2015

Design exploration of hardware accelerators for mitigating the effects of computational delay on digital control loops

Sudhanshu Prasad Vyas
Iowa State University

Design exploration of hardware accelerators for mitigating the effects of computational delay on digital control loops

by

Sudhanshu Prasad Vyas

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Electrical and Computer Engineering

Program of Study Committee:

Phillip H. Jones III, Co-major Professor

Joseph Zambreno, Co-major Professor

Arun Somani

Nicola Elia

Nathan Neihart

Iowa State University

Ames, Iowa

2015

DEDICATION

This thesis is dedicated to

the people whose labor allows me to pursue my goals

and

to the person who never saw the final fruits of her labor

my mother, Mrs. Savitri Vyas

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

First, thanks to my adviser, Dr. Phillip H. Jones III for giving me a chance. Rarely does one get a professor who allows their student to choose their own topic. Dr. Nicola Elia and Dr. Nathan Neihart for being on my committee and providing meaningful feedback. Dr. Joseph Zambreno for his timely advice and words of encouragement. Dr. Arun Somani for his advice in both technical and personal issues. Last, but not least, my father and his father, Dr. Hanuman Prasad Vyas and Dr. Madho Das Vyas, for being my source of inspiration and motivation.

This work was partly funded by the [NEED INPUTS FROM DR.JONES].

ABSTRACT

The field of modern control theory and the systems used to implement these controllers have developed rapidly and mostly exclusive of each other over the last 50 years. Digital control systems are traditionally designed assuming constant sensor sampling-rates and consistent processor response-times, with their implementation platform unaccounted for. Concurrently, embedded systems engineers focus on maximizing resource utilization by sharing processors amongst control and non-control tasks, causing unintended interactions. The result of this isolation between the two fields is that computing mechanisms meant to improve average CPU throughput, such as cache, interrupts, and task scheduling by operating systems, are contributing to this non-deterministic and unaccounted delays in the control loop. These deviations from design specifications degrade performance and sometimes completely destabilize the control-loop. This issue is being addressed by both the controls and the computer engineering communities and now more often in collaboration. This dissertation addresses this challenge by adding application specific hardware accelerators to computer architecture, while maintaining ease of implementation. The proposed solution is an on-chip co-processor that has been implemented on a Field Programmable Gate Array (FPGA) to support the servicing of many simple plants or a single plant of many states, while maintaining microsecond level response times, tight deterministic control loop execution while allowing the main processor to service non-control tasks. The effect of variations in digital control-loop delay on a plant's stability using an actual embedded platform consisting of a hardware-based plant emulator, as opposed to software-based simulations is also studied.

CHAPTER 1. INTRODUCTION

Embedded systems and digital control theory have independently developed into mature fields, despite the clear connection between controllers and embedded platforms. Initially, each digital control loop was implemented on a separate dedicated processor, thus maintaining an exclusivity between the two fields. The demand for tighter system integration and the use of economical commercial-off-the-shelf (COTS) products has blurred this separation [3]. In modern systems, the tasks running on processors unknowingly compete with each other for processor resources. These resources, meant to improve average resource usage for non-real-time systems, are becoming sources of non-deterministic execution time. Example causes include interrupts [3], cache misses [104], and task management through operating systems [105]. These features limit the degree to which deterministic timing can be guaranteed, and cause systems to break control engineers' key assumption of constant sample rates and processor response time [5]. Ultimately, control loop robustness is affected by this transition from a dedicated processor system to an environment of tasks competing for resources [17]. Thus, a more holistic view is now necessary to develop and deploy platform-robust controls that take into account cyber-architecture artifacts on a system's physical stability.

As a motivating example, Fig. 1.1 shows the timing response of an inverted pendulum model as the computational delay is varied (the time between receiving a sensor sample and sending the response), while holding sensor sample rate constant. In Fig. 1.1(a), a controller computing delay that is 15% of the state sampling rate has negligible impact on the system's stability. As the delay increases to 65% of the sample period (Fig. 1.1(b)), some ringing in the control signal becomes apparent. Progressing to a delay of 85% of the sample period (Fig. 1.1(c)) causes the plant to become less stable with oscillations that are now more pronounced. It is interesting to note that the state of the plant (i.e. cart position and pendulum angle) still appears stable.

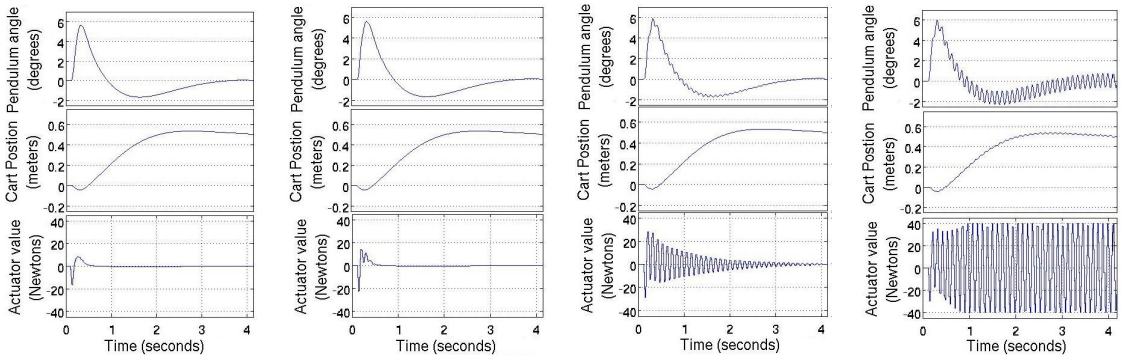


Figure 1.1 Effect of computational delay on a digital control system. With the sample period fixed at $15ms$, the delay is varied from 15% (a) to 90% (d). At 65%, a ringing begins to appear in (b), which becomes more pronounced at 85% (c). Finally, at 90% (d), the plant remains stable while the controller continuously oscillates.

A further increase in the computational delay (Fig. 1.1(d)) leads to loss of controller stability resulting in an eventual fall for the pendulum. If the amount of delay is constant, measurable, and within the closed-loop system's tolerance, it can be accounted for in the controller design. But this is not case when the controller task shares the processor resources with other tasks, be them control or non-control tasks, the delay varies with each iteration of the control-loop execution. Also, simply reducing the execution time may not mitigate the stability issue as seen in [41], where the authors show that reducing the jitter in a system previously stable with larger jitter, makes it unstable.

These issues open a field of research to mitigate and/or compensate for the effect of computational delay in both the embedded systems and control theory realms. This dissertation focuses on using hardware accelerators to guarantee the control engineers' assumption of constant execution time while still maintaining the ease of implementing the controller for the embedded systems engineer. The designed accelerators needed to be evaluated from two aspects, one for computation time and jitter and second for the controller performance. This resulted in a plant emulator, which would mimic the behavior of a plant as a linearized point while non-intrusively gathering and transmitting controller and plant performance.

1.1 Supporting Research

The effect of processor-induced jitter on a plant has been a topic of study for some time as seen in [104], a stepper motor is controlled using a processor that is scheduling tasks within a Linux operating system environment. It is observed that the torque applied by the actuator was affected by the computational load. The distribution of delay-jitter by the varying load showed clustering, resulting from various code-paths taken. The authors state that cache-misses, instruction pipelining and the scheduler are culprits of unpredictability. The authors' experimental results show, for the motors tested, jitter caused the motors to use 3.2-7.6% more of its available torque, then if no jitter was present. Additionally, an algorithm for jitter compensation was proposed and deployed that reduced this torque overhead to 0.8-0.18% at a cost of increasing the processor utilization by 20%. The same group characterized Linux for real-time applications [105] and found that the sources of jitter were implicit to the processor and were not completely correctable through software. In [13], the authors compare several scheduling methods and concluded that deadline advancement was the most consistent, with minimal degradation in performance of controllers as the number of tasks increased and had relatively consistent low jitter. Controls experts are developing toolflows, like JitterBug, to evaluate the impact of a controller's response-time jitter on closed-loop stability [17]. In [18, 36] the authors have developed a set of stability criteria for closed-loop systems in which the sample rate contains jitter. In [21], a quantitative metric similar to the concept of phase margin is proposed, called jitter margin, which is the upper-bound of delay that a control loop can tolerate before going unstable. In an approach closely related to ours, the delay and period of control loops are used in a cost function, which is then treated as a minimization problem [10], and later a convex optimization problem [128].

1.2 Organization of Dissertation

The following chapters of this dissertation are primarily derived from resultant publications that have either been accepted or are ready for submission. Chapter 2 describes our initial research that led to development and characterization of a hardware-accelerator that could sup-

port multiple single input single output plants. The difference between the software, hardware and co-designed implementations of a system with multiple PID controllers is shown, but only from the computational aspect. A platform was needed to analyze the effects of computation on plants. This led to developing the novel hardware emulator in Chapter 3, where we present the design and implementation of our control systems emulation framework that couples plant emulation hardware with an embedded processor, together on a Field-Programmable Gate Array (FPGA)-based platform. This hardware/software framework allows us to accurately study the interaction between an actual processor and a Plant-on-Chip (PoC). With this setup, more advanced controllers could be tested. Instead of testing just PIDs, a coprecessor which implemented linear iterative state-space calculations was designed, implemented, and tested with a PoC upgraded to use floating-point as opposed to fixed-point computations. The details are found in Chapter 4. The details of a scalable multi-core implementation of this co-processor is given in Chapter 5.

Chapter 6 contains research that I conducted in the area of fault tolerance for FPGAs before choosing the current thesis topic. My contribution to this work was the study of place and route quality of FPGA designs when routing matrices are damaged, by tricking the synthesis tools into believing that a targeted matrix is completely used by its corresponding configuration block.

CHAPTER 2. HARDWARE ARCHITECTURAL SUPPORT FOR CONTROL SYSTEMS AND SENSOR PROCESSING

A paper published in ACM Transactions on Embedded Computing Systems Special issue on application-specific processors (TECS), September 2013

Sudhanshu Vyas [1] Adwait Gupte[2], Joseph Zambreno[3], Chris Gill[4], Ron Cytron[4], and Phillip Jones[5]

Abstract

The field of modern control theory and the systems used to implement these controls have shown rapid development over the last 50 years. It was often the case that those developing control algorithms could assume the computing medium was solely dedicated to the task of controlling a plant. For example, the control algorithm being implemented in software on a dedicated digital signal processor (DSP), or implemented in hardware using a simple dedicated programmable logic device (PLD). As time progressed, the drive to place more system functionality in a single component (reducing power, cost, and increasing reliability) has made this assumption less often true. Thus, it has been pointed out by some experts in the field of control theory (e.g. Astrom) that those developing control algorithms must take into account the effects of running their algorithms on systems that will be shared with other tasks. One aspect of the work presented is this article is a hardware architecture that allows control developers to maintain this simplifying assumption. We focus specifically on the proportional-integral-derivative (PID) controller. An on-chip coprocessor has been implemented that can scale to

[1]Primary researcher and author
[2]Graduate Student, Department of Electrical and Computer Engineering, Iowa State University
[3]Associate Professor, Department of Electrical and Computer Engineering, Iowa State University
[4]Professor, Department of Computer Science and Engineering Engineering, Washington University, St.Louis
[5]PI and author of correspondence

support servicing hundreds of plants, while maintaining microsecond level response times, tight deterministic control loop timing, and allows the main processor to service non-control tasks.

In order to control a plant, the controller needs information about the plant's state. Typically this information is obtained from sensors with which the plant has been instrumented. There are a number of common computations that may be performed on this sensor data before being presented to the controller (e.g. averaging and thresholding). Thus in addition to supporting PID algorithms, we have developed a sensor processing unit (SPU) that off-loads these common sensor processing tasks from the main processor.

We have prototyped our ideas using Field Programmable Gate Array (FPGA) technology. Through our experimental results, we show our PID execution unit gives orders of magnitude improvement in response time when servicing many plants, as compared to a standard general software implementation. We also show that the SPU scales much better than a general software implementation. In addition, these execution units allow the simplifying assumption of dedicated computing medium to hold for control algorithm development.

2.1 Introduction

Control systems have a wide spectrum of applications, including aircraft flight-control, carbon-emission management, and national power grid management. In [4] Astrom notes that control theory has developed at an amazing pace over the last several decades, however its implementation on computer platforms has not maintained the same pace. When designing a computer-controlled system, the principle of "separation of concerns" is followed. Control engineers assume a hardware platform is dedicated to the controller being designed, while software developers implementing the design assume the controller can share hardware resources with other applications using scheduling schemes (e.g. rate-monotonic scheduling). The reasons for this disconnect can be seen by examining the evolution of digital platforms that have been used for implementing control systems.

A Brief History of Controls and Digital Computing Technology The increased availability of digital processors, in the 1960's, initiated a turning point for control theory. They

made developing modern control algorithms in the time-domain (i.e. the basis of "modern" control theory) and applying these algorithms to complex systems feasible [96]. These algorithms could now be implemented and tweaked quickly and economically. During the 1970's to early 1980's, it became clear that multiply accumulation (MAC) computations were at the heart of many digit signal processing algorithms. This lead some microprocessor architects to explore building architectures centered around MAC computation units, which resulted in the creation of digital signal processors (DSP) [35]. In addition to having hardware resources such as MACs and floating point units, a major difference between early general purpose microcontrollers and DSPs was the latter used a modified Harvard-architecture to allow their MAC units to fetch all operands in parallel [35, 94].

Digital signal processing typically requires keeping hard real-time constraints, thus early DSPs required all operations have completely deterministic timing. As a result, unlike general purpose microcontrollers, which are typically interrupt driven, early DSPs had little to no support for interrupts [35]. The reduced support for interrupts made keeping tasks deterministic easier, since tasks are not arbitrarily suspended to service system requests, however this made the use of DSPs for general purpose computing more difficult. Over time the increased performance of DSPs has enabled them to incorporate more features, such as interrupt support, to allow the implementation of multitasking for general purpose processing [35]. At the same time, general purpose microcontrollers have evolved more support for signal processing, incorporating hardware implemented MACs, floating point hardware, etc.

PID Basics Proportional-integral-derivative (PID) controllers are by far the most widely used method for stabilizing systems. In [95] it is estimated that PID controllers account for over 95% of the control approaches used in practice. These controllers have been effectively used to control systems that have a wide range of performance requirements in terms of response time, and deterministic timing of the control loop (i.e. sensor sample, PID computation, actuator update). At one extreme, a home environmental control system may require response times on the order of minutes, while the stabilization of an unmanned aerial vehicle (UAV) may require response times on the order of milliseconds. Even further on the high-performance side

of the spectrum, a fusion plasma containment field requires response times on the order of microseconds [101], with tight timing bounds placed on the determinism of the control loop.

Equations 2.1 and 2.2 give the classic equation of a Proportional-Integral-Derivative (PID) controller in its continuous-time and discrete-time form respectively.

$$u(t) = K_P e(t) + K_I \int_0^t e(t)\,\mathrm{d}t + K_D \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{2.1}$$

$$u[n] = K_P e[n] + K_I \sum_{j=0}^{n} e[j] + K_D (e[n] - e[n-1]) \tag{2.2}$$

Where:

- $u(t), u[n]$: is the correction given by the controller to the system at time t or discrete sample n.
- $e(t), e[n]$: is the error between the set point and current state of the system under control at time t or discrete sample n.
- K_P, K_I, and K_D: scale the error, integral (sum) of error, and derivative (difference) of the error respectively.

