set applications showed that the ‘productivity’ of an FPGA slice (the time × area product) is often an order of magnitude lower in GP-ASVP compared to a fully specialized hardware in FPGA.

Manual HDL: Manual HDL implementations were included to evaluate the impact of the more complex hardware in the FTL approach. In the FIR filter and MF-set benchmarks the higher resource utilization in the FTL approach is due to technical issues in embedding legacy FP cores. In the integer-only DCT benchmark the resource utilization and the time × area product of both approaches are practically the same.

Pipelining using FTL: In FTL the cost of additional hardware that implements bookkeeping and scheduling of token flows is negligible relative to the data path. Centralized finite state machine, typically used in manual approaches, is replaced by a distributed FSM. Issues exist in the way legacy pipelined cores are embedded in FTL-nets and in proper sizing of FIFO queues in cyclic (closed) pipelines.

Scalar implementation in the LEON3 processor: The observed IPC of the LEON3 processor was measured between 0.48 (in DCT) and 0.56 (in FIR). The stalls were mainly caused by the mul instructions that require 4 clock-ticks to complete instead of the usual 1 clock-tick for other
integer operations. In the benchmarks the D-Cache misses were not so prominent because the kernels have relatively small working sets that easily fit in caches.

**Microthreaded implementation in the UTLEON3 processor** : The observed IPC of the microthreaded UTLEON3 processor was measured between 0.67 (in FIR) and 0.90 (in MF-set). The higher IPC compared to LEON3 comes from overlapping the long-latency mul instructions and cache stalls with execution of instructions in the processor pipeline. However, because of the much higher resource utilization in FPGA and a slower clock frequency the current implementation of UTLEON3 does not deliver better performance than LEON3.

The low performance of the processors LEON3 and UTLEON3 compared to the other approaches ($S, F \ll 1$) is the price paid for a universal problem-independent hardware with an ‘easy’ programming model. The performance of the GP-ASVP approach is in between of the processors and specialized (problem-dependent) hardware (manual VHDL or FTL). Again, the price is due to the reusability of the GP-ASVP hardware over a range of applications by changing the firmware code.
Figure 10.12: DCT: LEON3 vs. UTLEON3, sensitivity of IPC to unpredictable latencies.

Figure 10.13: MF-set: LEON3 vs. UTLEON3, sensitivity of IPC to unpredictable latencies.
### Table 10.15: MF-set: Degradation of the execution time when the access latency to local memory is increased.

<table>
<thead>
<tr>
<th>Access Latency</th>
<th>GP-ASVP</th>
<th>MF-ASVP (FTL)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td>1</td>
<td>1.000×</td>
<td>1.000×</td>
</tr>
<tr>
<td>2</td>
<td>0.997×</td>
<td>1.000×</td>
</tr>
<tr>
<td>3</td>
<td>0.992×</td>
<td>0.992×</td>
</tr>
<tr>
<td>4</td>
<td>0.988×</td>
<td>0.992×</td>
</tr>
<tr>
<td>6</td>
<td>0.980×</td>
<td>0.992×</td>
</tr>
<tr>
<td>8</td>
<td>0.973×</td>
<td>0.992×</td>
</tr>
<tr>
<td>12</td>
<td>0.958×</td>
<td>0.992×</td>
</tr>
<tr>
<td>16</td>
<td>0.946×</td>
<td>0.983×</td>
</tr>
<tr>
<td>32</td>
<td>0.899×</td>
<td>0.975×</td>
</tr>
</tbody>
</table>

Table 10.15: MF-set: Degradation of the execution time when the access latency to local memory is increased.
Chapter 11

Conclusions

Figure 11.1: Practical composition of the three methods (ASVP, FTL, HWFAM) as discussed in the thesis. The list of contributions has been provided in Section 1.3.

This chapter concludes the thesis. In the first three sections the results from individual parts are reviewed. Afterwards in Section 11.4 the approaches discussed in the thesis are compared on a qualitative level. Section 11.5 discusses possible future work.

Figure 11.1 shows composition of the three methods–ASVP, FTL, and HWFAM–that were discussed in the thesis. The contributions have been listed in Section 1.3.

11.1 Results of Part I

The first part of the thesis studied a customizable processor architecture (ASVP) suitable for data-parallel tasks such as DSP and image processing. The architecture is instruction-driven and statically scheduled; the static scheduling is optimal in this case because all latencies are predictable.

The ASVP architecture was described in a previous work. The first contribution of this thesis is the new hardware implementation of the idea and the analysis of FPGA synthesis results and of the performance (Chapter 4). The first qualitative observation made by the author is that although the custom data-path in DFU is in theory easy to design (it is a combination of a linear pipeline embedded in a loop structure; Section 4.2), the practical realization in an HDL is complex due to many implementation details. Once the HDL code is optimized it becomes increasingly difficult to extend the DFU with new customized operations, which is in conflict with the intentions. This observation led me to the idea of increasing the level of design specifications of data-driven systems, discussed in the second part.
The second observation made in Chapter 4 is the limitation of the vector-based architecture to efficiently execute algorithms that are not completely regular and data-parallel. In the Mandelbrot set kernel the irregularity in loop iteration counts in distinct pixels causes redundant computations.

The last observation is that the vector architecture, which processes data in large and long-running batches still requires a high-frequency instruction issue that could not be satisfied by a simple general-purpose controller. In Chapter 5 the solution is proposed to partially specialize the controller that issues vector instructions. The fragments of long, horizontally encoded vector instructions are stored in a compact two-level form in a new dictionary table. The v-instruction issue is still fully under software control of the general-purpose controller. The experiments have shown that the speed-up is 1.1× for the Mandelbrot set kernel, and 1.3× for the image segmentation kernel which is more complex. The hardware cost of the method is between 3.3% and 7% of the ASVP core area depending on the implementation technology. A positive side-effect of the technique is also the reduction of the firmware code size.

11.2 Results of Part II

The second part of the thesis has been driven by the lessons learned from the practical implementation of the ASVP core in FPGA. The immediate motivation was to replace manual DFU implementations by modules automatically generated from a higher-level specification. Since higher-level specifications are presumably more compact and readable, it will become feasible to implement complex domain-specific custom functions directly in DFU.

Petri nets were chosen as the theoretical basis for hardware synthesis of data-driven systems. The first step towards this goal is hardware synthesis of synchronous Petri nets (SPN) described in Chapter 6. The key contributions are the introduction of the alternate choice nodes in Petri nets (the switch and select nodes), and the translation process from Petri nets to distributed FSM using activation nets (AN).

In previous work instances of Petri nets that could evolve into a marking with a conflict or overflow situations were banned because the synthesis method would produce functionally incorrect hardware. The introduction of the alternate choice nodes in Petri nets overcomes the necessity of checking for potential conflict and overflow situations during the design process because the generated hardware is always correct. Potential conflicts and overflows are resolved by a deterministic priority.

An activation net (AN) is a graph of boolean functions that evaluate firing rules in SPN and computes which transitions will fire at the next clock-tick. Each component class in a Petri net (place, transition, alternate choice) has a corresponding fragment of AN assigned. Hardware is generated by composition of the pre-defined fragments according to the topology of the SPN.

The use of synchronous Petri nets for modelling and synthesis of data-driven systems has been tackled in Chapter 7. The fundamental problem is the ‘bubble wall’: the maximal throughput of any arc in safe (1-bounded) SPN is 0.5 tokens per tick since a token can be transferred only if the target place is empty (holds a ‘bubble’). The ‘bubble wall’ is caused by the principle of locality which says that the behaviour of PN places and transitions depends exclusively on the local neighbourhood. The new cell component is a place extended with a so-called ‘flow-through’ state. In the flow-through state the cell unloads and loads tokens concurrently in the same firing event (clock-tick). The flow-through state effectively overcomes the ‘bubble wall’ because it eliminates the intermediate state when the place is empty. The realization of the cell components requires global information about the marking of distant nodes in Petri net. This information is mediated through the activation nets.

Fragments of pre-specified data paths are attached to the AN fragments of the SPN components.
The new components are: *cell, i-cell (place), fork, join, switch, select*. All the components are synthesisable via deconstruction into AN fragments.

The case study in Chapter 7 demonstrates a synthesis of a special-purpose pipeline for the *Mandelbrot fractal set* from the FTL net to hardware VHDL code. The MF-set pipeline has been integrated in the ASVP core as a new DFU. The speed-up is $22 \times$, although the price paid is the deep specialization of the DFU which precludes reconfiguration of the core via simple firmware upload.

### 11.3 Results of Part III

In the previous, second part of the thesis the dataflow processing has been studied at the level of hardware pipelines and the logic gates. At that detail the problems of pipelining, achieving the synthesis timing closure and the interfacing with hardware IP cores take place. The higher-level issues of task scheduling and block memory transfers (bandwidth, latency...) become visible once the dataflow processing is considered across an application on a chip in the third part.

The framework used for the chip-level dataflow is the microthreading and the Self-adaptive Virtual Processor (SVP) model. The SVP is in a good position to express data-driven concurrency across a chip. Within general-purpose processor cores scheduling is implemented at the microthread level, with latency detection/recovery at the level of instructions. At the inter-core level scheduling and synchronization is implemented at a coarser granularity of thread families.

The contribution is the proposal, prototype implementation and evaluation of a method for coupling of microthreaded general-purpose processors with a hardware accelerator, called *Hardware Families of Threads* (HWFAM). The two main features that distinguish the approach from the previous work is the concurrency of execution between the microthreaded processor and accelerators, and the use of a unified synchronization model between the HW and SW.

The microthreaded processor guarantees execution of threads according to data dependencies among instructions within threads and across different threads. The unit of work that can be moved from the microthreaded processor to accelerators is the family of threads. When execution of a family of threads is moved to accelerators, data dependencies to other threads that execute in the processor must be honoured. In HWFAM the interface hardware implements the same SVP protocol for data-driven synchronization through the processor register file as used in software.

Evaluation of the HWFAM in Chapter 9 gives several results:

The allocation of thread families to a processor or accelerator is decided at run-time when the family is created. The mapping of threads to a processor or accelerators is controlled by the blocksize parameter, a standard SVP mechanism for performance throttling. The experiments have shown that the values ‘blocksize = numberOfAccelerators’ or ‘blocksize = numberOfAccelerators+1’ are usually good choices.

The placement of thread families may also be controlled by the workload size (number of threads). If the work-load size correlates with the number of threads, the HWFAM scheme can automatically switch the creation events to accelerators or to the processor. Small-sized thread families will not be transferred to an accelerator (even if one is available) because the costs associated with the move itself could outweigh the benefits.

The microthreaded processor tolerates well the new latencies created by the additional processing of thread-family creation events. The additional processing (in TMT) can be lowered by marking the subset of families intended for execution in hardware as *virtual*; the processor then sends only these ‘virtual’ families into HWFAM. The marking is implemented by setting the least significant bit (LSB) of the thread function addresses to one.
11.4 Comparison of the Approaches

The author’s subjective estimation of the effort required for using the approaches discussed in the thesis is shown in Figure 11.2. The horizontal axis goes from the highest effort needed at the left to the lowest effort at the right. The LEON3 scalar processor is put at the far right as the easiest to use, while a manual (VHDL) approach is at the far left as the most challenging. Microthreading (UTLEON3) is not difficult to understand conceptually, though there are peculiar details that must be considered at the assembly level. The GP-ASVP approach was put in the middle because detailed hardware/software interactions must be observed by the programmer. At the moment the FTL approach is more difficult to use than the GP-ASVP, hence it was put to the left of it. Future improvements at the language and feature level might put it at least on-par with ASVP.

The statically scheduled hardware architecture—the ASVP discussed in Part I—is suitable for data-parallel applications only. The memory model requires that applications must be partitioned into tasks and all the I/O data for each task must fit in local memory banks. Dataflow scheduling is limited to the task level. The disadvantage of the architecture is the requirement to decompose algorithms over several layers: (1) operations constructed in hardware; (2) tasks executed in firmware; and (3) the management software in a host processor. The decomposition must respect that: (1) hardware operations implemented in the DFU should be simple and reusable; and (2) firmware code and the task I/O data must fit in local memory banks. Furthermore, vector-oriented processing requires high regularity of data parallelism down to the level of elementary operations. The advantage of the architecture is the software-oriented flow once the primitive hardware operations are defined and implemented in DFU. Evaluation in Chapter 10 shows that GP-ASVP, i.e. using general-purpose DFU constructed by the approach suggested in Section 4.2 and scheduling the algorithm in firmware, is not competitive with a direct implementation of the algorithm in FPGA hardware.

The FTL, presented in Part II, increases the specification level of hardware modules. Dataflow scheduling is performed for individual hardware operations. The hardware that performs the schedul-
11.4. COMPARISON OF THE APPROACHES

Parallel processing is implemented by pipelining. Individual high-level operations executed by algorithms are mapped to distinct pipeline stages. Irregular codes, such as conditional blocks, insert the alternate-choice nodes and split the pipelines. Pipelining is more flexible than the vector processing.

