Many-core Architectures with Time Predictable Execution Support for Hard Real-time Applications

by

Michel A. Kinsy

Bachelor of Science, Arizona State University, May 2007
Master of Science, Massachusetts Institute of Technology, May 2009

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

© Massachusetts Institute of Technology 2013. All rights reserved.

Author .................................................................

Department of Electrical Engineering and Computer Science

May 22, 2013

Certified by .................................................................

Srinivas Devadas
Edwin Sibley Webster Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .................................................................

Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students
Many-core Architectures with Time Predictable Execution Support for Hard Real-time Applications

by

Michel A. Kinsy

Submitted to the Department of Electrical Engineering and Computer Science on May 22, 2013, in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Abstract

Hybrid control systems are a growing domain of application. They are pervasive and their complexity is increasing rapidly. Distributed control systems for future “Intelligent Grid” and renewable energy generation systems are demanding high-performance, hard real-time computation, and more programmability.

General-purpose computer systems are primarily designed to process data and not to interact with physical processes as required by these systems. Generic general-purpose architectures even with the use of real-time operating systems fail to meet the hard real-time constraints of hybrid system dynamics. ASIC, FPGA, or traditional embedded design approaches to these systems often result in expensive, complicated systems that are hard to program, reuse, or maintain.

In this thesis, we propose a domain-specific architecture template targeting hybrid control system applications. Using power electronics control applications, we present new modeling techniques, synthesis methodologies, and a parameterizable computer architecture for these large distributed control systems.

We propose a new system modeling approach, called Adaptive Hybrid Automaton, based on previous work in control system theory, that uses a mixed-model abstractions and lends itself well to digital processing. We develop a domain-specific architecture based on this modeling that uses heterogeneous processing units and predictable execution, called MARTHA. We develop a hard real-time aware router architecture to enable deterministic on-chip interconnect network communication. We present several algorithms for scheduling task-based applications onto these types of heterogeneous architectures.

We create Heracles, an open-source, functional, parameterized, synthesizable many-core system design toolkit, that can be used to explore future multi/many-core processors with different topologies, routing schemes, processing elements or cores, and memory system organizations.

Using the Heracles design tool we build a prototype of the proposed architecture using a state-of-the-art FPGA-based platform, and deploy and test it in actual physical power electronics systems. We develop and release an open-source, small representative set of power electronics system applications that can be used for hard real-time application benchmarking.

Thesis Supervisor: Srinivas Devadas
Title: Edwin Sibley Webster Professor of Electrical Engineering and Computer Science
Acknowledgements

First, I would like to express my deep and sincere gratitude to Professor Srinivas Devadas for his extraordinary guidance, inexhaustible energy, and remarkable patience throughout this research. His support, encouragement, sound advice, and good teaching have been a true source of inspiration for me.

I am very fortunate to have Professor Arvind and Dr. Ivan Celanovic on my dissertation committee. I am deeply appreciative of Professor Arvind, who has been like a second advisor. He is always available for fruitful and engaged discussions; always ready with helpful advice and encouraging words. Dr. Celanovic has been both a mentor and a friend, from whom I have learned many important aspects of power electronics modeling techniques. I am truly indebted and thankful to his power electronics and control research group at the Institute for Soldier Nanotechnologies (ISN) for the power electronics applications. I am especially grateful to Jason Poon, Dusan Majstorovic, Nikola Celanovic, Pierre Haessig, Nathan Pallo, and Elaina Chai for their help in the experimental setup and measurements.

I would like to thank my colleagues, Michael Pellauer, Myong Hyon Cho, Mieszko Lis, Richard Uhler, Chris Fletcher, Myron King, Asif Khan, Keun Sup Shim, Rachael Harding, Ilia Lebedev, Victor Costan, Muralidaran Vijayaraghavan, and Abhinav Agarwal, for their valuable input and feedback at various stages of this work. I am grateful to Omer Khan and Marten van Dijk for their help in framing the computational problem. I will forever be thankful to my undergraduate advisor, Professor Sarma Vrudhula, for introducing me to academic research and encouraging me to pursue graduate studies.

Finally, I would like to thank my family; words alone cannot express my full appreciation. I owe sincere and earnest thankfulness to Heather R. (Storace) Kinsy, Michael R. Kinsy, Joseph D. Kinsy, and Gloria (Keller) Spencer for their love, support, and understanding. I would like to thank the Storace, Keller, and Yoder families for providing me a loving and supportive environment. This thesis is dedicated to my sister Mireille Audrey (Kinsy) de Pischof, who, many years ago, set us on this amazing journey to discover the world and ourselves.
Contents

1 Introduction .................................................. 19
1.1 Control Systems ........................................... 19
1.2 Power Electronics Control Systems ....................... 21
1.3 General Presentation of the Problem ..................... 22
1.4 Aims and Objectives ...................................... 24
   1.4.1 Aims .................................................. 24
   1.4.2 Thesis contributions ................................ 25
      1.4.2.1 Contributions to hybrid system modeling ..... 25
      1.4.2.2 Contributions to computer architecture design . 26
      1.4.2.3 Contributions to computer-aided design algorithms . 27
      1.4.2.4 Contributions to design tools and benchmarking . 27
1.5 Outline of the Thesis .................................... 27

2 Literature Review ............................................. 29
2.1 Introduction ................................................. 29
2.2 Formulation of Hybrid Systems ........................... 30
   2.2.1 Hybrid automaton ................................... 30
   2.2.2 Hybrid system modeling of power electronics ...... 31
2.3 Examples of Hybrid System Applications ................ 31
   2.3.1 Power electronics circuits .......................... 31
   2.3.2 Cyber-physical systems (CPS) ....................... 32
2.4 Hybrid System Computer Architectures .................. 32
   2.4.1 Predictable execution ................................ 32
   2.4.2 Predictable memory accesses ....................... 33
   2.4.3 Hard real-time aware routing ....................... 34
2.5 Computer-Aided Design Support .......................... 35
   2.5.1 Task scheduling ..................................... 35
   2.5.2 FPGA-based multicore design ....................... 36
2.6 Summary ..................................................... 37

3 Motivating Application Domain ............................. 39
3.1 Introduction ................................................. 39
3.2 Main Application: Control and Emulation of Power Electronics and Smart Grid Systems ....................... 40
   3.2.1 Brief Introduction to power electronics systems . 40
   3.2.2 Control and emulation of power electronics circuits . 41
3.3 Representative Power Electronics Circuits ................ 42
3.3.1 Wind turbine converter system ........................................ 42
3.3.2 Variable speed induction motor drive ............................... 47
3.3.3 Utility grid connected photovoltaic converter system .......... 48
3.3.4 Hybrid electric vehicle motor drive ................................. 49
3.4 Other Domains of Applications ......................................... 50
3.4.1 Complex embedded systems ........................................... 50
3.4.2 Cyber-physical systems ................................................ 50
3.5 Summary ................................................................. 51

4 Generalized Computation Framework For Hybrid Systems Using Power Electronics 53
4.1 Introduction .............................................................. 53
4.2 Modeling of Powers Electronics Circuits as Adaptive Hybrid Automata (AHA) 54
4.2.1 Formulation ........................................................... 54
4.2.2 Boost converter modeling example .................................. 55
4.3 Programming Model ...................................................... 56
4.4 Task-Based Program Decomposition ................................... 59
4.5 Task-Based Application Execution Model ............................... 60
4.5.1 Computational complexity .......................................... 61
4.5.2 Key desirable characteristics of computational engines for hybrid systems .................................................. 62
4.6 Summary ................................................................. 63

5 Predictable Computation ..................................................... 65
5.1 Introduction .............................................................. 65
5.2 RISC Processing Elements .............................................. 66
5.2.1 Predictable computing through core simplicity .................... 66
5.2.2 Five-stage RISC MIPS processing element ....................... 67
5.2.3 High-performance computing through core composition and heterogeneity .................................................. 69
5.3 Heterogeneous Computing ................................................ 69
5.3.1 Heterogeneous RISC MIPS processing elements .................. 69
5.3.2 MIPS vector execution unit ......................................... 72
5.3.3 Microcontrollers (MCU) and digital signal processors (DSP) .... 75
5.4 Design Approach Overview ............................................. 75
5.5 Summary ................................................................. 78

6 Predictable Memory Access ................................................ 79
6.1 Introduction .............................................................. 79
6.2 Multilevel Memories and their Effect on Access Predictability .................. 80
6.2.1 Working set of a task ............................................... 81
6.2.2 Working set windows ............................................... 82
6.2.3 Cacheless memory organization ................................... 83
6.2.4 Cache-based memory organization ................................ 84
6.2.5 Queue-based memory organization ............................... 85
6.2.6 Scratchpad-based memory organization ......................... 86
6.2.7 Main memory system ............................................... 87
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.8</td>
<td>Hybrid memory organization and management</td>
</tr>
<tr>
<td>6.3</td>
<td>Summary</td>
</tr>
<tr>
<td>7</td>
<td>Quality of Service in On-chip Interconnect</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Typical virtual channel router</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Predictability in virtual channel router</td>
</tr>
<tr>
<td>7.2</td>
<td>HRES Router Architecture</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Router input port micro-architecture</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Switching structures and switch allocations</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Other router architectures considered</td>
</tr>
<tr>
<td>7.3</td>
<td>Routing Algorithm</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Definitions and routing formulation</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Quality of service (QoS)</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Deadlock and livelock</td>
</tr>
<tr>
<td>7.4</td>
<td>Comparative Study of the Routers</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Router configurations</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Area and power estimates</td>
</tr>
<tr>
<td>7.4.3</td>
<td>Throughput and latency estimates</td>
</tr>
<tr>
<td>7.4.4</td>
<td>Discussion</td>
</tr>
<tr>
<td>7.5</td>
<td>Summary</td>
</tr>
<tr>
<td>8</td>
<td>Automated Compilation of Hybrid Control Applications</td>
</tr>
<tr>
<td>8.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>8.2</td>
<td>Automated Adaptive Hybrid Automaton Extraction</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Mixed-model representation of power electronics circuits</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Power electronics circuit analysis</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Automated adaptive hybrid automaton extraction</td>
</tr>
<tr>
<td>8.2.4</td>
<td>Automated multi-program application extraction</td>
</tr>
<tr>
<td>8.2.5</td>
<td>Automated task decomposition</td>
</tr>
<tr>
<td>8.3</td>
<td>Automated Task Scheduling and Processor Mapping</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Scheduling algorithm: taxonomy</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Definitions</td>
</tr>
<tr>
<td>8.3.3</td>
<td>ILP formulation of tasks-based scheduling</td>
</tr>
<tr>
<td>8.3.4</td>
<td>ILP formulation</td>
</tr>
<tr>
<td>8.3.5</td>
<td>Heuristic task-based scheduling algorithms</td>
</tr>
<tr>
<td>8.3.6</td>
<td>Scheduling algorithms application</td>
</tr>
<tr>
<td>8.3.7</td>
<td>Automated task mapping</td>
</tr>
<tr>
<td>8.4</td>
<td>Summary</td>
</tr>
<tr>
<td>9</td>
<td>Architecture Design Space Exploration Tool</td>
</tr>
<tr>
<td>9.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>9.2</td>
<td>Heracles Hardware System</td>
</tr>
<tr>
<td>9.2.1</td>
<td>System overview</td>
</tr>
<tr>
<td>9.2.2</td>
<td>Processing units</td>
</tr>
<tr>
<td>9.2.2.1</td>
<td>Injector core</td>
</tr>
<tr>
<td>9.2.2.2</td>
<td>Single hardware-threaded MIPS core</td>
</tr>
<tr>
<td>9.2.2.3</td>
<td>Two-way hardware-threaded MIPS core</td>
</tr>
</tbody>
</table>
9.2.2.4 Two-way hardware-threaded MIPS core with migration 127
9.2.2.5 FPGA synthesis data ........................................... 128
9.2.3 Memory system organization ..................................... 128
  9.2.3.1 Main memory configuration ................................ 128
  9.2.3.2 Caching system ............................................. 130
  9.2.3.3 Network interface .......................................... 130
  9.2.3.4 Hardware multithreading and caching ....................... 130
9.2.4 Network-on-Chip (NoC) ........................................... 131
  9.2.4.1 Flow control ............................................. 132
  9.2.4.2 Routing algorithm ......................................... 133
  9.2.4.3 Network topology configuration ......................... 133
9.3 Heracles Programming Models ...................................... 134
  9.3.1 Sequential programming model ............................... 134
  9.3.2 Parallel programming model ................................ 135
9.4 Programming Toolchain ........................................... 138
  9.4.1 Program compilation flow ................................... 138
  9.4.2 Heracles graphical user interface .......................... 139
9.5 Experimental Results ............................................. 140
  9.5.1 Full 2D-mesh systems ....................................... 140
  9.5.2 Evaluation results .......................................... 140
9.6 Summary .......................................................... 143

10 MARTHA family of architectures 145
  10.1 Introduction ...................................................... 145
  10.2 MARTHA-I: 16-bit Microprocessor ............................ 147
    10.2.1 Architecture ............................................. 147
    10.2.2 Hardware complexity and performance estimation ....... 147
  10.3 MARTHA-II: Single MIPS Core ................................ 149
    10.3.1 Architecture ............................................. 149
    10.3.2 Hardware complexity and performance estimation ....... 149
  10.4 MARTHA-III: Vector Machine ................................ 150
    10.4.1 Architecture ............................................. 150
    10.4.2 Hardware complexity and performance estimation ....... 151
  10.5 MARTHA-IV: 2D Mesh of Eight MIPS Cores .................... 152
    10.5.1 Architecture ............................................. 152
    10.5.2 Hardware complexity and performance estimation ....... 152
  10.6 MARTHA-V: Heterogeneous-I ................................ 153
    10.6.1 Architecture ............................................. 154
    10.6.2 Hardware complexity and performance estimation ....... 155
  10.7 MARTHA-VI: Heterogeneous-II ................................ 156
    10.7.1 Architecture ............................................. 156
    10.7.2 Hardware complexity and performance estimation ....... 157
  10.8 Summary ........................................................ 159
### 11 Results and Analysis

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1 Introduction</td>
<td>161</td>
</tr>
<tr>
<td>11.2 System Evaluations</td>
<td>162</td>
</tr>
<tr>
<td>11.2.1 MARTHA architectures</td>
<td>162</td>
</tr>
<tr>
<td>11.2.2 General-purpose central processing unit (CPU)</td>
<td>162</td>
</tr>
<tr>
<td>11.2.3 System-step latency comparisons</td>
<td>163</td>
</tr>
<tr>
<td>11.3 Other High-Performance Computer Systems</td>
<td>165</td>
</tr>
<tr>
<td>11.3.1 Evaluation of the IBM cell processor for hybrid control applications</td>
<td>166</td>
</tr>
<tr>
<td>11.3.2 Nvidia GTX 295 graphics processing unit (GPU)</td>
<td>168</td>
</tr>
<tr>
<td>11.3.3 Texas Instruments (TI) C66</td>
<td>169</td>
</tr>
<tr>
<td>11.4 Physical Deployment of the MARTHA Architecture</td>
<td>170</td>
</tr>
<tr>
<td>11.4.1 Industrial three-phase induction motor control</td>
<td>170</td>
</tr>
<tr>
<td>11.4.2 Emulator current and voltage measurements for the benchmarks</td>
<td>171</td>
</tr>
<tr>
<td>11.4.3 Comparing MARTHA to commercially available power electronics emulators</td>
<td>171</td>
</tr>
<tr>
<td>11.5 Discussions</td>
<td>172</td>
</tr>
<tr>
<td>11.6 Summary</td>
<td>176</td>
</tr>
</tbody>
</table>

### 12 Conclusion

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1 Computation Problem in Hybrid Control Applications</td>
<td>177</td>
</tr>
<tr>
<td>12.2 Thesis Contributions</td>
<td>179</td>
</tr>
<tr>
<td>12.3 Directions for Future Research</td>
<td>181</td>
</tr>
</tbody>
</table>
List of Figures

1-1 Components in a typical control system. ........................................... 20
1-2 Illustration of power electronics applications. ............................... 21
1-3 System affinity per architecture type for hybrid systems. ................. 23
1-4 Computation hardware domains. ...................................................... 25
1-5 Proposed design flow for hybrid systems-power electronics. .............. 26

3-1 Illustration of a hybrid control states using a boost converter circuit. .. 40
3-2 Basic power electronics system. ...................................................... 41
3-3 Basic boost converter circuit. ......................................................... 42
3-4 Basic buck converter circuit. ......................................................... 42
3-5 Basic control modules. ................................................................. 43
3-6 Single-chip solution of the control. ............................................... 44
3-7 Hard real-time and best-effort computation threads. .......................... 45
3-8 Wind turbine emulator module. ....................................................... 46
3-9 Emulator module evaluation steps. .................................................. 47
3-10 Module to processor type mapping. ................................................ 48
3-11 Variable speed induction motor drive system. .................................. 48
3-12 Utility grid connected photovoltaic converter system. ........................ 49
3-13 Hybrid electric vehicle motor drive converter system. ...................... 49

4-1 Hybrid modeling of power electronics systems as AHAs. .................... 56
4-2 Discretization of power electronics systems AHAs. ............................ 57
4-3 Hybrid Modeling of the boost converter. ......................................... 57
4-4 Translation of an adaptive hybrid automaton into a multi-program application. 58
4-5 Task-graph of the wind turbine system application. ........................... 61
4-6 Example of a program task-based decomposition. .............................. 62

5-1 Functional units and their service guarantees. ................................ 66
5-2 MIPS instruction formats. .............................................................. 67
5-3 Simple five-stage integer MIPS processor. ....................................... 68
5-4 Seven-stage integer MIPS processor. ............................................... 68
5-5 Seven-stage floating-point MIPS processor. ..................................... 70
5-6 Seven-stage floating-point/integer MIPS processor. ........................... 71
5-7 Seven-stage multiplier MIPS processor. .......................................... 71
5-8 Seven-stage floating-point MIPS processor. ..................................... 72
5-9 Datapath of the vector processing core. ......................................... 74
5-10 Lanes of the vector processing core. ............................................. 74
5-11 Microcontroller datapath. ............................................................. 75
5-12 Total execution time of wind turbine application tasks per processor.  
6-1 Working set window types. ............................................. 83  
6-2 Window flow graph. .................................................... 84  
6-3 Cacheless memory organization. ..................................... 84  
6-4 Cache-based memory organization. .................................. 85  
6-5 Queue-based memory organization. .................................. 85  
6-6 Window flow graph. .................................................... 86  
6-7 Scratchpad-based memory organization. ............................. 87  
6-8 Scratchpad and main memory subsystem namespace partitions.  
6-9 Main memory subsystem management view. .......................... 89  
6-10 Hybrid memory organization. ........................................ 89  
7-1 Typical virtual-channel router architecture. ....................... 92  
7-2 Typical flit organization and the proposed expansion. .......... 94  
7-3 Input port routing logic. .............................................. 94  
7-4 Switching logic of the two types of traffic. ....................... 95  
7-5 Network possible routes configurations. ........................... 96  
7-6 Two-network router. ................................................... 97  
7-7 Single shared crossbar router. ....................................... 97  
7-8 Total cell area utilization per router. .............................. 101  
7-9 Total power consumption per router. ................................ 101  
7-10 Hybrid electric vehicle application throughput and latency results.  
7-11 Latency results for the smart grid and motor drive applications.  
8-1 Computer-aided design flow of the compilation process for power electronics applications.  
8-2 Computer-aided design flow front-end design layers. .......... 108  
8-3 Computer-aided design flow front-end. ............................. 109  
8-4 Boost converter circuit with source and a load. .................. 109  
8-5 Node annotation of the boost converter circuit. .................. 110  
8-6 Partial boost converter netlist. ...................................... 110  
8-7 Direct graph representation of the boost converter circuit. .... 111  
8-8 Boost converter circuit incidence Matrix. .......................... 111  
8-9 System model to adaptive hybrid automaton transformation. .... 112  
8-10 C++ class declaration of circuit object and manipulator functions.  
8-11 Computer-aided design flow task decomposition. ............... 113  
8-12 Scheduling of wind turbine system application. ................ 120  
8-13 Scheduling of hybrid electric vehicle application. ............. 121  
8-14 Scheduling part of computer-aided design flow. ................. 122  
9-1 Heracles-based design flow. .......................................... 125  
9-2 Network node structure. .............................................. 126  
9-3 Possible physical memory configurations. .......................... 128  
9-4 Local memory sub-system structure. ................................ 129  
9-5 Effects of hardware multithreading and caching. ................ 131  
9-6 Virtual channel based router architecture. ........................ 132  
9-7 Network topology examples. .......................................... 134  
9-8 Examples of memory space management for sequential programs.  

14
11-15 Current and voltage measurements for the variable speed induction motor drive system. ............................................. 175
11-16 Current and voltage measurements for the utility grid connected photovoltaic converter system. ............................................. 175
11-17 Current and voltage measurements for the hybrid electric vehicle motor drive system. .......................................................... 175

12-1 Illustration of a unmanned aerial vehicle (UAV) flight modes. ............. 182
## List of Tables

5.1 Register space partitioning and instruction formats ........................................ 73
5.2 Wind turbine task name with abbreviations ....................................................... 77

7.1 Area comparison of router architectures .......................................................... 96
7.2 Power and clocking speed comparison of router architectures .............................. 98
7.3 Clocking speed of routers .................................................................................... 102

9.1 FPGA resource utilization per core type ............................................................. 128
9.2 FPGA resource utilization per coherence mechanism ........................................... 130
9.3 FPGA resource utilization per router configuration ............................................. 133
9.4 Clocking speed of different router types ............................................................ 133
9.5 2D-mesh system architecture details .................................................................. 141

10.1 *MARTHA-I* FPGA resource utilization ............................................................ 148
10.2 *MARTHA-II* FPGA resource utilization .......................................................... 150
10.3 *MARTHA-III* FPGA resource utilization ......................................................... 151
10.4 *MARTHA-IV* FPGA resource utilization .......................................................... 153
10.5 *MARTHA-V* FPGA resource utilization ............................................................ 155
10.6 *MARTHA-VI* FPGA resource utilization ........................................................... 157

11.1 Comparison of emulation time step for several state-of-the-art Hardware emulators for power electronics ................................................................. 173
Chapter 1

Introduction

This chapter presents an overview of the field of high-performance, hard real-time computation. It provides a compelling motivation to fundamentally rethink the computational models and architectures for next generation embedded systems. Particular emphasis is placed on distributed control systems for future "Intelligent Grid" and renewable energy generation systems. Indeed, the applications for high-performance, hard real-time computation systems go beyond future grid and energy conversion and span a range of applications from small-scale, safety critical, e.g., a pacemaker controller, to large-scale, distributed, e.g., automotive networks, and automated manufacturing systems just to name a few.

1.1 Control Systems

In many physical systems, it is necessary to keep the value of a variable at or near some desired value, called setpoint. The setpoint can be constant, therefore requiring regulation of the variable (e.g., voltage across a capacity), or it can vary, in which case the variable needs tracking (e.g., trajectory of an aircraft). Figure 1-1 depicts a block representation of the components in a typical control system.

The plant denotes the physical object under control. It has a set of input variables, which can be manipulated, through actuators (e.g., control valves, hydraulic actuators), and a set of output variables, some of which are controlled to meet setpoint values. The output variables are measured by sensors (e.g., accelerometers, thermocouples, tachometers). The plant controller generally has three main components: an emulator or plant model, a control unit, and a monitoring unit. The emulator uses a model of the system to help predict future system responses, instabilities, and failures. The control unit carries out the control strategy. It uses the emulator together with input and output measurements of the physical system to deduce the correct control scheme. The monitoring unit implements alarm conditions and backup procedures, and has the interface for human or external systems to interact with the controller. These three units work together to support the controller in its functions:
Figure 1-1: Components in a typical control system.

1. predict the plant output over some defined horizon,

2. calculate control values which will minimize the errors between system outputs and setpoint values,

3. apply the most optimal control strategy to the plant,

4. and repeat 1 through 3 at the next sampling period.

Most control systems are controlling analog processes. The plant as part of the physical world is an analog dynamic system. As such, changes in the system happen in a continuous fashion. The correctness, stability, and accuracy of the overall control system are important properties and they are closely related to the data propagation delay of the closed-loop system, so-called sampling time-step. To avoid truncation errors that can accumulation from step to step, which are the main cause of predicted and physical system states divergence, hard real-time control responses are often required.

In fact, the need to understand the dynamics and instabilities of the early control systems motivated Vannevar Bush to develop the world’s first, albeit mechanical, real-time computation engine in the 1920’s. This is one of the earliest examples of an advanced, reconfigurable, real-time, scientific, computational machine that later came to be known as MIT’s Differential Analyzer [1].

Early controllers were built out of either analog electronic circuits or switches, relays, and timers. But with the introduction of the inexpensive microprocessor, many of these control systems were redesigned to take advantage of the increased computation power that comes with the microprocessor. As a consequence, controllers became more sophisticated, and started including more complex features [2, 3, 4]. That trend has continued to today, where advances in the semiconductor domain and microprocessor technology lead to
more robust and more efficient control systems, e.g., industrial manufacturing, automotive, renewal energy, power grids.

The challenge in controlling these systems stems from the fact they are interacting directly with physical events characterized by dynamic continuous behaviors. Digital computers can only interact or compute continuous phenomena, at discrete frequencies or at discrete intervals of time and integrate over these discrete time intervals. Generic general-purpose architectures have not been able to meet some of the computation constraints required by these continuous, event-driven applications.

We do not address all the computational properties of control systems in this dissertation. We focus on power electronics control systems as an excellent representative of this domain of applications, namely, dynamic, multi-level, distributed control systems. Using the example of power electronics control systems, this thesis examines the computation requirements, in terms of performance, latency, and predictability, of current control systems, in the context of today’s processor technology and design trends to derive a new set of computer architectures for these systems.

1.2 Power Electronics Control Systems

![Figure 1-2: Illustration of power electronics applications.](image)

Today, we are witnessing a revolution that is poised to dramatically change the way our society generates, distributes and uses electrical energy, and energy in general. The push for a green and a sustainable energy future is fueling the drive for: a new smart grid, larger penetration of renewable energy generation, hybrid and electric vehicles, more efficient energy utilization, and better solutions for large scale energy storage. One of the key underlying physical layers of the smart grid is power electronics. Power electronics can broadly be defined as solid state energy conversion technology [5], that enables efficient and fully controllable conversion of electrical power, as shown in Figure 1-2. To understand how ubiquitous power electronics has become, consider a few power converter examples:
(1) Power electronics enables power flow interface of solar photovoltaics with the grid; (2) it provides efficient interface between variable speed wind turbines and the grid that enables maximum wind power conversion; (3) it allows for power flow control between electric vehicle motor and battery; and (4) it enables for power grid dynamic stabilization, etc. Power electronics could potentially reduce overall electricity consumption by more than 30% [6].

Power electronics is a solid-state energy conversion technology that enables efficient and agile processing and control of electrical power. The key elements of power electronic circuits are: electronic (solid-state) switches and energy storage elements on one side, and a control subsystem on the other side. Power processing components and controls need to work seamlessly, not only for nominal operating points, but also under faulty conditions, various system disturbances, and over the lifespan of the system [7, 5].

1.3 General Presentation of the Problem

Control systems are hybrid systems; mixtures of real-time continuous dynamics and discrete events [8, 9]. They consist of a network of computational elements interacting with physical inputs and outputs instead of as standalone hardware components. They read inputs from sensors, process the data, then emits actions through actuators in a continuous, latency-critical loop. They also vary wildly both in time and frequency domain. Their demand for high-fidelity models, low response latencies, and hard real-time execution requires a computation environment where programmability, performance and predictability must all be guaranteed.

Advances in microprocessor technology and design have allowed more effective control algorithms, which in turn have led to larger, more complex hardware system. Today, the push in these control systems is toward the single-chip solutions that provide better performance, communication reliability, and system integration over large hardware platforms of standard PCs linked by network cards.

With general-purpose processors (GPP) achieving all these three metrics is hard. Off-the-shelf, general-purpose systems primarily focus on programmability, and provide high average-case performance, through deep pipelines with speculative execution and dynamic dispatch, and multi-level caches and translation lookaside buffers, at the expense of predictability of execution time. Indeed, generic general-purpose architectures even with the use of real-time operating systems (RTOS) fail to meet the hard real-time constraints of hybrid system dynamics.

Field programmable gate arrays (FPGAs) are highly programmable, and have been used in the design of some of these systems. They need to be constantly resynthesized for applications. Very often the software/hardware co-design boundary needs to be redrawn and
debugged from one application to another. This generally leads to several issues pertaining
to module placement, routing, timing and software tools. Most important, performance gets
sacrificed for programmability and predictability in most FPGA-based designs targeting
hybrid system applications.

Application-specific integrated circuits (ASICs) can be used in designing systems-on-chip
(SoCs) with the predictability and the performance required by these control applications.
However, the lack of flexibility and programmability in traditional ASICs has been the
key obstacle to their adoption in most new complex hybrid dynamics applications. These
control algorithms are often expected to change over time, or they may be too complex to
consider hardware-only design.

In the past, a large sub-domain of these applications has been simply treated as em-
bedded systems. As their name indicates, embedded systems are often embedded as parts
of other hardware and mechanical components. And, as such, the emphasis in their design
has been on highly application-specific-closed systems. But over the last two decades, em-
bedded systems have evolved to become complex systems-on-chip (SoCs), in an effort to
provide more flexibility. Their compound annual growth rate has been 9% [10] during that
period. Yet, they lack the electronic design automation (EDA) support and domain-specific
computation engines seen in other domains of applications.

In recent years, there have been some efforts to highlight this lack of generalized method-
ology and hardware platform for hybrid control applications. One of these efforts is the
introduction of the term cyber-physical systems [11, 12, 13] and research conferences exclu-
sively dedicated the domain.

The design of computer systems for control systems in general, and power electronics
control in particular, poses a number of technical challenges different from those encountered
in general-purpose computers. General-purpose computer systems were primarily designed
to process data and not to interact with physical processes which are intrinsically concurrent
and time driven processes [14, 11]. This is the reason these applications have generally fallen
outside the general-purpose computation domain, and have been dealt with, on a case-by-
case basis, via ASICs or complicated, hard to reuse or maintain embedded systems.

Figure 1-3 illustrates the affinity to performance, predictability, and programmability for various architecture types when targeting hybrid system controls. GPPs and ASICs to date do not focus on more than two of these requirements, and the design of hybrid system processors (HSPs) where performance, predictability, and programmability are all first-class design targets, as required by the control systems, still remains an open research problem.

1.4 Aims and Objectives

1.4.1 Aims

The aim of this thesis is to define a general computation model and a suitable computer architecture for hybrid dynamical systems. These control systems model very complex dynamics, and therefore require high-performance computation engines. They interact with physical events, and therefore demand a high-degree of execution predictability, hard real-time, and often low-latency responses for stability. These days, they are part of larger distributed control systems, e.g., power grids, and need to programmable to a certain extent.