This controller is a combination of three basic control actions; 1) proportional, where the corrective signal varies proportionally to the error, 2) the integral, where the corrective signal is proportional to the integration of error over time and 3) the derivative, where the corrective signal is proportional of the gradient of error with respect to time. The sum of these individual control signals form the PID output, which is used to control the system. Let us take an example to illustrate the need for these three control actions. Suppose we have a vehicle that needs to maintain a certain heading, say North. If the vehicle deviates from its path by 15 degrees West, then the proportional element will apply a corrective signal to turn the vehicle 15 degrees East. When the vehicle is heading 1 degree West of its desired direction, the proportional element may not be sensitive enough to correct this small steady-state error. The integral part of the controller aides by accumulating the residual error to a point that the vehicle can make a corrective action. One ill-effect is the integral action causes overshoots (i.e. the vehicle may

reach North after swaying East and West in a damped sinusoidal manner). The derivative component of the controller compensates for this. However, a draw-back of having a derivative component is it amplifies system noise (e.g. vibrations, sensor error).

Computing Requirements The performance requirements of a physical system dictates, to a great extent, what computing medium is adequate to implement a PID controller. For example, the response time requirements of a home environmental control system would be on the order of minutes, which can easily be met by implementing a PID controller on a low cost microcontroller that is only capable of executing tens or hundreds of thousands of instructions per second. While stabilizing a UAV with response time requirements on the order of milliseconds, requires the PID controller be implemented on a microcontroller/processor that can execute millions or tens of millions of instructions per second. Systems with even tighter response time requirements, such as a plasma containment field controller, dictate the PID controller be implemented on a gigahertz processor or use specialized hardware to provide microsecond-level deterministic control loop timing. Furthermore, for such low latency systems, if a commodity gigahertz processor is used, the processor's operating system (OS) will need to be highly customized [101]. For example, interrupts may need to be disabled, which virtually defeats the purpose of having an OS.

The term *response time* refers to the interval of time from when the controller samples the current state of the system, using a sensor, to when the controller updates an actuator to adjust the system. This time is composed of three major components: 1) reading sensors, 2) computing the control algorithm to produce a correction value, and 3) the time for the correction value to be received and acted upon by an actuator. The term *jitter* refers to the variation between the best-case response time and worst-case response time of the controller. It is a measure of how deterministic the timing of a control loop is. Sources of jitter in an embedded system can stem from are varying peripheral latencies, cache misses, interrupts, branching, etc. Other sources are economical, where commercial off the self (COTS) vendors do not give the full specification of their devices. [100] implements a monitoring device on an FPGA to compensate for these unknown variables. In general, jitter can degrade a control system's performance [120]. One

aspect of our work presents experimental results that quantify some sources of jitter within a general purpose controller, and mitigates them without hampering the overall computing medium.

Bridging the Control Theory and Systems Implementation Gap While response time and jitter are two primary constraints placed on implementing a PID controller, often it is desirable to have the system perform tasks that do not have hard real-time constraints. Some examples are: 1) system management and health assessment tasks, 2) high-level planning for autonomous systems that are trying to accomplish a goal, such as navigating a maze, and 3) encryption and compression of military UAV communications. Control algorithms are now often deployed on computing platforms that are not dedicated (e.g. a DSP running a single tight control loop) for reasons such as increasing overall system performance or reducing cost (e.g. system-on-chip platforms). The assumption that control algorithms can be developed in a vacuum, not taking into account interactions with the underlying computing platform, must be addressed.

One common solution is to deploy a real-time operating system onto the computing medium (e.g. VxWorks, Lynx) or to develop custom software that makes use of hardware interrupts to give tasks hard real-time priorities. In addition, real-time scheduling algorithms such as rate monotonic or earliest deadline first need to be employed for assigning task priorities, and guaranteeing that it is feasible to schedule all tasks on to the underlying computing medium. Some drawbacks with this common approach is the design, implementation, and verification of the system from a software perspective becomes complex.

Even if these approaches are used to allow sharing of the computing medium, non-determinism still exist in the system. [56] have taken steps to close the gap between the development of control theory and non-deterministic computing behaviors. They have developed tools, such as "Jitter bug", to help algorithm developers quantify the sensitivity of their algorithms to jitter. A complementarity approach to using tools, such as Jitter Bug and deploying real-time operating systems is to use hardware support to isolate the control algorithm from non-deterministic aspects of the computing medium.

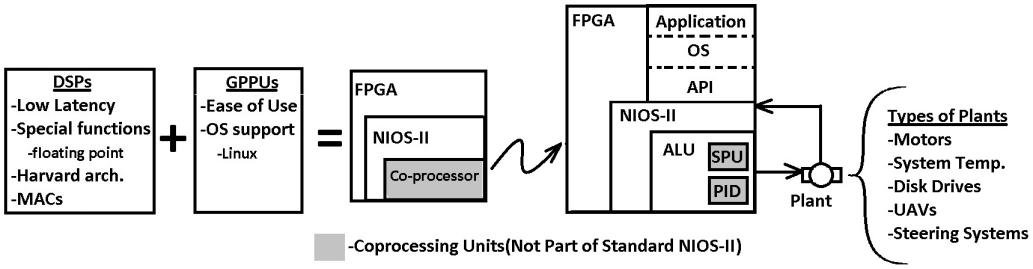


Figure 2.1 In this paper traits of both DSPs and general purpose processor units (GPPUs) are encapsulated in an FPGA-based system-on-chip architecture, which can be used to control a wide variety of plants.

Our work, illustrated in Figure 2.1, leverages the flexibility of Field Programmable Gate Array (FPGA) technology to develop and characterize a low cost embedded architecture that provides the aforementioned hardware supported isolation. At the heart of our approach is a time-multiplexed hardware PID controller and a sensor processing unit (SPU) [47] that are tightly integrated with an embedded processor as functional units. The architecture provides deterministic response times on the order of microseconds with low jitter, and can scale to support hundreds of PID control loops. The time-multiplexed PID functional unit does not share any on-chip resources with the softcore processor that could interfere with the PID control loop's real-time constraints. Thus, non real-time tasks can be safely run and efficiently scheduled onto the same device that services the PID controllers. Additionally, our hardware enforced mitigation of non-determinism simplifies software development and verification, since a real-time scheduling scheme is not needed for sharing the computing medium between real-time and non-real-time tasks. Section 4.3 provides a detailed overview of our architecture.

Contributions The primary contributions of this work are 1) the tight integration of a time multiplexed hardware PID controller within an embedded processor, 2) the characterization of our PID controller architecture and several alternative hardware/software hybrid-designs with respect to response time and jitter, 3) the tight integration of a hardware-based sensor processing unit (SPU) within an embedded processor and its evaluation with respect to software implementation of common sensor processing tasks in terms of response time [47].

Organization The remainder of this article is organized as follows. Section 6.2 gives an overview of the current state of the art for implementing PIDs in control systems and then discusses common sensor processing computing kernels that our SPU supports. In Section 2.3, the platform used for deploying and characterizing our architecture is described. Section 4.3 presents a detailed overview of our architecture. The evaluation methodology for quantifying the response time and jitter of a fully software implemented PID controller along with two software/hardware codesigns is given in Section 2.5. Our approach for evaluating the SPU is also discusses in Section 2.5. Section 2.6 summarizes the analysis and results of our evaluation experiments. Section 2.7 concludes this article and suggests avenues of future research.

2.2 Related Work

2.2.1 Implementing PIDs in Control Systems

The first publication of the PID concept was in 1922 [84],and intensive research on PIDs continues today. In [95] it was estimated that 95% of control loops in the world contain PIDs, and describes various research domains for PIDs including; tuning, cascading of PIDs, implementation techniques and controller management. [122] gives a background and comparison of different fuzzy-tuning methods.

Control systems can be implemented purely in software, in hardware, or a combination of the two.

Software Earlier, Section 2.1 gave a brief overview of the general nature of implementing controllers in software using DSPs and general purpose microcontrollers. An example of pure software high performance control system is a Linux based Plasma Control System (PCS) implemented on a Linux x86 server in [101]. The system was designed for confining the inherently unstable plasma of a tokamak. The authors explain that the system that they acquired from the market had an advertised response time of 25us. The authors had to make changes to the Linux kernel itself in order to even get a response time of 50us. These changes were drastic (e.g. disabling interrupts) and highly customized in order to get a deterministic system. Linux was virtually disabled in order for the control loops to meet timing, thus effectively undermining

major benefits of having an operation system, such as efficiently running multiple applications on a single processor at a time. Similar efforts have been made on the same tokamak plasma controller [15, 109, 43], which have used Linux servers in conjunction with dedicated hardware.

Hardware Limiting the discussion to digitally implemented controllers, industrial controllers (e.g. controllers in steel manufacturing facilities) are in most cases implemented using hardware-based devices called Programmable Logic Controllers (PLCs) [31]. According to a study performed by National Instruments [93], 77% of industrial applications having less than 128 I/Os use PLCs and are usually peripherals for computers that handle higher-levels of control. Though easier to use and free from issues like system reboots and varying latencies that are associated with computer-controlled systems, these devices have very limited resources. In response to market demands for more resource rich PLCs, researchers are attempting to implement PLC systems on FPGAs [31]. Our research is in alignment with this sprite, as we are evaluating a method that retains the benefits of hardware determinism, while enabling the ability to run higher-level applications and operating systems.

Field Programmable Gate Arrays (FPGAs) The practice of using FPGAs to implement control theory has been pursued since the 1990s, which was the point at which it was first feasible to use these devices for controls applications [49]. Examples of research implementing PIDs on FPGAs are [108] where various architectures of PIDs are tested on earlier FPGAs. In [82] an FPGA is used along with Matlab/Simulink to design PIDs on FPGAs for motor control. This method is known as Hardware in the loop (HIL). Such designs can later be merged with soft-processors on the FPGA to increase functionality. Other directions of research include methods to reduce the amount of power consumed by PIDs by implementing the computation using distributed arithmetic [22, 23]. [132] has implemented a time-multiplexed hardware PID controller that most closely matches the architecture of our PID computing unit. However one major difference is our tight coupling of the PID controller as a functional unit of an embedded softcore processor. We also characterize the tradeoffs associated with distributing the architecture across the hardware-software boundary, and characterize the jitter of the system [20].

Jitter analysis is critical for providing system stability guarantees. Though FPGAs are useful due to their quick turn-around time, they were not very popular with non-hardware experts. This changed once hard and soft-processors were available on FPGAs, allowing software and OS's to be incorporated in designs.

2.2.2 Common Kernel for Sensor Processing

Due to the large and diverse set of domains in which embedded systems are used, past works [32] [48] have tried to identify common computational kernels that are used across domains by abstracting away the specifics of individual applications. Similarly, we have tried to boil down the diverse set of sensor processing tasks to a small set of core kernels that are generic enough to find application in many fields, and common enough to warrant the allocation of processor chip real estate. While many applications require massive processing of sensor data using digital signal processing type algorithms, there is a large base of applications [55] [40][102][116] that consist of simpler processing tasks, and do not warrant the overhead of having a full blown Digital Signal Processor (DSP). We have identified and extracted five such tasks from the MiBench benchmark and various other sources: Linear Equations, Moving Average, Average, Delta Value, and Threshold/Range Check.

Linear Equations One of the simplest tasks that can be performed on sensor data is the execution of a linear mathematical operation on a single value or multiple values. In the automotive domain converting between radians and degrees is identified as a common task in [48]. In the controls system domain the Proportional-Integral-Derivative (PID) controller is one of the most common control algorithms used [63]. [117] and [102] give examples of advanced PID controllers that can be boiled down to calculating a set of linear equations involving sensor values along with other computational functions.

Moving Average Sometimes, sensors can be prone to spurious spikes in their output. The moving average is one method for reducing the effects of random "noise" on sensor output. Over time measurements are averaged together, and this average is used by the end application.

The impact of spikes in measurements are mitigated, while actual changes in the physical quantity being measured are eventually reflected by the average. The responsiveness of the moving average can be tuned to reflect the known physical dynamics of the quantity being measured by adding appropriate weights to the previously computed average, and the current sensor measurement (weighted moving average). The moving average is a common method for filtering sensor data [55][66] [44][79].

Average Applications, such as sensor networks, often measure physical quantities over a distributed area (e.g. temperature). Averaging these distributed measurements is a simple means for aggregating this information into a compact form. This approach was used by Goebel [40], Ganeriwal [37], and Nakamura [91]. In [55] Hellerstein uses averaging as means of "cleaning" data obtained from a group of sensors.

Delta Value Often, the difference between the current and previous value of the sensor (called delta) is used in a lot of computation. For example, the "communication" suite of benchmarks in [48] includes delta modulation, a process of encoding which may be used to encode the output of sensors before transmission. [116] presents another example of using the "delta" value in a sensor based system.

Threshold/Range Check There are various applications in which the sensors values are required to be within particular range or below/above a certain threshold for the proper operation of the system. In such systems, some sort of mechanism is needed to monitor the sensor values. Alternatively, the systems might need to take some kind of action when the sensor value breaches the defined range/threshold bound.

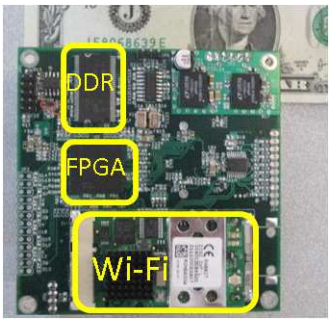
This is by no means an exhaustive list of such kernels, but having identified some common kernels of computation across fields, we describe a Sensor Processing Unit (SPU) in section 4.3 that supports these kernels.

2.3 Development Platforms

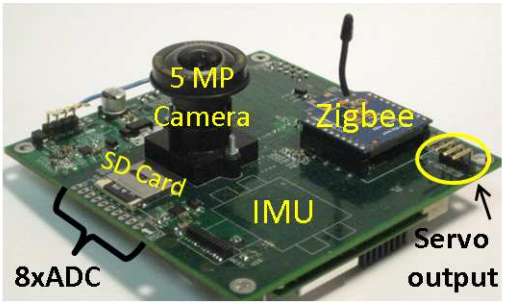
PID Coprocessor This work was deployed and evaluated on the Reconfigurable Autonomous Vehicle Infrastructure (RAVI) board, an in-house developed FPGA developement platform. The RAVI platform was specifically fabricated to develop efficient control systems for small battery-powered autonomous vehicles. These vehicles are often highly constrained in terms of power consumption, computational resources, and weight. The control of such vehicles involves the execution of computationally aggressive algorithms that must meet hard real-time constraints. Thus, it is critical that their software and hardware components efficiently map to the underlying host platform to maximize system performance.

RAVI leverages Field Programmable Gate Array (FPGA) technology to allow custom hardware to be tightly integrated to a soft-core processor on a single computing device. It enables the exploration of the software/hardware codesign space for designing system architectures that best fit an application's requirements. A major vision for RAVI is to act as a common research medium to bring control theorists, embedded system programmers, and hardware architects together to work on research issues that cross-cut all three disciplines.