Microthreading, discussed in Part III, implements dataflow scheduling in a general-purpose computing platform. Compared to previous dataflow machines and processors (Section B.4) the dynamic scheduling of threads is more coarse grained than scheduling of instructions. This has the advantage of lower hardware resource requirements. The cost is the requirement to rewrite programs to use families of threads. The HWFAM interface views families of threads as posted tasks of work. It takes advantage of the memory model which guarantees visibility of memory updates only after threads have been explicitly synchronized. A family of threads is a black-box that can be executed anywhere.

Microthreading employs sequential programming within threads, but requires decomposition of algorithms into families of threads. While this is easy to do theoretically (transform every loop and function call into a family), practical coding experience has shown that a blind use of the methodology leads to low IPC and poor utilization of resources. For instance, in the MF-set benchmark the inner while loop was not transformed into a family of threads because there would be no real overlap between the threads (iterations of the loop) anyway.

Experiments show that while microthreading is able to improve pipeline efficiency (IPC), the current implementation in UTLEON3 is not competitive with a classical scalar in-order processor such as the LEON3. Future implementation of microthreading must therefore achieve on-par clock-cycle delay, reasonable resource requirements (say at most $2 \times$ than a classical core) and--most importantly--it must support the cooperative multi-core operation (Section 8.4). This in turn requires high-performance pipelined memory buses and cache coherency.

Parallel execution in microthreading is based on efficient utilization of many independent but similar (interchangeable) processing units with fully programmable interconnect, realized partly in hardware and partly in shared memory. Pipelining in FTL on the other hand specializes each pipeline stage to a single program step, including the connections to the previous and next processing stages. The decision to use the first (microthreading) or the second (pipelining) approach is guided by an observation if there are ‘more instructions or more data in the algorithm’. Unfortunately, there is hardly any universal measure that could quantify the ratio of instructions vs. data.

In general, more instructions than data means a complicated algorithm operating over a small amount of data. Hence it makes sense to reuse a small number of hardware compute units (small relative to the number of instructions) over the whole program. More data than instructions means a simple algorithm applied over a large data set. The small number of instructions allows to implement them all as distinct instances of primitive compute cores; the interconnect is dedicated (independent wires between pipeline stages), hence all cores can work in true concurrency.

Dataflow scheduling is prone to resource deadlock. In microthreading resource deadlock happens when all hardware thread slots are occupied by waiting threads and no further progress is possible, while by using a different dynamic schedule (e.g. after changing the blocksize parameters) the same program could complete successfully. The solution in microthreading is to switch into sequential execution mode that does not use families of threads. In FTL resource deadlock is connected with FIFO queues getting full. Typically in cyclic pipelines, forward progress would be possible only if a token could be simultaneously pushed and popped off the queue.
11.5 Concluding Remarks and the Future Work

Mapping of user applications to compute fabrics is a central problem in computer science. Push-button synthesis of higher-level behavioural code into reconfigurable hardware (FPGAs) is something like a holy grail in our community. Besides the run-time performance optimality of tool-generated solutions the other main issue is the long compilation time caused predominantly by slow place&route processes (PAR). This discourages software developers from using FPGAs because developers are used to fast compilation times needed for nearly interactive coding and debugging. In ASVP and partly in HWFAM the answer was to partition the user application to software (= fast compilation, slow execution) and hardware (= slow compilation, fast execution) parts, with much of the interactive development hopefully occurring in the software component. In FTL the idea is to remove timing aspects of circuits from specifications so that the mapper and PAR tools could—in theory—autonomously insert pipelining registers where needed. This approach could achieve the timing closure of a design easier because pipelining would be performed exactly where needed.

The greatest potential for future work, both theoretical and practical, is in the Flow Transfer Level—the dataflow processing using Petri nets, presented in Part II. At the current theoretical level the synthesis is limited to FTL nets that decompose into acyclic activation nets. Cyclic activation nets would be transformed into cyclic combinatorial circuits in hardware. Attempts to overcome this limitations have already been made by the author, but the solutions obtained so far are unsatisfactory: while some practical experiments were successful (synthesis of testing FTL nets), the method itself could not be proved to be correct for all cases.

A practical limitation of FTL is the absence of a proper design input language. The CAD tool presented in Appendix D is being developed in the reverse direction: from the VHDL code generator back to the middle end. An input FTL net is given as a short program in Python that instantiates appropriate objects and constructs the graph. This strategy proved successful to test quickly the core transformation and synthesis algorithms.
Bibliography


[TM05] Emil Talpes and Diana Marculescu. Toward a multiple clock/voltage island design style for power-aware processors, 2005.


Publications of the Author

NOTE: The share of authorships of the publications listed below is given in percentages in superscripts (when the shares are not equal). The marking (S) means the author is a supervisor. All the publications are referred in the thesis.

Books (Non-Peer Reviewed)


Journal Papers (Peer Reviewed)


Conference Papers (Peer Reviewed)


The paper has been cited in:

The paper has been cited in:


The paper has been cited in:


**Other Non-Peer Reviewed Publications**

Appendix A

Proposed Doctoral Thesis (February 2012)

Title: Programmable and Customizable Hardware Accelerators for Self-adaptive Virtual Processors in FPGA

Abstract: (February 2012)

The author of the thesis suggests to explore the following:

1. Extension of the SVP paradigm to heterogeneous multicores by a technique for coupling general-purpose and special-purpose cores. The technique shall exploit hardware synchronization and communication primitives of the SVP paradigm, and it is expected to be latency tolerant.

2. A method for an efficient development of hardware accelerators of selected data-parallel algorithms in FPGA using a vector-based architectural template. The accelerators can be used as specialized processing cores in the SVP multicore grid.

3. A unified approach to designing hardware-accelerated systems and programming tools. The approach will be defined and evaluated by implementing a software tool that can generate a customized application-specific vector co-processor/accelerator based on architectural description. Efficiency of accelerators generated by the method will be compared to a manual approach.

Comments (September 2013):

Point 1 has been presented in Chapters 8–9.

Points 2 and 3 deserve an explanation. The intention was to create a method in which one could: (a) specify a function of a simple customized vector operation using a special-purpose declarative (text) format; (b) have a tool that would translate the specification into a synthesizable hardware (via VHDL code generation); and (c) integrate this hardware-oriented specification with a firmware toolchain so that a suitable customized DFU could be automatically generated or selected from a library based on
the analysis of kernel firmware (see Section 3.2). Strictly speaking, the point (c) has not been fulfilled. On the other hand, the original aim of points (a) and (b) was much more narrower than it is in the final work. The intended solution for (a) was to have the DFU specifications created using only a limited combination of pre-defined templates, such as the full reduction or the element-wise function map, as discussed in Section 4.2. However, in the final work the method described in the Part II is much more general and independent of the ASVP framework (Part I). The comparison of implementations of the Mandelbrot set generators in Section 7.6 shows that fully specialized compute pipeline is an order of magnitude faster than the original approach.
Appendix B

Survey of Related Work

B.1 Advanced Scheduling Techniques

**Out of Order Execution** (OOE) is a hybrid of the instruction-driven and data-driven scheduling. In OOE processors the *Instruction Set Architecture* (ISA) is von Neumann: program operations are stored as instructions in the program memory, the ISA defines a *Program Counter* register (PC, or IP='instruction pointer'). The selection of instruction from the program memory is controlled by the value currently stored in the PC. The execution of the selected instructions is driven by the availability of data that they require. The data comes from memory (unpredictable latencies) and from previous operations. The OOE processor has an instruction-driven pipeline *frontend* consisting of an I-Cache, instruction fetch and decode stages (including PC and branch predictions), and a data-driven pipeline *backend*. Data-driven scheduling in the backend is implemented by *reservation stations* where operations wait for their data, and by a *reorder buffer* where operations wait for an in-order commit to maintain the illusion of von Neumann architecture.

**Speculative execution** (SE) is an extension of the data-driven scheduling. In SE the prescribed data-dependencies among operations can be temporarily ignored: operations are executed using ‘guessed’ input data, they do not wait for the real committed data. The ‘guesses’ must be validated ex-post when the real input will have come. If the guesses were all correct the SE can overcome the dataflow limit [Wal91] of programs. If some of the guesses were incorrect the affected operations (and all that depend on their outputs) must be re-executed.

B.2 Gate-Level Dataflow: Synchronous vs. Asynchronous Circuits

Timing in digital circuits can be clock-*synchronous* or *asynchronous*. Different timing methods can be used for *local* circuits and for *global* interconnect. The following three combinations are possible (shown in Figure B.1): GALA (Globally Asynchronous Locally Asynchronous), GALS (Globally Asynchronous Locally Synchronous), and GSLS (Globally Synchronous Locally Synchronous).

<table>
<thead>
<tr>
<th>GALA</th>
<th>GALS</th>
<th>GMLS</th>
<th>GSLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Globally Asynchronous Locally Asynchronous</td>
<td>Globally Asynchronous Locally Synchronous</td>
<td>Globally Mesochronous Locally Synchronous</td>
<td>Globally Synchronous Locally Synchronous</td>
</tr>
<tr>
<td>= fully async. design</td>
<td>= sync. large IP blocks, async. global interconnect</td>
<td>= sync. large IP blocks at nominally the same freq. but varying phase</td>
<td>= fully sync. design</td>
</tr>
</tbody>
</table>

*Figure B.1:* Timing schemes in digital circuits.
Appendix B: Survey of Related Work

Clock, Datapath, Memory, Control, IO

Figure B.2: Power breakdown in a high-performance CPU [TSR+98].

Figure B.3: Recursive H-tree pattern to distribute global clock signal uniformly and with equal delay over area.

Table B.1: Summary of advantages of asynchronous circuits [JS02] and GALS.

<table>
<thead>
<tr>
<th>What?</th>
<th>Why?</th>
<th>GALS?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low power consumption</td>
<td>due to fine-grain clock gating and zero stand-by power consumption.</td>
<td></td>
</tr>
<tr>
<td>High operating speed</td>
<td>operating speed is determined by actual local latencies rather than global worst-case latency.</td>
<td></td>
</tr>
<tr>
<td>Less EMI noise</td>
<td>the local clocks tend to tick at random points in time</td>
<td>✓</td>
</tr>
<tr>
<td>Robustness towards variations in supply voltage, temperature and fabrication parameters</td>
<td>timing can be insensitive to wire delays</td>
<td></td>
</tr>
<tr>
<td>Better composability and modularity</td>
<td>due to handshake interfaces and local timing.</td>
<td>✓</td>
</tr>
<tr>
<td>No clock distribution and clock skew problems</td>
<td>no global clock signal that would require minimal phase skew across the circuit.</td>
<td>✓</td>
</tr>
</tbody>
</table>

GALA: Fully Asynchronous Circuits

Contemporary digital design industry is optimized around the clock-synchronous design method (GSLS). Standard commercial CAD tools exist that automate all development processes: RTL design, simulation & verification, ASIC layout, timing analysis, and testing of fabricated chips. The competing design method—asynchronous systems (GALA)—offers some advantages but there also many issues, especially in the CAD tools support. Asynchronous circuits potentially promise [JS02] low power consumption, less electromagnetic interference (EMI) noise, better composability and modularity, and other—see Table B.1. However, the lack of mature CAD tools, especially for timing analysis and testing, is often cited as the main drawback of the technology for practical use. Another drawback is increased die area because of the more complicated control circuitry. Tangram [vBKR+91] and BALSA [BE00] are CAD system for syntax-directed specification and generation of VLSI asynchronous circuits.

GALS: Globally Asynchronous Locally Synchronous

A somewhat middle ground are globally asynchronous locally synchronous (GALS) circuits. In GALS, individual large-scale hardware IP blocks are designed using the common synchronous-clock method, but communication of these blocks in chip (SoC) is done asynchronously. This combined approach allows to leverage existing CAD tools for individual components, and even directly reuse existing IP blocks. Synchronous blocks are extended by asynchronous wrappers that handle the sync/async interface for all data signals crossing the boundary. Several schemes were devised for the wrappers: pausable (stretchable) clocks [YD96] that dynamically shift clock pulses to avoid metastability; asynchronous
**B.3 SYNCHRONOUS ELASTIC FLOW (SELF) PROTOCOL**

FIFO buffers; boundary synchronization using standard registers.

Global distribution of low-skew clock signals in all-synchronous ICs may easily account for nearly half the power consumption of the whole chip (Figure B.2, [TSR+98]). The clock distribution network uses large buffers spread over the chip so that the delay (phase) of clock edges is equal at all termination points (registers). Figure B.3 illustrates one possible approach using recursive H-tree pattern.