Although these systems are deployed in various applications; automotive, automated manufacturing, smart grid, transportation network, biological systems, water flow control, disaster management, eldercare systems, or avionics, a large subset of them obey the same control theory [15, 4]. A very powerful implication is that one can leverage this commonality and develop a domain-specific computation framework for these systems which will lead to a more efficient, reusable, and effective hardware architecture. As it has been shown with digital signal processing (DSP) and graphics processing units (GPUs), domain-specific architectures can yield far better results than general-purpose workarounds or ad-hoc integrated hardware units [16, 17].

Creating a domain-specific architecture template targeted mainly at hybrid system control applications will lead to robust and stable control systems, since their specific performance, predictability, and programmability requirements can be better addressed. Furthermore, this is a very large application domain, and it is getting larger. Figure 1-4 illustrates how the new domain-specific architecture fits into the general universe of computation platform.

The work in this thesis centers around power electronics control systems, since it may not be tractable to address all relevant aspects of control systems and to explore all viable computer architectures for the domain. Even in this narrower field, we cannot give a complete account of all instances of power electronics controls, and we do not even attempt to do this. Instead, we focus on a small representative set of power electronics applications as the vehicle to induce a common system representation, a computational model, and a computer architecture that can be used as a template for control systems in general. The
aim is to make this work a starting point for a new kind of processors where support for continuous and discrete dynamics is made a central architectural feature via predictable execution are hard real-time regimes.

Figure 1-4: Computation hardware domains.

1.4.2 Thesis contributions

To solve the problem described in the previous section, this thesis uses power electronics applications as the foundation to build modeling techniques, synthesis methodologies, and computer architecture candidates for large distributed control systems. Figure 1-5 depicts the design flow process where circuit models combined with control software programs are expressed as adaptive hybrid models, which can be synthesized, mapped, and executed on a single multicore system hardware.

1.4.2.1 Contributions to hybrid system modeling

- Using power electronics control systems, and building on previous and ongoing work on dynamic controls, we define a systematic approach for converting different control design components: schematics, subsystem blocks, dynamic and continuous models, and software modules, into a single system without loss of specification parameters. We represent this mixture of models with a unique modeling abstraction called Adaptive Hybrid Automaton (AHA).

- The most direct implication of this abstraction is the multi-programs application (MPA) computation model it introduces. These applications have complex software/hardware and hardware/IO-interface interactions. Execution modes are interleaved, each with very different computation needs; each mode can be represented as a separate program. The introduction of a generalized computation model (i.e., MPA) for power system applications is new for the domain.
1.4.2.2 Contributions to computer architecture design

- We design a novel parameterizable computer architecture, called Many-core Architecture for Real-Time Hybrid Applications (MARTHA), with time-predictable execution, low computational latency, and high performance that meets the requirements for control, emulation and estimation of next-generation power electronics and smart grid systems. It uses core heterogeneity and computation migration for performance and programmability. It has a hybrid memory organization with different levels of cache, "software disable" caches, and a mix of multi-bank and distributed scratchpad memories for predictable data accesses.

- We develop a two-datapath network-on-chip router that provides predictable and deterministic communication latency for hard real-time data traffic while maintaining high concurrency and throughput for best-effort/general-purpose traffic with mini-
mal hardware overhead. This router requires less area than non-interfering networks, and provides better QoS in terms of predictability and determinism to hard real-time traffic than priority-based routers.

1.4.2.3 Contributions to computer-aided design algorithms

- We create an Integer Linear Programming (ILP) formulation and four heuristics for scheduling a task-based application onto a heterogeneous many-core architecture. Our ILP formulation is able to handle different application performance targets, e.g., low execution time, low memory miss rate, and different architectural features, e.g., Instruction Per Cycle (IPC), cache sizes. For large size problems where the ILP convergence time may be too long, we propose four different cost-function heuristics for mapping which show good performance as calibrated by the ILP formulation. We use these algorithms to map power electronics applications to our proposed architecture.

- We develop a deadlock-free bandwidth sensitive oblivious routing (BSOR) algorithm, and a hard real-time aware version, RT-BSOR, for applications where data traffic have different latency requirements.

1.4.2.4 Contributions to design tools and benchmarking

- We develop Heracles, an open-source, functional, parameterized, synthesizable many-core system design toolkit. Such a many-core design platform is a powerful and versatile tool for exploring future multi/many-core processors using different topologies, routing schemes, processing elements or cores, and memory system organizations. The design exploration of the MARTHA architecture is done in this environment.

- We release a small representative set of power electronics system applications that can be used as hard real-time application benchmarking.

These contributions have allowed us to define, for this domain of applications, a powerful computational model. They have led to the design of a heterogeneous, programmable, many-core architecture able to meet the high computation throughput, deterministic, and low-latency requirements for a spectrum of hybrid dynamical systems (e.g., power electronics, automotive, mechanical systems).

1.5 Outline of the Thesis

The remainder of the thesis is organized in eleven chapters. Chapter 2 reviews prior work in the area of control system and modeling, power electronics control design and hardware, embedded systems and hard real-time, predictable execution support in multicore systems.
and FPGA-based architectures. Chapter 3 frames the challenges facing the design automation and testing of power electronics, and the computational complexity and constraints of these applications. Chapter 4 presents the generalized computation framework used for dynamic hybrid systems. Chapter 5 has the architectural details of the processing units in our candidate multicore system. Chapter 6 describes the memory organization and caching techniques used in the proposed system template. Chapter 7 provides the implementation descriptions of the router and routing algorithms used. Chapter 8 shows the hybrid control application automated compilation process and presents algorithms to efficiently schedule task-based applications onto a heterogeneous many-core architecture. Chapter 9 details the architecture development platform and design environment. Chapter 10 presents different instances of the MARTHA architecture to show design trade-offs, scalability and composability. Chapter 11 describes the experimental results as well as the deployment measures and comparative study of other closely related current multicore systems. Chapter 12 summarizes the thesis.
Chapter 2

Literature Review

The computational framework presented in this dissertation incorporates key concepts from hybrid system formal definition and classification, power electronics system modeling and analysis, and hard real-time aware computer architecture design. The work does not fall entirely within one area of research. Therefore, in this chapter, we do not attempt a comparative study of prior works, but rather, we highlight and summarize important relevant works that inform our hybrid system formation and architecture design decisions.

We survey the general computation landscape of hybrid control systems, with a particular emphasis on power electronics systems. We discuss various methodologies and approaches used for modeling and analyzing hybrid control and power electronics systems. We study applicable computer methodologies and architectures currently in use for emulating and controlling these systems. We examine their performance, in terms of execution predictability and hard real-time support at the different architecture levels, namely, processing element, memory organization, and on-chip interconnect.

2.1 Introduction

It is difficult to find a unique definition of a hybrid system, and consequently a hybrid control system, because different scientific communities are often interested in different aspects of a hybrid system, even though these aspects may overlap. In computer science and computer engineering, the focus is on the verification and correctness of software and hardware interacting with continuous physical environments. In this field, the term embedded system traditionally has been used to characterize hybrid systems. Lately, the term cyber-physical system has been introduced to further generalize the domain [11, 12].

In this work, we adopt the definition that sees a hybrid system as a dynamical system whose evolution depends on a coupling between variables that takes values in a continuum and variables that take values in a finite set [15]. As such, hybrid systems encompass a larger and heterogeneous domain of systems, both in term of scale and performance.
In control theory, the main interest is on the control stability and equilibrium of linear, nonlinear, and dynamical behaviors of these systems [18]. In Mathematical modeling and simulation, the focal point is on the numerical representation that takes into account both continuous and discrete aspects of the system [9]. There have been few attempts to give a more unified mathematical representation, control, and implementation of hybrid systems. Branicky et al. [9] propose a general framework that tries to standardize the notion of a hybrid system. It combines differential equations and automata, and a controller that issues continuous-variable commands and makes logical decisions for the system. Stefan Pettersson and Bengt Lennartson [19] present a general hybrid model, which can be formulated in a unified mathematical description where only the transitions in the continuous and discrete states are considered. They propose hybrid petri nets as a viable candidate for graphic modeling of hybrid systems. Another unified framework [20] is proposed by Chai Wah Wu and Leon O. Chua. It focuses on the synchronization and control of dynamical systems, in particular chaotic systems.

Although the field of hybrid systems is increasingly relevant and has attracted considerable research interest in the past few years, many aspects have yet to be crystallized in a way that gives these systems a more universal representation. A universal representation is one that takes into account the full dynamical behavior of hybrid systems, and lends itself well to efficient and optimal hardware implementations with functional software and hardware adaptivity. Here, we concentrate on the most practical characteristics of these systems, and explore through the modeling and control of power electronics systems concrete hardware support implementations.

### 2.2 Formulation of Hybrid Systems

#### 2.2.1 Hybrid automaton

Branicky et al. [9] introduce a mathematical model of hybrid systems as interacting collections of dynamical systems, evolving on continuous-variable state spaces and subject to continuous controls and discrete transitions. In [8], Henzinger presents a formal definition and classifications of hybrid automata based on their algorithmic behavior. He defines a hybrid automaton as the formal model of a mixed discrete-continuous system. Bemporad et al. [21] prove the equivalence between piecewise affine systems and a broad class of hybrid systems described by interacting linear dynamics, automata, and propositional logic. Alur et al. [22] show that interesting classes of hybrid systems can be abstracted to purely discrete systems while preserving all properties that are definable in temporal logic.

Zhang et al. [23] examine Zeno executions in order to develop efficient tools for modeling, verification, simulation, and design of hybrid systems. In [24], they develop a unified framework for both Zeno and non-Zeno hybrid systems. In [25], Hui et al. discuss some of the
fundamental properties of hybrid dynamical systems. They establish sufficient conditions for uniform stability, uniform asymptotic stability, exponential stability, and instability of an invariant set of hybrid dynamical systems.

Stephen Prajna and Ali Jadbabaie [26] devise a methodology for safety verification of hybrid systems. In their framework, it can be proven that a given hybrid system does not enter an unsafe mode. In [15], Arjan van der Schaft and Johannes M Schumacher give an introduction to hybrid dynamical systems. They present modeling approaches and examples of hybrid dynamical systems. In [27], Goebel et al. focus on the analysis of asymptotic stability in hybrid systems and the design of stabilizing hybrid controllers. Lynch et al. [28] present a modeling framework called Hybrid Input/Output Automaton (HIOA), which includes a notion of external behavior for a hybrid I/O automaton, that captures its discrete and continuous interactions with its environment.

2.2.2 Hybrid system modeling of power electronics

Senesky et al. in [29] present a general model for power electronics circuits using a hybrid systems framework. They describe a hybrid systems perspective of several common tasks in the design and analysis of power electronics, and show scalability of the hybrid approach. Similarly, Geyer et al. [30] show that hybrid system methods can be applied to power electronics control applications. They outline several control approaches for power electronics circuits and systems that are based on hybrid systems and constrained optimal control methodologies.

2.3 Examples of Hybrid System Applications

2.3.1 Power electronics circuits

In [31], Graf et al. present a comprehensive review of current techniques used for real-time simulation of power electronics systems. They indicate that from the simulation point of view synchronous oversampling is desirable but requires $< 5\mu s$ sampling time for systems with the carrier frequency in the order of $20kHz$. They comment that such a low latency is out of reach of today's real time processors. In fact, there are multiple examples of grid-level digital simulator tools based on the off-the shelf high-performance processors and other software and hardware components [32, 33]. These systems can simulate in real-time the power grids' transient and sub-transient behavior, fault conditions, issues concerning the islanding behavior of the parts of the grid, and more, although many of them are constrained to modeling systems with low switching frequency and therefore slow dynamics. However, a majority of power converters operate well into the $kHz$ domain, thereby preventing these emulators from accurately modeling the fast switching behavior and non-linear dynamics.
of these power systems. Other platforms are designed for a single fixed topology [34]. As a consequence, they are impractical to use as a design tool for rapidly prototyping and controlling new power electronics systems.

In power electronics circuits emulation, like most hard real-time applications, tightly bound high-performance systems-on-chip (SoCs) with hard execution guarantees are often preferred [35]. Heterogeneity in these architectures, although it make them harder to program, can provide improved real-time behavior by reducing conflicts among processing elements and tasks, as well as improving programmability and flexibility [36]. Unfortunately, current high-performance processors, employ a number of techniques, namely, out-of-order execution, speculation, simultaneous multithreading, dynamic queues and memory hierarchies, which hinder strong quality-of-service (QoS) guarantees and worse case execution time (WCET) estimations.

2.3.2 Cyber-physical systems (CPS)

In [37], Poovendran presents an overview of CPS, their general characteristics and potential domain of applications. Dillon et al. [38] investigate QoS requirements for CPS and discuss the state of the art in CPS QoS techniques and models. Their two case studies are smart energy grids and intelligent vehicle systems. Rajhans et al. [39], using existing software architecture tools, develop techniques to model physical systems, their interconnections, and the interactions between physical and cyber processes. Similarly, Bhave et al. [40], in their attempt to address the lack of a unifying framework in modeling, analysis, and design of CPS, propose a schema consisting of defining system model relationships at the architectural level, instead of providing a universal modeling language. Due to the high degree of difficulty involved in complete system verification for CPS, Bak et al. [41] advocate and focus their work on an automatic method, based on reachability and time-bounded reachability of hybrid systems, to generate verified sandboxes.

2.4 Hybrid System Computer Architectures

There are several ongoing research efforts with the goal of providing some amount of predictability at different architectural levels in multi-processor systems.

2.4.1 Predictable execution

Bate et al. [42] investigate the usage of modern processors in safety critical applications. Many of these applications require hard real-time and predictable executions. They note the incapacity of past hardware solutions in meeting the performance requirements of new developments. Although they see many advantages with commercial off-the-shelf (COTS) processors when compared to custom-designed processors, they also observe that COTS
processors are not necessarily designed for predictability and their good average-case performance goal may not be appropriate from this domain of applications.

Bryce Cogswell and Zary Segall [43] propose MACS, a predictable architecture for real-time systems. They use a multi-context, shared pipeline processor where task-level parallelism is used to maintain high processor throughput while individual threads execute at a relatively slow, but very predictable, rate. Although this time-multiplexing approach works for some applications, it does not meet the low-latency requirement of future hybrid control systems.

Edwards and Lee argue that temporal behavior is as important as logical function for real-time systems [44]. They see the need for a fundamentally different computer architecture where execution predictability is at the heart of the design. In [45], they present the precision-timed (PRET) architecture. It is based on the SPARC V8 instruction set architecture, has a six-stage pipeline and uses hardware multithreading to eliminate data forwarding and branch prediction. The PRET architecture uses scratchpad memories instead of instruction and data caches.

Anantaraman et al. [46] try to reconcile the complexity and safety trade-offs by decoupling worst-case timing analysis from the processor implementation, through their virtual simple architecture (VISA). The architecture can be dynamically reconfigured from a complex simultaneous multithreading processor to a simple single-issue pipeline. Using application task decomposition, tasks are scheduled, executed, and monitored on the complex pipeline with the option of switching to a simple mode if task deadlines cannot be met in the complex mode. They show that a VISA-compliant complex pipeline consumes 43-61% less power than an explicitly-safe pipeline.

Martin Schoeberl evaluates the issues of current architectures with respect to worst-case execution time (WCET) analysis, and proposes solutions for a time-predictable computer architecture [47]. In T-CREST [48], he expands on these solutions by considering changes to the processing elements, interconnect, and memory organization in multicore and many-core system architectures. Paolieri et al. [49] propose a multicore architecture with shared resources that allows the concurrent execution of hard real-time and non-hard real-time application tasks. The architecture is meant to provide high-performance for the non-hard real-time tasks and the proper time analysis for the hard real-time tasks to meet their deadlines. Similarly, Ungerer et al. [50] set forth in the Merasa project a multicore processor architecture for hard real-time embedded systems and techniques to guarantee the analyzability and timing predictability of every feature provided by the processor.

### 2.4.2 Predictable memory accesses

The need for low-latency memory references with predictable access times in hybrid control or hard real-time applications exacerbates the speed gap between processors and memo-
ries. Many approaches have been proposed to bridge this gap or to reduce its effect while guaranteeing predictable memory access latency. Faced with time-predictable execution requirements in real-time applications and the difficulties associated with the worst-case execution time (WCET) analysis in standard cache organizations, Martin Schoeberl propose splitting the data cache into independent caches for different memory areas [51]. In his scheme, stack allocated data, static data, and heap data are decoupled and mapped onto different caches, with the goal of minimizing caching interferences. Reineke et al. [52] analyze the effects of cache replacement policies on execution predictability.

Lars Wehmeyer and Peter Marwedel [53] compare the impact of scratchpad memories and caches on worst case execution time (WCET) analysis. They highlight the negative impact of dynamic behavior of caches on the predicted WCET. Marwedel et al. [54] use scratchpad memories to ease the processor-memory speed gap. They describe a set of algorithms that can be applied at design time in order to maximally exploit the scratchpad memories. Their results show that using these compilation techniques can reduce the WCET by 48% and the energy consumption by 80%. In [55], they present a compiler-based algorithm that analyzes the application and selects the best set of variables and program parts to be mapped to the scratchpad.

Jack Whitham and Neil Audsley [56] demonstrate the use of trace scratchpad memory to reduce execution times in predictable real-time architectures. They observe that in addition to instruction scratchpads providing excellent WCET improvements with little overhead, trace scratchpads further reduce the WCET by optimizing worst case paths and exploiting instruction-level parallelism across basic block boundaries. In [57], Isabelle Puaut and Christophe Pais present a hybrid memory organization composed of locked caches and scratchpad memories for hard real-time systems. They propose an off-line data to memory type mapping algorithm and show examples of application and architecture instances where one on-chip memory structure is more appropriate than the other.

Another technique used to reduce some of the variations associated with caching in the memory access time, is prefetching. This approach, when feasible, yields better performance results than the complete disabling of caches. Aparicio et al. [58] combine prefetch with cache locking to improve the WCET in multitasking real-time systems. Their results show that a system configuration with prefetching and an instruction cache size that is 5% of the total code size performs better than a system with no prefetching and cache size 80% of the code.

2.4.3 Hard real-time aware routing

At the interconnect network level, Grot et al. [59] propose a new, lightweight topology-aware QoS architecture that provides service guarantees for applications such as consolidated servers on CMPs and real-time SoCs. Shi and Burns [60] also present a method for
evaluating at design time, the schedulability of a traffic-flow set with different quality of service (QoS) requirements in a real-time SoC/NoC communication platform. Their approach uses a priority-based wormhole switching policy and off-line schedulability analysis of traffic-flows. Ethereal [61] is a mesh-based NoC architecture that uses a centralized scheduler and time-division multiplexing to allocate link bandwidth to provide a guaranteed throughput. Authors made the point that with higher average latency, time-division multiplex access is not ideal for high-priority control traffic. Kakoee et al. [62] propose ReliNoC, a network-on-chip architecture that can withstand failures, while maintaining not only basic connectivity, but also quality-of-service support based on packet priorities. This work primarily focuses on QoS in the presence failure. Das et al. present in [63] an application-aware prioritization approach for On-Chip Networks. The main idea is to divide processor execution time into phases, rank applications within a phase based on stall-time criticality, and have all routers in the network prioritize packets based on their applications' ranks. To overcome the lower resource utilization associated with traditional QoS routers, Rijpkema et al. in [64] present a prototype router implementation which combines guaranteed throughput and best-effort routers by sharing resources. They describe the trade-offs between hardware complexity and efficiency of the combined router. D. Wiklund and D. Liu propose a switched NoC for hard real-time embedded systems called SoCBUS [65]. They acknowledged that the SoCBUS architecture is not suitable for general-purpose computation exhibiting certain traffic patterns. The IBM Colony router [66] found in the ASCI White supercomputer uses a hybrid datapath packet routing architecture. The router has three crossbars, and one of those crossbars is used by packets to cut through the router when contention is low to reduce latency.

2.5 Computer-Aided Design Support

There are also research efforts for compiling hard real-time applications onto different hardware architectures. Thomas A. Henzinger and Christoph M. Kirsch [67] propose a framework for a virtual machine, called the Embedded Machine, that mediates in real time the interaction between software processes and physical processes. It has a two-phase compilation process: first, a platform-independent intermediate code, that can be executed by the Embedded, Machine, is generated. This intermediate code is portable and has a predictable timing and output behavior; second, an architecture-dependent executable is obtained after performance and scheduling analysis.

2.5.1 Task scheduling

Application scheduling on heterogeneous resources has been shown to be NP-complete in general cases [68, 69], as well as in several restricted cases [70, 71, 72]. Task scheduling on a
heterogeneous system can be generally classified into static [73] and dynamic [74, 75]. With static assignment, task-to-core mapping is done at compile-time offline or once at the beginning of the application, and the schedule is maintained throughout the execution of the application. Compared with dynamic scheduling, task monitoring and migration are used to ensure a certain level of application performance. While dynamic scheduling can potentially achieve better application performance compared to static scheduling, it may become less feasible in large many-core systems. Brandenburg et al. [76], for example, consider the issue of scalability of the scheduling algorithms on multiprocessor platforms, particularly in the case of real-time workloads. K. Ramamritham and J. A. Stankovic [77] discuss the four paradigms underlying the scheduling approaches in real-time systems, namely, static table-driven scheduling, static priority preemptive scheduling, dynamic planning-based scheduling, and dynamic best-effort scheduling. A. Burns [78] also gives a detailed review on task scheduling in hard real-time systems. H. Topcuoglu and M. Wu [79], present two different scheduling algorithms for a bounded number of heterogeneous processors with an objective to simultaneously meet high performance and fast scheduling time. Their algorithms use rankings and priorities to schedule tasks with the objective of minimizing finish times. Arora et al. [80] present a non-blocking implementation of work-stealing algorithm for user-level thread scheduling in shared-memory multiprocessor system. Chaudhuri et al. [81] provide in-depth formal analysis of the structure of the constraints, and show how to apply that structure in a well-designed ILP formulation such as the scheduling problem. Yi et al. [82] present an integer linear programming formulation for the task mapping and scheduling problem. They use various techniques and architecture characteristics to reduce application execution time. Given an application, Lakshminarayana et al. [83] propose an age-based scheduling algorithm that assigns a thread with a larger remaining execution time to a fast core. Shelepov et al. [84] propose a heterogeneity-aware signature-supported scheduling algorithm that does not rely on dynamic profiling, where they use thread architectural signatures to do the scheduling.

2.5.2 FPGA-based multicore design

In [85] Del Valle et al. present an FPGA-based emulation framework for multiprocessor system-on-chip (MPSoC) architectures. LEON3, a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture, has been used in implementing multiprocessor systems on FPGAs. Andersson et al. [144], for example, use the LEON4FT microprocessor to build their Next Generation Multipurpose Microprocessor (NGMP) architecture, which is prototyped on the Xilinx XC5VFX130T FPGA board. However, the LEON architecture is fairly complex, and it is difficult to instantiate more than two or three on a medium size FPGA. Clack et al. [145] investigate the use of FPGAs as a prototyping platform for developing multicore system applications. They use a Xilinx MicroBlaze
processor for the core, and a bus protocol for the inter-core communication. Some designs focus primarily on the Network-on-chip (NoC). Lusala et al. [86], for example, propose a scalable implementation of NoC on FPGA using a torus topology. Genko et al. [87] also present an FPGA-based flexible emulation environment for exploring different NoC features. A VHDL-based cycle accurate RTL model for evaluating power and performance of NoC architectures is presented in Banerjee et al. [88]. Other designs make use of multiple FPGAs. H-Scale [89], by Saint-Jean et al., is a multi-FPGA based homogeneous SoC, with RISC processors and an asynchronous NoC. The S-Scale version supports a multi-threaded sequential programming model with dedicated communication primitives handled at run-time by a simple operating system.

2.6 Summary

In this chapter, we give a brief overview of hybrid systems formulation. We show some important related works on the modeling of power electronics applications as hybrid systems. The modeling and analysis approaches adopted in this dissertation borrow from these works. We present the most relevant works on predictable computer architectures. Most of these architectures only focus on one architectural area, processing core, memory organization, and on-chip interconnect. In our proposed architecture, predictability runs through the whole architecture stack.
Chapter 3

Motivating Application Domain

In this chapter, we describe some general control applications with a particular emphasis on power electronics control systems. Digital Control systems are everywhere around us these days. There are many reasons for this proliferation, but the four most important ones are performance, economics, safety, and reliability [90, 91, 92]. Adaptive cruise control, air conditioner control, aircraft guidance and navigation, power grid regulation, and assembly line monitoring are all examples of control systems.

3.1 Introduction

In the most general terms, a control system is an entity designed to maintain another system in some desired functional state. Although there are many types of control systems, this thesis focuses on computation models and hardware architecture support for hybrid control systems in general, and power electronics controls in particular.

From a system theory point of view, a hybrid control system is a combination of discrete dynamics and continuous dynamics. Its hardware architecture interacts with both discrete events and continuous physical processes. Indeed, a hybrid control system is a finite state machine describing possible discrete control or physical system states and their transitions, where each state represents some continuous change in the system, best described by a set of differential equations. Figure 3-1 shows an illustration of a boost converter control system.

There is a large body of work on these systems from a pure control theory point of view [93, 25, 92]. It is also true that the embedded systems field has been working on hardware solutions for these systems for as long as digital control has been around. But what is clear today, due to advances in IC technology and the growing complexity and scale of these applications, is that a renewed effort needs to be made in the attempt to provide a generalized computation model and architecture framework for these applications. The usefulness of these systems is directly related to the precision and reliability of the control
functions they can deliver. This relation is at the root of the enormous increase we see in the details of new models. Consequently, this leads to compute-intensive functional blocks. The fact that these systems are controlling physical processes that must obey physical laws both in time and space, further imposes additional design constraints (e.g., hard real-time, deterministic, low-latency) on the architecture. Power electronics control systems constitute one of those applications where there is a network of computational elements interacting with physical inputs and outputs.

3.2 Main Application: Control and Emulation of Power Electronics and Smart Grid Systems

The field of power electronics is concerned with the processing and distribution of electrical power using electronics devices [5, 7]. It is one of the key physical layers of the smart grid that enables highly efficient and fully controllable flow of electric power.

3.2.1 Brief Introduction to power electronics systems

Power electronics is a solid-state energy conversion technology. The key elements of power electronic circuits are: electronic (solid-state) switches, energy storage elements, and control systems. Figure 3-2 shows a basic illustration of power processing. Power electronics systems have changed rapidly over the past 20 years and their application and complexity
has also greatly increased, mainly due to developments in the semiconductor domain and microprocessor technology.

Figure 3-2: Basic power electronics system.

Today, dispersed power systems, including both renewable and non-renewable sources e.g., photovoltaic generators, wind turbines, hydroelectric and gas power station, are being integrated and delivered to consumers. The power converters serve as the interface between the energy sources and the electrical grid. Their presence also introduces a large hybrid control domain.

3.2.2 Control and emulation of power electronics circuits

We will illustrate control in power electronics through two simple and very popular power circuits, namely the boost converter and the buck converter.

A boost converter is a high efficiency direct current (DC) to direct current converter where the circuit output voltage is higher than the input voltage. It has at least two switching elements—a diode and a transistor (MOSFET, IGBT, or BJT)—and some energy storage components—capacitor and inductor. Figure 3-3 depicts the circuit representation of the boost converter. It works by using a control system (PWM in this case) that regulates the switching activities of the transistor which controls the current flow through the inductor. The boost converter system is used in many applications. For example, the Prius NHW20 hybrid uses only 168 cells delivering 202 volts and a boost converter for its 500 Volts motor instead of the 417 cells required [94].

The buck converter is very similar to the boost converter, but it is a set-down converter. The buck circuit output voltage is lower than the input DC voltage. It has a diode, a transistor, an inductor, and a capacitor arranged in slightly different topology. Figure 3-4 depicts the circuit representation of the buck converter. Buck converters are far more efficient in lowering supply voltage (95% or more) compared to linear regulators that lower the voltage by dissipating the excess power. They are widely used. For example, CPT DC 12V to DC 5V 15W converter voltage regulators are used in a car LED power supply.
3.3 Representative Power Electronics Circuits

In this section we will describe four representative power electronics applications, namely, a utility grid connected wind turbine, a variable speed induction motor drive, a utility grid connected photovoltaic converter system, and a hybrid electric vehicle motor drive. We use the wind turbine application to illustrate the control system computational requirements.

3.3.1 Wind turbine converter system

Wind turbines are an emerging renewable energy source. They generate power from the wind by means of moving aerodynamically designed blades and convert wind into rotating mechanical power. In the wind turbine, for example, the control unit must limit the converted mechanical power at higher wind speed, since the power in the wind is a cube of the wind speed. The control strategy may need to limit power delivery to the grid under high wind by doing a stall control, or an active stall, or even a pitch control. To converge to the proper strategy, the control algorithm may need a reference emulator of the physical system to check system responses, interfaces, and failure modes.

Wind turbine systems, like all current large power electronics systems, are multiple-input, multiple-output (MIMO) systems with complex interactions between software and hardware; Figure 3-5 illustrates the basic control modules. Input sensors collect the signals, both analog and digital, coming from the power electronics switching blocks and physical installations. Analog signals are converted into digital signals and they are coupled with
other digital signals to be fed into an *input analyzer*. The input analyzer is generally responsible for separating the various signals into their appropriate types and forwarding them to other modules. The control strategy unit uses a real-time emulator and incoming control targets to devise a new control scheme. The control unit signals may need to be converted before sending them to the actuators. These systems are part of large distributed power systems; through the monitoring and other distributed control units, current local system status can be monitored and coupled with other systems; and new control targets can be initiated.

Due to advances in processor technology and the increasing control complexity, the focus in the design of the hardware system has turn to multi/many-core system-on-chip (SoC) for control, monitoring, and communication with the main grid control center. This new approach promises better performance, communication reliability, and system integration over large hardware platforms of standard PCs linked by network cards [95]. Figure 3-6 shows the single-chip solution for the wind turbine control application.

These SoC architectures need to be deployed in environments that require concurrent execution of several tasks with different performance goals and execution guarantees. For example, in the wind turbine application, some of the tasks, namely, sensor, input analysis unit, control unit, reference emulator, and actuators are in the tight feedback loop directly interacting with physical processes. Therefore, they require deterministic and hard real-time guarantees, where other tasks can be dealt with in a best-effort fashion. Figure 3-6 also shows the data communication classification in the application.
Similar to the data traffic, the computation threads generating these data communications require different service guarantees in terms of performance and execution predictability. Threads are grouped into two service categories in this application: hard real-time and best effort. One of the implications of hard real-time and best effort service distinctions at the thread level is the memory management in providing these services. For example, the emulator needs fast and predictable accesses to its data storage unit to load system mode data, while the monitoring unit needs good average-case data access latencies for measurement recording. Figure 3-7 illustrates the services requirements at the thread level.

The need to decouple hard real-time threads from best effort threads calls for multiple computation contexts. The control modules also demand vastly different computation performances. For instance, the emulator is very compute-intensive and works on large data sets. For control stability and robustness, high-fidelity models are desired. It means that the emulator needs to perform its calculations in close time-step to the physical system to avoid state divergence. Figure 3-8 shows a model example for the wind turbine application.