Specifically, the RAVI board hosts an Altera manufactured Cyclone III (EP3C25) FPGA. This FPGA has enough logic and memory resources to easily support deploying the NIOS-II 32-bit soft-core processor and PID coprocessor, while still having a majority of its resources free to implement additional functionality.



(a) Bottom view of RAVI FPGA-based platform.



(b) Top view of RAVI FPGA-based platform.

Figure 2.2 RAVI: FPGA-based board for developing custom architectures for embedded systems.

Sensor Processing Unit (SPU) The SPU evaluation was performed on the Xilinx-based ML507 development platform, which hosts a Virtex-5 FX FPGA. The SPU prototype used less than .5% of the resouces available, and was tightly coupled to the Power PC processor embedded into this family of FPGA. [47] gives further details on some of the specifics of this platform. As indicated in Section 2.7, a future direction for this work is to couple the SPU with the PID coprocessor on the RAVI platform. The SPU would be used to condition sensor data before forwarding it to the PID controller.

2.4 Architecture Overview

This section describes the architecture of the time multiplexed hardware implemented PID controller, and a hardware implemented sensor processing unit (SPU)

2.4.1 Hardware implemented context-switching PID controller

2.4.1.1 Overview

In the world of digital control, a sensor is sampled at discrete time intervals, the sample is then used to decide the control value that is required for the plant to remain in a desired state. Between sampling intervals the controller for a given plant is idle, until the next sample is received. When a system implements multiple PID controllers on a single CPU, typically each PID is called as a function sequentially. This allows the the CPU to be time shared among the controllers. However each time the CPU switches to service a new plant, the information associated with the current plant and corresponding PID must be pushed onto a software stack. In addition, any interrupts that occur will force the CPU to push all of its state data onto the stack and then restore its state after the interrupt completes. When servicing many plants, this context switching overhead can become a bottle neck to performance (see section 6), and interrupts occurring during PID execution will add non-determinism to the overall control loop. In addition, as stated earlier, the PID controller will have further sources of non-determinism while competing for CPU time with general purpose non-periodic system tasks.

In our design we take advantage of the PIDs repetitive Sample, Compute, Write actions

to develop a resource efficient architecture that addresses the issues of context switching overhead, non-determinism, and competing with non-period tasks, when implemented on a general purpose CPU. Our architecture acts as a co-processor to the CPU, only relying on the CPU to configure and update each PID that is require for use in the system. Once configured, the PID co-processor executes each PID in a round robin fashion, as depicted in Figure 2.3(a). The the memory and logic resources allocated for the co-processor completely isolate the timing of its Sample, Compute, Write cycle from the general purpose CPU's behavior. Figure 2.3(b) provides the high-level structure of our architecture. A central PID compute unit is shared among all plants, and a single context memory is used to sequentially store and retrieve the context of each plant. This architecture requires only one clock cycle to context switch between plants.

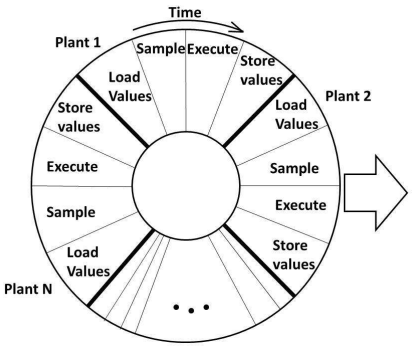
The primary factor that limits the number of plants (N) that can be support by this architecture is the required service time for the fastest plant. If the fastest plant requires a service period of $\tau_{min\_service}$, then the time to service all N plants in a round robin fashion is limited to $\tau_{min\_service}$. Thus, the number of plants serviced must satisfy the following constraint, where $\tau_{context\_switch}$ and $\tau_{compute}$ are the times required to perform a context-switch and PID computation, respectively:

$$N \leq \frac{\tau_{min\_service}}{\tau_{context\_switch} + \tau_{compute}} \tag{2.3}$$

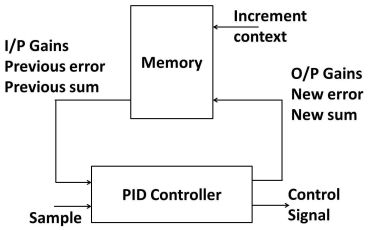
For example, let us assume a 10ns clock, and 2 clock cycles to service a plant (20ns). If an extremely high performance plant requires a control loop period of 10 us, then our architecture can service up to 500 plants (10us/20ns). If a 1 ms plant is the fastest, then the architecture can support up to 50,000 plants. As the speed of the fasts plant decreases, memory storage becomes the limiting factor. However, supporting 100's of plants easily satisfies the needs of most systems.

2.4.1.2 Detailed Architecture

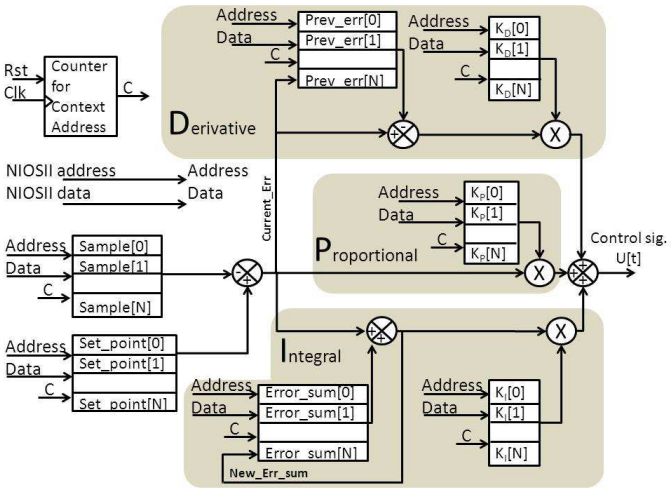
Figure 2.3(c) illustrates the detailed architecture of the PID execution unit. This architecture represents the custom hardware that is integrated into the NIOS-II, Figure 2.4. Except



(a) Conceptual idea for round-robin scheduling of PIDs.



(b) High-level architecture for implementing context switching between PIDs.



(c) Detailed architecture of the context-switch PID

Figure 2.3 A conceptual to detailed architectural illustration of the hardware implemented context-switching PID controller.

for configuring scaling constants, this execution unit runs independent of the software running on the soft-core NIOS-II processor. Thus, helping free the NIOS-II to perform higher levels of decision making (e.g. path planning, and task scheduling).

Context-switch In order to allow the PID execution unit to access the components of a given plant's context in parallel, each component is stored in a separate blockRAM (e.g. on-chip FPGA memory). These components are defined to be K_P, K_I, K_D, set-point, previous iteration error and previous iteration sum. A given blockRAM (e.g. K_P) sequentially stores the context component information for N plants. For example, the context information for

plant 0 is stored at address 0 of each blockRAM, and the context information of plant 1 is stored at address 1.

In the upper left-hand corner of Figure 2.3(c) is a counter. This counter is responsible for cycling through the context of the N plants in a round-robin fashion. In other words, it is used to synchronize all of the blockRAMs so that the PID execution unit has the appropriate plant's context available to it. If the counter has a value of 25, the context information for plant 25 will be provided to the execution unit. After the PID calculation completes, the counter increments to 26, and so forth. Once the counter reaches the value $N - 1$, it resets to 0 and begins counting upward again.

PID implementation The remainder of the design is a standard PID controller. We use one subtractor to calculate the error between the set-point and the sensor sample; one subtractor to compute the gradient between the current error and previous iteration's error; an accumulator to sum the error; three multipliers to calculate the proportional, integral and derivative corrections and a three-input adder to generate the correction output to be sent to the plant. The calculated values that need to be stored for the next update of the current plant are routed back to their respective block memories; specifically the current error is stored to the previous error blockRAM, and the error sum is stored to the error sum blockRAM.

Timing Both the PID execution unit and NIOS-II processor run on a 50 MHz (i.e. 20ns period) clock. The PID execution unit takes one clock to read values from the blockRAMs and one clock to compute and write updated values back to the blockRAMs, thus each plant takes 40ns to service (20ns x 2 clocks). Cycling through N plants takes $40xN$ns. This assumes that the sensor sampling rate is $40xN$ns or faster. If not, then the time to service all N plants will be limited by the sensor sampling rate.

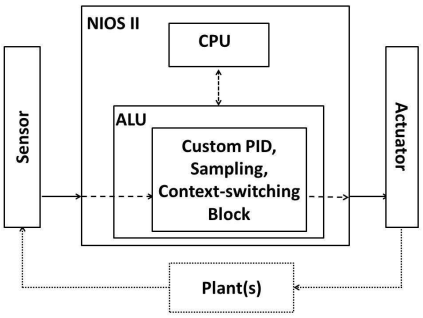


Figure 2.4 Hardware implemented time-multiplexed PID unit integrated into the NIOS processors ALU using user-defined instructions.

2.4.2 Sensor Processing Unit (SPU)

2.4.2.1 Overview

The SPU is designed to be a functional unit within an embedded processor, as shown in Figure 2.5. It has two main purposes 1) to efficiently offload the execution of common sensor processing tasks from the main ALU, and 2) to detect events that are a function of sensor values. Here we discuss the two major uses of the SPU, and how it could potentially be used for power management.

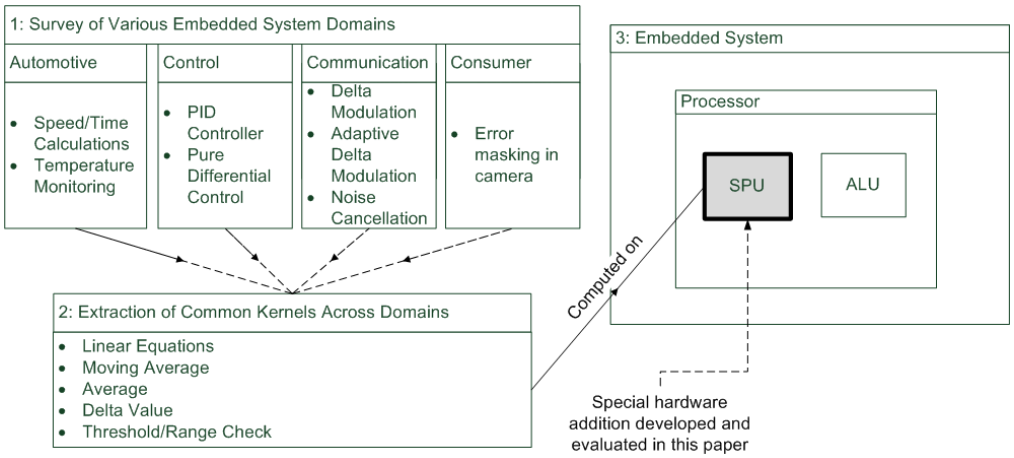


Figure 2.5 1) illustrates some high-level sensor dependent tasks identified within various embedded domains, 2) highlights common kernels of computation that were identified, 3) shows conceptually how our sensor processing unit (SPU) integrates into an embedded system.

Sensor processing offload The SPU supports the direct fusion of sensor data without intervention from the primary processor, allowing any arbitrary function involving weighting

the sensor values and summing them together. The SPU has been designed to operate in the following manner. First a user application programs the SPU with functions that need to be performed on the output of one or more sensors. Once programmed, the SPU computes these functions continuously as sensor data flows into the SPU. The output of the sensors are directly connected to the SPU, thus the SPU reevaluates its programmed functions autonomously of the rest of the processor. When the user application requires the result of one of the programmed functions, it simply issues a single instruction to fetch this value from a special register. This is opposed to the traditional approach of 1) reading all sensor values required by a function, and 2) computing the function in software. In addition to increasing the speed at which a given sensor processing function can be computed, discussed in Section 2.6.2, the SPU allows the rest of the processor to focus on other tasks.

Event detection Each function programmed into the SPU can have an event associated with it. An event checks if the result of a given function is <,>,or = to a fixed value, or if the result of a function is within or outside a given range. If the associated condition is true, then an interrupt is sent to the main processor. The purpose of this functionality is to allow the processor to work on other tasks until a given event fires, as opposed to continuously polling sensor values and performing event checks in software. This capability targets applications that take actions when sensor values surpass a given threshold (e.g. thermal shutdown condition), or fall outside an acceptable range (e.g. voltage supply stability). While, current processors have the ability to react to simple events such a thermal overload [62], the SPU is a lightweight means to generalize the types of events that a processor can detect and respond.

Power management Many processors and microcontrollers support a low power mode from which they can be woken up by an interrupt [61]. Given the SPU's ability to operate on sensor data autonomously of the rest of the processor, the SPU could potentially be used as a lightweight mechanism that allows the rest of the processor to go into, or come out of a low power state based on sensor data.

Listing 1 provides a sample code excerpt that shows how the common task of computing an average of multiple sensors could be implemented with and without the SPU. An important point to note is that while the time to compute the average in software is a function of the number sensors, the time to compute the average of 1 to N sensors using the SPU is constant. Where N is the number of sensors supported by the SPU. Of course a major hardware design-time decision that needs to be made is how many sensors should the SPU support, since the hardware resources of the SPU will be proportionate to the number of sensors supported.

```
WITHOUT SPU

int s1,s2,s3,avg;

while(1){
  // Sensor reads could be a large number of assembly instructions
  s1=read_sensor_1();
  s2=read_sensor_2();
  s3=read_sensor_3();

  // Time to compute Average will vary with # number of sensors read
  avg=(s1+s2+s3)/3
}



WITH SPU

int avg,sum;
int param1,param2;
param1=0x01010100; // Configure SPU to sum first
param2=0x11100000; // 3 sensor values.

// Initialize SPU once
setup_SPU(param1,param2)
```

```
while(1){
  // A single assembly instruction reads and sums sensor values.
  read_spu(sum);
  avg=sum/3;
}
```

Listing 1: Illustrates the difference between using the SPU vs. standard software for computing the average of multiple sensors.

2.4.2.2 Architecture

Figure 2.6 illustrates the architecture of the SPU. The following describes the five major components that comprise the SPU; Sensor Data Extractor, Configuration Storage, Processing Unit, Result Storage, and Interrupt Generator.

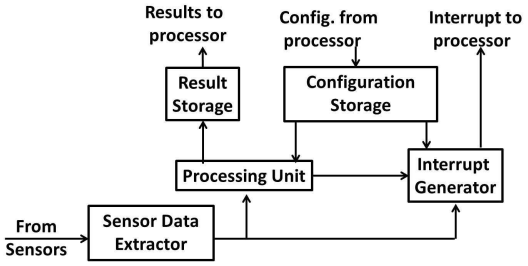


Figure 2.6 Architecture of the SPU

Sensor Data Extractor This block is responsible for connecting the SPU to the sensors. It continuously streams sensor data to the SPU's Processing Unit, and Interrupt Generator.

Configuration Storage This is where configuration information sent by a user application is stored. This information includes the set of sensors a given function operates on, the operations and constants that define a function, and threshold and range values associated with events that can generate interrupts to the main processor

Processing Unit This unit is responsible for all computations in the SPU. It is a simple Multiply-Accumulate module that multiplies sensor values with their associated weights (found in Configuration Storage). After a function has been computed, its value is stored in the Result Storage block, and the Processing Unit is reinitialized. Then the configuration parameters, and sensor values for the next function are loaded into the Processing Unit. This allows the Processing Unit to be time shared among many functions. If each function is assumed to executed in a single clock cycle, then this unit can scale to a large number of functions before noticeable latency issues associated with stale data arise.