The original hope in GALS circuits was to save power by not requiring any global, uniformly ticking clock signal [MHK+99]. Instead, many independent local clocks, possibly running at different frequencies, are used. However, practical evaluations showed that power savings are not the strong property of GALS circuits [KGGV07]. For instance, a superscalar CPU implemented using GALS was found to have only small decrease in power consumption, but relatively significant drop in performance of around 10% due to latencies of asynchronous interfaces [TM05].

The main strength of the GALS technique is top-level system integration. Large systems are broken into many independent clock domains, and timing constraints are more easily satisfied within the locally synchronous blocks [LBB+07]. Furthermore, dynamic voltage and frequency scaling (DVS) can be conveniently applied at the level of the synchronous blocks. Another GALS advantage is reduction of EMI noise due to un-synchronised clock switching in independent blocks. The noise reduction is welcomed in mixed analog/digital chips. Secure applications (cryptographic hardware) may gain increased immunity against power analysis attacks because of the less predictable timing that creates more evenly spread power consumption spectrum (and hence reveals less information) [GOK+05].

**B.3 Synchronous Elastic Flow (SELF) Protocol**

The SELF protocol [CKG06] (Synchronous ELastic Flow) can be used to transform synchronous circuits into elastic ones. The resulting circuits remain in the standard clock-synchronous domain, hence existing CAD tools and knowledge are reused; a major advantage over all-asynchronous approach.

Handshake protocol of the SELF scheme is illustrated in Figure B.4. Data vector is extended with two signalization wires called Valid and Stop, connecting the sender and receiver in the forward and reverse directions, respectively. Because SELF operates within standard clock-synchronous designs, there are four physical states that the Valid and Stop wires may assume. Of these, three logical states are recognized, as depicted in the figure:

- **Transfer (T)** \((V = 1 \land S = 0)\) – the sender provides valid data and the receiver accepts it.

- **Idle (I)** \((V = 0)\) – the sender does not provide valid data.


• \textit{Retry (R)} \((V = 1 \wedge S = 1)\) – the sender provides valid data but the receiver does not accept it (due to congestion or other unreadiness).

The protocol guarantees \textit{persistence} of the sender behaviour: once sender initiates data transfer while receiver is not ready the sender will continue to generate the \textit{retry (R)} requests until it eventually succeeds; it will not call off the transfer attempt.

In theory, any standard clock-synchronous design that uses only edge-triggered (flip-flop) registers can be transformed into equivalent elastic circuit using the SELF protocol. This transformation involves replacing the original flip-flop (FF) registers by \textit{Elastic Buffers} (EB). Figure B.5.a depicts a pipeline stage with elastic buffers at both ends. Each EB implements memory space for storing at least two data tokens. The forward latency (propagation delay) of the EB is \(L_f = 1\); this is the same as for a normal register. The backward latency \(L_b\) could be 0, but then the EB would directly propagate the \textit{Stop} signal from its output to its input via internal combinatorial path. In SELF, therefore, \(L_b = 1\), the \textit{Stop} signal is buffered and there is no combinatorial path between EB ports. (The case \(L_f = L_b = 0\) is a channel.) Since capacity \(C \geq L_f + L_b\), the EBs must have the capacity \(C_{eb} = 2\) data tokens.

As the capacity of normal edge-triggered flip-flop registers is only \(C_{ff} = 1\), not 2, the question is whether the EBs require double the physical silicon area compared to FFs. In CMOS custom-layout circuits the EB could be implemented using only two latches as shown in Figure B.5.c, plus a control automaton. This is indeed similar area as used by master-slave edge-triggered flip-flops, which are often also composed of two latches in series.

When the special EB cells are not available in the target technology a generic implementation using two standard flip-flops can be used. In Figure B.5.a-b there are two possible implementations, the first with a multiplexer at the output and the second at the input. Both have two flip-flops named ‘main’ and ‘aux’. The main flip-flop is used in uncongested situation when data flows through the EB. When the receiver signals \textit{Stop} the control logic in EB uses the auxiliary flip-flop to store incoming data that is already in flight. The \textit{Stop} signal is forwarded counter the data-flow direction with the latency 1 cc.

The generic EB implementations must be used whenever the silicon-optimized version is not available. This includes contemporary mainstream FPGAs. Some high-performance silicon technologies
use dynamic (capacitor-based) flip-flops, not the static master-slave FFs.

**B.4 Dataflow Computers**

In *data-driven (dataflow)* architectures operations are started in response to the availability of data. The main advantage of the data-driven approach is the natural ability to schedule operations dynamically according to the real execution latencies. In the true dataflow processor a program is encoded and stored as a graph in memory, with nodes and edges representing operations and dependencies, respectively. A comparison of the instruction-driven and data-driven processors is in Figure B.6. In the instruction-driven machines (left) the instructions encode the locations of the source and destination operands in the data memory. In the data-driven machines (right) the instructions (nodes) encode the locations of all the destination nodes(s) that require the output value. The source locations are not encoded because they are implicit: instructions fire when all their source values have arrived.

![Instruction-driven vs Data-driven Machines](image)

**Figure B.6:** A comparison of the instruction-driven (left) and data-driven (right) processors [Vee86].

Function units in dataflow processors produce output data in the form of *tokens*. Besides the value each token also encodes its (unique) destination location, i.e. the destination graph node in a program. When a token is produced the processor checks if the associated destination node is ready to fire: it is ready only if all the other source tokens have already arrived. (Practical implementations often support only monadic (one-input) and dyadic (two-input) operations.) Otherwise the token must be stored somewhere to wait for its companions. In the *static dataflow* machines the waiting tokens are stored directly at the graph nodes. The disadvantage of the static technique is that the concurrent execution of multiple instances of the same computation (using the same graph) is not possible because the instances would mix each other’s tokens in the nodes. This is especially painful for loops: loop iterations must be serialized (the graph is cleared between iterations), or the loop must be unrolled.

The tokens are usually defined:

- **Static dataflow:** \( \text{token}(v, d, p) \)
- **Tagged dynamic dataflow:** \( \text{token}(v, t \times d, p) \)

where \( v \) is the data value, \( d \) is the destination node location (instruction), \( t \) is the tag, and \( p \) is the port (operand index) in the destination node.
Figure B.7: Example of the tagged-token dynamic dataflow machine (Manchester Dataflow Machine [Vee86]. Tokens are produced in functional units and routed back to the input token queue. The matching unit pairs tokens destined to dyadic nodes with their waiting companion, or stores the token in the CAM memory if it is the first token for the given dyadic node. Tokens destined to monadic nodes bypass the matching unit altogether. The fetching unit retrieves the node instruction code given the location (address) in the incoming tokens.

Dynamic dataflow machines [Vee86, AN90] overcome the limitation by either (a) dynamically making a new copy of the pristine graph for each instance, or (b) by tagging (colouring, labelling) the tokens. The tags distinguish the different graph instances (e.g. loop iterations), but the waiting tokens may no longer be stored directly at the nodes. One possible solution is to use a content-addressable memory (CAM) to store and later look-up tokens based on the combination of the tag and the destination location.

Figure B.7 shows a functional diagram of one processing element in the Manchester Dataflow Machine [Vee86]. There are two memory units: (a) the memory for nodes is the read-only instruction memory, and (b) the memory for tokens is the intermediate storage for tokens waiting for their companions. The matching unit is based on a content addressable memory (CAM). Each incoming dyadic token (monadic tokens bypass the unit altogether) is looked up in the CAM using its tag \( t \) and the destination node location \( d \) as the key. If a match is not found the token is stored in the CAM (and not sent to the next unit). Otherwise the token stored in the CAM is retrieved, the CAM location is freed, and the retrieved token is paired with the current token that has triggered the look-up; both tokens are sent to the fetching unit.

The CAM in the matching unit should ideally be fully associative. In practice this is not feasible and only a limited number of ways can be supported (16 in [Vee86]). Moreover, the CAM must be relatively large (20 MB in [Vee86] in 1986) because it must hold intermediate working-set data (a graph cross-section). Overflows in the CAM due to the limited capacity and the limited number of ways (conflicts) can be handled in additional hardware by storing the tokens in other memory, but possibly with a higher lookup latency. Ultimately a dataflow machine is susceptible to deadlock if all the token memory is occupied by an unexpectedly wide expansion of the graph. This could happen for example in a naive implementation of the for-loop: the induction variable \( i \) depends only on itself \( (i := i + 1) \), therefore for a sufficiently high upper bound this expansion could fill the token store. Once this happens the machine is unable to continue because there is no intermediate storage left to finish the iterations’ loop bodies. There is no universal solution to the problem. Usually a some kind of throttle is used to regulate the expansion of parallelism in the machine based on heuristic metrics.
B.5. RECONFIGURABLE COMPUTING

B.5 Reconfigurable Computing

Reconfigurable arrays are *coarse grained* or *fine grained* depending functionality implemented in the smallest reconfigurable element.

In coarse grain arrays the elements are configurable functional units that include a data-path and a control logic. *Pleiades* [ASWR98, AZW+02] architecture targets low-power DPS application; it consists of a central ARMv8 processor and small satellite coarse-grained reconfigurable cells of different kinds (e.g. SRAM, MAC units, address generators, ALU, bit-level FPGA). *MorphoSys* [SLL+98] is a reconfigurable architecture targeted for acceleration of multimedia applications. It is an 8x8 array of reconfigurable cells (RC Array), context memory, 2kB frame buffer memory, DMA controller, and TinyRISC processor. A compiler from the SA-C language was developed for MorphoSys M1 [VNK+01]. *Chameleon/Montium* [HS03, SHRM04, SKW+07] is heterogeneous system-on-chip with reconfigurable elements. Each Montium unit is composed of a communication and configuration unit (NI) and Tile Processor (TP). Tile Processors contain 5 ALUs (16 bit), 10 local memories (512x16 bit), and a common control and communication/configuration unit. *RICA* [KNM+08] is a coarse-grained array of function units such as ADD, MUL, REG, MEM, SHIFT, MUX, JUMP, controlled by long instruction word (VLIW). Since the array includes a Program Counter it can control the instruction issue itself without a host processor.

Field-Programmable Gate Arrays (FPGA) are the most prevalent type of *fine grained* reconfigurable arrays. Contemporary SRAM FPGA architectures are usually based on look-up tables (LUT). In principle, LUT is a static memory element with \( n \) address lines (usually \( n = 4 \) to \( 6 \)) and 1 output data line. The LUT memory content (\( 2^n \) bits) is programmed during the reconfiguration process that takes place upon device start-up. Logic cells can contain multiple LUTs, synchronous registers (flip-flops), multiplexers, and carry-chain logic. They are arranged in an array and interconnected using configurable wires.

Reconfigurable arrays are used for stand-alone computing, or in a close cooperation with a processor as a *reconfigurable functional unit*. In *Chimaera* [HFHK04] a processor is extended by a small FPGA with a connection in the processor register file. Computation in the FPGA is state-less: the only inputs are register operands, the output is written in the register file.

In practice the usability of any reconfigurable architecture is critically connected with the existence of proper design entry tools [Guo06]. In FPGAs the VHDL and Verilog languages are most often used in practice because the knowledge of these HDLs is transferable to ASIC design. Synthesis from higher-level languages, such as C, was attempted many times, with varying success. Here we briefly summarize a couple of recently proposed approaches.

ROCCC [VPNH10] is a high-level behavioural synthesis compiler from C to VHDL. The compiler generates systems and modules. A module is a hardware block with a concrete set of inputs and outputs, some internal data-flow computation, and with a known delay. Modules are stored in a database and can be reused. Systems integrate modules and schedule computation and data transfers.

hArtes [RFB09] can synthesize parts of an application to different targets (FPGA, DSP, CPU), evaluate costs of the mappings and then make the decision on the final mapping and communication of different functions.

A different approach is taken in [BHLS10]: a domain-specific embedded language in C++ is presented that allows a program running on a host processor to construct an expression tree on arrays of floating-point numbers. Using lazy evaluation, custom hardware can be synthesized on-demand to implement the computation in an FPGA.

Computing on GPU devices has gained a great traction in recent years. GPUs are programmed in CUDA and OpenCL languages which expose concurrency and data communication to the programmer.
and thus allow for predictable performance. In [OBDA11], synthesis from OpenCL to an FPGA platform is presented. An OpenCL kernel function is split into a compute part and memory-access part (global memory loads and stores). The compute part is synthesized into a fully custom data path, and the memory-access part is synthesized as streaming units with custom address generators.

FCUDA [PGS+09] is a source-to-source compiler from NVidia’s CUDA language to AutoPilot C. FCUDA coarsens per-thread CUDA kernels by wrapping them into loop nests. Pragmas specify fetch/compute/write task separation, and control the coarsening process and data transfers.