The wind turbine control system is a hybrid system. As such, it has a set of discrete states, each with its own continuous dynamics. Therefore, to model its representative power electronics circuit, one must analyze all combinations of switching elements, based on external inputs and the internal state of the system during a given discrete time interval. Computation of continuous dynamics is done by integration over discrete time intervals. Thus, one must derive and solve a set of differential equations that describe the circuit topology in that time interval.
During each interval (or time step), four functions must be carried out, namely

- sample of input signals
- derive associate circuit topology (or system state)
- compute the system state after transforming the continuous differential equations into discrete differential equations, and
- emit output signals to the control unit.

Figure 3-9 illustrates these functions and the timeline. Discretization of the system state space model is generally done through matrix manipulations of the form:

\[
\begin{bmatrix}
    x((k+1)T) \\
    y(kT)
\end{bmatrix} =
\begin{bmatrix}
    A_d(T) & B_d(T) \\
    C_d & D_d
\end{bmatrix}
\begin{bmatrix}
    x(kT) \\
    u(kT)
\end{bmatrix}
\]

where \(x\) is the state space vector, \(u\) is the input vector, \(A_d, B_d, C_d,\) and \(D_d\) are system matrices, and \(T\) is the system time step. Using an exact discretization method, the discretized system matrices are given as:

\[
A_d(T) = e^{AT}
\]

\[
B_d(T) = \int_{kT}^{(k+1)T} e^{A[(k+1)T-\tau]} \cdot B \cdot d\tau
\]

under the assumption that \(u(t)\) is piecewise constant during the system time step. Therefore, the equations above become:
outputs

Figure 3-8: Wind turbine emulator module.

\[
A_d(T) = e^{AT}
\]

\[
B_d(T) = [e^{AT} - 1] \cdot A^{-1} \cdot B
\]

which must be calculated for each state.

Finally, it is worth noting that the control unit may be connected to a larger power plant control, and needs to be highly programmable to support different control algorithms and strategies. It is evident from this wind turbine illustrative example that the integration of these different services on the same computing platform requires rethinking of the cores, the memory hierarchy, and the interconnect network. A few direct system design goals can be drawn from this example:

- The SoC needs to be a heterogeneous multicore platform, with targeted support to the different computation demands and effective adaptation to application diversity. Targeted support implies dedicated hardware/processing units; which also calls for mechanisms for shuttling to and from those specified units computation threads.
- The network-on-chip (NoC) used to establish communications between the different
on-chip components needs to support hard real-time and best-effort.

- There needs to be a hybrid memory management scheme. This hybrid approach may consist of scratch-pad memories in addition to general-purpose cacheable memories, so that hard real-time and best effort threads have two separated memory domains and are managed according to their service needs.

- The emulator needs high-performance computation engines to perform fast matrix-vector manipulations.

Based on these goals, a simple mapping of function to processor-memory types yields an SoC with characteristics shown in Figure 3-10.

In power electronics systems, like most other hybrid control systems, the inability of the SoC to properly compute the steps described above and to return control responses in timely fashion can have catastrophic consequences. An unstable electrical power system carries severe injury or death risk to operators, can easily cause power outage, fire or damage to components.

### 3.3.2 Variable speed induction motor drive

Variable speed induction motor drives are used in applications that require variable speed with variable torque, such as, conveyor belt systems, household refrigerator compressors, machine tools, and pump or blower motors. The general power circuit topology of a variable
speed induction motor drive consists of an AC voltage input that is passed through a diode bridge rectifier to produce a DC output. The DC output voltage is then fed into an n-phase inverter sub-circuit to produce an n-phase variable-frequency variable-voltage output, which is filtered and fed into the inductive machine.

3.3.3 Utility grid connected photovoltaic converter system

We present in this subsection a second very important class of power electronics applications; namely, the utility grid connected photovoltaic converter system. This topology transfers
DC power produced by the photovoltaic (PV) array into AC power according to the voltage and power quality requirements of the utility grid. This is done by using a DC-to-DC converter to boost the PV voltage to a level higher than the peak voltage of the utility grid, such that we can use the inverter to generate the AC voltage without using a transformer.

![Utility grid connected photovoltaic converter system](image1)

**Figure 3-12:** Utility grid connected photovoltaic converter system.

### 3.3.4 Hybrid electric vehicle motor drive

Hybrid electric vehicles (HEVs) use two power sources, internal combustion and electric. They are fuel efficient because of their electric motor drive. The motor drive, with a controlled inverter system, is needed to deliver powerful and efficient drive to the electric motor. In general, a HEV motor drive power electronics system can operate in two modes; namely, conventional mode and regenerative breaking mode. In conventional mode, the internal combustion engine supplies the voltage that passes the inverter block to drive the induction motor; whereas in the regenerative breaking mode, the induction machine acts as a generator and the inverter block acts as a three-phase rectifier.

![Hybrid electric vehicle motor drive converter system](image2)

**Figure 3-13:** Hybrid electric vehicle motor drive converter system.
3.4 Other Domains of Applications

Large hardware systems reacting continuously to their environment at or close to the speed of the environment are not just limited to power electronics control systems. In fact, real-time embedded systems are becoming more complex and ubiquitous. They are part of geographically distributed systems, locally distributed systems, and lumped systems.

3.4.1 Complex embedded systems

Transportation systems, financial systems, biological systems, water resource monitoring and control, logistics and disaster management, and eldercare systems are all complex embedded systems.

Industrial computing systems are used to control, automate, power, or monitor many complex industrial processes. They enhance productivity, accuracy, reliability, and safety in assembly and production lines. In these control systems, time-critical services are closely monitored and various data are collected for maintenance, troubleshooting and analysis. In flight control systems where the objective is to force the plane or the missile to achieve the steering commands developed by the guidance system, many hard real-time factors need to be taken into account. In missile control for example, the types of steering commands vary depending on the phase of flight and the characteristics of the interceptor.

In new automotive applications, dynamic stability control (DSC) and engine control systems are coupled with advanced active safety programs, e.g., collision avoidance algorithms, to help prevent the driver and vehicle from a possible accident. Collision avoidance function determines the safe approaching speed to the front car by detecting the relative speed and distance through one or several types of sensors, like video, laser, or ultrasound, etc., and converting in hard real-time these parameters into a collision avoidance strategy. Intelligent transportation systems are using real-time vehicle tracking and transportation networks to mitigate congestions and to provide real-time advice to drivers and authorities. Real-time, high-quality data from multiple sources (including sensors) help to manage and measure transportation infrastructure performance.

3.4.2 Cyber-physical systems

The material in this subsection is included for completeness. Cyber-physical systems (CPS) refer to the integration of computational and physical resources [11, 12]. CPS is an emerging term designed to put new emphasis on the design and analysis of large, complex, and distributed systems where computation hardware engines and physical processes are more tightly conjoined.

The design methodology in CPS considers the physical plant and the control system as one unit. Future embedded systems will be more adaptive, autonomous, reusable, inter-
operable, distributed, and more context-aware. CPS attempts to develop a framework for designing, analyzing, and testing these systems. The aim is to establish new computation paradigms, inter-core communication abstractions, and new software/hardware partitions for these high-performance, complex, physical-computational systems.

3.5 Summary

Using the wind turbine application, we highlighted the design requirements of power electronics controls, and emphasize what they share in common with other dynamic control systems. We described three other power electronics control applications: a hybrid electric vehicle application, a utility grid connected photovoltaic converter system, and a variable speed induction motor drive application.

As shown through the illustrative applications many different activities are required to carry out an effective control strategy in these complex systems. Although in this thesis we are focusing on power electronics control systems, many of the algorithms and design approaches can be applied to other hard real-time applications. For example, in an aerial reconnaissance mission, continuous steaming images need to be processed at a given rate.
Chapter 4

Generalized Computation Framework For Hybrid Systems Using Power Electronics

In this chapter we present a generalized computation framework for dynamic hybrid systems. Building on previous work in the domain of control system and power electronics, we show how to express power circuits as adaptive hybrid automata. We transform these adaptive hybrid automata into multi-program models that can be efficiently executed on a single hardware platform.

4.1 Introduction

A general computation framework for hybrid systems must present a unified model for both continuous and discrete processes so that a single theory of control, stability, and analysis can be applied. The framework needs to be hard real-time-aware, due to fast responses required by these applications. It needs good abstractions to deal with the complexity and scalability issues of future large-scale, distributed control systems applications (e.g., next generation smart grid systems).

Dynamic physical systems are typically mathematically described in one of two domains: time-domain and frequency-domain. In the time-domain, systems are represented as sets of differential equations. For the modeling of these systems, time-domain analysis and design is often preferred over a frequency domain representation, because certain system characteristics may not be possible to represent in the frequency-domain [96].

In Power electronics, modeling algorithms can broadly be divided into two main categories [97, 98]:

- Nodal analysis based (e.g., SPICE), and
• State-space equation based (e.g., Matlab Simulink etc.)

Most modern power electronics simulations tools are based on the second approach, namely, the state space model formulation. The state space modeling approach for power electronics circuits can be further divided into two subcategories[97]:

• fixed matrix, and

• variable matrix modeling approach.

In the fixed matrix approach, all the non-linear elements (e.g., switching elements such as mosfets, diodes etc.) are modeled via nonlinear current-voltage elements which warrant uniform system description, i.e., fixed state-space matrix size. For example, when the switch changes the conduction state from “off” state to “on” state, the equivalent switch impedance changes the value from high impedance to low impedance during the switching time.

On the other hand, in the variable matrix approach, switches are treated as ideal switches with instantaneous transition from one state to another (most commonly with piece-wise linear characteristic). Hence, every combination of the switches in the circuit is described as a unique state-space representation. This leads to a system description where the upper bound on the number of state-space representations is $N = 2^m$, where $m$ is the number of switches. $N$ is the upper-bound since certain circuit configurations may not be feasible due to inherent physical topological constraints. This state-space equation time-domain representation is used in most hybrid systems.

4.2 Modeling of Powers Electronics Circuits as Adaptive Hybrid Automata (AHA)

As described in the previous section, power electronics systems are hybrid systems since they consist of a combination of discrete and continuous features. As such, they are best formalized as hybrid automata (HA) [99, 29]. The general behavior of a hybrid automaton consists of discrete state transitions and continuous evaluation. HA provide a natural modeling formalism for power electronics. This modeling approach allows certain properties of these systems to be verified and for these systems to be automatically synthesized. There are several classes of HA [22, 8, 9]; in our power electronics framework, we use the Adaptive Hybrid Automaton (AHA) model, a generalization of timed automata [28]. Figure 3-1 shows an illustration using a boost converter circuit example.

4.2.1 Formulation

Hybrid automata are a generalization of timed automata, where the changing rate of variables in each state is expressed as a set of differential equations.
Definition 1. We define an adaptive hybrid system to be a 6-tuple $H = (Q, \Lambda, f, E, I, \Phi)$ where: $Q = \{q_1, \ldots, q_k\}$ is the set of discrete states, $\Lambda \subseteq R^k$ is the continuous state space; $f : Q \mapsto (\Lambda \mapsto R^k)$ assigns to every discrete state a continuous vector field on $\Lambda$; $E \subseteq Q \times Q$ is the set of discrete transitions; $I : E \mapsto 2^\Lambda$ assigns each transition $e = (q_i, q_j) \in E$ a guard $\phi_e \in \Phi$.

The adaptive hybrid system model is given in state space form as:

$$\forall q \in Q \quad \dot{\lambda}(t) = A_q \lambda(t) + B_q x(t)$$

where $\lambda(t) \in \Lambda \subseteq R^k$ is the continuous state space vector, $A_q \in R^{k \times k}$ and $B_q \in R^{k \times n}$ are operation matrices; and $x \in R^n$ is the input vector to the system at time $t$. Any discrete state of the system belongs to a finite set $Q = \{q_1, \ldots, q_k\}$ and further defines the given state space representation. Every discrete state $q_i$ therefore has a unique dynamic behavior associated with it that defines the operation of the system.

Definition 2. In this framework we also define a system-mode, denoted $m_q$, where $q \in Q$, to be the operation of the system defined by given state space $\dot{\lambda}(t) = A_q \lambda(t) + B_q x(t)$ and a given $q$.

The state space representation of hybrid automaton modes, as defined in Equation 4.1, can be discretized based on a time-step $\tau$. We use the exact discretization method via state-transition matrix. The discretized state space system matrices, for a given mode are given as:

$$\dot{\lambda}(\tau + 1) = A_{q_i} \lambda(\tau) + B_{q_i} x(\tau)$$  \hspace{1cm} (4.2)

$$y(\tau) = C_{q_i} \lambda(\tau) + D_{q_i} x(\tau)$$  \hspace{1cm} (4.3)

$$\phi(\tau) = C_{\phi_{q_i}} \lambda(\tau) + D_{\phi_{q_i}} x(\tau)$$  \hspace{1cm} (4.4)

where $y$ is the output vector, and $\phi$ is the guard vector. In the discretized form, the set of operation matrices $\{A_{q_i}, B_{q_i}, C_{q_i}, D_{q_i}, C_{\phi_{q_i}}, D_{\phi_{q_i}}\}$ defines the dynamic behavior of the system. The adaptive hybrid model described above can be represented as a block diagram, as shown in Figure 4-1. The equivalent block diagram description of the discretized adaptive hybrid automaton is given in Figure 4-2.

4.2.2 Boost converter modeling example

The proposed adaptive hybrid system approach to power electronics converter modeling relies on piece-wise linear passive elements, piece-wise linear switches, and current and voltage sources (both controlled and independent). A standard boost converter is given in Figure 3-3, comprising an inductor, capacitor, load resistor, DC voltage source, ideal diode,
and an ideal switch. The figure shows the electrical circuit, the high-level switching modes, and some of the dynamics in the application.

Although there are four possible combinations for the switch states, in practical systems, we will only consider the three as shown in Figure 4-3(a), the fourth combination is invalid per power electronics component characteristics. The three combinations of the switches define a set of discrete states \( Q = \{q_0, q_1, q_2\} \), which define three modes \( \{m_0, m_1, m_2\} \).

\( x_c \) is the capacitor voltage, \( V_{\text{in}} \) is input voltage source, \( \phi_{\text{ds}} \) is the digital input gate drive signal controlling the switch \( S_1 \), \( E = \{e_{00}, e_{01}, e_{02}\} \) is the collection of discrete transitions for the given mode \( m_0 \) and \( \Phi = \{\phi_{00}, \phi_{01}, \phi_{02}\} \) is set of predicates associated with a corresponding discrete transition from \( E \).

Mode \( m_0 \), for example, is defined in the following way: \( \dot{x}_c(t) = -\frac{1}{RC}x_c(t) \), \( E = \{e_{00}, e_{01}, e_{02}\} \), with \( \phi_{00} : \phi_{\text{ds}} = 0 \land V_{\text{in}} - x_c > 0 \) (0 is the input to the system from the controller for the IGBT and \( V_{\text{in}} - x_c \) is the internal check of the diode state), \( \phi_{01} : \phi_{\text{ds}} = 0 \land V_{\text{in}} - u_c > 0 \), and \( \phi_{02} : \phi_{\text{ds}} = 1 \land V_{\text{in}} - u_c > 0 \).

4.3 Programming Model

The AHA formulation presented in the previous section provides a generalized modeling approach to dynamic hybrid systems like power electronics. It is able to capture both the dynamic and continuous behaviors of these systems. An adequate programming model needs to retain, leverage, and forward these application characteristics to the rest of programming toolchain. Computer systems perform discrete computation, but the programming model in the case of hybrid systems, needs to also deal with continuous physical quantities. Therefore,
we introduce the notion of \textit{system-step}, the system-level equivalent of the \textit{time-step} in the AHA formulation, and a set of definitions to help describe the model of computation.

\textbf{Definition 3.} For the AHA-based computation model, we define the \textbf{system-step} $\tau$ to be a single complete forward computation in the system. The safe, real-time interval it takes for $\tau$ to associate to a set of inputs, the proper set of outputs, is denoted $\delta$. The $\delta$ of an AHA is the maximum time interval one can use for discretization while still guaranteeing system stability.

Time is a crucial design characteristic of AHA-based hybrid system design. In this framework, the notion of time has three degrees of abstraction:

1. \textit{physical time} $(t)$ based on the physical world, also called hard real-time;

2. \textit{clock time} based on digital time used by processing hardware units;
3. system-step time, also called response latency $\delta$.

The clock time is expressed in function of the physical time, and the system-step time is function of the clock time.

**Definition 4.** Inputs and outputs for an AHA-based computation model may assume different values from one system-step to another. The set of input variables is defined as a function over $\tau$:

$$x(\tau) = \{x_1(\tau), ..., x_n(\tau)\} \quad \text{where} \quad \tau_{\text{initial}} \leq \tau \leq \tau_{\text{infinity}}$$ (4.5)

Similarly, the set of output variables is defined as a function over $\tau$:

$$y(\tau) = \{y_1(\tau), ..., y_m(\tau)\} \quad \text{where} \quad \tau_{\text{initial}} \leq \tau \leq \tau_{\text{infinity}}$$ (4.6)

The set of output variables is related to the set of input variables in the following manner:

$$y_1(\tau) = f_{1i}(x_1(\tau), ..., x_n(\tau))$$

...  

$$y_m(\tau) = f_{mi}(x_1(\tau), ..., x_n(\tau))$$ (4.7)

Where functional set $f_i = \{f_{1i}, ..., f_{mi}\}$ is the program, $p_i$, representing the discretized form of the set of operation matrices, state variables, guard variables, and execution directives for the hybrid system in mode $m_i$. Figure 4-4 illustrates this translation of modes from purely mathematical forms into software programs.

![Figure 4-4: Translation of an adaptive hybrid automaton into a multi-program application.](image)

Program $p_0$ is the master program, and it is active in all modes. It initializes modes, initiates their executions, evaluates transition guards, and orchestrates monitoring conditions.

**Definition 5.** In this framework, we define a Multi-Program Application (MPA) model to be a set of non-related or partially related programs of different complexities, data types,
and execution flows interacting only through a finite set of variables, over different time intervals, to provide a unique high-level application functionality.

**Definition 6.** In the MPA model, a program is stateless if it only associates to each $x(\tau)$ a $y(\tau)$ that is unrelated to any previous $x(\tau')$ or $y(\tau')$. In our illustrating case, programs $p_1$ to $p_k$ are stateless. On the other hand, a program is state-aware if it associates to each $x(\tau)$ a $y(\tau)$ that depends on the current state of the system and values of $x(\tau)$. Program $p_0$, in that sense, is state-aware.

For the MPA model to be functionally correct in accordance to the high-level application specifications, we maintain the invariant that during a *system step*, all state-aware programs in a dependency chain must be executed before the stateless programs.

**Definition 7.** In an AHA-based computation model, there are types of input and output variables, critical and monitor. The critical variables are the ones directly imported from the AHA formulation. Monitoring variables are the ones used to manage mode transitions and to observe system states.

This distinction in the variable types is at the root of the hard real-time and best-effort classification that we will later see at the program execution level.

**Definition 8.** Execution of all program dependency chains involving critical variables must be done under one system-step.

With these definitions, our formulations capture the four key characteristics of the hybrid systems [11]: (1) discrete computation, by introducing *system-step* $\tau$; (2) continuous physical quantities by relating the *system-step* to real time interval $\delta$; (3) concurrent actions by defining the program set $\{p_1, \ldots, p_m\}$; and finally (4) infinite execution by characterizing $\tau$ to be $\tau_{\text{initial}} \leq \tau \leq \tau_{\text{infinity}}$.

### 4.4 Task-Based Program Decomposition

Definition 8 states that all program dependency chains involving critical variables must be executed under one *system-step* which has a fixed safe, real-time interval $\delta$, dictated by the application. In fact, the main challenge in designing hardware platforms for hybrid systems stems from this key requirement.

In order to explore flexible arrangement of the computation modules to meet this requirement, we further decompose the application programs. As a starting point, we define a program as a composition of computation tasks (or simply tasks) providing one general compute service.
Definition 9. A task is a computational primitive that operates on a set of inputs \( I(i_1, i_2, ..., i_n) \), where \( i_\alpha \) is a data or a memory location read by the task. It has a set of outputs, denoted \( O(o_1, o_2, ..., o_m) \), where \( o_\alpha \) is data or memory location written by the task, and an internal working set data \( w \).

Definition 10. There are three elementary tasks: a feedback task, a sequential task, and a parallel task. For generality, they are defined for a given execution time \( t \), but also lend themselves to static application annotation and analysis.

1. At some time \( t \) during a system-step \( \tau \), task \( A \) in program \( p_j \) exhibits a feedback, denoted \( \hat{A} \), if \( I(i_1, i_2, ..., i_n)_A^t \cap O(o_1, o_2, ..., o_m)_A^{t-1} \neq \phi \).

2. At some time \( t \) during a system-step \( \tau \), tasks \( A \) and \( B \) in program \( p_j \) are sequential, denoted \( A \rightarrow B \), if \( O(o_1, o_2, ..., o_m)_A^t \cap I(i_1, i_2, ..., i_n)_B^t \neq \phi \) or \( O(o_1, o_2, ..., o_m)_A^{t-1} \cap I(i_1, i_2, ..., i_n)_B^t \neq \phi \).

3. At all time \( t \) during a system-step \( \tau \), tasks \( A \) and \( B \) in program \( p_j \) are parallel, denoted \( A \parallel B \), if \( O(o_1, o_2, ..., o_m)_A^t \cap I(i_1, i_2, ..., i_n)_B^t = \phi \) and \( O(o_1, o_2, ..., o_p)_B^t \cap I(i_1, i_2, ..., i_q)_A^t = \phi \). One can fuse together any two tasks \( A \) and \( B \), if and only if \( O(o_1, o_2, ..., o_m)_A^t = I(i_1, i_2, ..., i_n)_B^t \) and for all other tasks \( C \), \( O(o_1, o_2, ..., o_m)_A^t \cap I(i_1, i_2, ..., i_n)_C^t = \phi \) and \( O(o_1, o_2, ..., o_m)_A^{t-1} \cap I(i_1, i_2, ..., i_n)_C^t = \phi \). In other words, the intermediate \( A \) to \( B \) state is not observable by any other task.

Task fusion is useful for forcing a set of tasks to be assigned to the same processing engine, particularly when tasks share a large amount of state, and communication costs between processing units are prohibitive or expensive for those tasks.

4.5 Task-Based Application Execution Model

MPA definition allows us to transform a hybrid system application modeled as an AHA into a set of interacting programs. This interaction among programs is dynamic, dictated by the application, and controlled by a single master program which also holds the global state of the system. Task-based decomposition of programs permits a finer-grained modular representation of the high-level hybrid application. A hybrid control application, like the wind turbine application presented in Chapter 3, can be represented as AHA, which can be transformed into an MPA model. The MPA model can be further expanded and decomposed into fine-grained task graph modules with losing the dynamic behavior of the application. Figure 4-5 shows the wind turbine application task graph.
The application task-graph is constructed by first transforming the programs in the MPA model into sub task-graphs. We use task fusion and Tarjan's strongly connected components algorithm [100] to generate equivalent direct acyclic graph (DAG) representations of programs, illustrated in Figures 4-6(a) and 4-6(b). Second, we connect program sub task-graphs to form the final application task-graph. The connectivity of these sub-graphs are directly inferred from the dynamic behavior guard variables of the AHA-based system. Tasks involving critical variables form the critical execution chains of the applications. The execution of any critical chain must complete in one system-step to guarantee application correctness. These critical tasks chains are called hard real-time execution threads. In the wind turbine, Figure 4-5, the edges on those paths are colored in red.

The transformation of MPA models into application task-graphs is primarily done for easier hardware scheduling and to explore more concurrent execution of application modules. During that transformation, tasks are also annotated with meta-data informing on their working set, data types, execution complexity, and execution dynamics. Task characterization details are presented in the compilation chapter (cf. 8).

### 4.5.1 Computational complexity

AHA approach to modeling of control systems like power electronics circuits exhibits deterministic and bounded execution steps. The direct implication of the AHA modeling approach is the multi-programs application (MPA) computation model it introduces. These control applications have a large number of interleaving execution modes. Each mode has very different computation needs and may be represented in a different program. One of the drawbacks of this approach is the exponential growth of the number of AHA modes. Indeed, the number of modes in a power electronics circuit is $2^n$ where $n$ is the number of switching components in the system. This poses a serious challenge in using this approach for complex systems due to the exponential increase in memory resources on one side, and the exponential complexity increase of the finite state machine in program $p_0$ on the other.

In order to alleviate exponential growth of the problem, the system can be partitioned
into sub-systems that communicate via slowly changing state space variables (e.g., capacitor voltage). For example, if a system is partitioned into two sub-systems with $n$ and $m$ switching components respectively, the total number of modes in the system will be $2^n + 2^m$ instead of $2^{n+m}$. Once a system is partitioned, every partition is modeled as an independent Adaptive Hybrid Automaton, while independent AHAs communicate via input-output variables. Task fusion can also be used in reducing the complexity of the overall system.

4.5.2 Key desirable characteristics of computational engines for hybrid systems

Hybrid systems can be very simple or very complex. Power electronics, for example, encompasses a wide range of applications including motor drives with high performance pulse-width modulation (PWM) techniques, solid state transformers, power converters for renewable energy resources and power electronic controls. Therefore, a computational system aimed at this class of applications needs to be highly flexible and programmable. It needs to support design and testing of a simple buck converter to a cascaded multi-level converter. It should be programmable enough where new circuit topologies can be quickly tested, new control strategies can be studied before implementation, existing topologies can be analyzed for normal and fault conditions, and testing can be performed safely and quickly without the risk of harm for personnel or equipment.
Through our AHA-based modeling and MPA representation, a viable computer architecture for these applications must provide the following support:

1. Deterministic, low-latency system responses for system stability;
2. High-throughput, computation intensive hardware, since both continuous and discrete processes may require manipulation of large matrices;
3. Deterministic data access, for predictable computation of hard real-time threads;
4. Programmable hardware modules to support different control algorithms;
5. Ability to interact with other sub-systems and the external environment.

4.6 Summary

Through the *Adaptive Hybrid Automaton* modeling and Multi-Program Application representation, we present a generalized computation framework for dynamic hybrid systems. Through these modeling approaches, these hybrid dynamic applications can be reduced to forms for better hardware support, while they retain dynamic properties. In fact, these properties are transferred down to the hardware to provide better execution, data placement and management.
Chapter 5

Predictable Computation

As presented in Chapter 4, the correctness of the control application firmly depends on the system-step time duration in the discretization process of the state-space model formulation. A small time-step helps keep digitally emulated states of the plant close to the physical dynamics of the systems. This helps to avoid the accumulation of modeling errors from one step to another. These modeling errors are the main cause of emulated and physical system states divergence. Deterministic, ultra-low latency execution is key to high-fidelity, stable control in hybrid applications like power electronics systems.

A target computer architecture must guarantee certain services at the processing core level, at the interconnect level, and finally at the memory level. Figure 5-1 shows examples of those services. In this chapter, we develop a heterogenous many-core processing platform with the predictability and performance power required by these applications. We show the micro-architectural details of the processing elements, their organization, and their target performances.

5.1 Introduction

The design of a computer system to provide the functionalities needed for hybrid systems in general, and power electronics applications in our current case, poses a number of technical challenges different from those encountered in general-purpose computers. These hybrid applications, as previously mentioned, are multiple-input, multiple-output (MIMO) systems with complex interactions between software and hardware, hardware and I/O interfaces, analog and digital signals, real-time controls and digital monitoring.

The design space is larger because the system must be optimized for performance and programmability, but also for predictability. Some of these challenges can be alleviated by leveraging the key physical and dynamic behavioral characteristics in these applications. This is the guiding design principle behind our introduction of the AHA-based modeling with the goal of finding a common computation abstraction for a large subset of these
applications.

As previously stated, one of the main pitfalls of general-purpose or off-the-shelf computer systems is their one-dimensional push for average-case high performance above all else. In the case of hybrid control systems, predictability, performance, and programability are all equally important. Therefore, in order to achieve predictable and fast execution at the processor level, we start with a reduced instruction set computing (RISC) design approach. A RISC architecture trades complexity for the speed of execution, which can also translate into higher performance [101, 102]. Moreover, the simplicity of the architecture and instruction set (ISA) makes predicting program execution time more tractable.

5.2 RISC Processing Elements

5.2.1 Predictable computing through core simplicity

A RISC processor, with its simple, fixed-length instruction format, allows for low-latency, deterministic instruction decoding and execution. Furthermore, memory accesses are performed only through load and store operations and memory addressing modes are fairly simple, therefore providing better static analysis of the program code, compiler optimizations, and more run-time execution behavior predictability. Our processing element design is based on the Microprocessor without Interlocked Pipeline Stages (MIPS) architecture. This RISC architecture is widely used in commercial products and for teaching purposes [101]. Since the MIPS architecture [103, 104] is well known, we will not cover the ISA in its entirety. Instead, we will only give a brief overview and focus on the parts on the ISA that we have modified to fit our proposed architectures.
Figure 5-2 shows the two instruction formatting styles used in the MIPS ISA. The Opcode specifies the operation of the instruction. Fields Src reg 1 and Src reg 2 respectively specify the first and second source operand registers. The Dest reg field is the destination operand register. The Shift amt field is used for shift instructions (e.g., sll). The function field is used to specify additional operations in conjunction with the Opcode. Below is a MIPS binary snippet with the assembly translations:

Binary Address Assembly language representation

<table>
<thead>
<tr>
<th>Binary Address</th>
<th>Assembly language representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>27bdffdo</td>
<td>00000018 addiu sp,sp,-48</td>
</tr>
<tr>
<td>afbf002c</td>
<td>0000001c sw ra,44(sp)</td>
</tr>
<tr>
<td>23be0000</td>
<td>00000020 addi s8,sp,0</td>
</tr>
<tr>
<td>8fc40020</td>
<td>00000024 lw a0,32(s8)</td>
</tr>
<tr>
<td>0c000006</td>
<td>00000028 jal 18 &lt;gcd&gt;</td>
</tr>
<tr>
<td>00000000</td>
<td>0000002c nop</td>
</tr>
<tr>
<td>afc20018</td>
<td>00000030 sw v0,24(s8)</td>
</tr>
<tr>
<td>03e00008</td>
<td>00000034 jr ra</td>
</tr>
</tbody>
</table>

We select this architecture primarily because there is a larger body of open-source, readily available support resources (e.g., cross-compiler support). The general framework and techniques developed in this work can be implemented using other RISC architectures like SPARC, PowerPC, and ARM. In the rest of the chapter, a magic memory is assumed with no cache, main-memory, or off-chip memory distinction made. Issues related to the memory organization are treated in Chapter 6.

5.2.2 Five-stage RISC MIPS processing element

Our design space exploration for a candidate computer architecture for hybrid systems starts with a simple five-stage MIPS RISC processor. Figure 5-3 shows a simplified version of the
datapath of the architecture. Instructions are fetched from memory, decoded, executed, and written back in registers.

![Diagram of a simple five-stage integer MIPS processor.](image)

Figure 5-3: Simple five-stage integer MIPS processor.