Result Storage This block is used to store the results calculated by the Processing Unit. When an application requests the result of a given function, the value is fetched from this block. As the number of functions computed by the Processing Unit increases the staleness of the data stored in the Result Storage block increases. However, given that the clock rate of a processor is typically much higher than the rate of change of sensors, the relatively small time lag should be acceptable for most applications.

Interrupt Generator The Interrupt Generator is responsible for detecting when a sensor value, or function computed by the Processing Unit satisfies criteria specified by a user application. If a criteria is met, then an "event" is said to have taken place. On the occurrence of an event this unit sends an interrupt to the main processor.

2.5 Evaluation Methodology

Maintaining consistency is a key factor in being able to compare different methods of implementation. The RAVI development board allowed the use of a single platform for developing and evaluating each variation of our architecture. The portions of the board we used for our experiments included the Cyclone III FPGA, the on-board DDR DRAM and the UART port. The FPGA was used to implement the NIOS-II (Altera's soft-processor), the DDR stored software that was run on the NIOS-II and the UART port supported data collection. A pictorial description of the setup is shown in Figure 2.7.

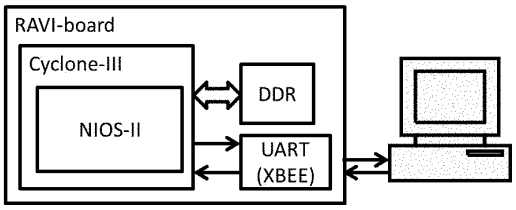


Figure 2.7 High-level depiction of experimental setup.

The FPGA allows all tests to run on a common processor, NIOS-II, and enabled implementing tightly processor-coupled hardware blocks. It should be emphasized that while our FPGA infrastructure is well suited for quantifying software/hardware tradeoffs, for an end product that is to be mass produced an application specific integrate circuit (ASIC) realization would be preferred from an economic, performance, and energy consumption perspective.

2.5.1 Hardware-base context switching PID

The metrics of interest for our evaluation were 1) response time (defined to be the time to service all plants once), and 2) response-time jitter. The system variables we varied to evaluate these metrics were:

1. the architecture (Case 1, 2, 3, 4),
2. the number of plants controlled (10,100, 1000),
3. the processor interrupt timer (1 ms, 100 ms),
4. the sensor sampling rate, in samples per second (SPS) (No Delay, 200KSPS, 819SPS),
5. software-implemented jitter compensation (used or not used) [126].

For profiling the system, software and hardware elements were added to take measurements. One advantage of hardware-based monitoring is it does not interfere with the timing of the system under test. For our evaluation a hardware based time-stamp counter (TSC) was implement to accurately measure plant response-time. Figure 2.8 depicts the flow of the evaluation process. First, the number of plants to be serviced (N) and the number of times to cycle through servicing the plants is set. For our experiments, we cycle through all plants one

thousand times. For each plant we 1) record the time servicing begins, 2) service the plant, and 3) record the time servicing completes. Once all plants have been serviced one thousand times, the collect data is transmitted to a standard PC for analysis. This procedure was run for each configuration evaluated. Section 2.6.1 discusses the results of these experiments.

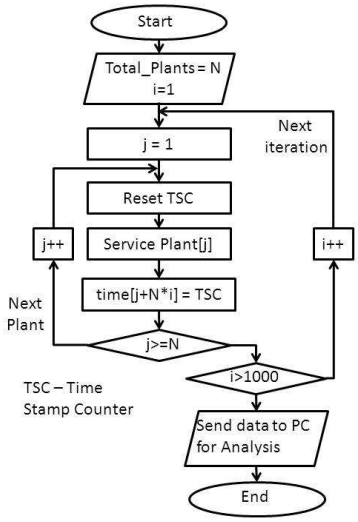
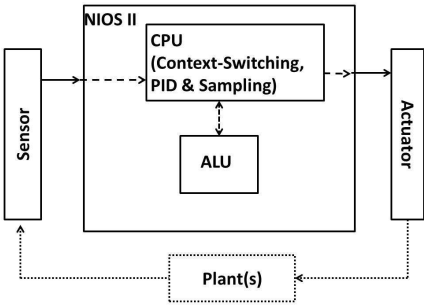


Figure 2.8 Test flow for measuring the response time of each architecture.

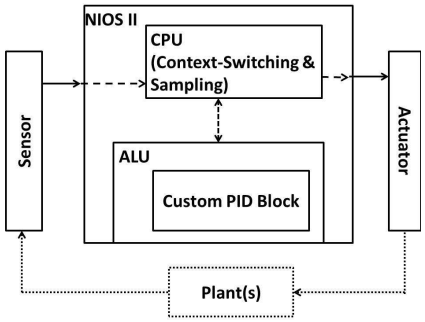
Reference Architectures We created three reference implementations to help clarify the speedup attained from our proposed approach (Case IV, Figure 2.9(d)). As we move from Case I to Case IV, larger portions of the PID control loop is migrated to hardware. A description of the four cases follows:

- Case I: A fully-software implemented solution on the NIOS soft-core processor (Figure 2.9(a)). This setup uses the system's standard software interface for sampling sensor data. This case's performance is expected to suffer from high bus-latencies and having to execute the PID functionality in software.

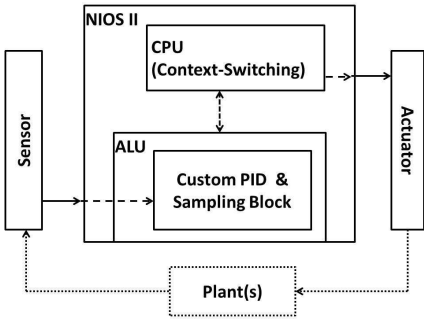
- Case II: The PID execution unit is used to accelerate PID computations. The standard software interface is still used for sampling sensor data, and context switching between each plant is the responsibility of software. It is expected this setup will still suffer from high bus-latency (Figure 2.9(b)).



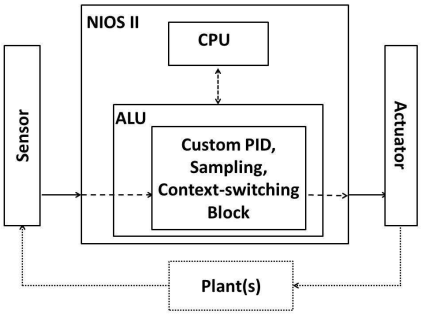
(a) Case I: Software implemented PID controllers.



(b) Case II: CPU-based context switching of hardware PID functional unit, and CPU driven sensor sampling. A context switching approach.



(c) Case III: CPU-based context switching of hardware PID functional unit, and PID functional unit driven sensor sampling.



(d) Case IV: Context switching of hardware PID functional unit, and sensor sampling in hardware (our approach).

Figure 2.9 Reference designs used to show how performance changes when moving from an all software solution (a), to software/hardware hybrid approaches (b), (c), compared to our full hardware PID off-loading solution (d).

- Case III: In addition to making use of the PID execution unit, this case also uses low latency custom instructions to give the processor quick access to senor data. The context switching between plants is still the responsibility of software. It is expected that having the software perform the PID context switching will have significant execution overhead (Figure 2.9(c)).

- Case IV: All functionality associated with sampling senor values, PID computation, and plant context switching are placed in hardware. It is expected this setup will greatly outperform Cases I - III (Figure 2.9(d)).

In addition to evaluating different distributions of functionality between software and hardware, we examine the impact of the following system parameters on the response time and response-time jitter of each distribution:

- processor's timer interrupt period: We use a programmable hardware timer to vary how often interrupts fired. This allowed us to isolate jitter caused by interrupts from jitter associated with the nature of an implementation approach (i.e. Cases I-IV). We examine the impact of setting the interrupt period to be 1ms versus 100 ms. Most general purpose processors require some form of hardware interrupt timer to function properly.

- software-implemented jitter compensation: We implemented a jitter compensation method as given in [126]. The pseudo-code for implementing this method is shown in Listing 2. During a calibration run, the longest latency of servicing a plant is recorded. Then, this estimated worst-case delay is used to extend the effective response time if a given plant is serviced in less than its estimated worst case. For example, if the worst-case estimated service time for a plant is 5 us, and the current servicing of the plant finishes in 3us, then the processor will wait for another 2 us. This gives the plant an effective service time of 5 us, thus helping reduce the response-time jitter of the system.

```
// Reduce response-time-jitter by forcing each cycle of the
// control loop to be equal to the worst case measured loop delay.
while(1){
  reset_time_stamp_counter();
  // start control algorithm
  ...
  ...
  // end control algorithm

  end_time = read_time_stamp();

  if(end_time > worst_delay){
    worst_delay = end_time;
```

```
  }
  wait(worst_delay - end_time);
}
```

Listing 2: Gives a traditional software approach for reducing the jitter of a control algorithm's control loop.

2.5.2 Sensor Processing Unit (SPU)

The SPU was implemented on the FPGA and connected to the processor using the coprocessor interface. The coprocessor interface allows User Defined Instructions (UDIs) to be used to read the sensor data. A Peripheral Local Bus (PLB) also connects the processor to a pseudo sensor. Evaluation experiments were performed for the following three setups:

1. PLB: A pseudo sensor is read over the PLB by the processor. This scenario is used to emulate an embedded processor reading sensor data over an on-chip general purpose peripheral bus.

2. UDI: The fact that the coprocessor interface is faster than the PLB contributes to the improved performance of the propose architecture. However, to show that this is not the sole reason for the improvement we have included this additional case, where the data is read over the low overhead coprocessor interface, but without the SPU.

3. SPU: The SPU implements a hardware unit to accelerate the common sensor processing kernels described in section 2.2.2. The processor uses UDIs to program the SPU, as well as to read data from the SPU.

The three setups were compared with respect to three metrics:

1. Execution Time: The amount of time required to complete a sensor processing operation.

2. Code Density: Size of the compiled programs.

3. Response Latency: Latency between an event occurring and the processor responding (used for the interrupt generation feature of the SPU).

2.6 Results and Analysis

First results related the response time of our PID execution unit are presented. This is followed by the results of our SPU evaluation experiments.

2.6.1 Hardware-based context switching PID

Figure 2.10 summarizes the average plant response time, in tabular form, when sensor delay and the number of plants are varied. Figure 2.11 shows how functionality is migrated from software to hardware when moving from Case I - Case IV.

1 Plant	No Delay	200KSPS	819SPS
Case I	16.85	27.60	1292.92
Case II	12.33	20.69	1295.73
Case III	9.13	9.63	1293.84
Case IV	0.04	5.00	1221.00

100 Plants	No Delay	200KSPS	819SPS
Case I	2249.01	3244.59	129856.34
Case II	1878.70	2571.55	129845.85
Case III	1423.04	1526.99	129833.02
Case IV	4.00	500.00	122100.00

10 Plants	No Delay	200KSPS	819SPS
Case I	227.79	331.45	12955.68
Case II	189.34	260.88	12985.27
Case III	146.99	151.78	12988.75
Case IV	0.40	50.00	12210.00

1000 Plants	No Delay	200KSPS	819SPS
Case I	22388.31	32342.28	1255871.24
Case II	18666.88	25614.60	1255733.47
Case III	14663.60	15169.76	1255794.38
Case IV	40.00	5000.00	1221000.00

Figure 2.10 Summary of response time across Cases I - IV. Note: Sensor update rate is measured in samples per second (SPS)

	Software Implemented	Hardware Implemented
Case I	• Set-point management • sensor sampling • PID • Context-switching	
Case II	• Set-point management • Context-switching • sensor sampling	• PID
Case III	• Set-point management • Context-switching	• PID • sensor sampling
Case IV	• Set-point management	• PID • Context-switching • sensor sampling

Figure 2.11 Illustration of movement of functionality from software to hardware.

Figures 2.12 - 2.14 present our results in the form of response-time histograms, showing the number of times an individual plant experienced a given response time (in microseconds). The response time is shown as a function of three variables; architectural implementation (Case I, Case II, Case III, Case IV), interrupt period (1ms and 100ms) and jitter compensation (yes or no). In these plots sensor sample time is held constant as No Delay, and the number of plants serviced is set to 100.

Each plot contains data from two identical experimental setups, with the only difference being jitter compensation enabled or not. The dotted line in each plot separates the non-compensation experimental run (left side) from the compensation implemented one (right). The experiments that do not implement jitter compensation have shorter average response times, but show more jitter than the experiments that implement jitter compensation. This is in alignment with our expectations. We have not plotted the histogram for the purely hardware architecture (Case IV) as the system is deterministic by nature (e.g. it cannot be unwantedly affected by software interrupts). We next organize our observations by test parameters; Architecture, Interrupt period, Jitter compensation.

Migrating functionality As expected, as functionality is migrated into hardware, response time decreases. Thus the fully hardware implemented PID architecture can service more plants than the other architectures. As the speed of the sensor decreases, the benefit of having dedicated hardware with respect to response time decreases since the bottleneck of the system becomes the sensor.

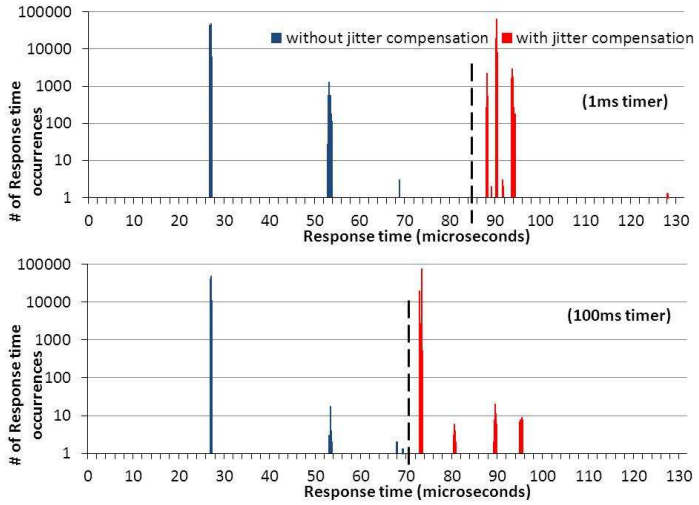


Figure 2.12 Response time and jitter: This plot shows the number of times plants experienced a given amount of response time. The distribution of these response times is a measure of how much jitter a give system configuration experienced. The parameter settings for this experiment were: # of plants = 100, sample rate = No Delay, Architecture = Case I.

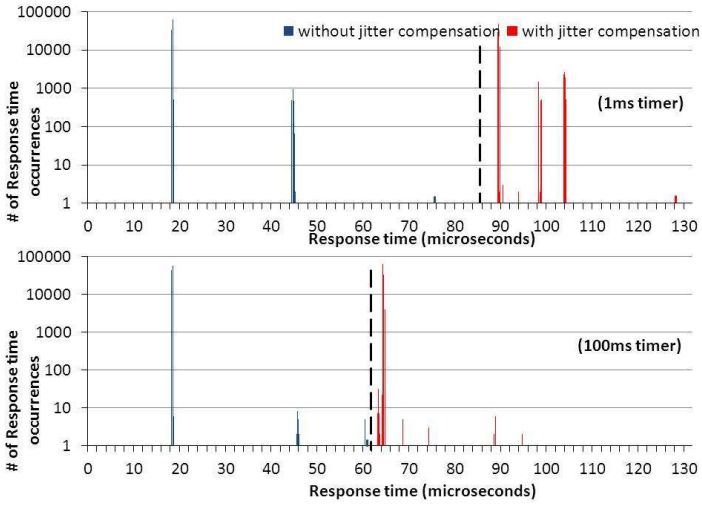


Figure 2.13 Response time and jitter: This plot shows the number of times plants experienced a given amount of response time. The distribution of these response times is a measure of how much jitter a give system configuration experienced. The parameter settings for this experiment were: # of plants = 100, sample rate = No Delay, Architecture = Case II.