Building on FCUDA, a high-level parameter space exploration tool is presented in [PLS+11]. The original FCUDA compiler can perform optimizations based on user input (e.g. unrolling, local memory partitioning), but given the long HDL synthesis times it is not feasible to explore all possible combinations of options to find the best performing one. This is overcome in the presented ML-GPS tool by actually synthesizing only a small number of sample configurations, then using the synthesis results to fit resource and clock-period estimation models. The tool ML-GPS synthesises a small number of sample configurations in FPGA and uses the synthesis results to fit resource and clock-period estimation models. The models are used in a binary search heuristic to find the optimal compiler/synthesis option setting that should minimize the total execution latency for a given kernel function.

B.6 Application-Specific Instruction-Set Processors

Standard general-purpose processors are optimized for a wide range of applications. In single-purpose (embedded) devices it is more efficient to optimize the processor for a given application domain, thus creating an Application-Specific Instruction-Set Processor (ASIP). Typically the processor function and/or structure is specified in Architecture Description Language (ADL), from which the synthesizable hardware implementation and the software toolchain can be generated.

LISA [ZPM96] is an industry standard language and tool that describes bit- and cycle-accurate processor models. Processor simulators and compilers can be generated from the description. In [SCL+04] RTL synthesis from LISA is presented.

EXPRESSION [HGG+99] can be used for fast design space exploration by automatic generation of optimizing compilers and simulators. Compared to LISA and other predecessors EXPRESSION can directly describe processor reservation tables, and it can model complex memory organizations.

B.7 Vector Processing Architectures

Vector processing was shown to be a good match for embedded applications. In [KP02] the VIRAM vector processor (Figure B.8) was evaluated using the EEMBC multimedia benchmark suite, and it was found (for example) to outperform VLIW processors by a factor of 10. The original VIRAM chip is 0.180 \( \mu m \) custom design clocked at 200 MHz, 1.6 GFLOP single-precision performance, with an on-chip 13 MB 8-bank embedded DRAM (a crucial feature). The large multi-banked local memory is used as a software-controlled staging buffer that balances the latency/bandwidth ratio of the memory hierarchy.

A centralized vector register file (VRF) cannot be scaled to support many chaining functional units, as each FU requires additional ports to the VRF. CODE [KP03] vector micro-architecture proposes to partition the architectural VRF into several physical VRFs distributed in clusters (Figure B.9). A renaming table is used to keep track of the location of architectural registers, and a history buffer is employed to support precise exceptions.
SCALE [KBH+04] is an implementation of a vector-thread architecture (Figure B.10) in a custom 0.180µm design, clocked at 400 MHz, with 4 vector-thread (VT) lanes. Vector lanes can either all execute the same instruction (as in a traditional vector processor), or each VT lane can fetch and execute different Atomic Instruction Blocks concurrently to the other lanes, allowing true multi-threaded operation. Similar to CODE, lanes are partitioned into clusters, but this partitioning is explicit and visible in ISA.

Following the success of VIRAM, several vector processors were implemented in FPGAs using the VIRAM ISA. Micro-architectures of these processors are usually traditional and not so advanced as in CODE or SCALE. VESPA [YSR08] is design-time configurable soft-core implemented in Stratix-I and -III FPGA. The processor allows ISA subsetting, and some of the primary design parameters that affect both performance and resource usage can be configured (e.g. the number of vector lanes, vector lane width, memory crossbar).

VIPERS [YEC+09] (Figure B.11) is similar to VESPA, but it is less strict to VIRAM ISA compliance, and more tailored to the FPGA target technology. It offers a few new instructions to take...
APPENDIX B. SURVEY OF RELATED WORK

Figure B.10: Abstract model of a vector-thread architecture [KBH+04].

Figure B.11: VIPERS soft vector processor consisting of scalar core and multiple vector lanes [YEC+09].

Advantage of the MAC (multiply-accumulate) and BRAM units in FPGAs. The new MAC instructions perform integer summation (reduction) within vector registers. The processor can be also augmented with a fast vector-lane-local memory that can accelerate table lookup functions.

FPVC [KL11] is an FPGA-based vector co-processor with floating-point units, implemented in Virtex 5. The processor has unified scalar and vector register file.
Appendix C

FTL: Details

C.1 Formal Definitions

C.1.1 SPN with Alternate Choices

Synchronous Petri net (SPN) with alternate choice nodes is a six-tuple: \(SPN^+ = (P, S, A, R, D, I)\), where:

1. \(P\) is a set of place nodes, \(P = \{p_i\}\),
   - Each place has at most one in-arc and one out-arc: \(\delta^+(p_i) \leq 1, \delta^-(p_i) \leq 1\).
2. \(S\) is a set of total transition nodes, \(S = \{s_j\}\),
   - (Total transitions are the normal transitions from standard P/T Petri nets.)
3. \(A\) is a set of alternate-choice transition nodes, \(A = \{a_k\}\),
   - \(a_k\) is a switch choice when: \(\delta^+(a_k) = 1 \land \delta^-(a_k) > 1\),
   - \(a_k\) is a select choice when: \(\delta^+(a_k) > 1 \land \delta^-(a_k) = 1\),
   - (other combinations of \(\delta^+(a_k)\) and \(\delta^-(a_k)\) are forbidden).
4. \(R\) is a set of arcs, \(R = \{r_l\}\),
   - place/transition (P/T) arcs: \((p_i, s_j), (p_i, a_k), (s_j, p_i), (a_k, p_i)\),
   - transition/transition (T/T) arcs: \((s_j, a_k), (a_k, s_j), (a_k, a_l)\).
5. \(D\) is an ordering of arcs in the alternate-choice transition nodes, \(D = \{d(a_k)\}\),
   - \(a_k\) is a switch choice when: \(d(a_k) = (r_1, r_2, ...), \text{ where } r_i \text{ are out-arcs of } a_k, r_i \in R, r_i = (a_k, *)\),
   - \(a_k\) is a select choice: \(d(a_k) = (r_1, r_2, ...), \text{ where } r_i \text{ are in-arcs of } a_k, r_i \in R, r_i = (*, a_k)\).
6. \(I\) is a set of places that hold a token (initial marking), \(I \subset P\).
7. The graph \((P \cup S \cup A, R)\) is directed graph.
8. The graph \((S \cup A, R)\) is polytree.\(^1\)

- Informative: When all place nodes are dropped from a complete SPN graph what remains is a forest (a disjoint union of trees). The individual components of the forest are directed acyclic graphs composed of the total and alternate-choice transitions and arcs. Additionally, the component graphs must be trees, i.e. the underlying undirected graph is also acyclic. Examples are given in Figure C.1.

### C.1.2 Activation Net

Activation net (AN) is a four-tuple \(AN = (O, F, E, I)\):

1. \(O\) is a set of boolean operator nodes: \(\text{wand } \square, \text{ or } \square, \text{ not } \square\).
   - \(\delta^+(\square) = \delta^+ (\square) = 1\).
2. \(F\) is a set of flip-flop nodes, \(F = \{\text{ff}_i, \text{ff}_{EN,i}\}\),
   - \(\text{ff}_i\) represents data input and output to the flip-flop, \(\delta^+(ff_i) = \delta^- ff_i = 1\),
   - \(\text{ff}_{EN,i}\) represents the clock enable input of the flip-flop, \(\delta^+(ff_i) = 1, \delta^- ff_i = 0\).
3. \(E\) is a set of directed edges, \(E = \{e_k\}, e_k = (O \cup F, O \cup F)\).
4. The graph \((O \cup F, E)\) is directed graph,
5. the graph \((O, E)\) is directed acyclic graph.

### C.1.3 WAND-split Transform

To split a \textit{wand} \(w_i, w_i \in O\):

1. where \(e_{IN,i}\) are in-edges of the wand, \(e_{IN,i} \in \{(*, w_i)\}\):

\(^1\)A polytree (also known as oriented tree) is a directed acyclic graph whose underlying undirected graph is a tree. A tree is an undirected graph that is connected and has no cycles.
C.1. FORMAL DEFINITIONS

2. Create a new wand $w'_i$.

3. replace $w_i$ with $w'_i$ in the in-edges: $e_{IN,i} \in \{(s, w'_i)\}$,

4. add edge $(w'_i, w_i)$.

C.1.4 WAND Multi-Split Transform

Transition subgraphs are the maximal connected graph components of $(S \cup A, R)$; one such example is in Figure C.1.c. In these complex transition subgraphs the decomposition transformation from SPN to AN is more involved than the simple wand splitting described above. In Figure C.1.c the problem is at the transition $F$ (a fork or join): as depicted in Figure C.2.b the $F$ is mapped to a single wand $s_F$, shared between three different choice nodes $A$, $B$, and $C$. If the $s_F$ is naively split in two there is a real danger that when the AN choice fragments are connected the resulting AN would contain a cycle.

The solution is shown in Figure C.2.c. A new wand $s'_F$ is split from $s_F$. The AN fragments of the three choice nodes are interconnected such that each of the ‘intention’ inputs $(s'_{A2}, s'_{B0}, s'_{C0})$ are connected to all ‘outcome’ outputs $(s_{A0}, s_{B0}, s_{C0})$ of the other choices.
C.2 Integration with Legacy Hardware Cores

In Section 7.5 the interface between a module in FTL and a surrounding hardware has been discussed. The reverse nesting is also possible: an existing (legacy) hardware core used from within an FTL design.

**Figure C.3:** Structure of a standard floating-point compute core. Depicted a variant with 3 pipeline stages.

**Table C.1:** The meaning of interface signals in a standard FP core.

<table>
<thead>
<tr>
<th>Name</th>
<th>Width</th>
<th>Dir.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>w</td>
<td>IN</td>
<td>Operand A</td>
</tr>
<tr>
<td>b</td>
<td>w</td>
<td>IN</td>
<td>Operand B</td>
</tr>
<tr>
<td>op Nd</td>
<td>1</td>
<td>IN</td>
<td>New data. Indicates that input operands a, b are valid.</td>
</tr>
<tr>
<td>ce</td>
<td>1</td>
<td>IN</td>
<td>Clock enable.</td>
</tr>
<tr>
<td>clk</td>
<td>1</td>
<td>IN</td>
<td>Clock.</td>
</tr>
<tr>
<td>rst</td>
<td>1</td>
<td>IN</td>
<td>Synchronous reset (clear).</td>
</tr>
<tr>
<td>result</td>
<td>w</td>
<td>OUT</td>
<td>Result output.</td>
</tr>
<tr>
<td>rdy</td>
<td>1</td>
<td>OUT</td>
<td>Output ready. Active when the Result output is valid.</td>
</tr>
</tbody>
</table>

**Figure C.4:** Integration of a legacy pipelined core in FTL.

The practical situation is embedding a pipelined compute core, e.g. a floating-point adder, in FTL-net design. Figure C.3 schematically shows internal structure of a common type of floating-point compute core with 2 inputs and 3 pipeline stages. The meaning of the I/O signals is summarised in Table C.1; signal names as based on Xilinx Floating Point core. The core has operand inputs a and b, a result output (result), and clock and reset inputs (clk, rst). The core is controlled by inputs op Nd and ce, and the output rdy. The ce input is the clock enable signal; when inactive (ce = 0)
C.3. EXAMPLE: FIFO QUEUE IN FTL

The core is ‘paused’. The op \(_{nd}\) input signalizes the validity of operand inputs; the core uses the validity information to decide if a stage contains meaningful data. The rdy (‘ready’) output signalises that data on the result output is valid. The signal is derived from op \(_{nd}\) via delay registers.

A core with the interface thus described can be used as a component in FTL design. Figure C.4(b) shows the integration of such hardware core in the activation net. The AN pattern is based on that for a cell (f-place) but without the FF register. Instead, the core’s control signals are connected to the AN.

- The input op \(_{nd}\) is active iff the transition wand \(s_1\) is enabled. Indeed, when \(s_1\) fires the input token is being consumed in the FTL component.

- The output rdy is asserted by the core when there is valid data at the result output. This makes the wand \(s_2\) potentially enabled. If indeed \(s_2\) is enabled the input ce (‘clock enable’) will be asserted and the result data value removed from the core at the next clock-tick. But if \(s_2\) is inhibited while rdy = 1, the ce will be inhibited and the core clock paused to prevent the loss of the result data value.

- When there is no valid data at the core’s result output the rdy output is 0. In that case the ce input is enabled by the AN pattern. New input data thus may be shifted in, and–more importantly–a computation already in flight inside the core may proceed.

C.3 Example: FIFO Queue in FTL

A FIFO queue in Figure C.5.(a) is constructed from two symmetrical parts (the input and output), each with a counter macro, a join component \((J_1/J_2)\) and a switch or select \((S_1/S_2)\), respectively. Data storage is realized by the i-cells (places) in the middle column \((C_0 - C_{2n-1})\). I-cells (0.5 tokens/tick) are used because in the queue there is no need to simultaneously read and write to the same storage place at a time.