The program counter (PC) manages the execution control flow. The program execution is in-order. There is no branch delay slot; the branch prediction is fixed and always non-taken. The advantage of the non-taken branch prediction approach in this architecture is that the processor continues to fetch instructions sequentially, and the pipeline is short enough that the branching penalty is only two cycles. There is no other speculative execution operation, and no translation lookaside buffer (TLB) unit. Although branches and jumps do introduce a certain amount of variation in the execution flow and in the exact cycle count, the architecture is very simple and can be clocked at a high rate. This helps make the execution, at the computational level, predictable in the context of the overall system response time.

![Diagram of a seven-stage integer MIPS processor.](image)

Figure 5-4: Seven-stage integer MIPS processor.

To keep the hardware structure simple and modular, we divide the fetch and memory stages, which interact with the memory system, into multiple, parameterized stages. Although in Figure 5-4, we show these stages split into two, it could be more stages. This approach has two advantages: one, it removes the memory access time from the critical
path and helps decouple the memory access rate from the processor clocking rate, and two, parameterization of memory access stages lends itself well to a hybrid cache or memory organization. Depending on the memory module accessed, data can be assumed ready at different stages.

5.2.3 High-performance computing through core composition and heterogeneity

The five-stage MIPS architecture and its seven-stage extension offer a good balance between execution predictability, with a simple architecture, and low-latency system response potential, with a fast processor type. Both are key requirements for hybrid control system applications. But, as shown in previous chapters, these applications are also very complex and they demand high processing throughput. A requirement that a single, simple, although fast, integer MIPS processor cannot deliver. This is also confirmed by the experimental data in the evaluation chapter 11.

To retain the advantages of the MIPS architecture while increasing execution throughput, a multicore approach is required. Furthermore, our proposed AHA/MPA-based modeling and task decomposition accommodate the multicore or many-core architecture framework well. The parallel processing of tasks increases the computation throughput while lessening the computation complexity of the application through the distribution of tasks on different processing cores.

In hybrid systems like power electronics, modes and their computation can be vastly different in terms of complexity, data types, and working set sizes, because each mode represents a different continuous evolution of the system. The execution of modes must also be contained within one system time duration. Even more important is the fact that the duration must be small enough to avoid control and physical system divergences. In other words, we need a fixed time interval and low latency for all computation modes.

Therefore, we introduce different types of processing elements or cores, each with slightly different processing capability. We then propose to map tasks in modes to different cores based on their computation needs with the goal to maintain a fixed latency execution for all modes. A heterogenous multicore architecture, where processing elements are specialized to different classes of tasks, is more effective in adapting to the application mode computation diversity.

5.3 Heterogeneous Computing

5.3.1 Heterogeneous RISC MIPS processing elements

Continuous state variables in hybrid control systems are continuous quantities, and as such their most faithful representations are with real numbers. Unfortunately, computer systems
can only deal with a finite subset of rational numbers, limited by the number of bits allocated to the representation. Single precision (32 bit), double precision (64 bit) floating-point, or fixed-point representations are commonly used in the computerized modeling and controlling of hybrid dynamic systems.

![Diagram of seven-stage floating-point MIPS processor.](image)

Figure 5-5: Seven-stage floating-point MIPS processor.

Figure 5-5 shows the floating-point version of the MIPS architecture. The MIPS ISA has 32 single precision floating-point registers \(f_0\) to \(f_{31}\). The prototyped MIPS floating-point architecture is a single precision architecture that follows the IEEE-754 standard. The most significant bit represents the sign bit, the mantissa is contained in the 23 least significant bits, radix is two bits, and the exponent is represented using the 8 bits between the sign and the mantissa. Below is a MIPS binary snippet with the assembly translations:

```
Binary Address Assembly Pseudo-instruction
--------------------------
41051eb8 // 00000400 num : .float 8.32
...
3c010100 // 00000000 lui at,0x100
c4220000 // 00000004 lwc1 $f2,0(at) l.s $f2, num

--------------------------
```

Figure 5-6 shows the floating-point/integer dual usage execution unit version of the architecture. The critical path latency has not increased from the architecture shown in Figure 5-5. During the architectural exploration, double precision and fixed-point direct support in hardware are examined for completeness, but their inclusion in the final architecture core types is not deemed necessary.

Another prevalent operation in these applications is multiplication. Although we can use the integer or floating-point execution units to simulate multiplications with add and shift operations, the frequency of multiplications and the low-latency system responses require
a fast, dedicated processing unit. The multiplier architecture, shown in Figure 5-7, also relies on the MIPS ISA instruction formatting. It is a 32x32 bit execution unit, where the result is stored in the $hi$ and $lo$ 32 bit registers. For the hardware implementation we explore different multiplication algorithms in hardware, before converging on the Baugh-Wooley algorithm [105] for its good balance between hardware simplicity and low-latency. The multiplier core has an integer multiplier, a floating-point adder, and a normalizer for the final output product.

Another architectural design challenge for a hybrid control system is the fact that both hard real-time and best-effort computations are tightly integrated. The complete isolation of cores executing hard real-time threads and best-effort threads will lead to a larger, more complex, and very inefficient architecture. Since during any given system-step, only one application mode is active, consequently, only one subset of tasks is being executed. Therefore, we introduced a two-context MIPS core, shown in Figure 5-8; one context for a hard real-time thread and another context for a best-effort computation thread. Each context
consists of a program counter, a status register, and 32 general-purpose registers. When both contexts are present on a core, the best-effort computation thread runs completely in the shadow of the real-time thread with no interference. Due to this non-preemption rule on the hard real-time context and the fact the application can theoretically stay in a mode indefinitely, the core has built-in logic to transfer the best-effort context to another core.

![Seven-stage floating-point MIPS processor.](image)

Best-effort computation core re-assignment and migration are orchestrated by program $p_0$ of the MPA model, also called the master program. Program $p_0$ is essentially the kernel program, in the absence of an operating system. For scalability reasons, $p_0$ tasks may be partitioned across a well-placed set of cores. By spreading these tasks on a large many-core system, one can create more manageable logical system domains (LSD). In an LSD, a $p_0$ task takes the responsibility of managing the nearby slave tasks. Chapter 9 deals with these mappings in more detail.

### 5.3.2 MIPS vector execution unit

State dynamics calculations are done through sets of differential equations in the form of matrices. This is the sort of computation and data operations at which single instruction multiple data (SIMD) architectures excel. For this reason, we explore an architecture under which the SIMD-style execution can be efficiently performed to take advantage of the inherent data parallelism in the state space equations, while providing a flexible programmable computation model. In our current implementation, we develop a vector-processing core that has a RISC-style architecture with vector operands. The instruction set architecture is based on the MIPS scalar ISA extended to support vector operations.

Figure 5-9 shows the datapath of the vector core. In the Decode stage, the registers involved in an execution determine whether the instruction is a scalar or a vector operation. Table 5.1 shows how the register space is partitioned and how instruction decoding is done. The ISA extension is done by manipulating the Opcode and function fields shown in Figure 5-2. The vector register is a 16-entry register file with 128-bit registers, divided into four
32-bit parts for the four lanes, illustrated in Figure 5-10. The number of active lanes is reconfigurable. The scalar register is a 16-entry, 32-bit unit used for scalar or scalar-vector operations. There is a vector length register (VLR) responsible for holding the number of elements in a vector operation. This allows for one instruction to execute an arbitrary number of vector operations.

Table 5.1: Register space partitioning and instruction formats

<table>
<thead>
<tr>
<th>Feature</th>
<th>scalar Module</th>
<th>Vector Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
<td>16 scalar registers numbered from 16 to 31</td>
<td>16 registers, numbered from 0 to 15, per lane</td>
</tr>
<tr>
<td>lw $17, 24($22)</td>
<td>scalar load operation</td>
<td>no vector operation involved</td>
</tr>
<tr>
<td>lw $4, 8($2)</td>
<td>no scalar operation</td>
<td>load register 4 based on address called only involving lane 0</td>
</tr>
<tr>
<td>lw $6, 12($25)</td>
<td>use scalar unit for address calculation</td>
<td>load register 6 based on address calculated for all lanes</td>
</tr>
<tr>
<td>lv $6, 12($25)</td>
<td>use scalar unit for address calculation based on register 25 content</td>
<td>load register 6 for each lanes based on the starting address calculated</td>
</tr>
<tr>
<td>sv $2, 4($21)</td>
<td>use scalar unit for address calculation based on register 21 content</td>
<td>store register 2 for each lanes based on the starting address calculated</td>
</tr>
<tr>
<td>add $17, $19, $23</td>
<td>scalar add operation</td>
<td>no vector operation involved</td>
</tr>
<tr>
<td>multv $4, $2, $3</td>
<td>no scalar operation</td>
<td>vector multiply</td>
</tr>
<tr>
<td>mult $16, $5, $7</td>
<td>take scalar register 16 content</td>
<td>multiple scalar register by register 5 for each lane</td>
</tr>
</tbody>
</table>

Each lane has a multiplication unit and an accumulator unit with bypass logic between the two units. Similarly, the multiplier and the accumulator are chained to other lanes, which allows for the result of one operation to be made immediately available to the next operation without passing through the vector registers. The architecture of the core is primarily optimized for the fast multiply-and-accumulate nature of matrix and matrix-vector multiplications. Many features in the core are designed to efficiently support these types of operations. The Load Unit serves as the main control logic to the memory. It takes care of the addressing of the memory, using a base and a stride, to automatically load.
vector instructions, and to initiate \( N \) loads or stores over multiple memory banks. The most important operation in this functional unit is address generation. The vector stride is simply the number of memory locations separating elements that are merged to form a single vector operation. For conditional operation on the elements of a vector, we use vector masks. The vector mask is loaded with a vector test instruction and the vector operation executed only on the elements whose corresponding entry in the vector mask is set to 1.

The SIMD format provides excellent code density and allows good usage of the Instruction Buffer module. Instructions are prefetched and streamed into the buffer. During static analysis of the program, branch instructions are checked to ensure that branch target instructions are in the buffer. If the program has a branch condition that may violate this requirement, then the buffer is disabled, and instruction fetch goes directly to memory. This enforces the design goal of deterministic execution time.
5.3.3 Microcontrollers (MCU) and digital signal processors (DSP)

Hybrid control systems are directly interacting with physical systems. The transfer of information in and out these control systems is done through various measurement devices, utility metering, sensor devices, motion, temperature, pressure, and actuators devices. Many of these functions do not need a 32-bit processor. Therefore, we introduce 8-bit and 16-bit microcontrollers to the architecture. Their basic datapath is shown in Figure 5-11. In our power electronics applications, the microcontrollers are used for signal generation, e.g., for generating 8-bit or 16-bit sinusoidal signal waveforms, and different I/O signal conditioning functions.

Digital signal processors (DSPs) are essential processing engines for control applications. In this work, we do not implement a DSP architecture. We use field-programmable gate arrays (FPGAs) as the prototype environment, and the FPGAs have built-in DSP building blocks. Analog-to-digital or digital-to-analog signal conversions, with or without filtering, are the connectors of the computer system to the physical plant under control in these applications. This makes DSP modules important to have on chip. Our FPGA-based DSP is built around a finite impulse response (FIR) filter. It is designed using the FPGA DSP blocks which have extra multipliers and pipelines for fast signal processing.

![Figure 5-11: Microcontroller datapath.](image)

5.4 Design Approach Overview

A computer system targeted at hybrid control systems like power electronics applications must have high computational power. It must be able to handle concurrent, hard real-time and best-effort computations. It must have low-latency system response. It must be predictable in its execution and programmable in order to support different control algorithms over time. These goals are best realized through parameterized, heterogeneous, multicore or many-core architectures with direct support for parallel task execution using specialized, predictable, and programmable RISC processing elements.

In this work, we proposed simple and fast processing cores, all based on the same MIPS ISA. The use of a single ISA for all the cores, from the vector core to microcontrollers, in the architecture removes one of the most daunting aspects of heterogeneous computer platforms, namely, the need for different compilers and software support stacks. It also
renders the architecture highly programmable. Figure 5-12 reports the execution time for the tasks in the wind turbine application on different processor types:

- 16-bit microprocessor (Processor1),
- single-cycle MIPS core (Processor2),
- 7-stage single-threaded MIPS core (Processor3),
- 7-stage 2-way threaded MIPS core (Processor4),
- Single lane vector machine (Processor5),
- 2-lane vector machine (Processor6),
- 4-lane vector machine (Processor7),
- 8-lane vector machine (Processor8).

Figure 5-12: Total execution time of wind turbine application tasks per processor.

Table 5.2 shows the full task names and the associated abbreviations used in Figure 5-12. Each task is compiled onto all the processors and the execution times are recorded. The data show an affinity between application tasks and core types. Some tasks are able to run efficiently on less powerful processors with the same execution time, e.g., State Selection (SS). While other tasks, e.g., Previous Distributed Control State (PDC), require certain processing power in order to meet their execution time constraints.
Table 5.2: Wind turbine task name with abbreviations

<table>
<thead>
<tr>
<th>Task full name</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start of System-Step</td>
<td>start</td>
</tr>
<tr>
<td>Input Collector</td>
<td>IC</td>
</tr>
<tr>
<td>State Selection</td>
<td>SS</td>
</tr>
<tr>
<td>Continuous State 1</td>
<td>CS1</td>
</tr>
<tr>
<td>Continuous State 2</td>
<td>CS2</td>
</tr>
<tr>
<td>Continuous State 3</td>
<td>CS3</td>
</tr>
<tr>
<td>Continuous State 4</td>
<td>CS4</td>
</tr>
<tr>
<td>State Variables</td>
<td>SV</td>
</tr>
<tr>
<td>Diversion Load</td>
<td>DL</td>
</tr>
<tr>
<td>Utility Meter</td>
<td>UM</td>
</tr>
<tr>
<td>Wind Turbine Sensors</td>
<td>WT</td>
</tr>
<tr>
<td>Battery Bank</td>
<td>BB</td>
</tr>
<tr>
<td>Rectifier</td>
<td>R</td>
</tr>
<tr>
<td>Inverter</td>
<td>I</td>
</tr>
<tr>
<td>Power Grid Sensors</td>
<td>PG</td>
</tr>
<tr>
<td>Previous State Variables</td>
<td>PSV</td>
</tr>
<tr>
<td>Input Analysis</td>
<td>IA</td>
</tr>
<tr>
<td>Previous Distributed Control State</td>
<td>PDC</td>
</tr>
<tr>
<td>Control Algorithm</td>
<td>CA</td>
</tr>
<tr>
<td>Monitoring Unit</td>
<td>MU</td>
</tr>
<tr>
<td>Output Signals</td>
<td>OS</td>
</tr>
<tr>
<td>Emulator Data Storage</td>
<td>EDS</td>
</tr>
<tr>
<td>Monitoring Unit Data Storage</td>
<td>MUDS</td>
</tr>
<tr>
<td>Distributed Control</td>
<td>DC</td>
</tr>
<tr>
<td>Control System</td>
<td>CS</td>
</tr>
<tr>
<td>Actuators</td>
<td>A</td>
</tr>
<tr>
<td>System Components</td>
<td>SC</td>
</tr>
<tr>
<td>End of System-Step</td>
<td>end</td>
</tr>
</tbody>
</table>

Using the processing cores presented in previous sections, we develop a parameterized architecture called MARTHA (Multicore Architecture for Real-Time Hybrid Applications). The MARTHA architecture features:

- General-purpose MIPS-core(s), used for mode transitions, controls, monitoring, and data collection.

- SIMD vector machine style core(s), used to model linear dynamics of power electronics, with fast, parallel, matrix manipulation operations.

- DSP core(s), used for I/O conditioning, analog/digital communication, and for non-linear machine models.
• Programmable microcontrollers, employed to model certain signal sources.

5.5 Summary

In this chapter we derive from the MPA modeling approach a hardware architecture where performance, programmability, and predictability are all first-class targets. We present a set of simple and fast processing elements that can be integrated into a single on-chip system to enable the efficient execution of the control and emulation of next-generation power electronics and smart grid systems. This parameterizable MARTHA architecture can potentially be used in other hybrid real-time applications. Even further refinement and optimization of the hardware units can be introduced for those applications.
Chapter 6

Predictable Memory Access

The emphasis in these general-purpose, multilevel memory systems is to minimize the average data access time in order to improve overall system performance. In contrast, real-time systems require predictable and repeatable execution time. In the previous chapter, we show how heterogeneous, RISC style, many-core architectures can provide coarse-grain fast parallelism and considerably reduce execution uncertainties in computer systems targeting hybrid control applications. In this chapter, the focus is on the memory organization, and the trade-offs between determinism and performance.

6.1 Introduction

A task’s execution time ($E_{\text{time}}$) on a processing element generally depends on three key factors: the instruction count ($I_{\text{count}}$) in the task’s program, the number of cycles per instruction ($C_{\text{inst}}$), and the cycle time ($C_{\text{time}}$). The number of cycles per instruction ($C_{\text{inst}}$) can be further expanded into the number of processor cycles per instruction ($C_{\text{inst}}(p)$), the number of memory references per instruction ($R_{\text{inst}}$), and number of cycles per reference ($C_{\text{ref}}$).

$$E_{\text{time}} = I_{\text{count}} \times C_{\text{inst}} \times C_{\text{time}}$$
$$E_{\text{time}} = I_{\text{count}} \times [C_{\text{inst}}(p) + (R_{\text{inst}} \times C_{\text{ref}})] \times C_{\text{time}}$$

Equation 6.1 shows the full expression of the execution time ($E_{\text{time}}$). The instruction count ($I_{\text{count}}$) depends on the instruction set architecture (ISA) and the compiler. The number of processor cycles per instruction ($C_{\text{inst}}(p)$) is determined by the ISA, the compiler, and the processor micro-architecture. The number of memory references per instruction ($R_{\text{inst}}$) is influenced by the ISA and the compiler. The number of cycles per memory reference ($C_{\text{ref}}$) depends on the system memory organization. The cycle time ($C_{\text{time}}$) is dependent on the processor micro-architecture and the memory technology. Although the
processor cycle time and the memory cycle time, or access time, are two distinctive system parameters, they are generally forced to be equal.

Of the five terms used to express the execution time, the number of cycles per memory reference, \(C_{ref}\), which depends primarily on system memory organization, is the main source of the memory reference latency unpredictability experienced in general-purpose computer systems. The aim of general-purpose systems is not necessarily to minimize the access time of any single memory reference. The goal in these systems is to minimize the average latency (number of cycles) of the \(C_{ref}\), therefore reducing the execution time \(E_{time}\) and increasing system throughput.

In fact, caches in the memory organization hierarchy are introduced based on the observation that general-purpose programs tend to be localized in time and space. Cache structures and cacheline replacement policies are designed to take advantage of those locality properties. Commonly used and nearby data are stored close to the processing units to minimize their access time and, therefore, the total program execution time.

For hard real-time applications, this biasing of the memory access time is a problem, because it makes the access time unpredictable and data locations in memory hierarchy non-deterministic. The unpredictability of the access time makes it difficult to estimate worst-case execution time (WCET) of the program. Often, the safest approach is to assume that every memory access is resolved at the last level of the memory hierarchy with the worse latency. The main challenge in architectures targeted for hybrid control applications is to develop a more balanced approach to the memory organization where determinism and high throughput efficiency obtained through multi-level memory features can coexist.

Future hybrid control systems differ from traditional embedded systems in their scale, complexity, and distributed nature. This is the reason single-chip heterogeneous many-core RISC-based architectures that provide coarse-grain parallelism are more suitable in these control applications. In this chapter, we explore the memory organization that best suits this class of many-core computer architectures. The logical organization of the memory (caches and main memory) embodies the following concepts: the physical distribution (or partition) of the memory, memory addressing scheme (from program view), and data placement and management schemes.

### 6.2 Multilevel Memories and their Effect on Access Predictability

The memory hierarchy in general-purpose multicore architectures is distributed and multilevel. The memory is organized from small-and-fast to large-and-slow. Small but very fast memories that buffer part of the system main memory help bridge the gap between the processor speed and the main memory access time. This approach has proven to be
a good technique for exploiting locality and improving the overall system performance by making the common case faster. The average memory access time $C_{A-ref}$ in such systems is function of the fast memory access time $C_{F-ref}$, the slow memory access time $C_{S-ref}$, and the percentage of times data is found in fast memory ($hit$).

$$C_{A-ref} = (C_{F-ref} \times hit) + (C_{S-ref} \times (1 - hit))$$ (6.2)

The emphasis in general-purpose multilevel memory systems is to minimize the average memory access time $C_{A-ref}$ in order to improve the overall system computation throughput. On the other hand, in hard real-time applications, memory references must have a bounded latency to guarantee predictable system responses. A task’s best-case execution time (BCET) has an associated best-case memory latency ($C_{BCET}$), and a task’s worst-case execution time (WCET) has an associated worse-case memory latency ($C_{WCET}$). $C_{BCET}$ and $C_{WCET}$ serve as the lower and upper bounds for the memory reference latency in the system.

$$C_{F-ref} \leq C_{BCET} \leq C_{ref} \leq C_{WCET} \leq C_{S-ref}$$ (6.3)

For hybrid systems computer architectures, the key goals at the memory level are: (1) to make the access time $C_{ref}$ constant across all memory references in hard real-time tasks, and (2) minimize the memory access time for both hard real-time and best effort computation tasks ($C_{ref} \approx C_{BCET}$).

### 6.2.1 Working set of a task

The key factors in determining the reference access time, under a memory organization, are: the program structure, the data structures in the program, and data (working) sizes. The program structure, in this work, follows the multi-program application (MPA) modeling approach. For the tasks related to the emulation functions, the dominant data structures are matrices and vectors. For the control and monitoring modules, the data structures in those tasks are less structured and can vary. The working set size, particularly for hard real-time tasks in control applications, is crucial in determining memory reference latencies. The working set size determines task data placement and management.

**Definition 11.** The working set for a task $i$, denoted $w_i$, is the collection of all memory locations referenced by the task during any execution mode.

The working set of a task is the smallest collection of data that must be present in memory to assure efficient execution of the task [106]. It includes both shared and private data. This applies to both hard real-time and best-effort computations. For hard real-time tasks, the access to the working set must also be deterministic, and low-latency for control applications. Recall that for hybrid control applications, like power electronics, system
states are evaluated through matrix manipulations of the form (cf. Chapter 3):

$$\begin{bmatrix} x((k + 1)\tau) \\ y(k\tau) \end{bmatrix} = \begin{bmatrix} A_d(\tau) & B_d(\tau) \\ C_d & D_d \end{bmatrix} \begin{bmatrix} x(k\tau) \\ u(k\tau) \end{bmatrix}$$

where $x$ is the state space vector, $u$ is the input vector, $A_d, B_d, C_d,$ and $D_d$ are system matrices, and $\tau$ is the system time step. The size of input and output vectors, in addition to the system matrices sizes, directly define the working set sizes for the tasks participating in the emulation and the direct control functions of the system. For a given hybrid control application, the adaptive hybrid automaton (AHA) model describes the dynamic states of the system, the switching frequencies of those states, and data transfers across states. The MPA transformation takes the AHA data ($x$, $u$, $A_d$, $B_d$, $C_d$, and $D_d$), adds control flow data, and partitions the data across the tasks in emulator and control modules. These tasks are the hard real-time tasks.

For monitor and auxiliary control tasks, which represent best-effort computation of the application, their exact working set sizes are less predictable. These tasks are given lower data placement priority. Simple minimum and maximum working set sizes are assigned to those best-effort tasks for the purpose of scheduling of tasks.

### 6.2.2 Working set windows

Per Definition 11, the working set can be a collection of non-contiguous memory blocks. For this reason, we introduce the notion of a working set window. There are two types of windows, moving windows and statically partitioned windows, shown in Figures 6-1(a) and 6-1(b). These windows are constructed by exploiting application information exposed through the AHA/MPA modeling.

**Definition 12.** A working set sliding window for a task $i$ during a system-step $\tau$, denoted $w_i(\tau, \Delta)$, is the memory block $[m_\tau - \Delta, m_\tau + \Delta]$, also called window size, which contains the current memory reference $m_\tau$. $\Delta$ is the working set window stride.

The window stride $\Delta$ is determined at task code compilation time. At each system-step, for all active tasks in the sliding window mode, memory operations are automatically initiated to maintain this window. The same memory range may appear across multiple windows.

**Definition 13.** In a statically partitioned window system, for a task $i$, memory blocks are numbered 1 through $n$. During a system-step $\tau$, the working set window, denoted $w_i(\tau, k)$, is the memory block $k$, which contains the current memory reference $m_\tau$.

The windows are constructed by analyzing the program code for each task, and determining the possible paths through the program. The analysis informs on function call
graphs, variable scope trees, and spatial locality of memory accesses. Another advantage of a RISC architecture is the small number of ways to address the memory. It makes the analysis of the dynamic behavior of the code from the memory standpoint more tractable. In the MIPS ISA, there are only four memory addressing modes:

1. register addressing, used in jump register instruction;
2. PC-relative addressing, used in branch instructions;
3. pseudo-direct addressing, used in jump instruction;
4. and base addressing, used in load and store instructions.

Modes 1 through 3 are used in determining instruction code placement and window construction. The data working set placement and window construction are done using the base addressing mode. Based on a task program code analysis, a window format, moving or static, is adopted, and a window flow graph is constructed. Figure 6-2 shows an illustration of a window flow graph.

For a given task, the working set windows can be of different sizes. The full working set of a task can even constitute a single window. Each window entry in the system has four fields: the starting address of the window, the range or size of the window, the preceding window ID, and the succeeding window ID. This is the general approach adopted in this work for hard real-time and best-effort tasks.

6.2.3 Cacheless memory organization

For the most part, general-purpose programs are written to run without knowledge of the underlying physical memory organization. Hybrid control applications cannot adopt this transparent memory approach due to their timing constraints and performance requirements.
The logical memory organization for hybrid systems needs to guarantee a deterministic access time $C_{ref}$ for all memory references in hard real-time tasks. One way to obtain this determinism is to implement a single-level memory system with fixed access priorities to tasks. Figure 6-3 shows the single-level main memory view. There are two primary implications of this single-level memory organization: one is the fact that all the accesses have close to constant latency, two is the fact that this latency is the worst-case across all the accesses. This second implication is a major disadvantage not just for hard real-time tasks, but also best-effort tasks.

6.2.4 Cache-based memory organization

Another approach is to implement a multilevel memory organization, where caches are used to exploit program localities. Figure 6-4 shows a simple two-level memory organization. This organization provides a good average case access time for best-effort references. It also introduces non-determinism in the latency of hard real-time memory references, due
to cache hits and misses. There are a few ways to address this lack of determinism in the caches.

![Diagram of cache-based memory organization]

Figure 6-4: Cache-based memory organization.

The hit rate for memory reference of task $i$ of working set size $w_i$ in the storage $k$ of size $s_k$ of latency $l_k$ is $h_{(i,k)}$. For $n$ memory references, $h_{(i,k)} = \sum_{j=1}^{n} p_j$ where $p_j$ is the probability of a hit in the storage $k$ on memory reference $j$. For random memory accesses,

$$\forall j \in n, \quad p_j = \begin{cases} 1 & \text{if } s_k \geq w_i \\ \frac{s_k}{w_i} & \text{otherwise} \end{cases}$$

In this formulation, cache cold misses are disregarded. We make the assumption that data placement also initializes caches. The hit rate of the whole run becomes: $0 \leq h_{(i,k)} \leq 1$. For hard real-time tasks, if $h_{(i,k)} < 1$, then the cache structure can be disabled. This is applicable to both instruction and data caches. If $h_{(i,k)} < 1$ and memory references are only locally random but globally deterministic, caches can still be used to buffer windows of the working set. Multiple windows can be stored in cache structures depending on cache and window sizes.

### 6.2.5 Queue-based memory organization

Using caching structures for window buffering may not be feasible. The cache line replacement policy may not align well with the window slicing of the working set and the management of those windows at runtime. For this reason, a queue-based memory organization is developed alongside the cache-based configuration. Figure 6-5 shows this memory arrangement, and Figure 6-6 illustrates the internal architecture of the queue.

![Diagram of queue-based memory organization]

Figure 6-5: Queue-based memory organization.

The lower bits of the address are used to address the whole queue. The queue structure is primarily used to facilitate the window replacement mechanism. The head window is the
next in line to be replaced. The rate of inserting a new item in the queue is dictated by Little’s Law\[107]:

\[ q_s = \alpha \times u_{wi} \]  \hspace{1cm} (6.4)

\[ \alpha = \frac{q_s}{u_w} \]  \hspace{1cm} (6.5)

Where \( q_s \) is the queue size, \( \alpha \) the replacement rate, and \( u_w \) the average window utilization level. The queue size, \( q_s \), is fixed in this scheme.

![Figure 6-6: Window flow graph.](image)

The window \( i \) utilization level, \( u_{wi} \), is defined as:

\[ u_{wi} = \frac{\Delta}{\left( \sum_{j=1}^{n} \Delta_j \right) / n} \]  \hspace{1cm} (6.6)

where \( \Delta \) is the window stride (c.f. Definition 12), and \( \Delta_j \) is the number of memory entries between reference \( j \) and \( j - 1 \). The total number of references on window \( i \) is \( n \). This implies that:

\( \forall j, \quad \Delta_j \leq \Delta \). In general, small \( \Delta_j \)s lead to a larger utilization level, which in turn gives lower window replacement rate. The average window utilization level, \( u_w \), is simply:

\[ u_w = \frac{\sum_{i=1}^{k} u_{wi}}{k} \]  \hspace{1cm} (6.7)

The number of working set window partitions is \( k \). The replacement rate \( \alpha \) is bound by the system threshold rate of \( \alpha_T \). The threshold rate is the maximum rate at which windows can be transferred from main memory to the buffers or caches based on the bandwidth allocated to task \( t \).

### 6.2.6 Scratchpad-based memory organization

The feasibility of streaming of the working set windows to caches or queues, for hard real-time tasks, depends on the deterministic nature of the runtime memory references, and the online bandwidth available to tasks. For any two consecutive memory references, if the target window cannot be determined or placed in the temporary storage under a bounded
latency, then the structure needs to be disabled for all memory accesses.

To mitigate some of the limitations associated with temporary storage disabling, we propose the integration of scratchpad memories to the memory organization. Figure 6-7 shows this memory configuration. The scratchpad memory is an extension of the memory space. It uses a separate namespace from the main memory subsystem. In this work, the most significant bit of the address specifies the namespace. When this bit is 1, accesses related to the address are directed to the scratchpad memory. The main memory subsystem is composed of the main memory and various cache/queue levels.

![Figure 6-7: Scratchpad-based memory organization.](image)

Unlike the main memory subsystem, the scratchpad memory is not-transparent. In other words, data stored and accessed in the scratchpad can be explicitly managed from the task program. Data can be moved from one memory space to another via simple load/store operations. Data movements are non-destructive, copies are independent data, and there is no consistency protocol maintained between the scratchpad and main memories. The two namespaces are disjoint, and no single address can refer to a location in both the scratchpad and main memory subsystem. In this work, scratchpad memories' access time is comparable to caches. There are no additional temporary storages between processing elements and scratchpad memories. Figure 6-8 depicts the namespace partitions. The scratchpad memory is envisioned as on-chip static random-access memories (SRAMs).