Interrupt period When we look at each plot, we notice that the jitter reduces when changing the interrupt period from 1ms to 100ms. This suggests that setting the rate of interrupts as low as possible may be a good practice for minimizing the response-time jitter of a system. However, one must use caution, as some tasks that rely on interrupts may suffer if a given interrupt type's rate is lowered below a given threshold.

Jitter compensation When jitter compensation is used, each plot shows a decrease in jitter and an increase in average response time. This matches our expected results, since the implemented algorithm forces the system to wait for the worst-case service time each time a given plant is served.

Summary Figure 2.10 tabulates response time as a function of number of plants, architectures and the sample rate of the sensors. It shows several interesting patterns. First, the response time of a system with a given architecture and sampling rate is directly proportional to the number of plants that are being controlled by the system.

Second, while keeping a fixed sampling rate and a fixed number of plants, and moving

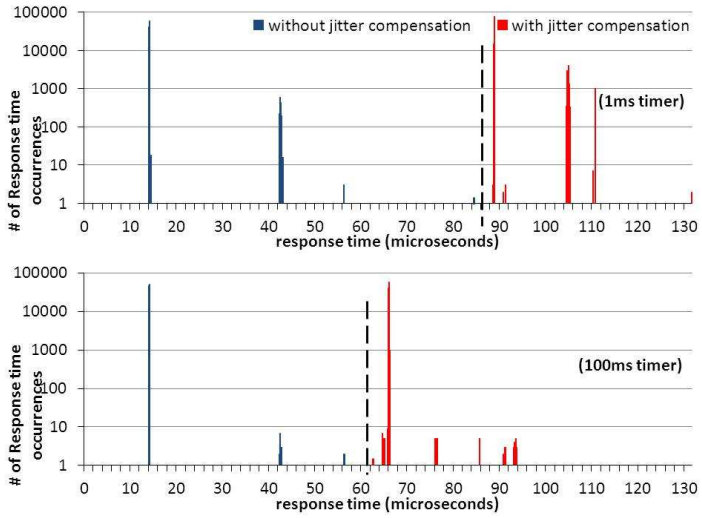


Figure 2.14 Response time and jitter: This plot shows the number of times plants experienced a given amount of response time. The distribution of these response times is a measure of how much jitter a give system configuration experienced. The parameter settings for this experiment were: # of plants = 100, sample rate = No Delay, Architecture = Case III.

from Case I (completely software) to Case IV (completely hardware) we see improved response times. However, there is a large decrease in response-time performance when moving form Case III to case IV. This indicates the vast majority of the main processor's time was spend executing context switching operations. It makes sense that hardware would perform this task much more efficiently since it stores all context information locally in BlockRAM (i.e on-chip memory), which can be accessed in a single clock cycle. While the main processor potentially has to fetch context information from main memory.

Third, all of the reference architectures have response-time jitter on the order of tens of (up to ∼100) microseconds, Figures 2.12 - 2.14. Digital control theory assumes the response time and sampling time are periodic, thus unbounded-jitter can cause serious instability issues [126]. Using jitter compensation methods help reduce this jitter, but having the PID loop deployed as a coprocessor eliminates jitter from such sources as interrupts and cache misses.

Resource utilization As can be seen in Figure 2.15, the resources utilization of the PID coprocessor is negligible with respect to the NIOS-II processor. Even the combination of the two utilize less than 30% of logic resources. In terms of on-chip memory utilization, the NIOS-

II uses less than 40% of these resources. Thus, well over 50% of the FPGA is available for implementing additional functionality.

	Resources			
	LUTs	Flipflops	Memory(bits)	18x18 Multiplier
PID coprocessor	0.32% (80/24,624)	0.28% (68/24,624)	0.16% (96,000/608,256)	4.55% (3/66)
NIOSII	25.67% (6,320/24,624)	22.67% (55,82/24,624)	37.95% (228,672/608,256)	3.03% (2/66)

Figure 2.15 Table showing the device utilization of our Context-switched PID system

2.6.2 Sensor Processing Unit

This section presents the results of the SPU experimental setup described earlier in Section 2.5.2.

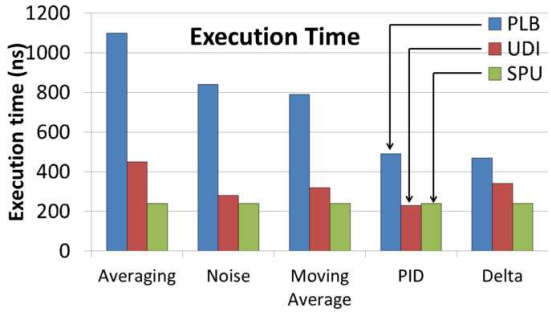


Figure 2.16 Comparing the execution time of various common kernels on all three experimental setups. As seen here, the SPU is typically fastest for most kernels.

Execution Time As can be seen from Figure 2.16, the execution time for the SPU is the lowest for most kernels. The execution time of both the PLB and UDI cases varies with tasks complexity. Also as the number of sensors increases, execution time for the PLB and UDI implementations increases (see Figure 2.17). On the other hand, the SPU execution time remains fairly constant for functions directly supported by the SPU. For functions not directly supported by the SPU, execution times do increase with the complexity. The execution time of the SPU remains constant as the number of sensors increases. However, as the number of sensors supported increases, the hardware resources required also increases. Across all cases, the average speedup obtained using the SPU was 2.48 fold faster than the PLB setup. Compared to the UDI setup, the SPU showed an average speedup factor of 1.38. This shows

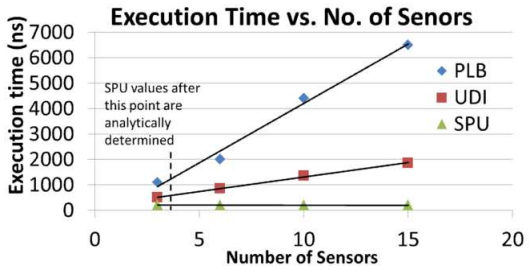


Figure 2.17 The effect of increasing the number of sensors on the execution time of the "Average" kernel. As seen here, regardless of the number of sensors, the time taken for reading the result from the SPU remains constant.

the speedups obtained over the PLB setup are not solely due to the SPU using the faster coprocessor interface.

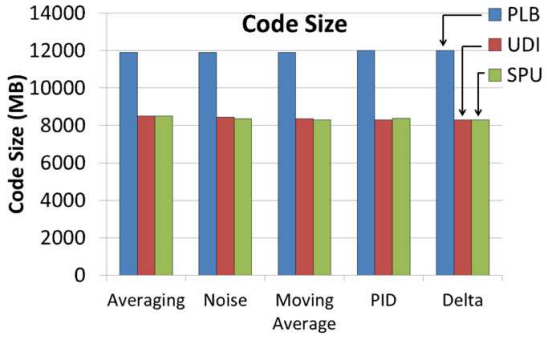


Figure 2.18 The executable binary size for various kernels for all three experimental setups. As seen here, the SPU case is the smallest in most cases.

Code Density As shown in Figure 2.18, the SPU presents a clear advantage over the PLB setup in terms of code density. This is expected since communicating over the PLB would entail some extra instructions for the arbitration of the PLB. On average the programs that used the SPU were 68.6% smaller than programs that used the PLB. The difference in code size between the SPU and UDI cases were negligible. Thus, even if the SPU itself is not implemented on a processor, these results suggest that implementing such a single instruction method of sensor access gives improvements in code size, as compared to using a standard shared peripheral bus.

Response Time As is clearly seen in Figure 2.19, the response time of the SPU is much higher than the other two setups due to the overhead of performing a context switch by the

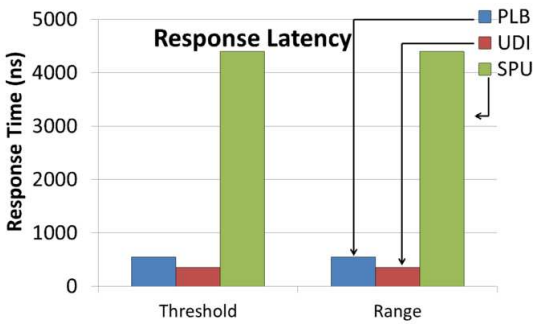


Figure 2.19 Comparing the response across setups. The PLB and UDI setups use polling, and the SPU generates interrupts. Although the SPU case has a large latency, offloading event monitoring to the SPU could relieve the main processor of a major computing burden when monitoring rare events for a high sample rate sensor.

interrupt handler. However, we have considered only the simplest of cases where in the PLB and UDI setups continuously polls the sensor without doing anything else. In situations where the response time is tightly constrained, such tight polling may be the only option, but in cases where the response time is allowed to be larger, the SPU presents another option. Using the SPU, the processor can continue executing other tasks while the sensor monitoring responsibility is relinquished to the SPU. The rest of the processor could even be put into a low power state, and woken up only when specific conditions are met. Exploring methods to reduce the context switching overhead in embedded microprocessors in order to allow for better response times (e.g. as in [129]) would be beneficial for such use-cases.

Resource utilization Figure 2.20 shows that the resource utilization of the SPU is well under 1% of the resources available on the Virtex5 FPGA (FX70).

	Resources	
	LUTs	Flipflops
SPU coprocessor	0.4%(206/44,800)	0.2%(108/44,800)

Figure 2.20 Table showing the device utilization of our Sensor-Processing Unit

2.7 Conclusion & Future Directions

An architecture capable of supporting the control of systems requiring tightly bound microsecond response times was presented. Our time-multiplexed hardware PID controller, which

is tightly integrated with a soft-core embedded processor, scales to support hundreds of PID control loops while maintaining response time performance. This architecture offloads the time critical PID computations to a custom functional unit, allowing non real-time tasks to execute on the soft-core processor without impacting the PIDs response times.

Response time and jitter characterization was performed on a software only implementation of multiple PID controllers, along with two alternative software/hardware codesign architectures that can be viewed as intermediate architectures between the software-only architecture and our final time-multiplexed hardware architecture.

A sensor processing unit (SPU) was discussed and evaluated for offloading common sensor processing kernels of computation.

Future directions for this work are: 1) deploying the Linux operating system on to the soft-core processor, while running PID controllers on the custom PID functional unit, 2) extending our time-multiplexed hardware implemented PID architecture to support cascaded PIDs, 3) developing additional custom functional units to support the design and evaluation of more complex control algorithms, 5) extending our SPU to support more advanced sensor processing tasks (e.g. sensor fusion using Kalman filtering), 6) potentially making these modules dynamically swappable at run-time, 7) implementing more advanced in hardware scheduling schemes (e.g. rate-monotonic, earliest deadline first) for scheduling plant servicing, and 8) using the SPU to condition sensor data and forward results directly to the PID coprocessor.

CHAPTER 3. AN FPGA-BASED PLANT-ON-CHIP PLATFORM FOR CYBER-PHYSICAL SYSTEM ANALYSIS

A paper published in IEEE Embedded Systems Letters Special Issue on Rigorous Modeling and Analysis of Cyber-Physical Systems, March 2014

Sudhanshu Vyas [1] Chetan Kumar N G[2], Joseph Zambreno[3], Chris Gill[4], Ron Cytron[4], and Phillip Jones[5]

Abstract

Digital control systems are traditionally designed independent of their implementation platform, assuming constant sensor sampling rates and processor response times. Applications are deployed to processors that are shared amongst control and non-control tasks, to maximize resource utilization. This potentially overlooks that computing mechanisms meant for improving average CPU usage, such as cache, interrupts, and task management through schedulers, contribute to non-deterministic interference between tasks. This response time jitter can result in reduced system stability, motivating further study by both the controls and computing communities to maximize CPU utilization, while maintaining physical system stability needs. In this paper, we describe an FPGA-based embedded software platform coupled with a hardware plant emulator (as opposed to purely software-based simulations or hardware-in-the-loop setups) that forms a basis for safe and accurate analysis of Cyber-Physical Systems. We model and analyze an inverted pendulum to demonstrate that our setup can provide a significantly more accurate representation of a real system.

[1]Primary researcher and author
[2]Graduate Student, Department of Electrical and Computer Engineering, Iowa State University
[3]Associate Professor, Department of Electrical and Computer Engineering, Iowa State University
[4]Professor, Department of Computer Science and Engineering Engineering, Washington University, St.Louis
[5]PI and author of correspondence

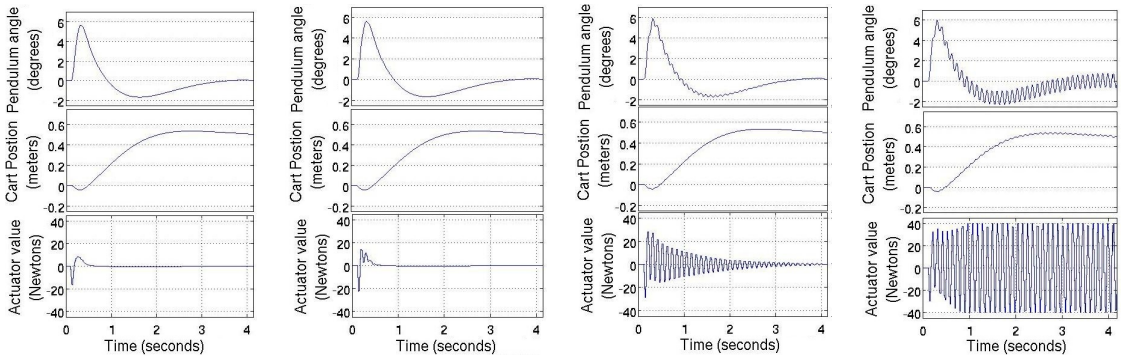


Figure 3.1 Effect of computational delay on a digital control system. With the sample period fixed at 15ms, the delay is varied from 15% (a) to 90% (d). At 65%, a ringing begins to appear in (b), which becomes more pronounced at 85% (c). Finally, at 90% (d), the plant remains stable while the controller continuously oscillates.

3.1 Introduction

Embedded systems and digital control theory have independently developed into mature fields, despite the clear connection between controllers and embedded platforms. Initially, each digital control loop was implemented on a dedicated processor, thus maintaining a *separation of concerns*. The demand for tighter system integration and the use of economical commercial-off-the-shelf products has blurred this separation [3]. In modern systems, the tasks running on the processor unknowingly compete for processor resources. These resources, meant to improve average resource usage for non-real-time systems, are becoming sources of non-deterministic computation time or *computation jitter*. Example causes include interrupts [3], branch misprediction [34], cache misses [105], and task management through operating systems [104]. These features limit the degree to which time invariance can be guaranteed, and cause systems to break control engineers' key assumption of constant sample rates and processor response time [5]. Ultimately, control loop robustness is greatly affected by this transition from a dedicated processor system to an environment of tasks competing for resources [17]. Thus, a more holistic view is now needed to develop and deploy controllers that take into account cyber-architecture artifacts on a system's physical stability.