All synchronization between the components is handled by tokens. The counters are kept in sync with input/output flows by the joins \(J_1, J_2\). In \(J_1\) the incoming token is paired with a token value from the counter. The current counter value explicitly selects one of the alternate pipes in the switch \(S_1\). At the next clock-tick the token is deposited in the chosen i-cell. If the i-cell is full the implicit back-pressure stops the incoming data token and the token from the counter. Queue overflow (full) and underflow (empty) conditions are detected by tokens. At the beginning (after a reset) both counters point to the first i-cell \(C_0\). As the i-cell \(C_0\) is empty there could not be a token produced on the FIFO output port (‘the queue is empty’). The first input token is deposited in \(C_0\) and the counter at input side is advanced by 1 to point to \(C_1\). At this time the output counter still points to \(C_0\), now full, and the output port can produce a token, if needed. The second input token is deposited in \(C_1\) and the counter will wrap to point to \(C_0\) again. If the first token in \(C_0\) has not been removed yet a third incoming token must wait because the queue is full. Once the first token in \(C_0\) is removed (the output counter will be advanced to point to \(C_1\)), the third input token is deposited in \(C_0\).

Counters (FSM)

The counters used in the queue are simple finite state machines (FSM). Two possible specifications of a counter in FTL are depicted in Figure C.5.(b-c). In (b) the token in cell \(C_0\) is duplicated in fork \(F_b\); one copy is re-loaded in the cell, the other is sent to the output. The trick here is that the duplication in the fork is controlled by readiness of the consumer connected to cnt. In (c) a new token is created in source \(F_c\) and loaded in cell \(C_c\) every time the token stored in the cell has been removed. The data
APPENDIX C. FTL: DETAILS

**Figure C.5:** FIFO queue.

**Figure C.6:** Expanded activation net of the FIFO queue. Data path in blue.

From a technical perspective the specification in Figure C.5.(b) is problematic because its AN contains a cycle. The version (c) has acyclic AN. Figure C.6 shows expanded activation net of the queue with data path drawn in blue colour.
C.4 Alternate Choices in Flowcharts and in FTL

The flowchart of the MF-set algorithm (Figure 7.13) shows that there are two junctions \( j_1 \), \( j_2 \) which represent merging of control-flows. The junction \( j_1 \) is at the point where control-flows from the input block \( B_0 \) and the main block \( B_2 \) merge. It is implemented in FTL-net (Figure 7.14) by priority select \( B \). The higher priority (port 1) is given to tokens coming from the loop, while new input tokens have the lower priority (port 2). The other junction \( j_2 \) is in the point where control-flows are exiting the algorithm. This junction is transformed into priority select \( L \). The higher priority is given to tokens from the loop.

In the flowchart there are two condition blocks: \( B_3 \) and \( B_4 \). In the model they are realized by switches \( K \) and \( D \), respectively. The switches are controlled by boolean condition flags computed in previous stages.

![Figure C.7: Representation of ‘alternate choices’ in flowcharts, FTL, and in hardware data path.](image)

Figure C.7 contrasts representations of ‘alternate choices’ in flowcharts, FTL and in hardware data-path. In flowcharts, typically used for software, there is no special symbol for merging of control/data-flows. In software, merging of alternate paths could be as simple as a jump to a common program address. In hardware, merging of several data-paths is implemented by a multiplexer which is potentially a costly structure. Consequently the special symbol for multiplexers is used in hardware diagrams.

Splitting of control flow in flowcharts is represented by a special condition block with the diamond shape. In software, a split of control-flow requires a conditional branch instruction which is usually costly relative to a straight sequential code. In hardware, the split of data path costs almost nothing (just a wire branching), as well as a split of control path in a controller automaton (all edges in a graph of a state machine cost the same, there is no preferred direction).

In classical Petri nets there is no special symbol for either merging or splitting (Figure 6.7). For splitting, the conditions are usually annotated at arcs. Merging is drawn by having multiple input arcs coming into a place.

In FTL, there is a special symbol for both merging (select) and splitting (switch).
Appendix D

FTL: Prototype CAD Tool

The prototype CAD tool translates a netlist of FTL components into synthesisable VHDL code. The tool itself is implemented in Python 3 programming language. The input netlist is created in the Python programming language by direct object instantiations. The tool is a library of functions called on the netlist objects.

There are two main data superstructures existing in the program over its run-time:

1. Flow netlist: composed of the FTL components (cells, i-cells, switches, select, ...) and connected by pipes.

2. Activation net (AN): derived from the flow netlist by pattern expansion (Figure 7.8).

D.1 Data Structures for Flow Netlists

The UML diagram of classes that implement the flow netlist is in Figure D.1. The classes with green (shaded) background are used for netlist construction, the other classes are internal. The base class for all FTL components is Element. Each Element has an array (list in Python) of ports and a name. Each port is an instance of PipeInFlow if it is an input port in element, or PipeOutFlow, if it is an output port in element. Two instances of PipeInFlow and PipeOutFlow are connected to form a pipe between two elements.

Pipes are connected using the « operator, and ports are accessed by index using the / (slash) operator. The code snippet below (Listing D.1) illustrates how three ICells (places in PN) A, B, C, and the priority select Select_Pri D are instantiated. The first constructor argument is the name of the element. The select D has 2 inputs (D.1, D.2) and 1 output (D.0), the icells have 1 input (x.0) and 1 output (x.1). The code shows how connections between the ports are established. The resulting netlist is the one shown in Figure D.3.

D.2 Data Structures for Activation Nets

D.2.1 Expression Trees

Activation net is represented by a collection of expressions. Expression trees are constructed from the following nodes (Python objects):

- NPrf – application of a primitive function.
Figure D.1: UML class diagram for construction of flow netlist.

Figure D.2: UML class diagram for activation nets.
Listing D.1: Three i-cells and a select.

- **NId** – a reference to AttrCore.
- **NAp** – application of a user (general) function.
- **NConst** – a special constant. For normal primitive constants (bools, ints, strings) the standard Python type is directly used in the AST, not wrapped in an object.

Of these the the NPrf node—the application of a primitive function—is the most important. The node contains a function identifier and an array of positional parameters, which are typically references to sub-trees. The primitive functions are:

- `@@ff(next-value, clock-enable, init-value)` = a flip-flop.
- `@@and'(v1, v2, ...)` = boolean AND of the v1, v2, ...
- `@@or'(v1, v2, ...)` = boolean OR of the v1, v2, ...
- `@@not(v1)` = boolean negation of v1.
- `@@cond(condition, choices-colon)` = ternary operator: the condition and a list of options given using the colon node.
- `@@colon(v1, v2, ...)` = list of values (v1, v2, ...), all of the same data type.
- `@@comma(v1, v2, ...)` = list of values (v1, v2, ...), different data types.
- `@@eq(v1, v2)` = comparison of two arbitrary values, the results is a boolean.
- `@@mpx(index, v0, v1, v2, ...)` = multiplexer, a selection from v0, v1, v2, ..., according to the index.
- `@@zero_const_tp(v)` = returns type name of the expression.
• \( @@nofo(v) \) = a pseudo-function, prevents dependency resolution ('no-follow').

• \( @@getfld(\text{struct-value}, \text{field-name}, \text{data-type}) \) = accesses a field in the structured value.

• \( @@downto(v1, v2) \) = descending range from \( v1 \) down to \( v2 \).

• \( @@to(v1, v2) \) = ascending range from \( v1 \) to \( v2 \).

• \( @@swisel(\text{object}, \text{port-index}, p1, p2, p3, ...) \) = pseudo-function for pattern expansion of the switch/select choice components.

### D.2.2 Example

The example flow diagram shown in Figure D.3 is initially expanded into a set of expressions shown below. Names of the variables (e.g. \( FX1 \)) encode their origin: the first letter is either \( F \) (= control path) or \( D \) (= data path), the second letter is the name of the original component (A, B, C, ...), and the third character is the original port number (0, 1, 2).

```
1  FX1 : std_ulogic := (!PAst0 & FX0);
2  FA1 : std_ulogic := (PAst0 & FD_cnd0 & @@swisel(D, 1, FD0, FA1, FB1));
3  DA1 : std_logic_vector(7 downto 0) := DAst0;
4  PAst0 : std_ulogic := @@ff(FX1, (FA1 | !PAst0), False);
5  DAst0 : std_logic_vector(7 downto 0) := @@ff(DX1, (FA1 | !PAst0), @@zero_const_tp());
6  FY1 : std_ulogic := (!PBst0 & FY0);
7  FB1 : std_ulogic := (PBst0 & FD_cnd1 & @@swisel(D, 2, FD0, FA1, FB1));
8  DB1 : std_logic_vector(7 downto 0) := DBst0;
9  PBst0 : std_ulogic := @@ff(FY1, (FB1 | !PBst0), False);
10  DBst0 : std_logic_vector(7 downto 0) := @@ff(DY1, (FB1 | !PBst0), @@zero_const_tp());
11  FD0 : std_ulogic := (!PCst0 & @@swisel(D, 0, FD0, FA1, FB1));
12  FC1 : std_logic := (PCst0 & FZ1);
13  DC1 : std_logic_vector(7 downto 0) := DCst0;
14  PCst0 : std.ulogic := @@ff(FD0, (FC1 | !PCst0), False);
15  DCst0 : std_logic_vector(7 downto 0) := @@ff(DD0, (FC1 | !PCst0), @@zero_const_tp());
16  FD_cnd0 : std_ulogic := FA1;
17  FD_cnd1 : std.ulogic := FB1;
18  DD0 : std_logic_vector(7 downto 0) := (FD_cnd0 ? (DA1 : DB1));
19  FX0 : std_ulogic := FX1;
20  DX1 : std_logic_vector(7 downto 0) := DX0;
21  FY0 : std_ulogic := FY1;
22  DY1 : std_logic_vector(7 downto 0) := DY0;
23  FZ1 : std_logic := FC1;
24  DZ1 : std_logic_vector(7 downto 0) := DC1;
```

The first compilation step is merging of \textit{wands} with a direct cyclical dependency. This is a technical artefact of how the AN is constructed. Example:

```
1  FX1 : std.ulogic := (!PAst0 & FX0);
2  FX0 : std.ulogic := FX1;
3  ->
4  FX1 := (!PAst0 & True & True & N.FX0.0)
```

This gives an intermediate set of expressions listed below. The \textit{select} D is not expanded, the expressions FA1, FB1, and FD0 still refer to the \( @@swisel \) pseudo-function.

```
1  FX1 : std_ulogic := (!PAst0 & True & True & N.FX0.0);
2  FA1 : std_ulogic := (PAst0 & True & @@swisel(D, 1, FD0, FA1, FB1) & True);
3  DA1 : std_logic_vector(7 downto 0) := DAst0;
```
D.2. DATA STRUCTURES FOR ACTIVATION NETS

To expand the \texttt{@swisel} pseudo-functions the \texttt{wands} must be split. This makes the inputs to the \texttt{@swisel} functions distinct from their outputs:

\begin{verbatim}
FA1 : std_ulogic := (N_FA1_3 & N_FA1_SSW_4);
FB1 : std_ulogic := (N_FB1_6 & N_FB1_SSW_7);
FD0 : std_ulogic := (N_FD0_9 & N_FD0_SSW_10);
N_FA1_3 : std_ulogic := (PAst0 & True & True & True);
N_FA1_SSW_4 : std_ulogic := @swisel(D, 1, N_FD0_SSR_11, N_FA1_SSR_5, N_FB1_SSR_8);
N_FA1_SSR_5 : std_logic := (N_FA1.3);
N_FB1_6 : std_ulogic := (PBst0 & True & True & True);
N_FB1_SSW_7 : std_ulogic := @swisel(D, 2, N_FD0_SSR_11, N_FA1_SSR_5, N_FB1_SSR_8);
N_FB1_SSR_8 : std_ulogic := (N_FB1.6);
N_FD0_9 : std_ulogic := (!PCst0 & True);
N_FD0_SSW_10 : std_logic := @swisel(D, 0, N_FD0_SSR_11, N_FA1_SSR_5, N_FB1_SSR_8);
N_FD0_SSR_11 : std_logic := (N_FD0.9);
....
\end{verbatim}