Scratchpad memory holds multiple advantages for hard real-time operations. The memory access time is fixed and comparable to first-level caches. The scratchpad is addressed in the same fashion as the main memory. A cache can hold any subset of the main memory data, therefore, it requires tag storage to differentiate one address from another. The scratchpad memory does not need tag storage, therefore, making higher usage of the same caching capacity. For the same capacity, scratchpad is shown to have 34% less area and consumes 40% less power [108]. It also adds more bandwidth to the system.

6.2.7 Main memory system

The main memory features a bank-based or interleaved flat memory system, where the on-chip and off-chip memory organization is fully exposed, and logically programmed before the start of the application. For an implementation of \( k \) independent memory banks, up to \( k \) concurrent memory accesses can be allowed. This increases the overall system memory.
bandwidth by k-fold. This approach provides better parallelism and performance than using a single, large bank main memory structure. Each bank has separate data and address lines, and the implementation of the bank-based memory is also relatively simple. The low-order bits of the address are used to select the appropriate bank, and the higher-order bits are used to address the location within the memory bank.

Part of the application compilation process is to make bank accesses contention-free through judicious scheduling of tasks onto processing units. The interaction between the other storage units or cores and the interleaved main memory structure is managed by the memory load unit. The low-order bits of the address contain information regarding starting bank index, stride, and single or multiple bank accesses; 3-bit, 2-bit and 1-bit respectively for the 8-bank memory structure. The load unit can also perform automatic scatter-gather operations. The scatter operation consists of storing the elements of a window across multiple banks, and the gather operation assembles a working set window by fetching data across different memory banks. Addresses for these operations are generated by adding the base-address and an index offset.

6.2.8 Hybrid memory organization and management

The memory organization that best lends itself to hybrid control applications, like power electronics, is the hybrid memory organization. This organization consists of dynamic random-access memories (DRAMs) for the main memory and SRAMs for caches, buffers, and scratchpad memories. The cache-based configuration of the memory follows the same design parameters as in the general-purpose computation: associativity, cache coherence, and consistency models. Buffers have simple, queue structures. Figure 6-9 shows the
For predictable memory access, the simplest scheme is to disable the caches and queues. Traditional locality properties of caching may not be feasible for these types of applications. In power electronics applications, for example, switching of modes leads to cache thrashing. With this approach, performance degrades but access time is highly deterministic. Caches and buffers can be used if application hard real-time tasks can be analyzed for prefetching. Using working set windows, prefetching allows for instruction and data address blocks to be streamed in and out of caches and buffers. The process involves an extensive static analysis of each task to determine full memory access patterns.

When it is not possible to safely devise a window prefetching strategy, through static analysis of the application task code, scratch-pad memories are used. Instructions and data only appear in these memories if the task program explicitly places them in there.
For example, in power electronics applications, different modes have different working sets. Therefore, part of the mapping and compilation process is the association of mode working sets to memory module types. The hybrid approach to the memory organization leads to a low-latency memory access, deterministic latency access, and higher bandwidth for both best-effort and hard real-time computations. Figure 6-10 illustrates this hybrid memory organization.

6.3 Summary

In conclusion, the physical organization of the memory system, the manner in which it is addressed, how it is searched, and how data are stored all affect the performance, predictability, and programmability of the computer system as a whole. When caches and buffers are used in implicit memory management, working set analysis and window partitioning allow for prefetching of data. This improves both performance and predictability of memory references. In the case that a prefetching mechanism cannot be safely inferred from the memory access patterns, scratchpad memories can be explicitly used for deterministic, low-latency memory accesses. The hybrid memory organization provides hard real-time tasks the required low-latency and deterministic memory access time, while retaining the high average-case performance desirable for best-effort computations.
Chapter 7

Quality of Service in On-chip Interconnect

The increasing complexity of hybrid control systems is accelerating the use of multicore architectures in these systems. This trend gives rise to new problems such as the sharing of the on-chip network resources among hard real-time and normal best-effort data traffic. In this chapter we propose a network-on-chip router that provides predictable and deterministic communication latency for hard real-time data traffic while maintaining high concurrency and throughput for best-effort/general-purpose traffic with minimal hardware overhead. The proposed router requires less area than non-interfering networks, and provides better QoS in terms of predictability and determinism to hard real-time traffic than priority-based routers. We present a deadlock-free algorithm for decoupled routing of the two types of traffic.

7.1 Introduction

With multicore and many-core architectures becoming mainstream computing platforms, they are deployed in many computation environments that require concurrent execution of different tasks. These tasks may require hard real-time and/or normal computation support with certain inter-core communications guarantees. Some data communications may be highly latency-sensitive while others may not. The current trend in system-on-chip (SoC) design is system-level integration of heterogeneous technologies consisting of a large number of processing units such as programmable RISC cores, memory, DSPs, and accelerator function units/ASIC [36, 109]. In fact, many control applications today have both general-purpose and real-time requirements. These hybrid applications SoCs must guarantee: (1) real-time operation reactive to external events, like traditional embedded systems; and (2) high average computing throughput and programmability, like general purpose multicore systems.
Traditionally, SoCs use buses and crossbar structures to establish communications between the different on-chip components. System-level latency and bandwidth constraints lead to a multi-level bus architecture, typically consisting of high-performance, low-latency processor bus, high-bandwidth memory bus, and IO bus (e.g., IBM CoreConnect). Besides the lack of concurrent communication support, bus-based SoC transactions lack latency predictability; one communication can be stalled by another transaction in progress. Crossbar structures, or point-to-point communications in general, are not scalable because they require a large number of wires.

Dally et al [109] argue that an SoC, composed of a number of hardware components: processors, DSPs, memories, peripheral controllers, and custom logic, should be connected by a network that routes packets between them, instead of connecting these modules via dedicated wires. Network-on-chip (NoC) architectures constitute an effective data communication infrastructure, providing both flexible connectivity and scalability [110].

7.1.1 Typical virtual channel router

![Typical virtual-channel router architecture](image)

Figure 7-1: Typical virtual-channel router architecture.

In conventional virtual-channel routers [66], the routing operation takes four steps or phases; namely, routing (RC), virtual-channel allocation (VA), switch allocation (SA), and switch traversal (ST), where each phase corresponds to a pipeline stage in the router. When a head flit (the first flit of a packet) arrives at an input channel, the router stores the flit in the buffer for the allocated virtual channel and determines the next hop for the packet (RC phase). Given the next hop, the router then allocates a virtual channel in the next hop (VA phase). Finally, the flit competes for a switch (SA phase); if the next hop can accept the flit, it moves to the output port (ST phase). Figure 7-1 illustrates such a virtual-channel router (VCR).
7.1.2 Predictability in virtual channel router

Unfortunately, virtual-channel based routing in multicore systems in general, and SoCs in particular, further weakens the notion of predictability and determinism so critical to hard real-time inter-core communications. The fact that various traffic must compete for physical link access leads to non-deterministic data transfer delays. Therefore the main drawback of conventional NoCs is their inadequacy in latency predictability. A few techniques, such as express virtual channels [111], dedicated virtual channels, priority-based NoC routing [60], QoS at the network level [112], and RTOS support for NoC-based architectures [113], have been used to mitigate the lack of deterministic latency guarantees in NoC-based communications. Still, they have not been able to meet the hard real-time constraints required by many distributed real-time applications.

The two main design requirements for the on-chip communication layer in these SoCs are: (1) hard real-time processes have absolute deadlines that must be met, and this includes processing time at the cores and inter-core data transfer latencies; (2) the network-on-chip is a shared resource and it needs to be used by all processes (real-time and best-effort). In this work, we propose a novel network-on-chip router, called the Hard Real-time Support (HRES) router. It provides predictable and deterministic communication latency for real-time data traffic while maintaining high concurrency and throughput for normal/general-purpose traffic with minimal hardware overhead.

7.2 HRES Router Architecture

As described in Section 9.1, a flit routing operation generally takes four steps. Resource sharing conflicts may arise in three of those four stages: at the buffer read and route computation level, at the virtual-channel allocation level, and at the switch arbitration level. The alternative to a multi-stage routing scheme is the bufferless approach. Bufferless routing generally consists of arbitrating between two flits competing for the same physical link at each network router or node. The flit that wins access to the link continues through the network competing for link access at each node. Some type of acknowledgment mechanism is implemented to inform the source node of a successful transmission or a failure. Commonly, time-out or negative acknowledgement schemes are used for the acknowledgment [114]. Although the process as just described generally has lower network traversal latency than the buffered approach, dropping of flits even with acknowledgment mechanism makes data communication through the bufferless datapath less predictable and deterministic, both key desirable characteristics for effective hard real-time communication.

The main challenge is to guarantee the hard deadline of certain packets while promoting a high degree of communication concurrency, optimal bandwidth utilization, as well as predictability, deterministic, and low latency, with no significant area or power increase.
The HRES-router uses a hybrid datapath with an interface identical to the conventional virtual-channel router shown in Figure 7-1. This simplifies the interface verification and allows for quick and seamless integration of the router into existing SoCs with little or no system-level change. Data communications are grouped into two categories: guaranteed latency for hard real-time traffic and best-effort latency for general-purpose traffic. Note that traditional quality of service based on priority can still be used in the case of best-effort latency traffic.

### 7.2.1 Router input port micro-architecture

![Figure 7-2: Typical flit organization and the proposed expansion.](image)

Flit type at the network interface is expanded by a single bit to mark the flit as hard real-time or normal traffic, as shown in Figure 7-2. Flits coming in to the router have two datapaths, one for real-time traffic or guaranteed latency and one for general-purpose (non real-time) traffic or best-effort latency. Just as the virtual-channel (VC) bits are read from the incoming flit for VC allocation, the service bit is read from the flit. If the guaranteed latency bit is set to 1, the flit is not buffered and is directly routed to the real-time crossbar. Figure 7-3 depicts the three-state logic at the input port used to make this decision.

![Figure 7-3: Input port routing logic.](image)

The datapath for general-purpose (best-effort) traffic consists of input buffers storing flits while they are waiting to be forwarded to the next hop. When a non real-time flit is ready to move, the switch connects an input buffer to an appropriate output channel via a series of actions: route computation, virtual-channel at the next hop allocation, and finally switch allocation. These stages are pipelined and stalled in case of resource contention.
7.2.2 Switching structures and switch allocations

There are two crossbars at the router, one for the buffered best-effort traffic and one for the bufferless real-time traffic. Figure 7-4 illustrates switching structures and their locations on the datapath. As previously mentioned, the buffered datapath is left unmodified. The only added logic is on the output port side. In addition for the switch allocator to grant requests, it now also checks that granted requests are not blocked by real-time flits during link traversal. If it is the case, the switch allocator proceeds as though the port request lost switch allocation. This modification is not on the critical path logic and can be done without introducing another cycle in the pipeline or affecting the timing characteristics of the datapath logic. In the Verilog code for the conventional virtual-channel router, a switch allocation stall is done on a VC at a port when request is 1 and grant is 0. In the HRES-router switch allocation stall is done on a VC at a port when request is 1 and grant is 0, or when valid bit for real-time flit assigned to the port is 1.

Hard real-time communications are more predictable in terms of source and destination processing elements participating in the transactions. This is due to the fact that these transactions are event-driven, and event handlers mapping is generally done offline. We use this property of real-time communication to derive the algorithm for determining routes. The details of the algorithm is presented in Section 7.3. In the HRES router, the switch for the bufferless traffic is supplemented with a programmable guaranteed-service selector table. This approach prevents flit dropping and removes the need for acknowledgment (and acknowledgment logic) while constraining route selection minimally.

Once routes are determined, routers are pre-configured by setting the proper selector bits at each router. For a 5-port router, the real-time traffic switch consists of five 5-to-1 multiplexers set up offline and per-application. The guaranteed-service selector table allows multiple concurrent real-time traffic through a router (e.g., Figure 7-5 node E) as long
as there is no sharing of physical links (e.g., Figure 7-5 link AD). Two types of real-time communications are supported in the HRES-router: one-to-one and one-to-many. Many-to-one will undoubtedly introduce potential conflicts. It is worth noting that in this router, one-to-many data communications are automatically supported with no additional logic.

It has been shown that for low to medium network load, significant network power savings, with minimal performance loss, can be had with bufferless routing [114, 115]. Therefore, one can envision disabling the buffered datapath at the routers even in the absence of real-time traffic for power savings under low network traffic applications.

### 7.2.3 Other router architectures considered

One approach to guarantee fixed latencies for the real-time traffic is to have a two-network routing scheme, where the real-time and the normal traffic share no physical link. Figure 7-6 shows such a router. The key disadvantages of this type of router are duplication of wires and logic that lead to more cell area, shown in Table 7.1, and changes in the network interface compared to the conventional router.

A third router architecture considered consists of a single large crossbar, where the switch arbitration logic is modified to give priority to the real-time traffic. Figure 7-7 depicts this router architecture. This approach increases the switch arbitration datapath and adds to the router critical path. The arbiter must serialize real-time and normal traffic requests.

All three routers use the same table-based routing algorithm presented in Section 7.3.

<table>
<thead>
<tr>
<th>Table 7.1: Area comparison of router architectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number ports</td>
</tr>
<tr>
<td>HRES router</td>
</tr>
<tr>
<td>Two-Network router</td>
</tr>
<tr>
<td>Single crossbar router</td>
</tr>
</tbody>
</table>
7.3 Routing Algorithm

Algorithms used to compute routes in network-on-chip (NoC) architectures, generally fall under two categories: *oblivious* and *dynamic* [116]. For the normal traffic using the buffered datapath, any traditional routing algorithm will still work and under the same assumptions. For bufferless, conflict-free, hard real-time traffic, the routing scheme is slightly more restricted. *Oblivious* routing with table-based support for both *minimal* and *non-minimal* routing is more appropriate. Considering application communication characteristics in terms of the real-time data constraints and normal data bandwidth requirements, the routing algorithm establishes the real-time traffic routes statically and offline. The key challenge is to find a fair and an effective tradeoff between load balancing of the network, to avoid premature congestion or lack of forward progress of normal traffic, and low data communication latency for both types of traffic.
Table 7.2: Power and clocking speed comparison of router architectures

<table>
<thead>
<tr>
<th>Router Architecture</th>
<th>Power (mW)</th>
<th>Clocking speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>HRES router</td>
<td>42.67</td>
<td>0.98</td>
</tr>
<tr>
<td>Two-Network router</td>
<td>46.94</td>
<td>0.98</td>
</tr>
<tr>
<td>Single crossbar router</td>
<td>44.11</td>
<td>1.55</td>
</tr>
</tbody>
</table>

7.3.1 Definitions and routing formulation

We first introduce standard definitions of flow networks.

**Definition 14.** A flow \( f_i \) is a data communication/traffic from one processing element-source node \( s_i \) to another processing element-destination node \( t_i \).

The set of packets/flits part of flow \( f_i \) passing through the network link from node \( u \) to node \( v \) consuming bandwidth and/or buffer space during a certain time interval is represented with the real-valued function \( f(u,v) \). We may have multiple flows with the same source and destination.

**Definition 15.** Given a multicore topology, we can derive a network flow \( G(V,E) \), a directed graph, where an edge \( (u,v) \in E \) represents a physical inter-router link and has capacity \( c(u,v) \). The capacities \( c(u,v) \) are the available bandwidths on edges. If \( (u,v) \notin E \), then \( c(u,v) = 0 \).

**Definition 16.** For the purpose of differentiating between real-time flows from normal flows, we denote real-time flow \( i \) from node \( s_i \) to node \( t_i \) as \( f_{i[r]} \). We also introduce \( c_{i[u]}(u,v) \) (the upper bound on flow demand \( f_{i[r]} \)), \( w_{i[u]}(u,v) \) (a real-valued weight function for the edge \( (u,v) \)), and \( k_i \) (the maximum number flow \( f_{i[r]} \) can be divided for routing).

We use the following mixed-integer linear programming (MILP) formulation to compute the set of admissible or feasible routes for the real-time flows.

\[
\text{Minimize} \quad \sum_i \sum_{(u,v) \in E} f_{i[r]}(u,v) \cdot w_{i[r]}(u,v) \tag{7.1}
\]

subject to:

\[
\forall i, \forall (u,v) \in E \quad f_{i[r]}(u,v) \in S_{[r]}(u,v), \quad |S_{[r]}(u,v)| \leq 1 \tag{7.2}
\]

\[
0 \leq \sum_i f_{i[r]}(u,v) \leq c_{i[u]}(u,v) \leq c(u,v) \tag{7.3}
\]

\[
\forall (u,v) \in E \quad \left( \sum_i f_i(u,v) + \sum_i f_{i[r]}(u,v) \right) \leq c(u,v) \tag{7.4}
\]
\[\forall i \ f_i[r] \in K_i[r], \ |K_i[r]| \leq k_i \quad (7.5)\]

\[\forall i, \forall j \in \{1, \ldots, k_i\} \sum_{(u,v) \in E} \frac{f_{(i,j)}[r](u,v)}{f_{(i,j)}[r]} \leq \text{deadline}_i \quad (7.6)\]

The real-valued weights \(w_{[i]}(u,v)\) are selected per application. A uniform weight of 1.0 will try to minimize the hop count. A good heuristic is a weight selection that is the inverse proportion of the number of adjacent nodes. This approach assigns higher weights to edge links, because flows using these edges have less path selection diversity. \(k_i\) allows a real-time flow \(f_i[r]\) to be spread across \(k\) different paths if the application permits. This MILP formulation, which routes real-time traffic in a predictable and deterministic fashion, also makes the splitting of flows more manageable; because it eliminates many of the problem encountered with buffered flow splitting, such as out-of-order packets delivery and destination buffering and rearrangements of packets. \(\text{deadline}_i\) is derived from the application specification, and Equation 7.6 enforces the condition that each sub-flow \(f_{(i,j)}[r]\) of flow \(f_i[r]\) is routed through a path where the deadline can be met. Although the algorithm allows splitting of real-time flows to relax routing constraints, no guarantee is made on the optimality of splits, since the classical splittable flow and unsplittable flow problems are NP-hard [117]. Equation 7.2 simply constrains the link \((u,v)\) to be used by at most one real-time flow. For the routing of normal flows and virtual-channel allocation, any traditional routing algorithm (oblivious or adaptive) and VC allocation scheme (static or dynamic) can be used.

### 7.3.2 Quality of service (QoS)

Quality of service is the method for guaranteeing that bandwidth and buffer utilization by the various flows in the network is done in a matter that promotes throughput fairness and latency fairness. QoS in this work needs to be ensured at the flow type and among flows of the same type. Real-time traffic is contention-free, therefore to enforce throughput fairness and avoid starvation of normal flows, the variable \(c_{[i]}(u,v)\) is used to ensure a minimum bandwidth availability to normal flows. The flow splitting property of the routing algorithm allows real-time flows that violate the minimal bandwidth threshold to be split. Due to the deterministic nature of real-time flow paths, any adverse effect of the splitting, e.g., out-of-order packets/flits, can be resolved through sender-receiver synchronization. Latency fairness is partially controlled by the application. In some cases, deterministic latency, especially for real-time flows, may be more important than low latency. The real-valued weights \(w_{[i]}(u,v)\) are used to control the latency fairness of real-time flows and load-balancing of the network at the routing level. For throughput fairness and latency fairness among buffered traffic, any traditional fairness mechanism can be applied.
7.3.3 Deadlock and livelock

The algorithm for routing real-time traffic is deadlock-free because it assumes bufferless paths and no sharing of a physical link. Similarly, it is livelock-free because there is no runtime misrouting of flow. Deadlock-freedom and livelock-freedom for normal traffic must be independently ensured by the algorithm used to route normal flows.

7.4 Comparative Study of the Routers

The router performance evaluations are done using the power electronics applications described in Chapter 3, namely, the hybrid electric vehicle (HEV) application, the smart grid system application (utility grid connected photovoltaic converter system), and the industrial motor control domain (variable speed induction motor drive).

7.4.1 Router configurations

We construct five different router configurations:

1. a virtual-channel reservation-based QoS (v-QoS) router, where VCs are statically partitioned into two sets, and one set is used by the real-time flows and the other by the best effort flows, VC allocation per packet is done dynamically within a set;

2. a priority-based QoS (p-QoS) router—we assign to the real-time flows the highest priorities, and packets can only be given VCs of the same priority, physical link access is also prioritized;

3. a lossy bufferless (L-Bless) router with acknowledgment;

4. a lossless bufferless (Ls-Bless) router—to void packet loss and retransmission we use a routing table to statically configure routes using the algorithm designed for the HRES-router;

5. and the proposed HRES-router.

For the virtual-channel based routers, we use 2 VCs, 4 VCs, and 8 VCs per port configuration.

7.4.2 Area and power estimates

Area and power estimates are obtained using Synopsys Design Compiler using IBM 45-nm SOI CMOS technology cell library. Typical operating point is 1.0 V, 25 C, and worst-case is 0.9 V, 125 C. Figure 7-8 shows the area utilization per router with different virtual-channel configurations. Overall, a VC-based router occupies significantly more cell area than a
bufferless router. The total cell area of the lossless bufferless router is less than 3% of the area recorded for the 2-virtual-channel VCR. Across different VC configurations, the HRES router takes less than 3% more cell area on average to build.

![Total cell area](image)

**Figure 7-8:** Total cell area utilization per router.

![Power (mW) pre-switching activities](image)

**Figure 7-9:** Total power consumption per router.

This shows that the HRES router, which guarantees predictability, deterministic, and low latency for hard real-time packets while maintaining all the functionalities of the conventional VC router, has negligible area overhead. Figure 7-9 shows the power consumption per router. The power summary does not include the clock tree power. The power consumption per router is inline with the area estimates. The HRES router has comparable power consumption to the conventional VC router. Dynamic power is relatively lower than the cell leakage power because application-based switching activity is not taken into account.

Table 7.3 shows the clocking speed of the various routers per number of virtual channels. The conventional VC router runs the fastest around 6.7GHz, but it has 4 pipeline stages.
On the other hand bufferless routers are single stage and run close to 3.6GHz. The HRES router, which architecturally inherits from both, runs closer to the bufferless router speed.

7.4.3 Throughput and latency estimates

Using the development platform presented in chapter 9, we construct: 2D-mesh, 46-bit flits, 8 VC depth, and variable packet length. We run the power electronics applications on a 16-core system synthesized on the Virtex6 XC6VL75T. We use XY-routing for all routers except Ls-Bless and HRES. Since our benchmarks are control applications we can increase the network traffic by increasing the number of state variables and monitoring variables in the application.

Figures 7-10(a), 7-10(b), and 7-10(c) show the throughput results for the HEV applications for 2 VCs, 4 VCs, and 8 VCs respectively. Other applications show similar trends. In Figures 7-10(d) and 7-11(a) we present the total system step latency per router for the smart grid and HEV applications, and Figure 7-11(b) shows the average per hop latency using the motor drive application. Overall, the HRES-router outperforms other routers in terms of throughput and latency.

7.4.4 Discussion

In hybrid applications, like power electronics, real-time data traffic have different latency requirements. As a result, pure priority-based schemes do not work well, because they treat real-time traffic within a priority class just like normal traffic by providing only best-effort quality of service to them, which does not guarantee conflict-free communication and therefore insufficient. Overall the developed router requires less area than non-interfering networks, and provides better QoS in terms of predictability and determinism to hard real-time traffic than priority-based routers. It also outperforms other routers in terms of throughput and latency.

7.5 Summary

In this chapter, we propose a network-on-chip router that provides predictable and deterministic communication latency for real-time data traffic, while maintaining high concur-
(a) Throughput per router using 2 VCs.

(b) Throughput per router using 4 VCs.

(c) Throughput per router using 8 VCs.

(d) System step latency per router.

Figure 7-10: Hybrid electric vehicle application throughput and latency results.

(a) System step latency per router for the smart grid application.

(b) Average per hop latency per router for the motor drive application.

Figure 7-11: Latency results for the smart grid and motor drive applications.
rency and throughput for normal/general-purpose traffic with minimal hardware overhead. The proposed router requires less area than non-interfering networks, and provides better QoS in terms of predictability and determinism to hard real-time traffic than priority-based routers.
Chapter 8

Automated Compilation of Hybrid Control Applications

The design of computer systems for hybrid applications requires sophisticated analysis tools and detailed design procedures involving functional analysis, application program decomposition, execution time analysis, hardware mapping, and runtime tolerance analysis. In this chapter, we discuss the automation process for mapping power electronics control applications onto instances of the system-on-chip hardware.

8.1 Introduction

The level of automation in the design, prototyping, verification and testing of hybrid control systems is, in many aspects, is in the early stages of development, especially when compared to electronic design automation (EDA) for generic integrated digital circuit design. This general lack of tools impedes faster development cycles and system integration, increases the cost and slows down deployment of these systems, including future smart grid systems.

The growth in complexity has exposed the limitations associated with the ad-hoc, failure-prone, design approach in hybrid control applications. These limitations can be summarized as follows:

- Non-uniform high-level design environment for hybrid control applications, and power electronics applications
- Lack of common system-level representation abstractions for easy computed-aided analysis and system decomposition
- Lack of well defined application process interfaces
- Lack of rapid prototyping tools for both computation hardware and control subsystems
• Inadequate hard real-time control system verification and testing, and poor test coverage against faults and parameter variations.

• Lack of direct hardware support for fast, predictable executions for a large subdomain of hybrid control applications.

For power electronics applications, the design process usually starts with functional specification of the converter. Based on expert knowledge, experience and analytical calculation, one arrives at a concept converter design. This step leads to preliminary circuit design, that includes switching topology, selection of passive elements and decision on the appropriate controller platform and control strategy. Once this stage is done, design tasks are divided into power processing part and control part which are often modeled in separate simulation platforms (circuit and control domain) and simulated to verify first functional, and then performance requirements. The utility grid connected photovoltaic converter system graphical representation shown in Figure 3-12, or the hybrid electric vehicle motor drive converter system in Figure 3-13, are produced at the end of this design stage.

In the past, the power processing and controller subsystem were separately prototyped, tested, and verified. This part of the design is often done using analog, low-voltage emulators that provide some level of confidence towards final integration and complete system testing, due to the lack of low-latency, high-fidelity, hard real-time, large digital hardware platforms. Beside the need for large, many-core, hard real-time aware computer architectures for control during in-the-field operation, the design process, as described, also highlights their need during development and testing of future power systems. High-accuracy, hard real-time emulation of power electronics circuits in Hardware-in-the-Loop (HiL) configurations will mitigate some of the current testing and validation limitations and bridge the gap between software-only simulation or analog low-voltage emulating and real operation conditions, while taking high-power equipments out of the testing environment with minimal loss of fidelity. With this approach, even more complex power electronics circuits can be designed. It will help to drastically reduce accidents during system testing and the risks of discovering an error in the very last stage of in-the-field testing and assembling after fabrication.

In this work, a predictable, heterogeneous, many-core architecture is developed for hybrid control systems in general, and for the emulation and control of power electronics applications, in particular. The dissertation provides a formulation for transforming generic control applications into adaptive hybrid automata (AHA), and an AHA into multi-program applications (MPAs). In this chapter, we focus on the automated process for converting power electronics applications: circuit model schematic, control subsystem blocks, and monitoring specifications into executable programs without loss of the dynamic and continuous behaviors of the high-level application.
Figure 8-1: Computer-aided design flow of the compilation process for power electronics applications.

The proposed computer-aided design (CAD) flow establishes a set of design methodologies to transform a power electronics application specification into levels of abstraction that formally represent the application and naturally lend themselves to a computer execution. Figure 8-1 shows the CAD flow of the compilation process for power electronics applications. Detailed discussions of the various data preparation and analysis methods for the application task programs, as outlined in the CAD flow are presented in following sections.
8.2 Automated Adaptive Hybrid Automaton Extraction

The front-end of the CAD environment comprises three layers: (1) the power electronics modeling layer, (2) formulation and analysis layer, and (3) the synthesis layer. Figure 8-2 illustrates these layers. The first layer consists of the Schematic Editor and model Netlist Extractor. The next two layers outline the circuit modeling and the application program analysis steps.

8.2.1 Mixed-model representation of power electronics circuits

In the Schematic Editor, the system designer represents the power electronics application comprising the emulator, control, and monitor modules, in a multi-domain, model-based design environment. There are few such tools both in the commercial and open-source space. Ptolemy [118], Kepler [119], Polis [120], and Matlab SimPowerSystems are a few examples of mixed-model design environments. For the applications presented in this work, Matlab SimPowerSystems is used for system specification. It provides component libraries and analysis tools for modeling and simulating electrical power systems. The libraries include models of electrical power components, three-phase machines, and electric drives.

The Schematic Editor allows for circuit schematics, control functions, and monitoring software components to be combined in mixed-model, block-based system diagrams. It gives designers the ability to convert their power electronics systems specification into model representations, where they can define component values, connectivity, and characteristics. These components range from linear circuit elements, like resistors, inductors, and capacitors, to machine models, including semiconductor devices, like diode, IGBT, voltage and
current, both independent and controlled sources.

![Diagram of computer-aided design flow front-end.](image)

**Figure 8-3:** Computer-aided design flow front-end.

The library of components has predefined power electronics switching blocks, such as a three-phase diode rectifier block, and control circuit blocks, such as a three-phase carrier-based pulse width modulator (PWM). The component library can also contain user-defined modules, blocks, or control/monitoring software programs. Given a power electronics circuit and a set of control code snippets, written in C++, the *Netlist Extractor* generates a unified system representation with components, their values or code, and connectivity. Figure 8-3 shows this phase of the design processing in the CAD flow.

### 8.2.2 Power electronics circuit analysis

![Diagram of boost converter circuit.](image)

**Figure 8-4:** Boost converter circuit with source and a load.

A commonly used approach in computer-aided analysis of power electronics systems is the matrix-form representation of circuits using graph theory [121, 122, 123]. Recall the boost converter circuit presented in Figure 3-3 (cf. chapter 3), Figure 8-4 shows the boost converter circuit with a source and a resistor load. Given the power electronics circuit, the *Netlist Extractor* generates the associated list of components used, their values, and connectivity. It also performs a node annotation of the circuit, shown in Figure 8-5.
The Netlist Extractor also provides circuit metadata, such as the hierarchical composition and degree of idealization of components. Figure 8-6 shows the partial netlist associated with the boost converter application. The node annotation of the circuit is followed by the directed graph representation where the vertices are the circuit nodes assigned, and the edges are the actual electrical components. Figure 8-7 shows the directed graph associated with the boost converter application. The next stage in the circuit analysis is the conversion of the directed graph into an incidence matrix. Figure 8-8 shows the incidence matrix of the boost converter application.

Figure 8-5: Node annotation of the boost converter circuit.

In the following stages of the circuit analysis, using the matrix representation form, components are grouped into tree and co-tree. A tree of the graph is simply a subset of the edges such that all graph vertices are connected by edges without forming a loop. The remaining edges or components become part of the co-tree. A systematic approach to finding a tree and co-tree is the reduction of the incidence matrix into reduced row-echelon form. In our algorithm, we use Gauss-Jordan elimination with pivoting to obtain our reduced row-
Figure 8-7: Direct graph representation of the boost converter circuit.

Figure 8-8: Boost converter circuit incidence Matrix.