As a motivating example, Fig. 3.1 shows the timing response of an inverted pendulum model as we vary the computational delay (the time between receiving a sensor sample and sending

the response), while holding sensor sample rate constant. In Fig. 3.1(a), a controller computing delay that is 15% of the state sampling rate has negligible impact on the system's stability. As the delay increases to 65% of the sample period (Fig. 3.1(b)), some ringing in the control signal becomes apparent. Progressing to a delay of 85% of the sample period (Fig. 3.1(c)) causes the plant to become less stable with oscillations that are now more pronounced. It is interesting to note that the state of the plant (i.e. cart position and pendulum angle) still appears stable. A further increase in the computational delay (Fig. 3.1(d)) leads to loss of controller stability resulting in an eventual fall for the pendulum.

Previous work has identified jitter in Cyber-Physical Systems (CPS) as a significant research challenge. The authors in [105] worked on characterizing Linux for real-time applications and found that the sources of jitter were implicit to the processor and were not completely correctable through software. A detailed analysis of branch-prediction schemes [34] concludes that static branching schemes work better for real-time systems than dynamic branch prediction. In [13], the authors compare several scheduling methods and concluded that deadline advancement was the most consistent, with minimal degradation in performance of controllers as the number of tasks increased and had relatively consistent low jitter. Controls experts are developing toolflows, like TrueTime-JitterBug, to evaluate the impact of a controller's response-time jitter on closed-loop stability [17]. In [18, 36] the authors have developed a set of stability criteria for closed-loop systems in which the sample rate contains jitter. In [21], a quantitative metric similar to the concept of phase margin is proposed, called jitter margin, which is the upper-bound of delay that a control loop can tolerate before going unstable. In an approach closely related to ours, the delay and period of control loops are used in a cost function, which is then treated as a minimization problem [10], and later a convex optimization problem [128]. A limitation of many of the previous approaches is their reliance on analytical tools and simulations of CPS which mask the jitter caused by hardware architectures.

In contrast, this paper presents the design and implementation of a control systems emulation framework that couples plant emulation hardware with an embedded processor, together on a Field-Programmable Array (FPGA)-based platform. This hardware/software framework allows us to more accurately study the interaction between an actual processor and a Plant-

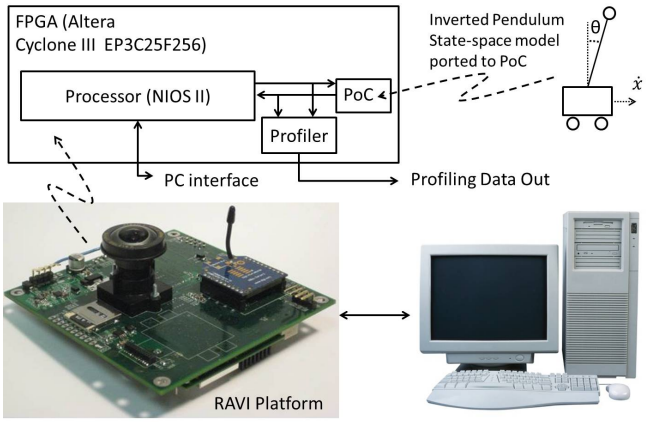


Figure 3.2 Our experimental setup implemented on our in-house reconfigurable platform, RAVI. Note θ and $\dot{x}$ of the plant model.

on-Chip (PoC). Our experimental results, using a state-space model of an inverted pendulum as captured in the PoC hardware, indicate that this proposed framework both safely and accurately captures the non-deterministic effects of modern processor architecture on a physical plant. Since the setup uses the same interfaces that the actual system would use, once the PoC is replaced by the real plant, the input and output jitter from sampling and actuating are already accounted for in the platform. The PoC could be integrated via on-chip or off-chip networking interface to emulate plants being controlled over a network.

3.2 Architecture

Figure 3.2 illustrates our FPGA-based infrastructure for CPS analysis. The FPGA is configured to implement the three main components: 1) an embedded processor (NIOS II) with conventional architectural features that is capable of running a modern operating system (OS), 2) a custom PoC emulator that implements a given model for the system under test, and 3) a profiler module that collects appropriate performance data and reports back to a host workstation. Our in-house reconfigurable platform, the RAVI (Reconfigurable Autonomous Vehicle Infrastructure) board, is also shown in Fig. 3.2. This small form factor (90 grams and 3.4"x3.4") board was specifically designed and fabricated at Iowa State University to promote the development of efficient control systems for mobile autonomous vehicles, hosting an Altera Cyclone III FPGA for deploying the computational stack, an inertial measurement unit (IMU)

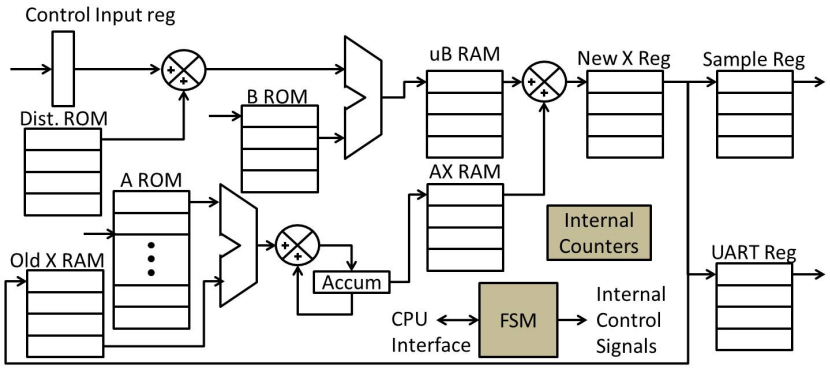


Figure 3.3 Register Level architecture of the state-space based Plant-on-Chip emulator.

for monitoring physical dynamics of vehicles, and other features that enable it to support a wide range of autonomous vehicles and applications.

Our proposed dedicated hardware (Fig. 3.3) emulates the state-space model of the chosen physical plant. Our example plant is an inverted pendulum from [97], where the state-vector, X consists of four variables, the pendulum's angle θ and angular rate $\dot{\theta}$, and its cart's position x and velocity $\dot{x}$. u is the input variable that comes from the controller to stabilize the plant and is stored in the 'Control Input reg'. The previous state of X is stored in the 'Old X RAM'. The feed-back matrix A and input matrix B are constants and thus stored in 'A ROM' and 'B ROM'. The new state of X is calculated by the hardware, with the help of a finite state machine (FSM) and internal timers, as follows. u is sequentially multiplied with the 'B' matrix and the result stored in 'uB RAM'. Next, the dot products of X with each row of A is sequentially calculated with the help of the accumulator and stored in the 'AX RAM'. Then, the addition of vectors Bu and Ax is performed, resulting in the new, updated state X and stored in the 'Xnew RAM'. The processor may sample X at any time through the 'Sample Reg'. A hardware interface is dedicated to non-intrusive transmission of X, u, and their respective time stamps through the 'UART Reg'. Other important evaluation metrics like sample-to-actuation time delay and the energy consumed by actuators are performed during post processing from the recorded data.

We require a noise source to emulate a noisy environment and test robustness in the same manner as JitterBug [17]. This is implemented with the 'Dist ROM' which contains a sample

array of a white-noise signal similar to JitterBug's disturbance. A value from the 'Dist ROM' is periodically injected into the system by adding it to the input u before starting a state-update. This emulates an external force being exerted on the cart and can be enabled or disabled through software by an application designer.

The current hardware utilization is fairly small with 2900 LUTs, 800 flip-flops, 32 DSP blocks and 1K of RAM/ROM. With a 50 MHz clock source, the emulator updates its state every 100μs, which is sufficient for emulating our example inverted pendulum plant. The advantage of our setup is that the states are periodically updated, independent of the processor controlling the hardware emulator. This eliminates the effects software simulations have on the computer they are usually running on (for example missing or late updates), especially when that computer is running the control algorithm, as well. The processor controlling this emulator cannot distinguish between the actual plant or its emulation, as the interface is unchanged and the hardware appears as an independent entity.

3.3 Experimental Setup and Results

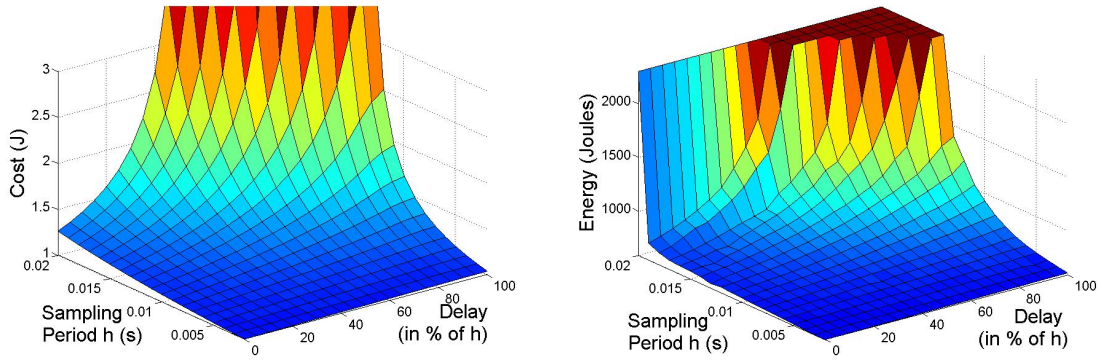
In evaluating our framework, we attempted a validation of the PoC against known control system evaluation tools and standards. Control systems can be evaluated based on transient response, energy consumption, or other cost functions. These metrics correlate with the amount of effort the controller exerts to keep the system stable after receiving a change either in reference value, or when experiencing an external disturbance. We shall now refer to this metric as J. For Jitterbug, J is an 'integration of square of error' [17], where error is the deviation of a designer specified variable from zero. The PoC's J is the energy (*Joules*) spent by the actuator. A secondary interest was in comparing the J's from JitterBug and the PoC. Initial experiments indicate that a JitterBug cost function of $J = \sum_{t=0}^{t=time_{sim}} \left({e_{\dot{\theta}}}^2 + {e_x}^2 + {e_{\dot{x}}}^2 \right)$ was closest to the amount of energy used by the actuator to keep the pendulum upright. Method 1 describes our routine for characterizing system costs. We explored the design space by varying sample period and computational delay and measured the cost in JitterBug and the energy in our setup to keep the system stable. The points where JitterBug's plots trend to infinity (equivalent to the plateau region of our setup's plots) correspond to the unstable regions of

the system. To give a physical perspective, these regions correspond to our pendulum example losing balance.

Method 1 Control system cost profiling

```
1: procedure INITIALIZE
2:     for period = 2ms → 20ms do
3:         for delay = 0% → 100% do
4:             run simulation for time_sim
5:             J ← calculate energy                          ▷ or cost
6:             Array_J ← J; delay ← delay + 5%
7:         Matrix_J ← Array_J; period ← period + 1ms
```

We conducted two sets of experiments. First, we attempted to maintain the pendulum cart at a fixed location, given various external disturbances. This can be done in JitterBug and in our setup. Next, we tested our setup with a step-response, which JitterBug does not permit. We performed a profiling of the relevant cost, as outlined in Method 1, and fed this data into Matlab to create the following surface plots.



(a) JitterBug (b) Plant-on-Chip

Figure 3.4 Surface plots of cost (a) and energy (b) while injecting disturbance in cart position x

Figure 3.4 gives a summary of the first experiment's results. We see common trends in both setups. As we increase the computational delay from 0% of the sample period to a full sample period, the cost (Fig. 3.4(a)) of keeping the system stable and the amount of energy (Fig. 3.4(b)) needed by the system to keep the system stable increase in a similar fashion. Both setups show an increase in cost and energy as the sample period of the controller is increased.

The region of instability is almost the same in both setups, with the PoC setup showing a slightly smaller region. An example point is where sample period is 15ms and delay percentage is 70%. JitterBug shows that the system will be unstable whereas the PoC setup indicates that the system will be stable, but will spend more energy to maintain stability. This difference is because the pattern and magnitude of JitterBug's external disturbance is unknown and an estimated pattern is used in the PoC setup. The major difference between the setups is that JitterBug predicts that the system will be stable when the sample period is 20ms and delay is roughly 40% or less. Since the PoC is a more realistic setup and shows that a 20ms sample period even with no delay will be unstable, we can safely say that JitterBug's prediction is less accurate.

While analyzing our setup's step response (Fig. 3.5) to different combinations of sample period and delay, we can refer back to Fig. 3.1 for additional clarity. Keeping the sample period fixed to 15ms, let us observe the impact of increasing delay. At 15% delay, the system is very stable in the time response plot (Fig. 3.1(a)) and is in the dark-blue plain of Fig. 3.5. As we increase delay, we start seeing a damp oscillation in the controller signal begin to increase in Figs. 3.1(b) and 3.1(c) and the energy increase and climb the cliff of the surface plot of Fig. 3.5. At 95%, the system is unstable (Fig. 3.1(d)) and the corresponding point on the surface plot is on the plateau, further indicating instability. A JitterBug version of this test is not possible as the reference value cannot be set by a user to produce a step input.

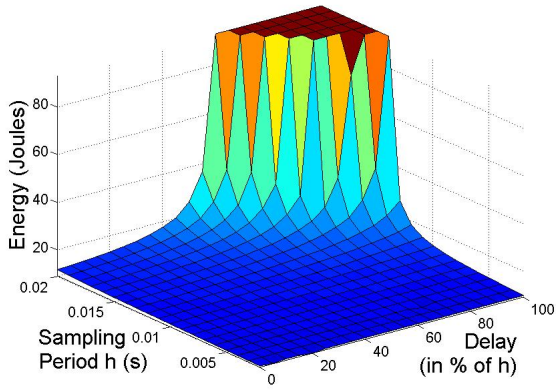


Figure 3.5 Characterization plot of PoC's step-response

3.4 Conclusion

We presented a method for analyzing Cyber-Physical Systems using a hardware plant emulator we designed and integrated with an embedded processor in an FPGA-based platform. Our framework provides insight for embedded designers into how computer architecture can influence control loops. Though current simulation-based design tools provide a good approximation of a system's robustness to sample-period and delay, they work in environments and with assumptions that the delay can be modeled as a probability distribution function [13] [17]. Research [34] [105] [104] shows this to be not realistic and that computer elements cause non-deterministic time-varying delay and sample-period. With an actual processor under test, our setup inherently contains these non-deterministic sources of delay jitter and thus gives a more accurate result, when characterizing a system's robustness against sample period and delay variation.

In the future, we plan to control a plant-on-chip emulator while sharing processor resources with other tasks, using a real-time operating system (e.g. RT-Linux).