After that the \texttt{@swisel} functions are expanded into the control fragment (Figure 6.10), and the expressions are minimized:

\begin{verbatim}
FX1 : std_ulogic := (!PAst0 & N_FWD.0_0);
FA1 : std_ulogic := (N_FA1.3 & N_FA1_SSW.4);
DA1 : std_logic_vector(7 downto 0) := DAst0;
PAst0 : std_ulogic := @ff(FX1, (FA1 | !PAst0), False);
DAst0 : std_logic_vector(7 downto 0) := @ff(DX1, (FA1 | !PAst0), @zero_const_tp());
FB1 : std_ulogic := (PBst0 & True & @swisel(D, 2, FD0, FA1, FB1) & True);
DB1 : std_logic_vector(7 downto 0) := DBst0;
PBst0 : std_ulogic := @ff(FY0, (FB1 | !PBst0), False);
DBst0 : std_logic_vector(7 downto 0) := @ff(DY1, (FB1 | !PBst0), @zero_const_tp());
FD0 : std_logic := (!PCst0 & @swisel(D, 0, FD0, FA1, FB1));
FC1 : std_ulogic := (PCst0 & True & True & N_FZ1_2);
DC1 : std_logic_vector(7 downto 0) := DCst0;
PCst0 : std_ulogic := @ff(FD0, (FC1 | !PCst0), False);
DCst0 : std_logic_vector(7 downto 0) := @ff(DX0, (FC1 | !PCst0), @zero_const_tp());
DD0 : std_logic := (FA1 ? (DA1 : DB1));
DX1 : std_logic_vector(7 downto 0) := DX0;
FY0 : std_logic := (True & N_FY0_1 & !PBst0 & True);
DY1 : std_logic_vector(7 downto 0) := DY0;
DZ1 : std_logic_vector(7 downto 0) := DC1;
N_FA1_3 : std_ulogic := PAst0;
N_FA1_SSW_4 : std_logic := N_FD0_SSR.11;
N_FA1_SSR.5 : std_logic := N_FA1.3;
\end{verbatim}
N_FB1_6 : std_ulogic := PBst0;
N_FB1_SSW_7 : std_ulogic := (N_FD0_SSR_11 & !N_FA1_SSR_5);
N_FB1_SSR_8 : std_ulogic := N_FB1_6;
N_FD0_9 : std_ulogic := !PCst0;
N_FD0_SSW_10 : std_ulogic := (N_FA1_SSR_5 | N_FB1_SSR_8);
N_FB1_SSR_11 : std_ulogic := N_FB1_6;

The final expressions from above are translated into VHDL source code. For instance, the equation
PAst0 := ... (see above) is printed as:
if rising_edge(clk) then
  if to_bool(rst) then
    PAst0 <= '0';
  else
    if to_bool((FA1 or not PAst0)) then
      PAst0 <= FX1;
    end if;
  end if;
end if;

D.2.3 Tree Traversals

Boolean Constant Folding

The majority of transformations of expressions are implemented by recursive traversals over the tree. The listing below illustrates a Python class that implements simplification of boolean expressions by constant folding. In code the class is instantiated and the root of the expression tree is passed in the method run_on(). Machinery in the base class DefaultTrav analyses the type of node and calls one of the trav_ methods, such as trav_ap() for NAp, trav_prf_and() for NPrf(Prf.AND) etc.

class SimplifyExprTrav(DefaultTrav):
  *** Simplify an expression by applying a set of trivial transformations
  (constant folding).
  ***
  def trav_prf_not(self, eqd):
    eqd1 = self.do_trav(eqd.get_args()[0])
    if eqd1 == True: return False
    if eqd1 == False: return True
    if isinstance(eqd1, NPrf) and eqd1.get_prf() == Prf.NOT:
      # two negations, skip the two levels
      return eqd1.get_args()[0]
    else:
      # REPACK
      return NPrf(Prf.NOT, [eqd1], loc=eqd.get_loc())
  
def simil_variadic(self, args):
    *** Traverse (simplify) the sub-trees in args[0..n].
    Count the number of False and Trues in the resulting list.
    ***
    ns_args = []
    num_false = 0
    num_true = 0
    for i in range(0, len(args)):
      v = self.do_trav(args[i])
      if v == False:
        num_false += 1
      else:
if v == True:
    num_true += 1
else:
    ns_args.append(v)
return (ns_args, num_false, num_true)
def level_prf_args(self, prf, args):
    """ Try to level a nested prf of the same kind, eg. 
    (a & b) & c =>> a & b & c 
    (a | b) | c =>> a | b | c 
    """
    n_args = []
    for a in args:
        if a.get_prf() == prf:
            # the prfs are identical, skip the level 
            n_args.extend(a.get_args())
        else:
            n_args.append(a)
    return n_args

def trav_prf_and(self, eqd):
    """ Traverses the & operator. """
    ns_eqd, num_false, num_true = self.simpl_variadic(eqd.get_args())
    if (num_false > 0):
        # any false in &
        return False
    if (num_true == len(eqd.get_args())):
        # all true in &
        return True
    if len(ns_eqd) == 1:
        # only one component 
        return ns_eqd[0]
    else:
        # REPACK 
        return NPrf(eqd.get_prf(),
                    self.level_prf_args(eqd.get_prf(), ns_eqd) )
def trav_prf_or(self, eqd):
    """ Traverses the | operator. """

The code above works as follows:

If the current node is a negation (NPrf(Prf.NOT, ...)) the method trav_prf_not is called. The method recursively traverses the argument of the node at line 6. If the value obtained from the recursive traversal is the boolean constant (True, False) the inverted constant is immediately returned (e.g. NPrf(Prf.NOT, [False]) -> True). If the node obtained from the recursive traversal is itself a negation node these two negations are annihilated: NPrf(Prf.NOT, [NPrf(Prf.NOT, [x])]) -> x. Otherwise the negation node is recreated and returned from the method.

Simplification of the AND and OR nodes is similar, and will be explained on the former. If the current node is an AND (NPrf(Prf.AND, [a, b, c, d, ...]) – the AST allows any number of arguments to ANDs and ORs) the method trav_prf_and() is called. The method calls
simpl_variadic() which: (a) recursively traverses all arguments of the original node, and (b) counts the number of cases the simplified argument evaluated to constant False and True in variables num_false and num_true. Back in trav_prf_and() these counts are analysed: a non-zero num_false means that the whole AND node is False, and the full count in num_true means that the result is True. Otherwise the AND node is rebuilt (‘REPACK’) from the simplified arguments. Note that constant True/False arguments have been already removed in simpl_variadic().

The function level_prf_args flattens trees when nodes of the same kind are needlessly nested: NPrf(Prf.AND, [a, b, NPrf(Prf.AND, [c, d]), e]) -> NPrf(Prf.AND, [a, b, c, d, e]).

Absorption Laws

Absorption law is a powerful symbolic minimization technique. Typically it is written in the following form:

\[ a \land (a \lor b) = a \]  \hspace{1cm} (D.1)
\[ a \lor (a \land b) = a \]  \hspace{1cm} (D.2)

The fruitful approach is to think about the inner references to \( a \) as being restricted in values by their outer reference. In Equation D.1 if \( a = 1 \) then the value of all inner references to \( a \) is 1 (obviously); if \( a = 0 \) then due to the outer \( \land \)-operation the result is 0 and the value of \( a \) in any inner reference does not matter. Similarly in Equation D.2, if \( a = 0 \) then the value of all inner references to \( a \) is 0 (obviously); if \( a = 1 \) then due to the outer \( \lor \)-operation the result is 1 and the value of \( a \) in any inner reference does not matter. Therefore, the following is correct:

\[ a \land (a \lor b) = a \land (1 \lor b) \]  \hspace{1cm} (D.3)
\[ a \lor (a \land b) = a \lor (0 \land b) \]  \hspace{1cm} (D.4)

It is evident that all the inner references to \( a \) can be replaced by a constant 1 or 0, depending only if the outer operator was \( \land \) or \( \lor \), respectively. Moreover, the contents of the inner expression (in the parenthesis) is irrelevant:

\[ a \land (...a...) = a \land (...1...) \]  \hspace{1cm} (D.5)
\[ a \lor (...a...) = a \lor (...0...) \]  \hspace{1cm} (D.6)
\[ \neg a \land (...a...) = \neg a \land (...0...) \]  \hspace{1cm} (D.7)
\[ \neg a \lor (...a...) = \neg a \lor (...1...) \]  \hspace{1cm} (D.8)

The four substitution patterns shown above are implemented in the AbsorpLawTrav class.
Appendix E

Microthreaded Processor UTLEON3

The UTLEON3 processor is an implementation of the SVP model. For more information see the works [X.1, X.3, X.8, X.11, X.12]. The model is multicore-aware, meaning concurrent threads are to be automatically spread in a processing micro-grid environment: a 2D array of SVP processors. The UTLEON3 implementation is written in VHDL, it has a 7-stage in-order single-issue multi-threaded execution pipeline, and it is fully FPGA-synthesizable. It is based on the LEON3 SPARCv8 embedded processor from Aeroflex Gaisler [GCH07].

Microthreading is a multi-threading variant that decreases the complexity of context management. The goal is to tolerate long-latency operations (LD/ST and multi-cycle operations such as floating-point), and to synchronize computation on register access. A possible speed-up generated by microthreading comes from the assumption that while one thread is waiting for its input data, another thread has its input data ready and can be scheduled in a few clock cycles and executed. Another assumption is that load and store operations themselves need not be blocking since the real problem arises just when an operation accesses a register that does not contain a valid data value.

E.1 Overview of the micro-architecture

E.1.1 Processor functional model

Figure E.1 shows the structural block diagram of the processor. Figure E.2 shows a simplified functional model of the UTLEON3 processor. Several things can be noted: Instructions originate either in the I-Cache or in the scheduler. The I-Cache supplies instructions either to the fetch stage or to the scheduler to implement a 0 clock cycle (CC) switch on an I-Cache miss. The program counter is calculated either in the processing pipeline, or a new value is supplied by the scheduler on a context switch. On a D-Cache hit the pipeline operates as the normal SPARC pipeline; on a D-Cache miss (LD) the pipeline marks the destination register as pending and continues processing; the register update happens autonomously in the background; on a D-Cache miss (ST) the value to be written is put in a store queue (not shown).

An overview of the pipeline operation on an instruction or data miss is shown in Fig. E.3 (legacy mode) and Fig. E.4 (micro-threaded mode). The numbers in the boxes between the pipeline stages denote the size of the incurred bubble in the pipeline. In the legacy mode the pipeline stalls on instruction or data misses, while in the micro-threaded mode a context switch brings in a new thread that has data ready for execution.

A thread switch can be triggered in three pipeline stages:
Figure E.1: Structural diagram of the UTLEON3 processor with connections to the HW-FAM accelerator subsystem, which is detailed in Figure 9.1. Pipeline Stages: FE = Instruction Fetch, DE = Decode, RA = Register Access, EX = Execute, ME = Memory Access, XC = Exception, WB = Writeback. RAU = Register Allocation Unit, RUC = Register Update Controller, A-Cache = AMBA–Cache Interface.

Figure E.2: A functional model of the UTLEON3 processor.
E.1. OVERVIEW OF THE MICRO-ARCHITECTURE

- **FE_SWITCH** - occurs on an I-Cache miss, the scheduler supplies a new instruction from another thread.

- **DE_SWITCH** - occurs on the `swch` instruction modifier, the scheduler supplies a new instruction from another thread.

- **EX_SWITCH** - occurs on reading a pending register that does not contain valid data, the pipeline is flushed and 3CCs are lost.

![Figure E.3: Latencies in the legacy LEON3 processor pipeline](image)

![Figure E.4: Latencies in the micro-threaded UTLEON3 processor pipeline](image)

E.1.2 Register File

Traditional multi-threaded architectures often replicate the whole processor state for each thread context supported, including all the architectural (program visible) registers [AKK+93, MD96]. This simplifies the software programming model, but it requires a large register file to support just few thread contexts. As the processor register file is one of the most expensive units in a CPU [RDK+00], its optimal utilization is very important. Previous experimental work [REL03] has shown the advantage of reducing the per-thread context size. In [REL03] the authors reduced the context from 32 to 16 registers (i.e. two mini-threads in one context) without much performance impact.

The UTLEON3 processor allows each thread to individually specify the number of required registers, from 1 up to the architectural limit of 32 registers per thread given by the SPARCv8 instruction encoding. (This is a more advanced mechanism than in [WW93].) The code specifies the number of registers required for the thread using the `.registers` pseudo-instruction given at the beginning of each thread routine.

Fine-grained synchronization and communication between threads in one family, and between the processor pipeline and long-latency units (cache, FPU) is accomplished by a self-synchronizing register file based on I-structures [ANP89]. Each 32b register in the file is extended with a state information.
A register can be either FULL when its data is valid, or PENDING when its data is scheduled to arrive, or WAITING if there is a thread waiting for the data.

When a register contains valid data, it is in the FULL state. If data are to be delivered into the register asynchronously (from cache, FPU etc.), but they have not been required by the pipeline yet, the register is in the PENDING state. Finally, if the data were required by a thread, but were not available at the time, the register is in the WAITING state; only on the PENDING⇒WAITING transition the thread has to be switched out and suspended.