Echelon form. Since the main goal in manipulating the incidence matrix is to facilitate the extraction of Kirchhoff’s Current Law (KCL) and Kirchhoff’s Voltage Law (KVL), certain component organizations into tree and co-tree have been shown to be more desirable. In general, all voltage sources are forced in the tree and all current sources in the co-tree. More specifically, the circuit elements are organized in the following order: voltage sources, capacitors, resistors, inductors, and current sources.

8.2.3 Automated adaptive hybrid automaton extraction

The AHA model formulation is done at the second layer, which constitutes the first part of the Application Synthesizer design software module. Here, the set of matrices representing the discrete states of the power electronics systems, transition equations, control functions, and output signals are generated. This is the pure mathematical representation of the application. The output files at this stage can be fed to an offline simulator for soft simulation of the system. System-level decompositions are done at this stage. Large, complex circuits are partitioned into subcircuits that communicate via slowly changing state space variables (e.g., capacitor voltage) and annotated.

Figure 8-9 shows the adaptive hybrid automaton form of power electronics control model at the end of the circuit analysis. The mathematical framework for the AHA form transformation is presented in chapter 4. Figure 8-10 shows C++ class responsible for the circuit analysis and the AHA extraction. Given a netlist file (e.g., boost-converter.txt, an ECircuit class object is instantiated with the function call: ECircuit boost = ECir-
Figure 8-9: System model to adaptive hybrid automaton transformation.

cuit("boost_converter.txt"). With the execution of boost.extract_circuit() a circuit object is created and all the appropriate fields are populated. When boost.generate_circuit_modes() is invoked, it generates the complete set of switching modes of the circuit under analysis. The next logical operation we perform on the ECircuit object is boost.generate_matrices(), which produces the state-space equations for each switching mode and discretizes the equations.

8.2.4 Automated multi-program application extraction

The third layer is the end of the Application Synthesizer software module, where the AHA model is transformed into MPA model. In the first stage of this transformation, the analyzer simply makes each continuous system mode a program with a set of inputs and outputs. The matrices of the modes represent the working set of the program. Control algorithm, monitoring, and state transition modules form another set of programs. This stage of the power electronics control compilation process also classifies programs in terms of their hard real-time or best-effort computation requirements.

The process of decomposing these initial programs into finer-grained tasks with execution constraints, working set size, and execution time requirements depends on the application complexity and the targeted architecture configuration. In the case of a field-programmable gate array (FPGA), hardware resource limitations may only allow a certain number of core types, memory modules, or network routers. Similarly, for a non-reconfigurable system-on-chip (SoC), the power budget may be a factor for activating only a subset of available on-chip hardware units. This process, including the scheduling of these tasks and their mapping on an architecture, is discussed in detail in the following sections. Figure 8-11 shows the phase of the CAD tool that partitions the power electronics
Figure 8-10: C++ class declaration of circuit object and manipulator functions.

Figure 8-11: Computer-aided design flow task decomposition.

modules across parallel computing engines, while taking into account electrical component characteristics.
8.2.5 Automated task decomposition

In many-core system environments, parallel computing comes naturally. But this parallel computing paradigm also forces the application running to examine the type of parallelism it exhibits. Broadly, an application exhibits some instruction-level parallelism, some data parallelism, and some task parallelism. Instruction Level Parallelism (ILP) allows multiple instructions to be executed concurrently, and techniques to increase ILP in a program (application), are as relevant in uniprocessor platforms as multiprocessor platforms. ILP has been well researched [124, 125]. Data parallelism (DP) allows multiple processes to work on the same problem over disjoint memory spaces. However, one of the main challenges with data parallelism exploitation is the fact that obtaining accurate data dependency sets is hard, because certain data dependencies can only be resolved during runtime after address calculations. Task-level parallelism on heterogeneous many-core architectures seems more attractive because it decouples an application from the core ISA, or the number of cores. This reduces the problem of executing such an application on a given platform to a scheduling problem. The formal definition of a task and the application task decomposition were given in chapter 4.

8.3 Automated Task Scheduling and Processor Mapping

The problem of finding a schedule for a heterogeneous parallel architecture is complicated by a number of factors: different processing elements, not every processor may be able to execute all processes, the run time of a given process may be different on different processing elements, and communication time may vary. Before proceeding with a discussion of scheduling algorithms, we first give a set of standard definitions.

8.3.1 Scheduling algorithm: taxonomy

For understanding and completeness, we list some basic scheduling terminology [126]:

- A processor has a processing power $\rho$.
- Processor allocation: on which processor a task should execute.
- Task priority: when, and in what order with respect to other tasks, should each task execute.
- Fixed task priority: Each task has a single priority that remains the same through all phases of the application.
- Dynamic priority: a task may have different priorities at different execution points.
- Pre-emptive: tasks can be pre-empted by a higher priority task at any time.
• Non-pre-emptive: once a task starts executing, it will not be pre-empted and will therefore execute until completion.

• Co-operative: tasks may only be pre-empted at defined scheduling points within their execution.

• No migration: Each task is allocated to a processor and no migration is permitted

• Task-level migration: threads of a task may execute on different processors; however each thread can only execute on a single processor.

In this work, many of these characteristics or classifications are not examined, simply because prior work may have done it, or this particular instance of the problem does not warrant such examination, or it is obvious how to extend the algorithm to support such a characteristic. Our framework provides a set of algorithms for mapping an application onto a heterogeneous many-core architecture, starting from simple ones to more complex ones, depending on the level of available system characterization and application analysis data.

8.3.2 Definitions

In our framework we define an application \( A = \{T_1, T_2, ..., T_k\} \) where task \( T_i = (w_i, d_i, r_i, c_i) \), in addition to the characteristics described above, has the following properties: \( w_i \) (working set of task \( T_i \)), \( d_i \) (deadline of task \( T_i \)), \( r_i \) (input rate of task \( T_i \)), and \( c_i \) (instruction count in task \( T_i \)). We define a processing element \( p_i = (\varphi_i, L_{min}, L_{max}) \), where \( \varphi_i = f(IPC, EU_s, CS_s) \), \( L_{min} \) represents the memory latency on a cache hit, and \( L_{max} \) the maximum miss latency. \( \varphi_i \) is a function of the IPC (instructions-per-cycle), EUs (execution units), and CSs (cache sizes). The EU factor helps specify which processing unit has what execution functional unit, e.g., floating-point unit, multiplication unit.

Definition 17. A task admissible schedule (TAS) for an application \( A \) is a set of tuples that associates with each task \( T \) a set of processing elements such that the data dependencies and timing constraints between tasks are respected.

Definition 18. A task \( T_i \) is schedulable on a processing element \( p_j \), denoted \( T_i \triangleright p_j \), if its worst-case execution time \( p_j \) is less than or equal to its deadline. In this work, equations 8.7 and 8.8 are used to determine task schedulability.

Definition 19. An application is schedulable according to a TAS algorithm if all of its tasks are schedulable.

Definition 20. An application is said to be feasible with respect to a given heterogeneous many-core system if there exists at least one TAS solution that can schedule all possible sequences of tasks that may be generated by the application on that system without missing any deadlines or violating any inter-task dependency.
Definition 21. Given two tasks \( T_i \) and \( T_j \), a third task \( T_k \) can be composed out of \( T_i \) and \( T_j \) for mapping purposes if \( T_i \) and \( T_j \) are sequential, with no intermediate observable state, and cannot be pipelined.

8.3.3 ILP formulation of tasks-based scheduling

An application task graph \( G = (A, E) \) is a directed acyclic graph (DAG) in which each vertex \( T_i \in A \) represents a task and each edge \( e(T_i, T_j) \in E \) represents a dependency between tasks \( T_i \) and \( T_j \). Given a DAG \( G = (A, E) \), we want to find a schedule that minimizes the finishing time of the last critical task. For the formulation, let us assume that all tasks in \( A \) are critical and \( T_k \) is the last task. We want to assign to each task \( T_i \in A \) a pair \((t_s, t_f)\) where \( t_s \) and \( t_f \) represent the starting and finishing time of task \( T_i \) under a given schedule \( \theta \). For all edge \( e(T_i, T_j) \in E \), \( w_{i,j} \) represents the amount of data transferred from \( T_i \) to \( T_j \) during execution \( O(o_1, o_2, ..., o_m)_{T_i} \cap I(i_1, i_2, ..., i_n)_{T_j} = w_{i,j} \).

For \( k \) number of tasks and \( n \) processors, the exhaustive listing of schedules will produce \( \frac{k!}{(k-n)!} \) schedules. Therefore, this may be prohibitive for large \( k \). However, a user can limit the ILP runtime and find a solution over a subset.

8.3.4 ILP formulation

The objective function is:

\[
\text{minimize } T_k(t_f) \tag{8.1}
\]

Subject to:

\[
T_i(t_f) \leq d_j \tag{8.2}
\]

\[
T_i(t_s) \leq T_i(t_f) \tag{8.3}
\]

\[
\text{if } e(T_i, T_j) \in E \quad T_i(t_f) < T_j(t_s) \tag{8.4}
\]

\[
\forall e(T_i, T_j) \in E, \quad T_j(t_s) - T_i(t_f) = d_{i,j} \tag{8.5}
\]

\[
\forall (P(T_i) = p_u, P(T_j) = p_v), \quad w_{i,j} \times b_{p_u, p_v} \leq d_{i,j} \tag{8.6}
\]
\( \forall T_i \in A, \forall p_u \in P, \)
\[
E_{(T_i,p_u)} = (p_u \times c_i) \\
+ = (w_i \times \text{hitrate}_{i,u} \times l_{\text{min}_u}) \\
+ = (w_i \times (1 - \text{hitrate}_{i,u}) \times l_{\text{max}_u})
\]

(8.7)

For \( T_i \in A, p_u \in P, T_i(t_f) - T_i(t_s) \geq E_{(T_i,p_u)} \) (8.8)

\( \forall T_i \in A, p_u \in P, M(T_i,p_u) - (E_{(T_i,p_u)} \times b_{i,u}) = 0 \) (8.9)

\( \forall T_i \in A, \sum_{u=1}^{n} b_{i,u} = 1 \) (8.10)

\( \forall p_u \in P, \sum_{i=1}^{k} M(T_i,p_u) \leq T_k(t_f) \) (8.11)

8.3.5 Heuristic task-based scheduling algorithms

For applications with a large number of tasks and tight constraints where the convergence of the ILP formulation onto a satisfiable solution may take too long, we examine a set of heuristics to provide a fairly effective alternative to the ILP formulation. Heuristic H1 is very simple and converges quickly. It assigns to each task a set of processing units based on execution affinity, and tries to minimize processor-sharing among those sets. Its output is a set of processors that can be used for a task. In general it is good to select the fastest processing unit out of the set as the final mapping processor. Our second heuristic H2 takes into account task deadlines in addition to processor affinity in mapping tasks to processors. It tries to minimize the finishing time per processor as opposed to the global finishing time as done in the ILP formulation.

Algorithm 1. Minimizes intersections across all mapping sets (H1).
1. **Assumption:** An application \( A \) composed of a number of tasks, \( A = \{T_1, T_2, \ldots, T_k\} \) and a system with a list of processing elements \( P = \{p_1, p_2, \ldots, p_n\} \).
2. **Objective:** Find a set of task-processor mapping \( S = \{S(T_1), S(T_2), \ldots, S(T_k)\} \) such that: \( \forall T_i \in A, S(T_i) = \{p_u, \ldots, p_v\} \) with \( 1 \leq u < v \leq n \) while minimizing \( \forall (i, j) S(T_i) \cap S(T_j) \).
Algorithm 2. Minimizes the finishing time on each processor (H2).

1: Assumption: An application $A$ composed of a number of tasks, $A = \{T_1, T_2, \ldots, T_k\}$ and a system with a list of processing elements $P = \{p_1, p_2, \ldots, p_n\}$.

2: Objective: Find a set of task-processor mapping $\{S(p_1), S(p_2), \ldots, S(p_k)\}$ such that

$\forall p_i \in P, S(p_i) = \{T_a, \ldots, T_b\}$ where $D(p_i) = \text{minimum}(\sum_{u=1}^{k} d'_u)$.

3: Where $d'_u = \begin{cases} 
    d_u & \text{where } T_u \triangleright p_i \\
    0 & \text{otherwise}
\end{cases}$

4: Begin
5: $\forall p_i \in P, S(p_i) = \phi$ and $D(p_i) = 0$
6: for $i = 1; i \leq n; i++ : do$
7: for $j = 1; j \leq k; j++ : do$
8: if $(T_j \triangleright p_i)$ then
9: $S(p_i) = S(p_i) \cup \{T_j\}$
10: $D(p_i) += d_j$
11: end if
12: end for
13: end for
14: while $(\forall T_i \in A, |S(T_i)| > 1 \text{ and } \forall (i,j) S(T_i) \cap S(T_j) \neq \phi)$ do
15: if $(\exists (T_i, T_j) \mid S(T_i) \cap S(T_j) \neq \phi)$ then
16: if $(|S(T_i)| > 1 \land |S(T_j)| > 1)$ then
17: $S(T_i) = \begin{cases} 
    S(T_i) - \{S(T_{\text{min}}) \cap S(T_i)\} & \text{where } S(T_i) \subseteq \{S(T_{\text{min}}) \cap S(T_i)\} \\
    \{p_e\} & \text{for any } p_e \in S(T_i) \text{ otherwise}
\end{cases}$
18: end if
19: end if
20: end while
21: End
8.3.6 Scheduling algorithms application

Using the task decomposition of the wind turbine and hybrid vehicle applications (cf. chapter 3), we show the effectiveness of the scheduling algorithms on heterogenous architectures of the different processing elements presented in chapter 5. In general, H1 performs poorly compared to the ILP-based mapping, because it does not take into account many of the system architectural features. If a compute-intensive task is inadvertently mapped onto a weaker core, it can impact the whole application execution time. H2 performs better H1, and it is within 20% to 70% of the ILP-based mapping efficiency. Figures 8-12 and 8-13 show system-step latency for each algorithm.

8.3.7 Automated task mapping

The scheduling process is iterative. Even when a feasible task-to-core schedule is discovered, the on-chip network communications need to satisfy the required application data movement bandwidth and latency requirements. Figure 8-14 shows the final step of the CAD tool chain. The network-on-chip routing algorithm is presented in chapter 7. Once a schedule and a routing are determined, the tasks are compiled with their target processing element parameters and memory management directives. In this work, a MIPS GCC cross-compiler is used to get executable binaries.

8.4 Summary

To summarize, we provide a CAD tool for extracting power electronics system models, for analyzing these models, and transforming these models into forms that can be efficiently executed on a computer system. We present an ILP formulation and two non-iterative
Figure 8-12: Scheduling of wind turbine system application.
(a) ILP schedule solution for the hybrid electric vehicle application.

(b) H1 schedule solution for the hybrid electric vehicle application.

(c) H2 schedule solution for the hybrid electric vehicle application.

Figure 8-13: Scheduling of hybrid electric vehicle application.
heuristics that can handle different application performance targets and architectural features for scheduling a task-based application onto a heterogeneous many-core architecture. Most of the techniques presented are easily expandable to other hybrid control applications.
Chapter 9

Architecture Design Space Exploration Tool

In this chapter we present the hardware design tool used to generate the core, memory, and network configuration of the various hard real-time architectures presented in the dissertation. *Heracles* is an open-source, functional, parameterized, synthesizable multicore system design toolkit. Such a multi/many-core design platform is a powerful and versatile research and teaching tool for architectural exploration and hardware-software co-design. The *Heracles* toolkit comprises the soft hardware (HDL) modules, application compiler, and graphical user interface. It is designed with a high degree of modularity to support fast exploration of future multicore processors of different topologies, routing schemes, processing elements (cores), and memory system organizations. It is a component-based framework with parameterized interfaces and strong emphasis on module reusability. The compiler toolchain is used to map C or C++ based applications onto the processing units. The GUI allows the user to quickly configure and launch a system instance for easy factorial development and evaluation. Hardware modules are implemented in synthesizable Verilog and are FPGA platform independent. The *Heracles* tool is freely available under the open-source MIT license at: http://projects.csail.mit.edu/heracles.

9.1 Introduction

The ability to integrate various computation components such as processing cores, memories, custom hardware units, and complex network-on-chip (NoC) communication protocols onto a single chip has significantly enlarged the design space in multi/many-core systems. The design of these systems requires tuning of a large number of parameters in order to find the most suitable hardware configuration, in terms of performance, area, and energy consumption, for a target application domain. This increasing complexity makes the need for efficient and accurate design tools more acute.
There are two main approaches currently used in the design space exploration of multi/many-core systems. One approach consists of building software routines for the different system components and simulating them to analyze system behavior. Software simulation has many advantages: i) large programming tool support; ii) internal states of all system modules can be easily accessed and altered; iii) compilation/re-compilation is fast; and iv) less constraining in terms of number of components (e.g., number of cores) to simulate. Some of the most stable and widely used software simulators are Simics [127]—a commercially available full-system simulator—GEMS [128], Hornet [129], and Graphite [130]. However, software simulation of many-core architectures with cycle- and bit-level accuracy is time-prohibitive, and many of these systems have to trade off evaluation accuracy for execution speed. Although such a tradeoff is fair and even desirable in the early phase of the design exploration, making final micro-architecture decisions based on these software models over truncated applications or application traces leads to inaccurate or misleading system characterization.

The second approach used, often preceded by software simulation, is register-transfer level (RTL) simulation or emulation. This level of accuracy considerably reduces system behavior mis-characterization and helps avoid late discovery of system performance problems. The primary disadvantage of RTL simulation/emulation is that as the design size increases so does the simulation time. However, this problem can be circumvented by adopting synthesizable RTL and using hardware-assisted accelerators—field programmable gate arrays (FPGAs)—to speed up system execution. Although FPGA resources constrain the size of design one can implement, recent advances in FPGA-based design methodologies have shown that such constraints can be overcome. HAsim [131], for example, has shown using its time multiplexing technique how one can model a shared-memory multicore system including detailed core pipelines, cache hierarchy, and on-chip network, on a single FPGA. RAMP Gold [132] is able to simulate a 64-core shared-memory target machine capable of booting real operating systems running on a single Xilinx Virtex-5 FPGA board. Fleming et al [133] propose a mechanism by which complex designs can be efficiently and automatically partitioned among multiple FPGAs.

RTL design exploration for multi/many-core systems nonetheless remain unattractive to most researchers because it is still a time-consuming endeavor to build such large designs from the ground up and ensure correctness at all levels. Furthermore, researchers are generally interested in one key system area, such as processing core and/or memory organization, network interface, interconnect network, or operating system and/or application mapping. Therefore, we believe that if there is a platform-independent design framework, more specifically, a general hardware toolkit, which allows designers to compose their systems and modify them at will and with very little effort or knowledge of other parts of the system, the speed versus accuracy dilemma in design space exploration of many-core
systems can be further mitigated.

To that end we present Heracles, a functional, modular, synthesizable, parameterized multicore system toolkit. It is a powerful and versatile research and teaching tool for architectural exploration and hardware-software co-design. Without loss in timing accuracy and logic, complete systems can be constructed, simulated and/or synthesized onto FPGA, with minimal effort. The initial framework is presented in [134]. Heracles is designed with a high degree of modularity to support fast exploration of future multicore processors—different topologies, routing schemes, processing elements or cores, and memory system organizations by using a library of components, and reusing user-defined hardware blocks between different system configurations or projects. It has a compiler toolchain for mapping applications written in C or C++ onto the core units. The graphical user interface (GUI) allows the user to quickly configure and launch a system instance for easily-factored development and evaluation. Hardware modules are implemented in synthesizable Verilog and are FPGA platform independent.

9.2 Heracles Hardware System

9.2.1 System overview

Heracles presents designers with a global and complete view of the inner workings of the multi/many-core system at cycle-level granularity from instruction fetches at the processing
core in each node to the flit arbitration at the routers. It enables designers to explore different implementation parameters: core micro-architecture, levels of caches, cache sizes, routing algorithm, router micro-architecture, distributed or shared memory, or network interface, and to quickly evaluate their impact on the overall system performance. It is implemented with user-enabled performance counters and probes.

Figure 9-1 illustrates the general Heracles-based design flow. Full applications—written in single or multithreaded C or C++—can be directly compiled onto a given system instance using the Heracles MIPS-based GCC cross compiler. The detailed compilation process and application examples are presented in Section 9.4. For a multi/many-core system, we take a component-based approach by providing clear interfaces to all modules for easy composition and substitutions. The system has multiple default settings to allow users to quickly get a system running and only focus on their area of interest. System and application binary can be executed in an RTL simulated environment and/or on an FPGA. Figure 9-2 shows two different views of a typical network node structure in Heracles.

Figure 9-2: Network node structure.

### 9.2.2 Processing units

In the current version of the Heracles design framework, users can instantiate four different types of processor cores, or any combination thereof, depending on the programming model adopted and architectural evaluation goals.

#### 9.2.2.1 Injector core

The injector core (iCore) is the simplest processing unit. It emits and/or collects from the network user-defined data streams and traffic patterns. Although it does not do any useful computation, this type of core is useful when the user is only focusing on the network on-chip behavior. It is useful in generating network traffic and allowing the evaluation of
network congestion. Often, applications running on real cores fail to produce enough data traffic to saturate the network.

9.2.2.2 Single hardware-threaded MIPS core

This is an integer 7-stage 32-bit MIPS-Microprocessor without Interlocked Pipeline Stages-Core (sCore). This RISC architecture is widely used in commercial products and for teaching purposes [101]. Most users are very familiar with this architecture and its operation, and will be able to easily modify it when necessary. Our implementation is generally standard with some modifications for FPGAs. For example, the adoption of a 7-stage pipeline, due to block RAM access time on the FPGA. The architecture is fully bypassed, with no branch prediction table or branch delay slot, running MIPS-III instruction set architecture (ISA) without floating point. Instruction and data caches are implemented using block RAMs, and instruction fetch and data memory access take two cycles. Stall and bypass signals are modified to support the extended pipeline. Instructions are issued and executed in-order, and the data memory accesses are also in-order.

9.2.2.3 Two-way hardware-threaded MIPS core

A fully functional fine-grain hardware multithreaded MIPS core (dCore). There are two hardware threads in the core. The execution datapath for each thread is similar to the single-threaded core above. Each of the two threads has its own context which includes a program counter (PC), a set of 32 data registers, and one 32-bit state register. The core can dispatch instructions from any one of hardware contexts and supports precise interrupts (doorbell type) with limited state saving. A single hardware thread is active on any given cycle, and pipeline stages must be drained between context switches to avoid state corruption. The user has the ability to control the context switching conditions, e.g., minimum number of cycles to allocate to each hardware thread at a time, instruction or data cache misses.

9.2.2.4 Two-way hardware-threaded MIPS core with migration

The fourth type of core is also a two-way hardware-threaded processor but enhanced to support hardware-level thread migration and evictions (mCore). It is the user's responsibility to guarantee deadlock-freedom under this core configuration. One approach is to allocate local memory to contexts so on migration they are removed from the network. Another approach which requires no additional hardware modification to the core, is using Cho et al [135] deadlock-free thread migration scheme.
9.2.2.5 FPGA synthesis data

All the cores have the same interface, they are self-contained and oblivious to the rest of the system, and therefore easily interchangeable. The cores are synthesized using Xilinx ISE Design Suite 11.5, with Virtex-6 LX550T package ff1760 speed -2, as the targeted FPGA board. The number of slice registers and slice lookup tables (LUTs) on the board are 687360 and 343680 respectively. Table 9.1 shows the register and LUT utilization of the different cores. The two-way hardware-threaded core with migration consumes the most resources and is less than 0.5%. Table 9.1 also shows the clocking speed of the cores. The injector core, which does no useful computation, runs the fastest at 500.92 MHz whereas the two-way hardware-threaded core runs the slowest at 118.66 MHz.

Table 9.1: FPGA resource utilization per core type

<table>
<thead>
<tr>
<th>Core type</th>
<th>iCore</th>
<th>sCore</th>
<th>dCore</th>
<th>mCore</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>227</td>
<td>1660</td>
<td>2875</td>
<td>3484</td>
</tr>
<tr>
<td>LUTs</td>
<td>243</td>
<td>3661</td>
<td>5481</td>
<td>6293</td>
</tr>
<tr>
<td>Speed (MHz)</td>
<td>500.92</td>
<td>172.02</td>
<td>118.66</td>
<td>127.4</td>
</tr>
</tbody>
</table>

9.2.3 Memory system organization

The memory system in Heracles is parameterized, and can be set up in various ways, independent of the rest of the system. The key components are main memory, caching system, and network interface.

9.2.3.1 Main memory configuration

The main memory is constructed to allow different memory space configurations. For Centralized Shared Memory (CSM) implementation, all processors share a single large main memory block; the local memory size (shown in Figure 9-2) is simply set to zero at all nodes.
except one. In Distributed Shared Memory (DSM), where each processing element has a
local memory, the local memory is parameterized and has two very important attributes:
the size can be changed on a per core-basis, providing support for both uniform and non-
uniform distributed memory, and it can service a variable number of caches in a round-robin
fashion. Figure 9-3 illustrates these physical memory partitions. The fact that the local
memory is parameterized to handle requests from a variable number of caches allows the
traffic coming into a node from other cores through the network to be presented to local
memory as just another cache communication. This illusion is created through the network
packetizer. Local memory can also be viewed as a memory controller. Figure 9-4 illustrates
the local structure of the memory sub-system. The \textit{LOCAL ADDR BITS} parameter is used
to set the size of the local memory. The \textit{Address Translation Logic} performs the virtual-to-
physical address lookup using the high-order bits, and directs cache traffic to local memory
or network.

![Diagram of Local Memory Sub-system](image)

Figure 9-4: Local memory sub-system structure.

For cache coherence, a directory is attached to each local memory and the MESI protocol
is implemented as the default coherence mechanism. Remote access (RA) is also supported.
In RA mode, the network packetizer directly sends network traffic to the caches. Memory
structures are implemented in FPGA using block RAMs. There are 632 block RAMs on the
Virtex-6 LX550T. A local memory of 0.26\(MB\) uses 64 block RAMs or 10\%. Table 9.2 shows
the FPGA resource used to provide the two cache coherence mechanisms. The RA scheme
uses less hardware resources than the cache-coherence-free structure, since no cache-line
buffering is needed. The directory-based coherence is far more complex resulting in more
resource utilization. The \textit{SHARERS} parameter is used to set the number of sharers per data
block. It also dictates the overall size of the local memory directory size. When a directory
entry cannot handle all sharers, other sharers are evicted.
Table 9.2: FPGA resource utilization per coherence mechanism

<table>
<thead>
<tr>
<th>Coherence</th>
<th>None</th>
<th>RA</th>
<th>Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>2917</td>
<td>2424</td>
<td>11482</td>
</tr>
<tr>
<td>LUTs</td>
<td>5285</td>
<td>4826</td>
<td>17460</td>
</tr>
<tr>
<td>Speed (MHz)</td>
<td>238.04</td>
<td>217.34</td>
<td>171.75</td>
</tr>
</tbody>
</table>

9.2.3.2 Caching system

The user can instantiate direct-mapped Level 1 or Levels 1 and 2 caches with the option of making Level 2 an inclusive cache. The INDEX.BITS parameter defines the number of blocks or cache-lines in the cache where the OFFSET.BITS parameter defines block size. By default, cache and memory structures are implemented in FPGA using block RAMs, but user can instruct Heracles to use LUTs for caches or some combination of LUTs and block RAMs. A single 2KB cache uses 4 FPGA block RAMs, 462 slice registers, 1106 slice LUTs, and runs at 228.8MHz. If cache size is increased to 8KB by changing the INDEX.BITS parameter from 6 to 8, resource utilization and speed remain identical. Meanwhile if cache size is increased to 8KB by changing the OFFSET.BITS parameter from 3 to 5, resource utilization increases dramatically: 15 FPGA block RAMs, 1232 slice registers, 3397 slice LUTs, and speed is 226.8MHz. FPGA-based cache design favors large number of blocks of small size versus small number of blocks of large size ¹.

9.2.3.3 Network interface

The Address Resolution Logic works with the Packetizer module, shown in Figure 9-2, to get the caches and the local memory to interact with the rest of the system. All cache traffic goes through the Address Resolution Logic, which determines if a request can be served at the local memory, or if the request needs to be sent over the network. The Packetizer is responsible for converting data traffic, such as a load, coming from the local memory and the cache system into packets or flits that can be routed inside the Network-on-chip (NoC), and for reconstructing packets or flits into data traffic at the opposite side when exiting the NoC.

9.2.3.4 Hardware multithreading and caching

In this section, we examine the effect of hardware multithreading (HMT) on system performance. We run the 197.parser application from the SPEC CINT2000 benchmarks on a single node with the dCore as the processing unit using two different inputs—one per thread—with five different execution interleaving policies:

¹Cache-line size also has traffic implications at the network level
- setup 1: threads take turns to execute every 32 cycles; on a context switch, the pipeline is drained before the execution of another thread begins.

- setup 2: thread switching happens every 1024 cycles.

- setup 3: thread context swapping is initiated on an instruction or a data miss at the Level 1 cache.

- setup 4: thread interleaving occurs only when there is a data miss at the Level 1 cache.

- setup 5: thread switching happens when there is a data miss at the Level 2 cache.

Figure 9-5 shows the total completion time of the two threads (in terms of number of cycles). It is worth noting that even with fast fine-grain hardware context switching, multithreading is most beneficial for large miss penalty events like Level 2 cache misses or remote data accesses.

![Execution Cycles](image)

Figure 9-5: Effects of hardware multithreading and caching.

### 9.2.4 Network-on-Chip (NoC)

To provide scalability, *Heracles* uses a network-on-chip (NoC) architecture for its data communication infrastructure. A NoC architecture is defined by its topology (the physical organization of nodes in the network), its flow control mechanism (which establishes the data formatting, the switching protocol and the buffer allocation), and its routing algorithm (which determines the path selected by a packet to reach its destination under a given application).
9.2.4.1 Flow control

Routing in *Heracles* can be done using either bufferless or buffered routers. Bufferless routing is generally used to reduce area and power overhead associated with buffered routing. Contention for physical link access is resolved by either dropping and retransmitting or temporarily misrouting or *deflecting* of flits. With flit dropping an acknowledgment mechanism is needed to enable retransmission of lost flits. With flit deflection, a priority-based arbitration, e.g., *age-based*, is needed to avoid livelock. In *Heracles*, to mitigate some of the problems associated with the lossy bufferless routing, namely retransmission and slow arbitration logic, we supplement the arbiter with a routing table that can be statically and off-line configured on a per-application basis.