CHAPTER 4. A SOFTWARE CONFIGURABLE COPROCESSOR-BASED STATE-SPACE CONTROLLER

A paper published in International Conference on Field-programmable Logic and Applications (FPL), September 2015

Aaron Mills[1], Pei Zhang[1] Sudhanshu Vyas[1] Joseph Zambreno[2], and Phillip Jones[3]

Abstract

We present a software configurable coprocessor-based state-space controller that can control physical processes representable by a linear state-space model. Our proposed architecture has distinct advantages over purely software or purely hardware approaches. It differs from other hardware controllers in that it is not hardwired to control one or a small range of plant types (e.g. only electric motors). Via software, an embedded systems engineer can easily reconfigure the controller to suit a wide range of controls applications that can be represented as a state-space linear model. Additionally, we introduce a novel design methodology to help bridge the gap between controls and embedded system engineering. Control of the well-understood inverted pendulum on a cart is used as an illustrative example of how the proposed hardware accelerator architecture supports our envisioned design methodology for helping bridge the gap between controls and embedded software engineering.

[1]Graduate student researched and author
[2]Associate Professor, Department of Electrical and Computer Engineering, Iowa State University
[3]PI and author of correspondence

4.1 Introduction

Field-programmable gate arrays (FPGAs) are of growing interest in the area of applied control theory [85]. In addition to the massive parallelism available on FPGAs that can potentially be utilized to obtain high controller update rates, software-hardware co-design using FPGAs can help separate embedded software concerns (e.g. real-time scheduling feasibility), from controls concerns (e.g. accounting for update-rate jitter).

Implementing an efficient controller on an FPGA can be challenging for engineers unfamiliar with hardware architecture design. One solution is software programmable hardware. In our work, we describe a software-configurable FPGA co-processor architecture that can implement a wide range of linear state-space controllers, up to the complexity of a Linear Quadratic Regulator (LQR) coupled with a Luenberger Observer.

For the purpose of evaluation, the controller can be interfaced to a hardware-based emulation of a physical plant (i.e. Plant on Chip) [125]. This arrangement is depicted in Fig. 4.1. The Plant on Chip (PoC) allows for rapid, and consistent testing of control algorithms and system platform configurations. Once stability of the emulated plant is achieved, it can be replaced with an interface to the actual plants sensors and/or actuators. All control computations are done in hardware, while software running on the CPU is used to initialize the co-processor. The software is also free to perform other tasks, for example, task scheduling, path planning, video processing, and interactive communications.

The remainder of this paper is organized as follows. In Section 6.2, we discuss and compare related works in this problem space. In Section 4.3, we describe the detailed design of our software configurable co-processor based state-space controller, and provide a brief overview of our plant on chip architecture. In Section 4.4, we present an illustrative example of using our co-processor to evaluate the use of hardware verses software for an embedded controls application, and explore the performance and scaling of our coprocessor-based controller. Section 6.6 concludes this paper and provides avenues of future work.

4.2 Related Work

Kozak, in [75], surveys trends in the field of applied controls, in which we see controls have evolved from manually-tuned single-input single-output (SISO) controllers to multiple-input multiple output (MIMO) H_∞ controllers and model-predictive controllers (MPC). The latter types of algorithms are computationally intense and can introduce significant latencies when implemented with off-the-shelf processing platforms. Kozak additionally suggests that a software-hardware co-design approach for implementing advanced controllers (e.g. H_∞) in FPGAs would enable designers to make better use of these complex controllers in high-speed systems. Monmasson, in [86], makes a similar suggestion, pointing out how different parts of a control algorithm are better suited for different types of hardware. However, locating the optimal software-hardware partition is still a challenge.

There are numerous examples of application-specific FPGA-based controllers in the literature. An example of a system requiring very fast control update rates appears in [98], in which a high-speed pan/tilt camera is designed to track objects. In order to reach the 3.5ms update rate, a dedicated PC is used to perform image processing and produce motor control signals. It is noted that the PC introduced considerable delay in the feedback loop. Another application requiring very high update rates appears in [119], which presents an application-specific design that used machine vision to control an inverted pendulum. In [51], the authors developed a self-tuning state-space controller using a multiply-accumulate unit which is interfaced with a digital signal processor (DSP). This paper demonstrated the use of FPGAs to control a plant with non-constant plant parameters. In [8], the design of a high-speed, hardware-only, fixed-point MPC is discussed. Finally, in [67], an MPC is implemented on an FPGA and is shown to allow for a significantly faster sample rates than a PC running at a higher clock frequency.

Compared to software, implementing high-performance control algorithms in hardware is time consuming and leads to application-specific solutions. A proposed solution to this issue appears in [123] and [11], which use a co-processor to perform low-level repetitive matrix operations for MPC. This allows control designers to use software for the high-level logic; however, to do so they must work with a custom floating-point format and instruction set.

A general summary of approaches used to implement controllers on FPGAs appears in [90]. In [90] a call is made for designs that make efficient use of the massive parallelism available on FPGAs, while retaining the generality and flexibility available to software solutions. Our work pursues this goal. In summary, a number of works exist describing controllers that achieve reduced computational delay. However, these controllers are designed to solve specific problems, unlike our fully software configurable solution. Garbergs, in [38, 39], presents the closest vision and architecture to our approach. The main differences being: 1) their design does not take into account scaling to different sized controllers (e.g. if controller coefficients change the design must be re-implemented), 2) their design is intended to be standalone, as compared to being a memory-mapped co-processor, and 3) their vision focuses more on developing a fast hardware controller as opposed to supporting a design methodology that bridges the gap between embedded software developers and controls engineers.

4.3 System Architecture

The targeted hardware platform is the Xilinx Zynq 7000 series system-on-a-chip (SoC). The Zynq SoC consists of an ARM Cortex A9 processor coupled with an FPGA. The Xilinx toolchain for the Zynq directly supports hardware-software co-design, making the platform ideal for developing co-processor based applications. FPGA components are written in VHDL and software components are written in C.

The AXI bus is a 32-bit wide standardized interface which allows a co-processor to communicate with the CPU, or allows independent communication among co-processors. As shown in Fig. 4.1, the PoC and the controller both have slave interfaces, which allow their internal memory spaces to appear to the CPU as memory-mapped peripherals. Meanwhile, the AXI Bus master interface on the LQR controller allows it to sample the output of the PoC while the system is running.

Besides the model coefficients, both controller and PoC must be configured with three constants which represent the size of the state-space model, and allow boundaries to be computed for internal memory fetches. The controller also is configured with a particular sample rate, which represents the number of clocks it will wait before sampling the PoC output memory.

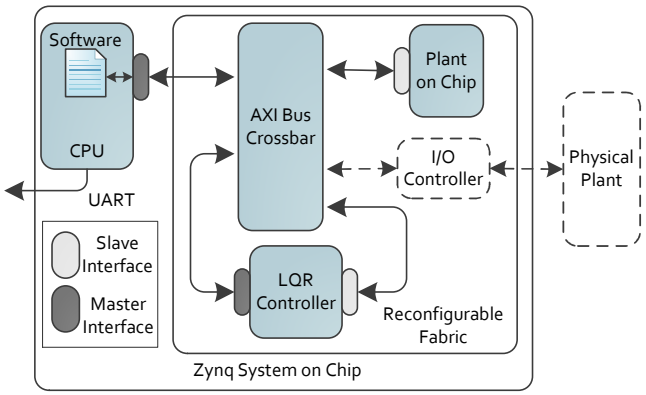


Figure 4.1 System Overview. Both the controller and Plant on Chip (PoC) are fully software configurable over the shared AXI bus. A software or hardware controller can control the PoC without requiring hardware reconfiguration.

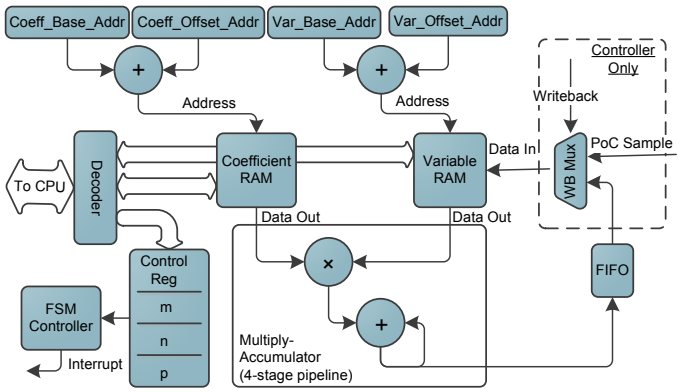


Figure 4.2 Architecture datapath. The dot-product result for each row is stored in the FIFO. After all rows are processed, the FIFO writes results back in the same cycle as they are read back out for the next update operation. An interrupt signal can be used to notify the CPU when values are updated.

1. m: the number of plant model inputs.

2. n: the number of plant model states.

3. p: the number of plant model outputs.

Dot-products between the coefficient rows and variable vectors are performed in a straightforward manner using a pipelined multiply-accumulate unit, as shown in Fig. 4.2. These operations are performed as single-precision floating point to avoid the limitation on precision and tedious preprocessing work associated with fixed-point math, as well as to increase the

ease with which the co-processor integrates with software. The resource usage of the system is summarized in Table 4.1; in total 5% of available Slices are consumed.

Table 4.1 System Resource Usage on Zynq XC7Z020

	Slices	LUTs	BRAM/FIFO	DSP48E1
PoC	443	1376	4	7
Controller	447	1378	5	3
Total	**890**	**2754**	**9**	**10**

4.3.1 General Linearized Model of Plant

Both the PoC and controller computations are centered on a standard linear state-space model of a physical plant. This generic system model consists of matrices A,B, and C[4] and is formulated as follows:

$$x_{k+1} = Ax_k + Bu_k \tag{4.1}$$

$$y_k = Cx_k \tag{4.2}$$

These equations allow one to compute the next system state x_{k+1} based on the current state x_k, given a particular input u_k. Additionally the output y_k is computed at each time step k based on the current state and input value.

4.3.2 Co-processor Memory Space

The controller and PoC each have two separate dual-ported memories to facilitate parallel data access: one for coefficients (A, B, C, L, K), and one for variables (x, y, u). The memory map as seen by the CPU is shown in Fig. 4.3. Each memory is dual ported with independent outputs. One port is dedicated to interfacing with the CPU for memory initialization and read back, and the other port dedicated to internal operations.

As a simple optimization, matrices which are involved in the same dot-product are concatenated in memory where it is possible. This prevents two extra, unnecessary load-store cycles,

[4]We omit matrix D which is rarely required.

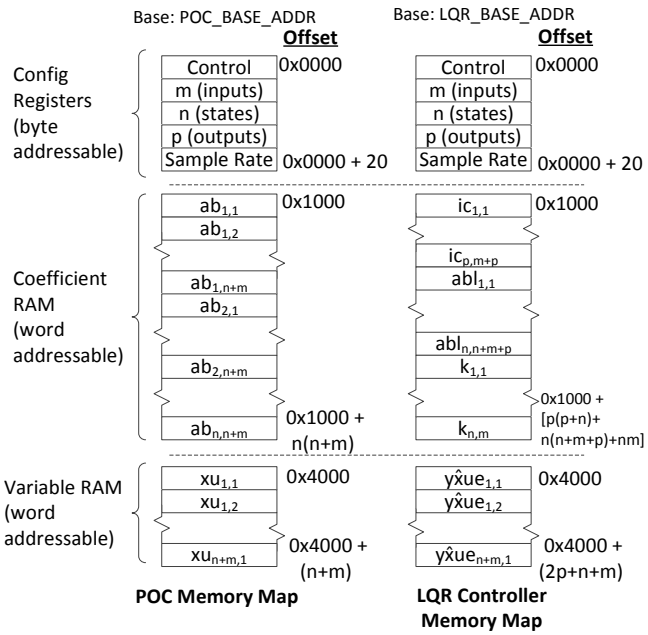


Figure 4.3 Coefficients and variables occupy independent, packed memory spaces, with matrices laid out in row-major order.

and is explained in greater detail in subsequent sections. All data is loaded by the CPU into a contiguous block at the coefficient or variable memory base address, with only the values of m, n and p being needed to compute arbitrary matrix boundaries. Compared to providing a fixed address for each matrix, this scheme maximizes memory efficiency and plant model flexibility, since there is no real architectural constraint on m, n or p other than the overall memory size of the target FPGA.

4.3.3 Plant on Chip Algorithm

The A and B matrix, and the x and u vectors, are concatenated in order to allow more efficient pipelining of multiply-accumulate operations. All computations are performed in single precision floating point format. The PoC computes the equations in Equations 4.3 and 4.4. In most applications the D matrix is not needed and therefore excluded.

$$\begin{bmatrix} x_{1|k+1} \\ \vdots \\ x_{n|k+1} \end{bmatrix} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} & b_{1,1} & \cdots & b_{1,m} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} & b_{n,1} & \cdots & b_{n,m} \end{bmatrix} \times \begin{bmatrix} x_{1|k} \\ \vdots \\ x_{n|k} \\ u_{1|k} \\ \vdots \\ u_{m|k} \end{bmatrix} \tag{4.3}$$

$$\begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} c_{1,1} & \cdots & c_{1,n} \\ \vdots & \ddots & \vdots \\ c_{p,1} & \cdots & c_{p,n} \end{bmatrix} \times \begin{bmatrix} x_{1|k} \\ \vdots \\ x_{n|k} \end{bmatrix} \tag{4.4}$$

4.3.4 LQR Controller Algorithm

An LQR controller is implemented to create a complete closed-loop plant control system. An LQR controller is based around the gain matrix K. A particular K is sought such that the feedback law $u_k = -Kx_k$ minimizes the quadratic cost function in Eq. 4.5 [25].

$$J(u) = \sum_{1}^{\infty} x_k^T Q x_k + u_k^T R u_k \tag{4.5}$$

Generating K requires the controls engineer to select state-cost matrix Q and performance index matrix R which work well for a given plant.

Although the complete value of state vector x can be read from the PoC at any time, many plant models include internal states which cannot be directly measured. Therefore, to increase its flexibility our controller also integrates a Luenberger-type observer model. Here L denotes the gain matrix of the observer, and y is the measurable states sampled from the plant.

$$\hat{x}_{k+1} = A\hat{x}_k + Bu_k + L(y - C\hat{x}_k) \tag{4.6}$$

This addition allows the controller to estimate the value of the unmeasured plant states as $\hat{x}$; if all states are observable then one merely configures n equal to p for the co-processor. It is this estimated state vector which is used to generate the control vector u. This process is shown in Fig. 4.4.

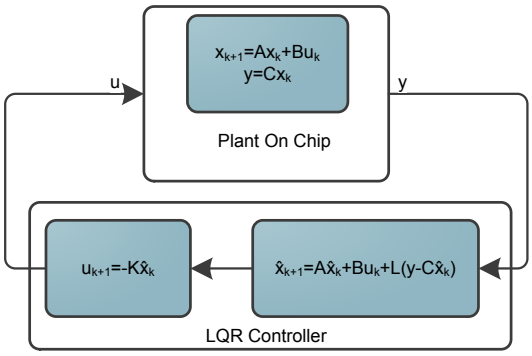


Figure 4.4 LQR controller with observer controlling an emulated plant (PoC).