### E.1.3 Thread Scheduling

The processor implements blocked (coarse) multithreading, meaning a thread is switched out of the pipeline only when an unsatisfied data dependency (i.e. a dependency on a long-latency operation) has been encountered. This improves the single-thread performance, but it requires the pipeline to have fully bypassed stages. When a thread switch occurs, the affected thread's instructions in the previous pipeline stages must be flushed. For example when a thread in the Execute Stage of the pipeline reads a register in the PENDING state, it must be switched out as the data in the register is not valid, and thus the two previous pipeline stages (Decode, Register Access) must be cleared as well – but only if they contain instructions from the same thread. To optimize this case the architecture allows to annotate each instruction with a swch modifier that will cause the thread to voluntarily switch itself out of the pipeline early in the Fetch Stage (if a compiler or assembler programmer anticipates the instruction depends on a long-latency producer instruction). The voluntary switch causes zero overhead in most cases as there is no pipeline bubble inserted. However, the dependent instruction must be cleared in any case and later re-executed, so the synchronization cost in the typical case is 1 cycle, while without the modifier it is 3 cycles. The processor allows instructions from different threads to be present in distinct pipeline stages at the same time.

### E.1.4 Families of Threads

Contrary to some previous multi-threaded architectures which often provided only a few (4-8) hardware thread contexts, the UTLEON3 processor supports tens to hundreds of concurrent hardware threads. To be able to take advantage of the large-scale parallelism offered the threads are created in families.

Conceptually, threads in one family share data and implement one piece of a computation. In a simple view one family corresponds to one for-loop in the classical C language; in microthreading each iteration (each thread) of a hypothetical for-loop (represented by a family of threads) is executed independently according to data dependences.

As the families of threads can be nested similarly to the classical for-loops, the processor can execute many different families, each comprising of many threads, at the same time. Families compete for processor resources, mainly for entries in the thread table and registers in the register file. To regulate resource allocation and sharing among families of threads, the architecture allows to specify (for each family individually) a so-called blocksize parameter. The blocksize value limits the maximal number of threads of a given family that are allowed to co-exist at any moment in the thread table. This enables a compiler or assembly-level programmer to optimize the rate of thread creation in each family. This parameter is optional for it does not affect the semantics of the computation, but it affects its pace.
E.2 Previous Multithreaded Processor Architectures

Single-threaded architectures focus on extraction of independent instructions (ILP) that can be executed during a stall. One of the in-order execution techniques to accomplish it is the run-ahead execution [BNS+03, NAJ+08] that allows a program to proceed past stalled instruction. The result register of the stalled instruction is marked as poisoned, and so all the dependent instructions can be identified. Once the stall is over, the dependent instructions must be re-executed, and their results correctly integrated into the architectural state. Other techniques try to decouple the producer and consumer instructions [PG01, RVVA04]. By executing the producer instructions (such as a memory load) early its effective latency can be partly hidden.

Multicore architectures require many concurrent tasks to be fully utilized. Existing architectures usually presume a multi-programmed environment (e.g. OpenSPARC T1/T2), where processes communicate through shared memory or by message passing. Others provide ISA extensions to support multiple thread execution [URv03], but they do not automatically spread threads over multiple cores. The MIPS MT [Kis08] architecture introduces only two new unprivileged instructions: fork to create a new thread, and yield to make a thread wait for an event. The Sparcle/Alewife [AKK+93] system is based on a slightly modified SPARC architecture. It employs block (coarse) multithreading. The SPARC register windows are used to implement 4 independent thread contexts rather than as a register stack; context switching and thread scheduling is done in software via fast traps; and fine-grain synchronization through empty/full bits is implemented in an external Communications and Memory Management Unit and in the cache controller. The MSparc [MD96] architecture is similar to Sparcle, but the context switching mechanism is provided in hardware.
Appendix F

HWFAM: Example Mapping of a FIR Filter

F.1 Description of the FIR Filter

The Finite Impulse Response (FIR) filter is one of the basic algorithms in the Digital Signal Processing (DSP) field. It is widely used in many applications because of the inherent stability of the filter due to the lack of feedback, and a simple data-parallel implementation.

The FIR filter output is defined by the following equation:

$$z_k = \sum_{i=0}^{t-1} (x_{k+i} \cdot b_i)$$  \hspace{1cm} (F.1)

where $\vec{z}$ is the output vector, $|\vec{z}| = n - t + 1$; $\vec{x}$ is the input vector, $|\vec{x}| = n$; $\vec{b}$ is the vector of coefficients (taps), and $t$ is the length of the vector $\vec{b}$. We assume $n \geq t$, typically $n \gg t$. The value $z_k$ is the dot product of $\vec{b}$ and $\vec{x}_{[i..i+t]}$.

F.1.1 Implementation in the Microthreaded Processor

In a sequential program the computation is implemented using two nested for loops: the inner loop computes the dot product, while the outer loop runs over the whole index space of the vector $\vec{z}$, as shown in Listing F.1 below.

```c
for (int k = 0; k < n; ++k) {
    sum = 0;
    for (int i = 0; i < t; ++i) {
        sum += x[k+i] * b[i];
    }
    z[k] = sum;
}
```

**Listing F.1:** The basic FIR filter sequential algorithm in C pseudo-code.

The basic FIR filter algorithm is embarrassingly data-parallel. Each element $z_k$ can be computed independently of the other, and also the inner summation loop can be parallelized. In SVP both loops are implemented by families of threads. Decomposition of the algorithm to threads is in Figure F.1. At the top the thread labelled ‘FIR’ represents the entry point to the function. This thread creates the family of threads that conceptually implements the outer for-loop in Listing F.1 at line 1. Subsequently,
... /* other code */
/* fir_top: Create (call) the FIR outer family */
/* Registers (calling convention): */
/* %tl0 = (input) xvec = address of the x vector */
/* %tl1 = (input) zvec = address of the z vector */
/* %tl2 = (input) bvec = address of the b vector */
/* %tl3 = (input) bv4_len = (t-1)*sizeof(int), t is the number of coefficients. */
/* %tl4 = (input) xv4_len = (n-1)*sizeof(int), where n is the length of x */
*/
allocate %tl5 /* allocate family handle in %tl5 */
setlimit %tl5, %tl4 /* set: limit */
setstep %tl5, 4 /* ... step */
set fir_outer, %tl6 /* ... thread fun. addr. */
setthread %tl5, %tl6
setblock %tl5, BLOCKSIZE_1
create %tl5, %tl5 /* run */
... /* other code that could be overlapped with the FIR execution */

/* wait for the termination of the created family: */
ymov %tl5, %tl6

... /* other code */

/***************************************************************************/
/* Registers: */
/* %tl0 = offset of the word in xvec/zvec [thread index] */
/* %tg0 = xvec */
/* %tg1 = zvec */
/* %tg2 = bvec */
/* %tg3 = bvec_len limit */
.registers 4 0 8 0 0 0 /* require 4 global (%tgX), 8 local registers (%tlX) */

fir_outer:
/* prepare to create the inner family: */
add %tg0, %tl0, %tl1 /* sub-%tg1 = xvec, current ptr */
mov %tg2, %tl2 /* sub-%tg2 = bvec */
set 0, %tl3 /* sub-%ts0 = running inner sum */
allocate %tl4
setlimit %tl4, %tg3
setstep %tl4, 4
setthread %tl4, %tl5
setblock %tl4, BLOCKSIZE_2
create %tl4, %tl4 /* run the inner family */
mov %tl4, %tl5 /* wait for termination */

/* store result to zvec */
st %tl3, [%tg1+%tl0]; END

Listing F.2: Assembly source code of the FIR top and outer-loop thread functions.
F.1. DESCRIPTION OF THE FIR FILTER

The FIR filter is decomposed into threads, represented by ovals. The family tree is dynamically unwound in the processor. Dashed lines represent data dependencies between sibling threads in inner families, responsible for the summation.

**Figure F.1:** The FIR algorithm decomposed in threads, represented by ovals. The family tree is dynamically unwound in the processor. Dashed lines represent data dependencies between sibling threads in inner families, responsible for the summation.

**Listing F.3:** Assembly source code of the FIR innermost family threads.

```
/* require 3 global (%tgX), 1 shared (%td0+%ts0), */
/* 3 local (%tlX) regs. */

/* %tg1 = xvec, current ptr
 * %tg2 = bvec
 * %ts0 = running inner sum */

fir_inner:
    ld [%tg1+%tl0], %tl1 /* load x[k+i] */
    ld [%tg2+%tl0], %tl2 /* load b[i] */
    umul %tl1, %tl2, %tl2
    add %td0, %tl2, %ts0 ; END /* partial sum */
```

Each thread in the outer family creates an ‘inner’ family of threads to implement the summation loop at line 3. The summation requires passing of the sum variable from the thread $i$ to $i + 1$. This communication pattern, depicted in Figure F.1 by dashed lines, is supported in the SVP architecture using the sibling-shared registers.

The complete assembly source code of the FIR algorithm executable on the microthreaded processor UTLEON3 is shown in Listings F.2 and F.3. The subprogram calling sequence is in Listing F.2 at lines 2–22, and the outer family thread function called ‘fir_outer’ at lines 33–51. The inner family thread function called ‘fir_inner’ is in Listing F.3.

The FIR subprogram calling sequence at lines 2–22 could be a part of other higher code. It creates thread family implementing the outer loop. First, a new family handle is allocated (line 11) and stored in the local register %tl5. Then the family parameters are set using the set-X instructions. The instructions setstart, setlimit, and setstep write the loop bound values (start and limit), and the increment (step) value. Each thread in the new family will correspond to one iteration of the hypothetical loop. The instruction setthread writes the program address of the thread function of the new family (‘fir_outer’ in this case). The thread function corresponds to the iteration body code...
APPENDIX F. HWFAM: EXAMPLE MAPPING OF A FIR FILTER

block of the hypothetical loop. Once the parameters are set, the new family is created (line 17). This begins the actual execution of the new family in the processor.

The new family executes \( n \) threads (\( \leq (\text{limit} - \text{start} + 1)/\text{step} \)), each running the code ‘fir outer’ listed at lines 33–51. The .registers directive before the beginning of the thread function is used to reserve the required number of architectural registers for the thread. The ‘fir outer’ thread needs 4 global registers, called \%tg0–%tg3 in the code, and 8 local (private) registers, called \%tl0–%tl7. Global registers are physically shared in the processor register file among all threads in the family and the parent thread. The global registers \%tg0–%tg3 in family threads are mapped to the local registers \%tl0–%tl3 in the parent thread. The code in the ‘fir outer’ threads creates new families executing the ‘fir inner’ thread functions.

The assembly code of the ‘fir inner’ thread families is in Listing F.3. According to the .registers directive the thread function expects 3 global registers (\%tg0–%tg2), 1 sibling-shared register (\%td0+\%ts0), and 3 local (private) registers (\%tl0–%tl2). Global registers are shared between the parent thread and all the family threads. The registers are written in the parent thread before the family starts (Listing F.2 at line 37), and they stay constant throughout the execution of the family. In the ‘fir inner’ function the register \%tg1 contains pointer to the current position of the outer loop within the \( \vec{x} \) vector. This could be represented in the original sequential C code in Listing F.1 as ‘&\( x[k] \)’.

The initial contents of the local register \%tl0 in threads is special. The register is written by hardware prior to thread start with the thread’s individual index value. The first logical thread in a family receives the index value equal to the family’s start value, the second receives start+step, the third receives start+2*step, and so on until the limit value is reached. The family of ‘fir inner’ threads is iterated from the index 0 to 4(\( t-1 \)) (where 4 is sizeof(int), and \( t \) is the length of the vector \( \vec{b} \)).

The sibling shared registers are used for communication between adjacent threads in family. One sibling-shared register is required in ‘fir inner’ threads. In each thread in the family the register is accessible via two names: \%td0 and \%ts0. The \%td0 register name refers to input shared register of this thread and the \%ts0 to the output shared register. The output shared register \%ts0 in thread \( i \) is physically mapped to the input shared register \%td0 in the following thread \( i+1 \), and so on. The input shared register in the first thread and the output shared register in the last thread in family are physically mapped to the family’s parent thread. The mapping of the sibling shared registers is depicted in Figure F.1 using dashed lines.

The code in ‘fir inner’ threads first loads two elements \( x[k+i] \) and \( b[i] \), and multiplies them together (lines 7–9). The add instruction at line 10 encodes several operations. It reads values from the registers \%td0 (input sibling shared register) and \%tl2 (result of the umul instruction). If any of the two registers are not in the full state, the thread execution is suspended and another thread is injected in the processor pipeline. Once the inputs are gathered the addition is performed and the result is written to the output shared register \%ts0. The write may potentially wake up the following thread’s add instruction waiting on the register. Finally, the END modifier specifies that the thread terminates once the instruction is completed.