![Diagram](image)

**Figure 9-6:** Virtual channel based router architecture.

The system default virtual-channel router conforms in its architecture and operation to conventional virtual-channel routers [66]. It has some input buffers to store flits while they are waiting to be routed to the next hop in the network. The router is modular enough to allow user to substitute different arbitration schemes. The routing operation takes four steps or phases, namely routing (RC), virtual-channel allocation (VA), switch allocation (SA), and switch traversal (ST), where each phase corresponds to a pipeline stage in our router. Figure 9-6 depicts the general structure of the buffered router. In this router the number of virtual channels per port and their sizes are controlled through *VC.PER.PORT* and *VC.DEPTH* parameters. Table 9.3 shows the register and LUT utilization of the bufferless router and different buffer configurations of the buffered router. It also shows the effect of virtual channels on router clocking speed. The key take-away is that a larger number of VCs at the router increases both the router resource utilization and the critical path.
Table 9.3: FPGA resource utilization per router configuration

<table>
<thead>
<tr>
<th>Number of VCs</th>
<th>Bufferless</th>
<th>2 VCs</th>
<th>4 VCs</th>
<th>8 VCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>175</td>
<td>4081</td>
<td>7260</td>
<td>13374</td>
</tr>
<tr>
<td>LUTs</td>
<td>328</td>
<td>7251</td>
<td>12733</td>
<td>23585</td>
</tr>
<tr>
<td>Speed (MHz)</td>
<td>817.18</td>
<td>111.83</td>
<td>94.8</td>
<td>80.02</td>
</tr>
</tbody>
</table>

9.2.4.2 Routing algorithm

Algorithms used to compute routes in network-on-chip (NoC) architectures, generally fall under two categories: oblivious and dynamic [116]. The default routers in Heracles primarily support oblivious routing algorithms using either fixed logic or routing tables. Fixed logic is provided for dimension-order routing (DOR) algorithms, which are widely used and have many desirable properties. On the other hand, table-based routing provides greater programmability and flexibility, since routes can be pre-computed and stored in the routing tables before execution. Both buffered and bufferless routers can make usage of the routing tables. Heracles provides support for both static and dynamic virtual channel allocation.

9.2.4.3 Network topology configuration

The parameterization of the number of input ports and output ports on the router and the table-based routing capability give Heracles a great amount of flexibility and the ability to metamorphose into different network topologies; for example, k-ary n-cube, 2D-mesh, 3D-mesh, hypercube, ring, or tree. A new topology is constructed by changing the IN_PORTS, OUT_PORTS, and SWITCH_TO_SWITCH parameters and reconnecting the routers. Table 9.4 shows the clocking speed of a bufferless router, a buffered router with strict round-robin arbitration (Arbiter1), a buffered router with weak round-robin arbitration (Arbiter2), and a buffered router with 7 ports for a 3D-mesh network. The bufferless router runs the fastest at 817.2MHz, Arbiter1 and Arbiter2 run at the same speed (~ 112), although the arbitration scheme in Arbiter2 is more complex. The 7-port router runs the slowest due to more complex arbitration logic.

Table 9.4: Clocking speed of different router types

<table>
<thead>
<tr>
<th>Router type</th>
<th>Bufferless</th>
<th>Arbiter1</th>
<th>Arbiter2</th>
<th>7-Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed (MHz)</td>
<td>817.18</td>
<td>111.83</td>
<td>112.15</td>
<td>101.5</td>
</tr>
</tbody>
</table>

Figure 9-7(a) shows a 3x3 2D-mesh with all identical routers. Figure 9-7(b) depicts an unbalanced fat-tree topology. For a fat-tree [136] topology, routers at different levels of the tree have different sizes, in terms of crossbar and arbitration logic. The root node contains
the largest router, and controls the clock frequency of the system.

Figure 9-7: Network topology examples.

9.3 Heracles Programming Models

A programming model is inherently tied to the underlying hardware architecture. The Heracles design tool has no directly deployable operating system, but it supports both sequential and parallel programming models through various application programming interface (API) protocols. There are three memory spaces associated with each program: instruction memory, data memory, and stack memory. Dynamic memory allocation is not supported in the current version of the tool.

9.3.1 Sequential programming model

In the sequential programming model, a program has a single entry point (starting program counter–PC) and single execution thread. Under this model, a program may exhibit any of the follow behavior or a combination thereof:

- the local memory of executing core has instruction binary;
- PC pointer to another core’s local memory where the instruction binary resides;
- the stack frame pointer–SP points to the local memory of executing core;
- SP points to another core’s local memory for the storage of the stack frame;
- program data is stored at the local memory;
- program data is mapped to another core local memory.
Figure 9-8: Examples of memory space management for sequential programs.

Figure 9-8 gives illustrating examples of the memory space management when dealing with sequential programs. These techniques provide the programming flexibility needed to support the different physical memory configurations. They also allow users:

- to run the same program on multiple cores (program inputs can be different); in this setup the program binary is loaded to one core and the program counter at other cores points to the core with the binary;
- to execute one program across multiple cores by migrating the program from one core to another.

### 9.3.2 Parallel programming model

The *Heracles* design platform supports both hardware multi-threading and software multi-threading. The keywords *HHThread1* and *HHThread2* are provided to specify the part of the program to execute on each hardware thread. Multiple programs can also be executed on the same core using the same approach. An example is shown below:

```c
...  
int HHThread1 (int *array, int item, int size) {  
    sort(array, size);  
    int output = search(array, item, size);  
    return output;  
}  

int HHThread2 (int input,int src,int aux,int dest){  
    int output = Hanoi(input, src, aux, dest);  
}```

135
return output;
}
...

Below is the associated binary outline:

@e2 // <HHThread1>
27bdffd8 // 00000388 addiu sp,sp,-40
...
0c00004c // 000003c0 jal 130 <search>
...

@fa // <HHThread2>
27bdffd8 // 000003e8 addiu sp,sp,-40
...
0c0000ab // 00000418 jal 2ac <Hanoi>
...

@110 // <HHThread1_Dispatcher>
27bdffa8 // 00000440 addiu sp,sp,-88
...
0c0000e2 // 000004bc jal 388 <hThread1>
...

@15e // <HHThread2_Dispatcher>
27bdffa8 // 00000440 addiu sp,sp,-88
...
0c0000fa // 000004d8 jal 3e8 <HHThread2>
...

*Heracles* uses OpenMP style pragmas to allow users to directly compile multi-threaded programs onto cores. Users specify programs or parts of a program that can be executed in parallel and on which cores. Keywords *HLock* and *HBarrier* are provided for synchronization and shared variables are encoded with the keyword *HGlobal*. An example is shown below:

...  