F.1.2 Performance Analysis

The data-flow graph of one thread of the ‘fir inner’ family is depicted in Figure F.2. Instructions are issued in the order from top to bottom. Each instruction takes at least 1 clock-cycle (cc) to complete. The additional sources of latencies are the D-Cache reads, the multiplication operation, and the dependency on the previous thread in the add instruction. D-Cache hits take 1 cc, but D-Cache misses may stall for unspecified number of cycles. Technically, the instruction issue continues even after the D-Cache miss, but the destination register of the missed load instruction is marked pending to cause the stall on the
first attempt to read it. In this case this would be at the time the umul instruction is issued. All stalls are eventually overcome by a thread switch. The stalled thread is suspended; the stalling instruction is aborted and the program counter at the instruction is stored in the thread table. Replacement thread is fetched from the ready queue and its next instruction is injected in the pipeline at the fetch stage.

In ‘fir_inner’ threads the two load instructions read consecutive words from the arrays $\vec{x}$ and $\vec{b}$. The array $\vec{b}$ is repeatedly scanned in each instance of the ‘fir_inner’ family, i.e. $(n - t + 1)$-times. Similarly, elements in the middle part of the array $\vec{x}$ are each accessed $t$-times. The default cache-line length in UTLEON3 is 16 words. In run-time the first thread that hits a word in a missing cache-line will be suspended at the umul instruction. However, once the cache-line is loaded into D-Cache the failing load instructions are not re-executed. Instead the data is delivered automatically by the register update controller directly to the load’s destination register (%tl1 or %tl2).

The multiplication instruction umul typically has non-unit latency. In the default configuration the latency is 4 clock-cycles. In contrast to the D-Cache the multiplier’s latency is pipelined (the D-Cache latency is non-pipelined). This means that a new operation in the multiplier can be started each clock-cycle (= the initiation interval is 1 cc). It is only a matter of having the necessary minimal number of threads (4) ready in the queue to keep the processor utilized.

In the presented assembly source code integer numbers are used to represent the values in the filter. This is because the current implementation of the processor does not support floating-point computations in the microthreaded mode. However, the performance figures would not change much because, using the typical 32-bit single-precision arithmetic in DSP algorithms, the data sizes are equal, and latencies in Floating Point Unit (FPU) are comparable to the latency of umul.

The complete analysis of performance impacts of the pipelined and non-pipelined latencies is in my work [X.11]. The conclusion drawn from the paper and the description above is that microthreading in principle can achieve the throughput close to 1 instruction per cycle (IPC=1) in a single core. This requires sufficient parallelism in the form of independent threads.

Assuming the ideal IPC=1 and neglecting the code that sets-up thread families in Listing F.2 we can trivially compute the minimal number of cycles the filter algorithm will take on a single core. The threads in ‘fir_inner’ families contain 4 instructions, hence they take at least 4 cc each, and there is $(n - t + 1) \cdot t$ of them. The minimal execution time of the FIR filter in UTLEON3 is thus:

$$T_{\text{UTLEON3}} = 4 \cdot (n - t + 1) \cdot t \quad \text{[cycles]}$$

where $n = |\vec{x}|$, $t = |\vec{b}|$. 

![Figure F.2: Data-flow in one thread of the ‘fir_inner’ family.](image)
The single-core \( T_{UTLEON3} \) could be improved by the way of specialization in the processor architecture. Inspirations can be drawn from DSP processors. As the first step the mul and add instructions could be fused to form a new multiply-add instruction. The fused multiply-add requires 3 source registers, hence it would necessitate more ports in the register file. The two load instructions cannot be executed in parallel unless the D-Cache would provide two independent ports. In DSP processors there are often two independent local memories (X, Y) in the core to support multiple loads in the same cycle.

In SVP multiple cores can be employed to cooperatively execute a family of threads. In the filter example the threads in the ‘fir_outer’ family in Listing F.2 are data-independent, hence they could be partitioned over multiple processor cores without much synchronization effort. The overhead for this partitioning scheme is the duplication of the vector \( \mathbf{b} \) and of the parts of the vector \( \mathbf{x} \) in multiple L1 D-Caches.

### F.2 Mapping of the FIR Filter in HWFAM

The FIR filter example contains two levels of nested thread families. The inner family computes a single element at a given index \( k \) of the result array \( z \) by the way of the dot-product between \( x \) and \( b \). The outer family simply replicates the inner family over all indexes \( k \). The hierarchy of the families and the data dependencies in the inner family are shown graphically in Figure F.1.

Internal structure of the hardware FIR accelerator core is in Figure F.4. The accelerator has three local memory blocks (BlockRAMs) denoted A, B, and C. The memory blocks A and B must be preloaded with contents of the vectors \( \mathbf{x} \) and \( \mathbf{b} \), respectively. The computation is implemented using a single multiplier and adder units. All data is 32bit wide integer values. A single-purpose embedded automaton (not shown in the picture) reads values from the memory blocks A and B according to the FIR access pattern. The values from A and B are multiplied and the intermediate result is accumulated in the adder. Each time the index in the memory block B (\( \mathbf{b} \)) reaches the maximal limit value, the current value in the accumulator is stored in the memory block C (\( \mathbf{z} \)) and the accumulator is cleared.
F.2. MAPPING OF THE FIR FILTER IN HWFAM

211

\[ \vec{x} : \begin{array}{cccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \quad \text{BlockRAM ‘A’} \]

\[ \times \quad \ast \quad + \quad \vec{z} : \begin{array}{cccc} 0 & 1 & 2 & 3 \\ \text{BlockRAM ‘C’} \end{array} \]

\[ \vec{b} : \begin{array}{cccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \quad \text{BlockRAM ‘B’} \quad \rightarrow \text{32 bit data} \]

Figure F.4: Internal structure of the FIR accelerator.

\begin{align*}
\text{Loc.:} & \quad \text{Parameter:} \\
%tg0 & \quad \text{ptr to } \vec{x} \\
%tg1 & \quad \text{ptr to } \vec{z} \\
%tg2 & \quad \text{ptr to } \vec{b} \\
%tg3 & \quad 4(t-1) \\
\text{start} & \quad 0 \\
\text{step} & \quad 4 \\
\text{limit} & \quad 4(n-1) \\
\end{align*}

Table F.1: Software FIR parameters.

\begin{align*}
\text{Loc.:} & \quad \text{Parameter:} \\
1 & \quad n \\
2 & \quad \text{offset of } \vec{x} \text{ in A} \\
3 & \quad \text{offset of } \vec{b} \text{ in B} \\
4 & \quad \text{offset of } \vec{z} \text{ in C} \\
5 & \quad t \\
\end{align*}

Table F.2: Configurable parameters in the FIR accelerator core.

Compared to the software microthreaded implementation the speed-up comes from the following sources:

- fetching of the values \( x[k+i] \) and \( b[i] \) is done concurrently because each is stored in a physically separate memory block;
- all operations (load, mul, add) are performed in parallel;
- the input data arrays are pre-fetched into the local memory blocks, hence there cannot be a cache-miss.

The procedure calling parameters of the original FIR software family are listed in Table F.1. The parameters \%tg0–%tg3 are stored in the family-global registers, physically allocated within local registers of the parent thread. As detailed in the table they contain pointers to the arrays \( \vec{x}, \vec{b}, \) and \( \vec{z} \), and the value of \( t \). The parameter \%gain is located in family’s creation event and it corresponds to \( n \).

The hardware accelerator core for the FIR filter is configurable by parameters shown in Table F.2. Besides the \( n \) and \( t \) parameters it is also possible to specify the starting addresses of the \( \vec{x}, \vec{b}, \) and \( \vec{z} \) arrays within the local memory blocks A, B, and C, respectively.

The diagram of interactions in the HWFAM subsystem when executing the FIR family in hardware is displayed in Figure F.4. Compared to the previous general discussion this picture includes the specific details of the motivational example. The execution proceeds according to the following script:

1. The family creation event is raised once the top parent family executes the create instruction (Listing F.2, line 17). The creation event/message contains the start/step/limit parameters and a pointer to the global registers in the CPU register file (but not their actual values).
2. The creation message is sent to the Thread Mapping Table (TMT)
3. If the creation message succeeds in TMT a notification is sent to the resource manager (RM).
(4) The RM extracts the base pointer to the global registers from the creation message and fetches their values by directly accessing the CPU register file. This operation makes all the FIR family software parameters (Table F.1) available in hardware.

(5) The RM configures the DMA engine to load the \( \vec{x} \) and \( \vec{b} \) arrays from the main memory to the local memory blocks in the accelerator core.

(6) The input arrays are read by the DMA engine in coarse blocks.

(7) The RM sets execution parameters in the accelerator core (Table F.2) and issues a start command to it.

(8) Once the computation is over the result data in the form of the arrays \( \vec{z} \) are transferred from the local memory block C to the main memory.

(9) Finally, the RM writes the thread termination register in RF and invokes a Family Completion Message to the CPU scheduler. Parent thread waiting on the child completion is unblocked. The scheduler deallocates the family entry in its data structures.
Appendix G

HWFAM: Experimental Data

G.1 Experimental Set-up

The work-flow of the experimental set-up is depicted in Figure G.1. Simulation runs are performed in Modeltech’s ModelSim tool, a Verilog/VHDL simulator. The setting of various option switches in software (assembler macros) and hardware (VHDL generics) source codes is specified in Simulation configuration files that use the file extension *.cfg. Each option in the file can be specified as a vector of values. Compilation, simulation, and aggregation of results is controlled by the Masstest tool. Masstest is a collection of tools written in perl. Since the options in simulation configuration files may have multiple values each, the file describes a volume of configuration space. Each point in the space is a complete simulation run. Masstest walks through the configuration space and runs the simulator for each point. Partial results from all simulation runs are collected and a single aggregated table file is generated. In the table each row is a simulation run, each column is either a parameter of the run or an event counter. Data processing and visualization of the result tables, including the plotting of graphs used in this thesis, is done in the R system [R C12].
G.2 Experimental Configurations

```
# hardware parameters:
hw-dsets 4
hw-dsetsize 1

# software parameters:
program h17

sw-BLSZ_NFIRS 4
sw-BLSZ_FIRZV 4
sw-BLSZ_REDU 4

sw-USEHWT;sw-FORCEFLUSH 0;0 0;1 1;1
sw-AFLUSH_HACK 0

# fir:
sw-NFIRS 0
sw-XVECLEN 48
sw-BVECLEN 24

# dct:
sw-NDCTS 1 2 3 4
sw-BLSZ_NDCTS 1 2 4
sw-BLSZ_DCTROWCOL 4
```

Listing G.1: h17-dct.cfg, Ad: Occupancy-Based Placement, DCT
G.2. EXPERIMENTAL CONFIGURATIONS

Listing G.2: h17-fir-large.r.cfg, Ad: Occupancy-Based Placement, FIR

Listing G.3: h17-fir-minsz-sw.cfg, Ad: Magnitude-Based Placement, FIR
APPENDIX G. HWFAM: EXPERIMENTAL DATA

Listing G.4: h17-fir-minsz-hw.cfg, Ad: Magnitude-Based Placement, FIR

Listing G.5: h17-fir-minsz-hyb.cfg, Ad: Magnitude-Based Placement, FIR
G.2. EXPERIMENTAL CONFIGURATIONS

Listing G.6: h17-fir-minsz-hyb-t.cfg, Ad: Magnitude-Based Placement, FIR

Listing G.7: h17-fir-virt2.cfg, Ad: Tolerance of the TMT Latency Scaling, FIR
G.3 Ad: Occupancy-Based Placement

Figure G.2: Benchmark: FIR, 1x, larger instances (32x128). Two accelerators (‘Acc 1’, ‘Acc 2’). Config.: Listing G.2.
G.3. **AD: OCCUPANCY-BASED PLACEMENT**

**Figure G.3:** Benchmark: FIR, 1x, larger instances (32x512). Two accelerators ('Acc 1', 'Acc 2'). Config.: Listing G.2.

**Figure G.4:** Benchmark: FIR, 1x, larger instances (16x1024). Two accelerators ('Acc 1', 'Acc 2'). Config.: Listing G.2.
Figure G.5: Benchmark: FIR, 4x, larger instances (32x128). Two accelerators (‘Acc 1’, ‘Acc 2’). Config.: Listing G.2.

Figure G.6: Benchmark: FIR, 4x, larger instances (32x512). Two accelerators (‘Acc 1’, ‘Acc 2’). Config.: Listing G.2.
**G.3. AD: OCCUPANCY-BASED PLACEMENT**

<table>
<thead>
<tr>
<th>Time [cycles]</th>
<th>a) CPU only</th>
<th>b) BLOCKSIZE=1</th>
<th>c) BLOCKSIZE=2</th>
<th>d) BLOCKSIZE=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>600000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure G.7:** Benchmark: FIR, 4x, larger instances (32x1024). Two accelerators (‘Acc 1’, ‘Acc 2’). Config.: Listing G.2.