```c
#pragma Heracles core 0 {
  // Synchronizers
  HLock lock1, lock2;
  HBarrier bar1, bar2;
```
/ Variables
HGlobal int arg1, arg2, arg3;
HGlobal int Arr[16][16];
HGlobal int Arr0[16][16] = { { 1, 12, 7, 0,...
HGlobal int Arr1[16][16] = { { 2, 45, 63, 89,...

// Workers
#pragma Heracles core 1
{ start_check(50);
check_lock(&lock1, 1000);
matrix_add(Arr, Arr0, Arr1, arg1);
clear_barrier(&bar1);
}

#pragma Heracles core 2
{ start_check(50);
check_lock(&lock1, 1000);
matrix_add(Arr, Arr0, Arr1, arg2);
clear_barrier(&bar2);
}
}
...

Below is the intermediate C representation:
...
int core_0_lock1, core_0_lock2;
int core_0_bar1, core_0_bar2;
int core_0_arg1, core_0_arg2, core_0_arg3;
int core_0_Arr[16][16];
int core_0_Arr0[16][16] = { { 1, 12, 7, 0,...
int core_0_Arr1[16][16] = { { 2, 45, 63, 89,...

void core_0_work (void)
{
    // Synchronizers
    // Variables
    // Workers
    // Main function
    .
main();
}
void core_1_work (void)
{
    start_check(50);
    check_lock(&core_0_lock1, 1000);
    matrix_add(core_0_Arr, core_0_Arr0, core_0_Arr1, 
               core_0_arg1);
    clear_barrier(&core_0_bar1);
}
...

Below is the associated binary outline:

@12b // <Dispatcher>
27bdffe8 // 000004ac addiu sp,sp,-24
...
0c000113 // 000004bc jal 44c <core_0_work>
...

9.4 Programming Toolchain

9.4.1 Program compilation flow

![Software toolchain flow diagram](image)

Figure 9-9: Software toolchain flow.

The *Heracles* environment has an open-source compiler toolchain to assist in developing software for different system configurations. The toolchain is built around the GCC MIPS cross-compiler using GNU C version 3.2.1. Figure 9-9 depicts the software flow for compiling a C program into the compatible MIPS instruction code that can be executed on the system. The compilation process consists of a series of six steps.
• First, the user invokes mips-gcc to translate the C code into assembly language (e.g., ./mips-gcc -S fibonacci.c).

• In step 2, the assembly code is then run through the isa-checker (e.g., ./checker fibonacci.s). The checker’s role is to: (1) remove all memory space primitives, (2) replace all pseudo-instructions, and (3) check for floating point instructions. Its output is a .asm file.

• For this release, there is no direct high-level operating system support. Therefore, in the third compilation stage, a small kernel-like assembly code is added to the application assembly code for memory space management and workload distribution (e.g., ./linker fibonacci.asm). Users can modify the linker.cpp file provided in the toolchain to reconfigure the memory space and workload.

• In step 4, the user compiles the assembly file into an object file using the cross-compiler. This is accomplished by executing mips-as on the .asm file (e.g., ./mips-as fibonacci.asm).

• In step 5, the object file is disassembled using the mips-objdump command (e.g., ./mips-objdump fibonacci.o). Its output is a .dump file.

• Finally, the constructor script is called to transform the dump file into a Verilog memory, .vmh, file format (e.g., ./dump2vmh fibonacci.dump).

If the program is specified using Heracles multi-threading format (.hc or .hcc), c-cplusplus-generator (e.g., ./c-cplusplus-generator fibonacci.hc) is called to first get the C or C++ program file before executing the steps listed above. The software toolchain is still evolving. All these steps are also automated through the GUI.

9.4.2 Heracles graphical user interface

![Heracles Designer graphical user interface](image-url)

Figure 9-10: Heracles designer graphical user interface.

The graphical user interface (GUI) is called Heracles Designer. It helps to quickly configure and launch system configurations. Figure 9-10 shows a screen shot of the GUI. On
the core tab, the user can select: (1) the type of core to generate, (2) the network topology of the system instance to generate, (3) the number of cores to generate, (4) traffic type, injection rate, and simulation cycles in the case of an injector core, or (5) different pre-configured settings. Generate and Run buttons on this tab are used to automatically generate the Verilog files and to launch the synthesis process or specified simulation environment. The second tab—memory system tab—allows the user to set: (1) main memory configuration (e.g., Uniformed Distributed), (2) total main memory size, (3) instruction and data cache sizes, (4) Level 2 cache, and (5) FPGA favored resource (LUT or block RAM) for cache structures. The on-chip network tab covers all the major aspects of the system interconnect: (1) routing algorithm, (2) number of virtual channels (VCs) per port, (3) VC depth, (4) core and switch bandwidths, (5) routing tables programming, by selecting source/destination pair or flow ID, router ID, output port, and VC (allowing user-defined routing paths), and (6) number of flits per packet for injector-based traffic. The programming tab is updated when the user changes the number of cores in the system; the user can: (1) load a binary file onto a core, (2) load a binary onto a core and set the starting address for another core to point to that binary, (3) select where to place the data section or stack pointer of a core (it can be local, on the same core as the binary or on another core), and (4) select which cores to start.

9.5 Experimental Results

9.5.1 Full 2D-mesh systems

The synthesis results of five multicore systems of size: 2×2, 3×3, 4×4, 5×5, and 6×6 arranged in 2D-mesh topology are summarized below. Table 9.5 gives the key architectural characteristics of the multicore system. All five systems run at 105.5MHz, which is the clock frequency of the router, regardless of the size of the mesh.

Figure 9-11 summarizes the FPGA resource utilization by the different systems in terms of registers, lookup tables, and block RAMs. In the 2×2 and 3×3 configurations, the local memory is set to 260KB per core. The 3×3 configuration uses 99% of block RAM resources at 260KB of local memory per core. For the 4×4 configuration the local memory is reduced to 64KB per core, and the local memory in the 5×5 configuration is set to 32KB. The 6×6 configuration, with 16KB of local memory per core, fails during the mapping and routing synthesis steps, due to the lack of LUTs.

9.5.2 Evaluation results

We examine the performance of two SPEC CINT2000 benchmarks, namely, 197.parser and 256.bzip2 on Heracles. We modify and parallelize these benchmarks to fit into our evaluation framework. For the 197.parser benchmark, we identify three functional units: file
Resource Utilization

Figure 9-11: Percentage of FPGA resource utilization per mesh size.

Figure 9-12: Effect of memory organization on performance for the different applications.

Table 9.5: 2D-mesh system architecture details

<table>
<thead>
<tr>
<th>Core</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ISA</td>
<td>32-Bit MIPS</td>
</tr>
<tr>
<td>Hardware threads</td>
<td>1</td>
</tr>
<tr>
<td>Pipeline Stages</td>
<td>7</td>
</tr>
<tr>
<td>Bypassing</td>
<td>Full</td>
</tr>
<tr>
<td>Branch policy</td>
<td>Always non-Taken</td>
</tr>
<tr>
<td>Outstanding memory requests</td>
<td>1</td>
</tr>
<tr>
<td>Level 1 Instruction/Data Caches</td>
<td></td>
</tr>
<tr>
<td>Associativity</td>
<td>Direct</td>
</tr>
<tr>
<td>Size</td>
<td>variable</td>
</tr>
<tr>
<td>Outstanding Misses</td>
<td>1</td>
</tr>
<tr>
<td>On-Chip Network</td>
<td></td>
</tr>
<tr>
<td>Topology</td>
<td>2D-Mesh</td>
</tr>
<tr>
<td>Routing Policy</td>
<td>DOR and Table-based</td>
</tr>
<tr>
<td>Virtual Channels</td>
<td>2</td>
</tr>
<tr>
<td>Buffers per channel</td>
<td>8</td>
</tr>
</tbody>
</table>
reading and parameters setting as one unit, actual parsing as a second unit, and error reporting as the third unit. When there are more than three cores, all additional cores are used in the parsing unit. Similarly, 256.bzip2 is divided into three functional units: file reading and cyclic redundancy check, compression, and output file writing. The compression unit exhibits a high degree of data-parallelism, therefore we apply all additional cores to this unit for core count greater than three. We also present a brief analysis of a simple Fibonacci number calculation program. Figures 9-12 (a), (b), and (c) show 197.parser, 256.bzip2, and Fibonacci benchmarks under single shared-memory (SSM) and distributed shared-memory (DSM), using XY-Ordered routing. Increasing the number of cores improves performance for both benchmarks; it also exposes the memory bottleneck encountered in the single shared-memory scheme. Figures 9-13 (a), (b), and (c) highlight the impact of the routing algorithm on the overall system performance, by comparing completion cycles of XY-Ordered routing and BSOR [137]. BSOR, which stands for Bandwidth-Sensitive Oblivious Routing, is a table-based routing algorithm that minimizes the maximum channel load.
(MCL) (or maximum traffic) across all network links in an effort to maximize application throughput. The routing algorithm has little or no effect on the performance of 197.parser and 256.bzip2 benchmarks, because of the traffic patterns in these applications. For the Fibonacci application, Figure 9-13 (c), BSOR routing does improve performance, particularly with 5 or more cores. To show the multithreading and scalability properties of the system, Figure 9-14 presents the execution times for matrix multiplication given matrices of different size. The Heracles multicore programming format is used to automate of the workload distribution onto cores.

9.6 Summary

In this chapter, we presented the Heracles design toolkit used to construct the hardware components of the proposed architectures. It is a generic multicore architecture design exploration tool that can be used even in purely general-purpose architectures. It is comprised of soft hardware (HDL) modules, an application compiler toolchain, and a graphical user interface. It is a component-based framework that gives researchers the ability to create complete, realistic, synthesizable, multi/many-core architectures for fast, high-accuracy design space exploration. In this environment, the user can explore design tradeoffs at the processing unit level, the memory organization and access level, and the network on-chip level. The Heracles tool is open-source and can be downloaded at http://projects.csail.mit.edu/heracles. In the current release, RTL hardware modules can be simulated on all operating systems, the MIPS GCC cross-compiler runs in a Linux environment, and the graphical user interface has a Windows installer.
Chapter 10

MARTHA family of architectures

In this chapter, we present different degrees of complexity and capability of the MARTHA (Multicore Architecture for Real-Time Hybrid Applications) architecture using the Heracles (c.f Chapter 9) design tool. MARTHA is a parameterized system-on-a-chip (SoC) architecture targeted at the emulation and control of future power electronics applications. The architecture is highly programmable and can be used in other hybrid control applications.

10.1 Introduction

The proposed and developed MARTHA architecture is a heterogeneous, reconfigurable, multicore architecture. It is designed to meet the high computation throughput and low latency requirement for a spectrum of hard real-time hybrid control systems (e.g., mechanical systems, industrial systems, unmanned aerial vehicles). In this work, we show MARTHA’s implementations for real-time emulation, prototyping, and estimation of power electronics systems and controllers. It is flexible enough to cover a wide range of power electronics systems, from a simple buck converter to a multi-level converter.

A SoC can be simply defined as an on-chip integration of a variety of functional hardware units. As such, MARTHA uses a modular and parameterized design framework to enable different system configurations of hardware units depending on the target power electronics application. Each system configuration can have a varying level of parallelism, hardware complexity, heterogeneity, and scale. Given an application, a designer can automatically build a minimal instance of the architecture that best suits the application requirements. This is the approach we adopt in the FPGA-based designs presented in this work. Although not considered in this thesis, one can imagine spinning a MARTHA domain-specific integrated circuit, where the same techniques of deriving the FPGA-based architecture are used to identify the set of hardware components to activate for a given application.

The architecture efficiently integrates support for predictable execution-time while providing high-performance and programmability for power electronics applications, smart
grid systems, and potentially other hybrid real-time applications. Its building blocks are: general-purpose MIPS-core, MIPS-based RISC SIMD vector machine style core, microcontroller, cache, scratchpad memory, buffer queue, and hybrid-path network router. This heterogeneity of the architecture is dictated by the complex interactions between the computer system and the physical environment under control. In most instances the architecture has at least one DSP functional unit. It is used for analog-to-digital conversions when reading sensor data from the physical environment, and for digital-to-analog conversions when transmitting command instructions.

Per the multi-program application framework developed in chapter 4, the main functionalities of the software running are:

1. in each system-step, the control system samples inputs (both analog and digital), performs A-to-D conversions and error corrections if necessary, and stores input data into memory,

2. it calculates the current discrete mode of the physical system based on the inputs and the previous system state,

3. it evaluates the continuous behavior of the selected discrete mode,

4. it emits control commands to the plant,

5. while collecting monitoring data on itself and the plant.

Figure 10-1 shows the typical functional modules for control hybrid systems.

Figure 10-1: Generalized functional modules for control hybrid systems.
10.2 MARTHA-I: 16-bit Microprocessor

10.2.1 Architecture

In this section, we present a 16-bit processing core version of the architecture, called MARTHA-I, shown in Figure 10-2. It comprises:

- one 16-bit programmable microprocessor,
- two digital signal processors (DSP),
- one memory controller for off-chip communication,
- one 16Kb direct-mapped cache,
- one 32Kb scratchpad memory,
- two 1Kb buffers, and
- one 8-bank main memory, with 1Mb per bank.

All the memory structures are parameterized: the number of banks, the bank size, the cache, the scratchpad, and the buffer. In this implementation, each buffer has four windows of four blocks each, where each block is a four 16-bit word line. Each network router has two virtual channels, each channel has four entries.

![Figure 10-2: MARTHA-I architecture.](image)

10.2.2 Hardware complexity and performance estimation

The architecture synthesis is done using the Heracles default design environment. The default FPGA board is the Xilinx Virtex-6 LX550T 1760 speed -2. The board has 687,360 total registers, 343,680 slice lookup tables (LUTs), 864 DSP48E1 25×18 Multipliers, and 632 block RAM with 36Kbits each. The synthesis resource utilization percentages are based on these FPGA resources.
Table 10.1: MARTH A-I FPGA resource utilization

<table>
<thead>
<tr>
<th></th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>692</td>
<td>0.1%</td>
</tr>
<tr>
<td>LUTs</td>
<td>2423</td>
<td>0.7%</td>
</tr>
<tr>
<td>Block RAMs</td>
<td>268</td>
<td>42%</td>
</tr>
<tr>
<td>DSP48E1 Blocks</td>
<td>8</td>
<td>0.9%</td>
</tr>
</tbody>
</table>

Table 10.1 shows the FPGA resource utilization for MARTH A-I. The DSP48E1 blocks are used from the DSP signal manipulations. The core is not pipelined and its speed of 84.6 MHz also constitutes the whole design speed. The block RAM utilization of 42% is a design choice.

Figure 10-3 shows the mapping of the boost converter application on MARTH A-I. The output DSP is used to generate the pulse-width modulation (PWM) signal needed to control the MOSFET switch. The 16-bit core emulates the circuit and compares its internal values with the measured values. The emulation and monitoring operations are used to maintain a constant output voltage of the system under control. The application specification switching frequency is 20MHz or 50\( \mu \) system-step latency.

The MARTH A-I architecture can only support simple power electronics circuits, like the boost converter. When the wind turbine application, presentation in chapter 3, is compiled onto the architecture, the program code severely stresses the memory. All the floating point and matrix multiplication operations need to be simulated at the software level. The
10.3 MARTHA-II: Single MIPS Core

10.3.1 Architecture

In the MARTHA-II architecture the 16-bit processing core is replaced by a 7-stage pipelined 32-bit MIPS core, shown in Figure 10-4. The system comprises:

- one 7-stage pipelined 32-bit MIPS core,
- two digital signal processors (DSP),
- one memory controller for off-chip communication,
- one 16Kb direct-mapped cache,
- one 32Kb scratchpad memory,
- two 1Kb buffers, and
- one 8-bank main memory, with 1Mb per bank.

The architecture is identical to MARTHA-I except that it has a more powerful processor core.

Figure 10-4: MARTHA-II architecture.

10.3.2 Hardware complexity and performance estimation

The MARTHA-II architecture runs at 172.02 MHz, which is the speed of the MIPS core. Table 10.2 shows the FPGA resource utilization of the architecture. For the boost converter application, both architectures are fast enough to meet the application required switching frequency. For the wind turbine application, MARTHA-II improves the system-step latency by 38.5%, running at 73.8 seconds.
Table 10.2: MARTHA-II FPGA resource utilization

<table>
<thead>
<tr>
<th></th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>2116</td>
<td>0.3%</td>
</tr>
<tr>
<td>LUTs</td>
<td>5527</td>
<td>1.6%</td>
</tr>
<tr>
<td>Block RAMs</td>
<td>268</td>
<td>42%</td>
</tr>
<tr>
<td>DSP48E1 Blocks</td>
<td>8</td>
<td>0.9%</td>
</tr>
</tbody>
</table>

10.4 MARTHA-III: Vector Machine

10.4.1 Architecture

Hybrid control systems, like power electronics applications, evaluate the continuous dynamics of the plant under control in the form of matrix computations. Vector-style processor cores can provide an efficient execution support for these matrix operations. In the MARTHA-III architecture, the processing core is a 4-lane vector core. The details of the core is presented in chapter 5. MARTHA-III comprises:

- one 4-lane MIPS-style vector core,
- two digital signal processors (DSP),
- one memory controller for off-chip communication,
- one 8-bank main memory, with 1Mb per bank.

Caches, scratchpads, and data buffers have been disabled. The vector core has an internal instruction buffer, and the data memory references are directed to the interleaved main memory, and they are managed by the memory load unit of the core. Figure 10-5 shows an illustration of the MARTHA-III architecture.

![MARTHA-III architecture](image)

Figure 10-5: MARTHA-III architecture.
10.4.2 Hardware complexity and performance estimation

Table 10.3 shows the FPGA resource utilization for *MARTHA-III*. The vector core makes use of the FPGA DSP48E1 multipliers for its lanes. The selection of the 4-lane for vector core is a compromise between the FPGA resource utilization and the matrix execution efficiency. The compromise is based on the study of the effect of vector lane number on execution time. Figure 10-6(a) shows the results of the study. The execution times are reported on a fixed matrix size of 256x256. The vector core and architecture run at 102.16 MHz.

<table>
<thead>
<tr>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>6984</td>
</tr>
<tr>
<td>LUTs</td>
<td>8024</td>
</tr>
<tr>
<td>Block RAMs</td>
<td>186</td>
</tr>
<tr>
<td>DSP48E1 Blocks</td>
<td>246</td>
</tr>
</tbody>
</table>

Table 10.3: *MARTHA-III* FPGA resource utilization

![Effect of Number of Vector lanes on 256x256 Matrix size](image1)

(a) Number of vector lanes vs. execution time.

![Effect of Matrix Size on Execution Time](image2)

(b) Matrix size vs. Execution time for 6-Lane Vector Core.

Figure 10-6: Vector core performance tuning.

Figure 10-6(b) highlights the impact of the matrix size, which grows with the complexity of a hybrid control system, on the execution time. Results are shown for a 6-lane vector core configuration using square matrices of size 1 through 1024. Given a system-step latency target (e.g., 1μs), state space matrices need to be bounded in size based on system resources. This in turn sets a limit on the allowable dynamic behavior complexity that can be supported effectively by a given system. For the wind turbine application, *MARTHA-III* has a system-step latency of 54 seconds. The lack of substantial improvement of the latency over the *MARTHA-II* architecture is due to the latency associated with scaler operations in the other part of the application.
10.5 MARTHA-IV: 2D Mesh of Eight MIPS Cores

10.5.1 Architecture

The MARTHA-IV architecture is a multicore architecture of eight homogenous MIPS cores, shown in Figure 10-7. The memory organization is non-uniform across all cores. The architecture comprises:

- eight 32-bit MIPS cores,
- two digital signal processors (DSP),
- two memory controllers for off-chip communication,
- three 4/8/16Kb direct-mapped caches,
- two 16/32Kb scratchpad memories,
- two 1Kb buffers, and
- one 8-bank main memory, with 1Mb per bank.

Figure 10-7: MARTHA-IV architecture.

10.5.2 Hardware complexity and performance estimation

The MARTHA-IV architecture uses most of the FPGA resources, except for the DSP48E1 blocks. Table 10.4 shows the FPGA resource utilization. Most of the reconfigurable fabric
Table 10.4: MARTHÄ-IV FPGA resource utilization

<table>
<thead>
<tr>
<th>Resource</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>465408</td>
<td>67.7%</td>
</tr>
<tr>
<td>LUTs</td>
<td>310244</td>
<td>90.3%</td>
</tr>
<tr>
<td>Block RAMs</td>
<td>630</td>
<td>99.7%</td>
</tr>
<tr>
<td>DSP48E1 Blocks</td>
<td>124</td>
<td>14.5%</td>
</tr>
</tbody>
</table>

is used to realize the hybrid memory organization and the network routers. Depending on the control application, some the storage structures can be disabled or reduced in size. Hybrid control applications exhibit two types of parallelism, namely, task parallelism and data parallelism.

The MARTHÄ-IV architecture lends itself well to task parallelism by decoupling hard real-time control functions, monitoring tasks, and external control interactions and executing them on different MIPS cores. On the other hand, for the data parallelism part, the architecture needs to simulate SIMD operations across multiple MIPS cores. This leads to larger application program code and a more congested network. It does not perform well when evaluating continuous system dynamics. For the wind turbine application, MARTHÄ-IV provides a system-step latency of 45 seconds.

10.6 MARTHA-V: Heterogeneous-I

Recall that in chapter 3, a simple mapping of function to processor-memory types for the control modules in Figure 10-1 yields a heterogenous SoC with the hardware characteristics shown in Figure 10-8. The MARTHÄ-V architecture is the implementation of the SoC.

![Figure 10-8: MARTHÄ-V architecture.](image-url)
10.6.1 Architecture

The *MARTHA-V* architecture is a heterogenous, multicore architecture, shown in Figure 10-9. It comprises:

- one 4-lane MIPS-style vector core,
- two 32-bit MIPS cores,
- one 16-bit programmable microprocessor,
- two digital signal processors (DSP),
- two memory controller for off-chip communication,
- one 16Kb direct-mapped cache,
- one 32Kb scratchpad memory,
- three 1Kb buffers, and
- one 8-bank main memory, with 1Mb per bank.

![Figure 10-9: MARTHA-V architecture.](image-url)
10.6.2 Hardware complexity and performance estimation

Table 10.5 shows the FPGA resource utilization. The system-step latency for the wind turbine drops to 3.4μs. The MARTHA-V architecture has the heterogeneity to properly take advantage of both task and data parallelism. The system mode computation, which is a hard real-time task, is placed on the cacheless MIPS core. This core is connected to the scratchpad memory, where both instruction and data for the task are placed. The memory references are fixed and low latency. The best-effort application modules, like the monitoring tasks, are mapped onto the MIPS core with the cache structure. The matrix-vector manipulations for the continuous dynamics are done on the vector core. The mode matrices data is striped across the different banks of the main memory.

Figure 10-10: Mapping of the model selector onto the MARTHA-V architecture.

Figure 10-10 is the mapping of the model selector tasks of the adaptive hybrid automaton (c.f Chapter 4) onto the architecture. Figure 10-11 illustrates the scheduling of the continuous dynamic evaluations onto the vector core.
10.7 MARTHA-VI: Heterogeneous-II

10.7.1 Architecture

The MARTHA-VI is a larger version of the heterogeneous MARTHA-V architecture. It is the largest system we place on the FPGA board. It is shown in Figure 10-12 and comprises:

- one 8-bit programmable microprocessor,
- one 16-bit programmable microprocessor,
- two 32-bit MIPS cores,
- two 4-lane MIPS-style vector cores,
- two digital signal processors (DSP),
- two memory controllers for off-chip communication,
- one 64Kb direct-mapped cache,
- two 16/32Kb scratchpad memories,
- three 1Kb buffers, and
- one 8-bank main memory, with 1Mb per bank.
Table 10.6: *MARTHA-VI* FPGA resource utilization

<table>
<thead>
<tr>
<th>Resource</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>681240</td>
<td>99.1%</td>
</tr>
<tr>
<td>LUTs</td>
<td>342108</td>
<td>99.5%</td>
</tr>
<tr>
<td>Block RAM</td>
<td>630</td>
<td>99.7%</td>
</tr>
<tr>
<td>DSP48E1 Block</td>
<td>632</td>
<td>73.1%</td>
</tr>
</tbody>
</table>

### 10.7.2 Hardware complexity and performance estimation

Table 10.6 shows the FPGA resource utilization for the *MARTHA-VI*. The processing cores only consume a small fraction of the resources. The memory structures and the logic around them use the bulk of the FPGA resources. The network routers, although there are only two virtual channels per physical link with our entries per channel, still have considerable FPGA resource utilization footprint. The *MARTHA-VI* architecture has a clocking speed of 102.16 MHz, which is essentially the vector core speed. The system-step latency for the wind turbine drops is registered at 2.1 μs.

The improvement in the system-step latency comes from the fact that the *MARTHA-VI* architecture has two vector cores. The continuous dynamic evaluations are mapped onto the vector cores in a way that reduces the latency associated with the switching from one discrete mode to another. In this configuration, the data associated with the previous mode can be lazily stored while the current mode’s continuous dynamic operations are executing on the other vector core. Figure 10-13 illustrates this mode mapping.
Figure 10-13: Mapping of the continuous dynamic operation onto the MARTHA-VI architecture.

Figure 10-14: Mapping of sub-circuits onto the MARTHA-VI architecture.
In Figure 10-13, during any given system-step, only one of the vector cores is computing on the active mode continuous dynamic. Another alternative is to partition the system into sub-systems and to use the vector cores to evaluate them in parallel. Figure 10-14 shows this configuration for the wind turbine application. The system-step latency does improve slightly to 1.9μs, but the circuit partitioning and the application mapping are more complex. In general, having multiple vector cores enlarges the domain of power electronics applications that the architecture can support.

10.8 Summary

In this chapter, a family of MARTHA architectures are presented. Each architecture has different computation engines and hardware resource requirements, and supports a given range of hybrid control applications. The MARTHA architecture framework is based on modularity and promotes scalability. This allows for different instances of the architecture for different domains or scales of hybrid applications. The heterogeneity of the architecture provides targeted support to the different computation demands and more effective adaptation to application diversity.
Chapter 11

Results and Analysis

In this chapter, we present the results of the experiments performed to validate our design framework. Using the power electronics control applications presented in Chapter 3, the performance of different architectures, including an off-the-shelf general-purpose multicore system, is evaluated and analyzed. The FPGA-based prototype of the MARTHA-VI architecture (cf. chapter 10) is deployed and tested in an actual physical power electronics system, where measurements are taken and compared against different state-of-the-art, commercially available, power electronics controllers.

11.1 Introduction

Computer architectures for future hybrid control applications need to equally guarantee performance, predictability, and programmability. Power electronics applications, for example, require: low to high-performance processing elements, low-latency system responses, predictable execution, and a programmable computation environment.

Here, we evaluate and compare several computer architectures, including the MARTHA architectures, to determine their affinities to the computational needs of power control applications. The analysis of the various architectures focuses on four key areas:

1. computation power, which affects the system-step latency and allowable system dynamic behavior complexity;

2. memory, and its effects on the system-step latency and data access time predictability;

3. on-chip communication, which influences the deterministic aspect of data transfers; and

4. I/O signal processing, which affects the system-step latency of the control system.

The three representative power electronics applications used in this chapter to evaluate and analyze the architectures are:
• variable speed induction motor drive,
• utility grid connected photovoltaic converter system, and
• hybrid electric vehicle motor drive.

The switching frequencies of semi-conductor devices in these applications require a control system-step of the order of 1μs. For each application, the same model is executed on all the architectures. This approach allows for a direct, comprehensive scheme for comparing the aggregate performance of the different architectures by simply measuring their system-step latencies.

11.2 System Evaluations

The performance, in terms of system-step latency, of the MARTHA architectures is compared to the baseline multicore commodity computation platform, namely, the Intel Xeon 5600 series. We also examine the potential performance of the IBM Cell [138] and the TI-C66 [139] for these power electronics applications.

11.2.1 MARTHA architectures

From the MARTHA family of architectures (cf. Chapter 10), four architectures are used for the direct comparison with other architectures. The four instances of the architecture are:

1. MARTHA-II composed of a single MIPS core;
2. MARTHA-IV, a homogenous mesh network of eight MIPS Cores;
3. MARTHA-V, a heterogenous architecture with a MIPS core and vector core; and
4. MARTHA-VI, a heterogenous architecture with multiple MIPS and vector cores.

11.2.2 General-purpose central processing unit (CPU)

The commodity CPU used for the system evaluation is the Intel Xeon 5600 series, 32nm, 6-Core server processor. It uses the x86 ISA and targets workstation, server, and embedded system workloads. The CPU has a top execution speed of 2.93 GHz. Each core has its own 256KB L2 cache and up to 12MB of shared L3 cache that can be accessed by all processor cores. Intel QuickPath Interconnect, which is a point-to-point processor interconnect, is used for the inter-core data communications. The architecture has streaming SIMD extensions. For the benchmark applications, no specific compiler optimization is performed and the SIMD extension utilization is left to the operating system scheduler to manage. The results presented on this architecture are for software-only implementation of the applications.
11.2.3 System-step latency comparisons

To execute the benchmarks on the various hardware systems, we use the *Heracles* (cf. Chapter 9) GCC MIPS cross-compiler environment for the *MARTHA* architectures, and use the POSIX threads (pthreads) library for the CPU version of the applications. Figure 11-1 shows the end-to-end *system-step* latency per benchmark per platform. These *system-step* latencies are the worst-case latencies. On the CPU, all benchmarks are run on the native operating system and the average execution time is 783µs. The CPU memory hierarchy proves to be a bottleneck, caches perform poorly due to thrashing, and a substantial amount of time is spent on DMA operations during discrete state switching.

![Figure 11-1: End-to-end latency comparison on all five systems for the three benchmarks.](image)

The *MARTHA-II* architecture lacks the computation power to effectively execute the matrix manipulations involved in continuous state evaluations. Its average *system-step* is 6770µs. Having multiple homogenous MIPS considerably improves the system performance in terms of *system-step* latency. Execution time drops from 6770µs to 838µs on average, highlighting the amount of parallelism that can be exploited in the benchmarks. The system performance improves the most when we take full advantage of the key characteristics of these applications, by using dedicated processing structures for a given task. The average *system-step* latency of *MARTHA-V* and *MARTHA-VI* are 2 and 1.4 µs, respectively.

![Figure 11-2: Best-case end-to-end latency comparison on all five systems for the three benchmarks.](image)

Figure 11-2 depicts the best-case end-to-end *system-step* latency for the different systems. This figure is shown for completeness. The best-case latency happens when the
hybrid application stays in the same continuous mode for an extended period and there is no I/O signal change. In general, there is less data movement, and the cache locality of the data helps powerful processors perform extremely well. In these regimes, the CPU shows an end-to-end system-step latency that is under 1µs and outperforms both MARTHA systems. Unfortunately, for these hard real-time applications, the worse-case end-to-end system-step latency, Figure 11-1, is the relevant system performance metric.

Figure 11-3: End-to-end latency comparison on all five systems for the Motor Drive benchmark.

Figure 11-4: End-to-end latency comparison on all five systems for the Utility Grid Connected benchmark.

Figure 11-5: End-to-end latency comparison on all five systems for the Hybrid Electric Vehicle benchmark.

Figures 11-3 through 11-5 show the execution time breakdown that sums up the end-to-
end latency, for all three benchmarks. This provides some valuable insight into the runtime execution behavior of these systems which can lead to further submodule optimizations. It is worth noting that these execution time breakdowns are susceptible to small error due to measurement artifacts (it is, in some cases, hard to count a given system cycle under a particular region, e.g., computation, or only one). In the case of the CPU measurements, we use the Intel PIN tool [140] to instrument the benchmark code. Due to our lack of visibility into the system on-chip communication, we just characterize the CPU performance along computation, memory, and I/O communications.

The MARTHA-V and MARTHA-VI architectures, being designed specially for these type of hybrid control applications, have many advantages over the homogeneous and general-purpose architectures. Some of these advantages are smaller code size, fewer and very predictable memory accesses, and concurrent accesses to different memory banks. The vector core, with its large register set and heterogenous lanes for fast multiply-and-accumulate operations with forwarding capability, helps to evaluate system dynamics and avoid the large number of memory accesses associated with intermediate data. The memory layout for matrix data is aggressively optimized for all the MARTHA architectures.

The percentage of execution time spent in each region is fairly constant across all benchmarks as shown in Figure 11-6. For the MARTHA-II architecture, only 2% of the execution time is spent on I/O operations, most of the system time is spent in computation, at 87%. In the case of the MARTHA-IV architecture, more parallel execution of application tasks stresses the memory subsystem to a great degree. The memory operations consume 37% of the total system-step execution time compared to the 11% seen with the MARTHA-II system.

The trend is similar for MARTHA-IV and MARTHA-V. The increase in computation power, through the addition of processing units, stresses the memory bandwidth more, while improving overall system-step latency. For the MARTHA-V and MARTHA-VI architectures, memory operations count for 52% and 63% of the execution time, respectively. The CPU results show the extreme case where the memory bandwidth and hierarchy fail to keep up with the computation power therefore limiting the whole system. Up to 68% of execution time is in memory access and operating system bookkeeping.

11.3 Other High-Performance Computer Systems

The MARTHA architecture in many aspects is not as flexible as a pure general-purpose architecture. For this reason, we expand our system evaluation to consider other high-performance and domain-specific architectures.
11.3.1 Evaluation of the IBM cell processor for hybrid control applications

IBM CBE (Cell Broadband Engine) is a multiprocessor system dedicated to distributed data processing. It has four different processing elements: PowerPC Processing Element (PPE), array of Synergetic Processing Element (SPE), Memory Flow Controller (MFC), Internal Interrupt Controller (IIC). The functional elements are connected via the EIB (Element Interconnect Bus). PPE and SPE elements are the main processing units. SPE element has a special vector processor (SPU, Synergetic Processor Unit) capable of executing SIMD instructions. It has 128 128-bit wide registers capable of storing multiple scalar data types. The cell processor is currently available in two forms, as part of Sony’s gaming console (Sony PS3) and part of IBM’s QS Cell BladeCenter mainframe. In this work, we use the Sony PS3 platform running a Linux operating system.

Figure 11-7 shows the IBM architecture. It is a heterogenous architecture that shares some of the characteristics of the MARTHA architecture. Instead of directly pouring the power electronics benchmarks onto the cell architecture, the testing is done in a more targeted fashion. As first presented in chapter 4, the system states are evaluated through matrix manipulations of the form:

\[
x((k+1)\tau) = \begin{bmatrix} A_d(\tau) & B_d(\tau) \\ C_d & D_d \end{bmatrix} \begin{bmatrix} x(k\tau) \\ u(k\tau) \end{bmatrix}
\]

Where \(x\) is the state space vector, \(u\) is the input vector, and \(A_d, B_d, C_d,\) and \(D_d\) are system matrices. Our first goal is to examine the matrix calculations. All data types are floating point (32 bit wide). This means that an SPU processor can process four values with one instruction. To simplify the evaluation, we focus solely on the calculation state space vector \(x\).

Input data, matrix \(A_d, B_d\), vector \(x, u\) data, are loaded onto the system as input files. After the calculation, the new values of \(x\) are stored into an output file. An SPU processor
only sees its cache memory (256 KB of cache memory). To get the data needed for a calculation, the SPU has to initiate a special direct memory access (DMA) transfer that will copy data from main storage to its local cache. This means that the SPU processor has a copy of matrix $A_d$, $B_d$ and vector $u_i$, where $i$ is the current iteration, since theoretically, data for vector $u$ may not fit into local storage.

The memory access of the SPU processor is 128-bit wide, therefore, with one load or store instruction, the processor can access 128 bits of memory. In order to fully exploit the fact that memory access is 128-bit wide, four output values are calculated in parallel. This is done by unrolling the matrix multiplication loop. This scheme essentially guarantees that the full potential of SIMD instructions is used.

The results show that an SPU processor element needs 3810 cycles to calculate one $x$ output vector when matrix $A_d$ has size of $32 \times 32$ elements, matrix $B_d$ has size of $16 \times 32$ elements, vector $Uu$ has size of 16 elements, and vector $x$ has size of 32 elements. Since each SPU processor runs at 3.2 GHz, one iteration takes 1.19 $\mu$s. Although the matrix-vector evaluations are the most compute intensive, they only represent one operation in the series of operations that must be performed in one system-step with the expected latency of 1 $\mu$s.

Even if one leaves out the mode selection phase of the system-step, the input and output processing phases remain a challenge since the cell processor has no direct support for signal processing. An FPGA or a DSP connected through the Flex I/O interface can be used for the input and output phases. In this work, we use Analog Devices Blackfin 32-bit DSP for the input and output processing. Data transfers across the two boards (DSP-CELL) to the
PPE, and when measured in a simple loop-back test on a single analog signal, are in the order of 385-526 μs, captured with the oscilloscope.

### 11.3.2 Nvidia GTX 295 graphics processing unit (GPU)

Another commercially available high-performance architecture considered here is the Nvidia GTX 295 GPU. This architecture has 480 processor cores with a theoretical single precision compute power of 1192.32 GFLOPS. The memory size is 2 × 896 MB, the bandwidth is 2 × 111.89 GB/s with an interface width of 2 × 448-bits. Our tests show that the system can deliver 820 GFLOPS when computing on the matrices of the emulator model with the input data present in memory. Unfortunately, the GPU is not flexible enough to perform well across all the stages of the system-step evaluation. Beside the matrix operations in the emulator unit, there are no other obvious SIMD-friendly tasks in the applications. Most of the parallelism in the application is task-based parallelism. Even in the emulator unit, the number of matrix operations in the different modes is too small when compared to the GPU capability.

![Figure 11-8: NVIDIA GeForce 8800 GTX GPU architecture (source: NVIDIA).](image)

Getting data onto the chip and close to the processor to take advantage of their computation power has proven to be difficult. The architecture has a PCI-Express bus for interface. When connected to the Spartan-6 FPGA for input/output signal processing, the
scheduler on the GPU side has a hard time managing this type of data. The latency for a simple loop-back test on a single analog signal is in the order of millisecond. The GPU architecture lacks the flexibility, the fine-grained synchronization, and the heterogeneity to lend itself to these hybrid control applications as a viable compute platform. The architecture is optimized for maximum throughput when running graphics applications. Figure 11-8, for example, shows the NVIDIA GeForce 8800 GTX GPU architecture. The computations in control applications are simply too diverse for these massively parallel architectures. In our experiments, this type of architecture is best suited for high-latency high-throughput applications, and not for low-latency, low to medium-throughput applications like power electronics controls.

11.3.3 Texas Instruments (TI) C66

The TMS320C6678 Multicore Fixed and Floating Point Digital Signal Processor is based on TI's KeyStone multicore architecture. It has eight C66x CorePac DSPs; each core runs at 1.0 to 10.0 GHz. This architecture is designed to support high-performance signal processing applications. Each core has 32KB of L1 program cache, 32KB L1 data cache, and 512KB of L2 unified cache. The system has a 4MB on-chip share memory. Figure 11-9 shows the TI TMS320C6678 Multicore architecture. The DSPs run a lightweight real time native operating system called SYS/BIOS. The compiler provided with the platform supports OpenMP 3.0, which is used to do the matrix evaluation and data transfer rate on the system.

Our results show that for small-sized matrices (less than $256 \times 256$), the system performs poorly even with a good layout of data in the memory. But for larger sized matrices, the execution of single precision matrices is around 64.2 GFLOPS when using all eight cores. Overall, the system has enough computation power from the matrix operations in the emulator. The memory organization and management are simple enough to enforce predictable memory references. Getting data in and out of the system has long latencies, but further I/O interface design effort may alleviate some of the latency. The architecture lacks direct support for general, best-effort computation. This last issue is addressed with TI's release of the KeyStone II system-on-chip architecture.

Figure 11-9 shows the KeyStone II heterogenous multicore architecture. It has four RISC ARM cores running at 5.6GHz and a DSP core running at 9.6GHz. This architecture is particularly interesting because it has many of the architectural characteristics of MARTHA. Although the memory management and the interconnect do not provide some of the key guarantees seen in the MARTHA architecture, the simplicity of the cores and the heterogeneity follows the same design principles. Due to the available software support, the testing and comparative study of the KeyStone II architecture could not be fully done and included in this dissertation.
11.4 Physical Deployment of the MARTHA Architecture

In this section, we present the measurements when the MARTHA-V architecture is used to control an industrial 186 Watt Leeson three-phase, four-pole, squirrel-cage, induction machine system. We also show some current and voltage measurements for the other benchmarks.

11.4.1 Industrial three-phase induction motor control

The Leeson three-phase induction motor is used in many applications, from compressors to conveyors. The model KXRMT in the experiment operates at 1/6 to 400 horsepower with 3600 revolutions per minute (RPM). Figure 11-11 shows the experimental setup, where the same control unit in MARTHA is used for both the real system and the emulator reference model. We test the fidelity of the emulator model to the actual physical power electronics system. Measured waveforms on both the real system and emulator on MARTHA have an almost one-to-one matching. Figures 11-12 and 11-13 show the side by side reading of the emulator and the physical system for the line-to-line inverter voltage and the motor phase current, respectively. Emulator response latency is measured to be less than (1.3μs). Figure 11-14 shows the recording on the oscilloscope.
11.4.2 Emulator current and voltage measurements for the benchmarks

Figure 11-15 shows the measurements for the variable speed induction motor drive. Channels 1 and 3 are the inverter output current and line-to-line motor voltage. They are the same signals as on channel 1 in Figures 11-12 and 11-13. The signal on channel 3 is the stator current.

Figure 11-16 shows measurements of the hard real-time emulator model waveforms, voltages at the three-Level inverter, shown on channel 1-2, and current at utility grid, shown on channel 3-4.

Figure 11-17 shows the DC link current, the rotor current, and the inverter output line-to-line voltage for the hybrid electric vehicle motor drive system, with switching frequency of 4 kHz.

11.4.3 Comparing MARTHA to commercially available power electronics emulators

Table 11.1 compares the simulation time step, which includes input/output latency, computation time, and A-to-D and/or D-to-A conversion, of three real-time hardware digital emulators with our proposed platform. Note that OPAL's eDriveSim is an electric car only
11.5 Discussions

These performance results show that power electronics systems require different architectural mechanisms than those provided by commodity CPUs or GPUs. Power electronics applications exhibit a high degree of task parallelism and a mixture of hard real-time and best-effort tasks. For these hybrid control applications, massively parallel architectures, like GPUs, or high-frequency architectures, like multicore CPUs, expose further the disparity of speed between the processing elements and the memory organization supporting these processing units.

Furthermore, these applications are interacting with physical events, and some of the data needed in their computation are not resident in any particular memory location, and...
must be continuously collected from the physical environment. This characteristic of the application makes, on one end, the processing unit execution efficiency dependent on the memory access latency, and on the other end, the memory itself subject to the I/O processing latency.

With the MARTH A-II architecture, time multiplexing the different tasks on a single core and using memory to store task contexts, although functional, performs very poorly because it removes all parallelism from the application. The MARTH A-IV architecture solves the performance bottleneck associated with the time multiplexing of the tasks as in the MARTH A-II case. Tasks are decomposed and executed on different cores. The cores are homogeneous therefore making the mapping and software support effort simple. The key drawback of this architecture is the homogeneity of the core because different tasks have different computation and performance requirements.

The performance results of the MARTH A-IV architecture highlight the limitation of pure parallelism and multithreaded implementations. A heterogeneous multicore architecture, like MARTH A-V or MARTH A-VI, can exploit the power electronics application task parallelism and can accelerate the matrix-vector multiplication in the emulator to meet the low latency requirement of the control.

The Intel CPU is added for two main reasons: (1) to verify that there is a need for a
domain-specific architecture; (2) for completeness of the study. The CPU performs poorly for two reasons: (1) the memory organization is not favorable to the dynamic mode changes in these applications; and (2) the complexity in the hardware and software stack makes the system response time both slow and unpredictable. Since we know that the CPU has the raw power to perform well when the control system is in a steady state, a more specialized I/O board and slimmed-down operating system will improve its performance results.

The general observation with GPUs and the IBM Cell is that they can all do the matrix computation part in the emulator very fast, but their lack of flexible and fast inter-task communication makes other data communication (e.g., mode selection) slow. The TI SoC architecture shares MARTHA’s processing element combination. But scalability and predictability at all levels, including memory access and on-chip interconnect, are not addressed.

We did not do a direct comparison with real-time oriented architectures such as PRET [44] and VISA [46]. This is because there is no available register-transfer level (RTL) code. We can qualitatively argue their differences. For example, PRET shares the timing predictability property of MARTHA, but PRET’s focus is on executing multiple hardware contexts on a shared hardware platform. By having multiple interleaved hardware threads, the timing requirement on each thread gets relaxed enough to be predictable. Similarly, VISA provides, like most real-time frameworks, some timing predictability at the expense of performance. The goal in MARTHA, due the application domain requirements, is to deliver programmability and time predictability while maintaining high-performance and low-latency.
Figure 11-15: Current and voltage measurements for the variable speed induction motor drive system.

Figure 11-16: Current and voltage measurements for the utility grid connected photovoltaic converter system.

Figure 11-17: Current and voltage measurements for the hybrid electric vehicle motor drive system.
11.6 Summary

In summary, we evaluate the performance of the power electronics benchmarks on different computation platforms, namely, commodity CPU, and a set of MARTHA architectures. The MARTHA-VI, prototyped on an FPGA, is deployed and tested in a physical environment. It enables a high-fidelity (with 1μs latency and emulation time-step), safe, and fully realistic testing and validation of detailed aspects of power electronics systems. Evaluation results show that the proposed MARTHA architecture guarantees an end-to-end latency of 1μs for our benchmarks; at least a 10× better performance compared any commercial platform. To the best of our knowledge, no current academic or industrial control system, for hybrid systems, or power electronics systems, has such a fast emulation response time.
Chapter 12

Conclusion

This dissertation proposes and develops MARTHA (Multicore Architecture for Real-Time Hybrid Applications), a heterogeneous, parameterized, multi/many-core architecture, designed to meet the high computation throughput, deterministic, and low-latency requirements of a spectrum of hybrid control systems, and especially, future high-efficiency power electronics applications (e.g., smart grids, automotive electrical systems, process industry power supplies).

We developed a generalized, mixed-model representation for hybrid control systems; we derive a common abstraction and programming model for these systems, and we used the proposed architecture for their efficient execution. Evaluation results show that the MARTHA architecture guarantees an end-to-end latency of at least 10× better than any commercial available platform.

12.1 Computation Problem in Hybrid Control Applications

The need for high-efficiency, automated systems coupled with recent advances in computer technology has given rise to complex control hardware. Examples of these systems range from small-scale and safety critical, e.g., a pacemaker controller, to large-scale and distributed, e.g., automotive, aerospace and power grids. Applications in this domain can vary widely from one to another. An important subclass of these applications is the hybrid control system domain.

Hybrid control systems are characterized by the interaction of continuous dynamics, expressed as differential or difference equations, and by discrete events, best described by finite state machines. It is a characterization that takes into account both the changing phases and the natural continuous progression in the physical plants under their control. Hybrid control systems encompass a large domain of applications. Prior work in the domain has shown how difficult it is to find one single formulation, computation model, or hardware platform for all hybrid systems.
Our general approach in this dissertation, is to make the problem more tractable by focusing on one key representative application subdomain, namely, power electronics applications. Our motivation in using power electronics benchmarks stems from the fact they possess many characteristics common to most hybrid control systems and good benchmarking metrics:

1. Large domain of application: ventilation, air conditioning, lighting system, healthcare, automotive, motor controls, and drives, just to name few;

2. Efficiency and reliability are important: they are time critical, they deal with high, sometimes extreme, currents, voltages, frequencies, temperatures, and pressures;

3. Scalability of design methodologies and architecture can be easily tested: one can make an application execution context larger by adding more circuit components (e.g., more filtering stages);

4. Effects of spatial and temporal localities can be easily isolated;

5. Effects of I/O processing speed on the overall architecture performance can be exposed;

6. An architecture’s ability to fully exploit both the data or task parallelism in the application.

The design, verification, and deployment of hybrid control systems, in general, and power electronics applications, in particular, require a tightly-coupled software and hardware environment. This arises from the fact that they are reactive, hardware-in-the-loop systems. Their computational requirements can be summarized as follows:

1. Deterministic, low-latency execution, for fast system responses to minimize control error and to avoid control and physical plant divergences;

2. High-fidelity control reference models, for high-efficiency control algorithms, therefore high-performance compute engines;

3. Support for both continuous and discrete processes, and both linear and non-linear systems, therefore, a highly flexible processing unit or a multi-processor architecture;

4. Support for both hard real-time computation (predictable execution, deterministic data accesses and on-chip communications) and best-effort computation for monitoring and distribution of control in large-scale systems;

5. Programmable hardware modules to support different control algorithms.
Generic general-purpose, including GPU, architectures even with the use of real-time operating systems fail to meet the hard real-time constraints of hybrid system dynamics. This is because they lack fine-grained execution predictability, due to their complex hardware structure (multi-level caches, TLBs, etc), and their complicated software stack. The execution results for the power electronics applications on the Intel Xeon 5600 series 6-Core processor expose these drawbacks.

The use of a real-time operating system (RTOS) and worst-case execution time (WCET) analysis on these computer systems alleviate some of their unpredictability at the expense of control efficiency. WCET forces the physical system to be over-engineered by adding redundant or larger, slower components to compensate for the long hardware response time. In power electronics applications, for example, larger capacitors are often used to slowdown the switching frequency of a system and to make the system more controllable through RTOS and WCET analysis.

A traditional ASIC design approach, which derives the computer architecture directly out of the application specification, has led to very specialized systems-on-chip (SoCs) that are difficult to reprogram, reuse, or maintain. Similarly, past FPGA approaches have not proven successful, because they lack a clear and generalized design methodology for this domain of applications. Design and resynthesis efforts frustrate control designers, and often their single or Multi-FPGA solutions lead to complicated, low-performing systems.

12.2 Thesis Contributions

The objectives of the dissertation are to define a generalized computation model and a suitable computer architecture for future, high-performance, hybrid control systems, in general, and power electronics applications, in particular. To that end, we develop a small but representative set of large-scale power electronics applications, namely, a utility grid connected wind turbine converter system, a variable speed induction motor drive, a utility grid connected photovoltaic converter system, and a hybrid electric vehicle motor drive. We use these applications to derive a set of quantitative system specifications and performance requirements. We plan to release the applications as open-source benchmarks for evaluating future architectures or software targeting large-scale, hard real-time applications.

We develop a unifying abstraction for these applications that can be generalized to other power electronics and control applications, called adaptive hybrid automaton (AHA). The AHA formulation is based on prior work on control systems, hybrid controls, switched automata, dynamical systems, and power electronics circuits modeling and analysis. This allows us to not have to treat many of the formulation properties, assumptions, and proofs in complete detail. The AHA formulation is used to derive the Multi-Program Application (MPA) computational model used in the rest of the application compilation process. Using
the MPA model, applications, represented as AHA, can be reduced to efficient, hardware, execution forms, while they retain their dynamic properties and high-level specifications. These properties and specifications are used to guide the application task decomposition and scheduling, data placement, and runtime execution management.

The introduction of the AHA formulation and the MPA computation model is meant to avoid some of the pitfalls associated with the direct ASIC-based design approach, by providing an intermediate application representation before hardware mapping and execution. It makes the targeted architecture less application-specific and more domain-specific. For the hybrid control domain, a candidate architecture must guarantee an efficient execution of all applications that can be reduced to the MPA computation model, regardless of their high-level application domain. The MARTHA architecture is a novel, parameterized, heterogenous, multicore system that provides such a guarantee.

It is worth noting that the MPA computational model represents a new compute paradigm. One that sees an application as a set of programs in continuous interactions. Each program represents a mode in the application. Since the application modes can be vastly different, in terms of dynamic behavior, the associated programs can also be vastly different, in terms of working data sizes, data types, data and program structures. In fact, the MPA computation model commands a compute architecture where the heterogeneity of the processing elements is not merely driven by performance optimization, as often seen with accelerators, or fault, power, or technology driven, like we see in multi-frequency multicore systems.

The MARTHA architecture features general-purpose MIPS-core(s), used for mode transitions, controls, monitoring, and data collection; SIMD vector machine style core(s), with heterogenous, parameterized lanes, optimized for fast multiple-and-accumulate (MAC) operations, used to model linear dynamics of a hybrid application through matrix manipulation operations; DSP core(s), used for I/O conditioning, analog/digital communication, and for non-linear machine models; and programmable microcontrollers, employed to model certain signal sources.

All the cores are made simple for fast and predictable execution, even the vector core has a RISC-style architecture. They all are based on the same MIPS ISA. This approach keeps the compiler and the software support simple and manageable. The memory in MARTHA is hybrid, partly transparent, with a multi-namespace organization. Data placements and accesses are done on a per task basis, using main memory, caches, queues, and scratchpad memories, depending on the task hard real-time requirements and working set size.

For scalability and concurrency of on-chip communication, the MARTHA architecture uses a hybrid datapath router approach. It is a low hardware overhead, hard real-time aware interconnect router where the hard real-time traffic is decoupled from the best-effort traffic. An automated compilation tool is developed to assist in the various application transformations and hardware mappings. We present a framework for decomposing MPA
computation model into a finer-grained task-based application representation for easier scheduling and to explore more concurrent task execution.

We develop a generalized integer linear programming (ILP) formulation and various heuristics for scheduling a task-based application decomposition onto a heterogeneous many-core architecture. We create a deadlock-free real-time aware bandwidth sensitive oblivious routing (RT-BSOR) algorithm to direct on-chip communication. The MARTHA architecture design exploration is enabled by the creation of the Heracles tool, which is a platform for fast RTL-based design space exploration for future multi/many-core architectures.

An FPGA-based prototype of the MARTHA architecture is built, deployed, and tested in actual physical power electronics systems. This is an important outcome of this work, since the main approach in the thesis is to focus on a small subset of hybrid control systems, namely, power electronics applications, to re-examine the full architecture stack (core, memory, and network-on-chip), and to develop a functional computer system where performance, predictability, and programmability are all equally enforced. This is in contrast to the approach where the focus is in a single area (e.g., core level) with an attempt to cover all hybrid control applications.

12.3 Directions for Future Research

Hybrid control applications have generally fallen outside the general-purpose computation domain, and have been dealt with, on a case-by-case basis, via ASICs or embedded systems. The results from the testing and deployment of the MARTHA architecture show that a large set of these applications can be modeled under a unique computational abstraction and executed efficiently on a domain-specific architecture.

The MARTHA architecture is only one step in the search for a generalized hybrid control application architecture. More research effort is needed to refine and expand the main results and conclusions of this dissertation work. For example, when software support becomes available and stable, the full system evaluation and testing of the Texas Instruments KeyStone II heterogenous multicore architecture will provide some valuable insights into the class of architectures best suited for these applications. Another research direction consists of reevaluating the general-purpose architecture without the complex operating system stack where some of the techniques developed for MARTHA can be applied (e.g., static memory access analysis and window-based prefetching).

Also, the robustness of the AHA formulation, and MPA computation model need to be tested by expanding the class of hybrid control applications we can currently execute on the MARTHA architecture. One such class of applications is unmanned aerial vehicle (UAV) control applications. In the UAV, we have the autopilot, which is responsible for the high-level navigation algorithm; sensors for position measurements (speed, latitude, longi-
Figure 12-1: Illustration of a unmanned aerial vehicle (UAV) flight modes.

titude, and altitude) and attitude measurements (heading, roll, and pitch); and a reference trajectory. Evaluation of the reference trajectory is based on the mathematical model of the airframe using state space representation or transfer function. The UAV, as described, shares with power electronics, and many other hybrid control applications, the same basic functional blocks. Figure 12-1 shows an illustration of possible modes during a UAV flight session. It would be interesting exercise to develop a full system specification with help from a UAV domain expert and to run the application through the MARTHA toolchain.
Bibliography


188


[133] Kermin Elliott Fleming, Michael Adler, Michael Pellauer, Angshuman Parashar, Arvind Mithal, and Joel Emer. Leveraging latency-insensitivity to ease multiple fpga


[142] dSPACE Inc. dspace launches world's fastest hil simulation technology.