The control update operation is split into three phases, shown in Equations 4.7-4.9. Matrix $I_{p\times p}$ is the identity matrix, and matrices K and C are negated before loading into memory to eliminate the need for subtraction. Matrix sizes in terms of the user-configured constants m, n and p are included as subscripts.

$$E_{p\times n} = \begin{bmatrix} I_{p\times p} & -C_{p\times n} \end{bmatrix} \times \begin{bmatrix} y_{p\times 1} \\ \hat{x}_{n\times 1|k} \end{bmatrix} \tag{4.7}$$

$$\hat{x}_{n\times 1|k+1} = \begin{bmatrix} A_{n\times n} & B_{n\times m} & L_{n\times p} \end{bmatrix} \times \begin{bmatrix} \hat{x}_{n\times 1|k} \\ u_{m\times 1} \\ E_{p\times 1} \end{bmatrix} \tag{4.8}$$

$$u_{m\times 1} = \begin{bmatrix} -K_{m\times n} \end{bmatrix} \times \begin{bmatrix} \hat{x}_{n\times 1|k+1} \end{bmatrix} \tag{4.9}$$

Additionally, the addresses applied to the Coefficient RAM and Variable RAM are split into base and offset addresses in order to increase flexibility. The pseudocode in Algorithm 2 demonstrates the computation of memory addresses for the dot product operation. Note that address calculation arithmetic is comprised of only small integers, and is only computed once during initialization.

4.4 Experimentation

The performance of the co-processor is tested in this section. For the experimental setup, the Zynq's ARM processor and FPGA fabric are both clocked at 50Mhz, and memory caching

Algorithm 2 LQR with Observer using variable m, n, and p

1: **procedure** INITIALIZE
2: $v_{base} \leftarrow \{0, p, p\}$ ▷ Compute memory boundaries (implemented as mux).
3: $c_{base} \leftarrow \{0, 2p(p+n), (2p+2n+m)(2p+3n)\}$
4: $v_{max} \leftarrow \{(p+n)-1, (n+m+p)-1, n-1\}$
5: $c_{max} \leftarrow \{2p(p+n)-1, (n+m+p)(3n)-1, mn-1\}$
6: **procedure** UPDATE
7: **for** $j \leftarrow 0$ ... 2 **do** ▷ For each computation phase...
8: $sum \leftarrow 0$
9: **for** $c_i \leftarrow 0$ ... $c_{max}[j]$ **do** ▷ For each coefficient...
10: $sum \leftarrow sum + \text{COEF\_MEM}[c_{base}[j] + c_i] \times \text{VAR\_MEM}[v_{base}[j] + v_i]$
11: **if** $v_i = v_{max}[j]$ **then**
12: WBFIFO $\leftarrow sum$ ▷ Save dot product for this row into writeback FIFO.
13: $v_i \leftarrow 0$
14: $sum \leftarrow 0$ ▷ Reset accumulator.
15: **else**
16: $v_i \leftarrow v_i + 1$

is disabled. We configure the ARM processor to 50 Mhz to represent a lower-powered embedded processor, and disable cache to emulate safety critical systems that require highly deterministic timing. Traditionally, both *delay* and *jitter* are considered as critical parameters for determining the stability of a controller [1]. In this paper, we only consider delay, since the software controller is single-threaded, and the hardware controller is naturally jitter-free. In particular we are concerned with the effect that controller update delay has on plant stability. The experiment performed in this section presents a test plant requiring a sample rate of 2ms to maintain stability. As shown, the hardware controller shows a clear advantage over the software controller, as the former can maintain plant stability for large state-space models, whereas the latter cannot due to computational delay.

4.4.1 Test Plant

The plant used during testing is the classic inverted pendulum, a non-linear model illustrated in Fig. 4.5. The model parameters are shown in Table 4.2. Meanwhile, the state vector consists of four states: x, $\dot{x}$,ϕ, and $\dot{\phi}$. This model requires the CPU set co-processor parameters $n = 4$, $m = 1$, and $p = 2$.

A partial derivation of the system follows. The two nonlinear equations which describe the physics of the pendulum [58, 81] are shown below:

$$(M+m)\ddot{x} + b\dot{x} + ml\ddot{\theta}\cos\theta - ml\dot{\theta}^2\sin\theta = u \tag{4.10}$$

$$(I+ml^2)\ddot{\theta} + mgl\sin\theta = -ml\ddot{x}\cos\theta \tag{4.11}$$

We linearize the equations around the upright equilibrium position of the pendulum ($\theta = \pi$), assuming that the system will maintain a small deviation from this position. We introduce ϕ as this deviation (that is, $\theta = \phi + \pi$). We use small angle approximations of the trigonometric functions in the system equations to obtain a linearized version.

$$(M+m)\ddot{x} + b\dot{x} - ml\ddot{\phi} = u \tag{4.12}$$

$$(I+ml^2)\ddot{\phi} - mgl\phi = ml\ddot{x} \tag{4.13}$$

Finally we rearrange the expressions as a set of first-order differential equations so they can be put into state-space form.

$$\begin{bmatrix} x_{k+1} \\ \dot{x}_{k+1} \\ \phi_{k+1} \\ \dot{\phi}_{k+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & \frac{-(I+ml^2)b}{I(M+m)+Mml^2} & \frac{m^2gl^2}{I(M+m)+Mml^2} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{-mlb}{I(M+m)+Mml^2} & \frac{mgl(M+m)}{I(M+m)+Mml^2} & 0 \end{bmatrix} \begin{bmatrix} x_k \\ \dot{x}_k \\ \phi_k \\ \dot{\phi}_k \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{I+ml^2}{I(M+m)+Mml^2} \\ 0 \\ \frac{ml}{I(M+m)+Mml^2} \end{bmatrix} u \tag{4.14}$$

Note that the output expression in Equation 4.15 is configured to reflect the fact that only position x and angle ϕ are directly observable.

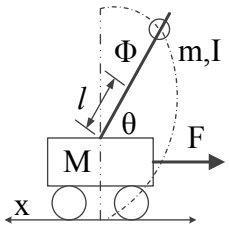


Figure 4.5 Inverted Pendulum Model

Table 4.2 Inverted Pendulum Model Symbols

Symbol	**Meaning**	**Initialization**
M	cart mass	2.725kg
m	pendulum mass	1.09kg
b	coefficient of friction	0.1 N/m/sec
l	length to pendulum center of mass	0.2 m
I	pendulum moment of inertia	$0.006kg \cdot m_2$
u	applied force	0
x	position displacement	0
θ	angle from downward vertical axis	N/A
ϕ	angle from upward vertical axis	$-5°$

$$y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_k \\ \dot{x}_k \\ \phi_k \\ \dot{\phi}_k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} u \tag{4.15}$$

At this point, Matlab is used for discretization and to solve the LQR minimization problem, thereby providing LQR gain matrix K.

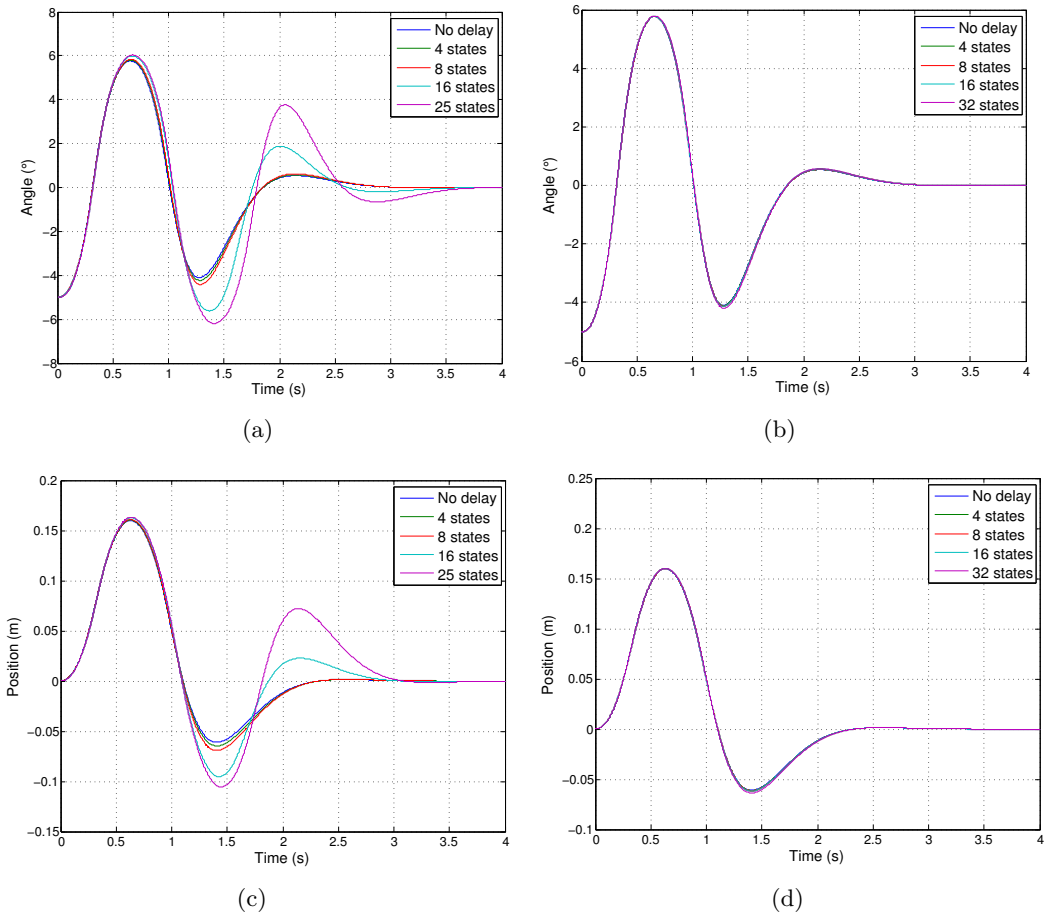


Figure 4.6 Impact of computation delay on plant stability. Software controller: Pendulum angle response (a) Pendulum position response (c). Hardware controller: Pendulum angle response (b) Pendulum position response (d). The plant responses are nearly indistinguishable with increasing delay.

4.4.2 Performance Analysis

The effect of controller computational delay on plant behavior is tested by setting the pendulum vertical displacement angle ϕ to $-5°$ so that it is initially unstable, and then increasing controller workload over repeated trials. Extra zeroed 'dummy' states are added to the 4 base states in order to modulate the controller delay without impacting the model itself.

Fig. 4.6 shows the response of the inverted pendulum plant as the number of control states is increased. At 16 states, the software response shows decaying oscillations, and beyond 25 states the plant becomes unstable. However, Fig. THe hardware implementation results show that increasing the computational workload has very little observable effect on the stability of

Table 4.3 Execution Time Comparison (μs)

States	Software	Hardware
4	183	9.9
8	428	20.66
16	1254	48.64
32	4071	127.14

the plant. Table 4.3 compares the software execution time to the hardware execution time as the number of states is increased.

It is also possible to estimate the maximum number of states supported by this architecture. Based on the memory map, the total memory required for both the PoC and controller can be computed as follows:

$$f(m,n,p) = 2n^2 + p^2 + 3nm + 2pn + 2(p+n+m) \tag{4.16}$$

Given that total amount of block RAM on the target device is 560kB, we consider two cases: one actuator and many actuators. Letting $n = p$, which is worst-case memory wise, if $m = 1$ then the maximum number of states is approximately 166. If we constrain $m = n = p$ then the maximum number of states is approximately 131.

4.5 Conclusion

We have presented a novel approach to designing a hardware-based co-processor for control applications, and have illustrated how this approach could be used to ease the transition from control theory to embedded control implementation. Avenues of future work include: 1) increase the modest parallelism exploited by our current co-processor implementation and preform more aggressive pipelining, 2) describe a more formalized procedure for moving controller designs from theory to implementation, and 3) explore the support of higher complexity controllers, such as, H_∞. Algorithm steps which involve irregular data or computation, and are therefore usually pre-computed offline, could be transfered to the CPU. Meanwhile, steps which involve regular data or computation are ideal for hardware accelleration.

CHAPTER 5. TIME-SPACE ANALYSIS OF A SCALABLE PROGRAMMABLE STATE-SPACE COPROCESSOR FOR DIGITAL CONTROL LOOPS

A paper submitted to Real-Time Embedded Technology & Applications Symposium (RTAS) 2016

Sudhanshu Vyas [1] Joseph Zambreno[2], and Phillip Jones[3]

Abstract

With the increasingly tighter constraints on digital control loops for higher performance, new methods of implementing controllers are needed. Embedded-system engineers face an ever growing challenge to meet control engineers' assumption of deterministic execution cycles with negligible variance while optimizing resource usage. Simultaneously, shortening the design cycle to cope with demands of faster design iterations is also needed. We propose a software programmable co-processor onto which control algorithms represented in state-space form can be offloaded to for systems with tight resource constraints . The host processor is freed to execute task that are less time-critical while the co-processor executes the controller with deterministic delay in the order of microseconds. We propose two scalable architectures which run multiple cores in parallel to reduce the computation time for state-space controllers with many coefficients. Implementing our designs on a field programmable gate array, we present our post place and route results show that systems as large as 173 states can be executed in less than 2.5μs of computational delay when running on a 32 core implementation.

[1]Primary researcher and author
[2]Associate Professor, Department of Electrical and Computer Engineering, Iowa State University
[3]PI and author of correspondence

5.1 Introduction

Traditionally, digital control theory and embedded systems have been treated as mutually exclusive topics of engineering. In embedded systems design, the processor must execute a task within a specific time-frame as opposed to the assumption in controller design that the task will execute at a certain point in time. As the number of tasks on an embedded system grows and the response specifications of the control-loop become more strict, the negative effects of unintended variance on the control loop's stability increase and may lead to total system failure [16]. Though state-of-the-art software scheduling schemes maintain the computation time within a bound, the controller may still be unstable. Gomez [42] shows that for a given plant which one may attempt to control, the stability may degrade by reducing the computational delay variance and even improve by distributing it.

There is a growing interest for FPGAs in the control engineering field as [87] argues that software-hardware co-designs unlock their potential as parts of the control algorithms by providing fast cycle-to-cycle update rates in hardware while implemented less time critical parts of the controller in software. For example one can implement adaptive control [6] as demonstrated in [50] where the controller in hardware may be reconfigured through software to account for model inaccuracies (due to steps for simplification like linearization) or changes in the system parameters for example from degradation with time. Unfortunately, implementing controllers on FPGAs itself requires training and can be challenging for unfamiliar engineers. A solution is using software reprogrammable hardware, where the control engineer provides the designed controller to an embedded systems engineer or on their own ports the design to the hardware. Many control systems with observers can be represented in state-space form as shown below

$$\hat{X}_{k+1} = A\hat{X}_k + BU_k \tag{5.1}$$

$$Y_k = C\hat{X}_k + DU_k \tag{5.2}$$

where $\hat{X}$ is the controller's estimated state vector, Y is the actuation vector A,B,C, and D are the feedback, input, output and feed-forward matrices. If a simple state-space controller

is being implemented, the control engineer can calculate the needed coefficients and pass this information to the embedded systems engineer who enters this information into a hardware state-space controller. We analyze the effects on execution time and hardware resource utilization as we introduce and increase parallelism in the architecture. The lower computation time in our architecture is achieved by manifolding the state-space computing core and the memory which stores the coefficients of the corresponding cores. We propose two strategies of scaling the controller memory while keeping the parallelizing architecture the same. The architectures are implemented on a Xilinx Zynq-7020 device. In the first architecture a memory module, implemented in look-up tables (LUTs) is dedicated to each core. The other architecture uses a single yet $32N$-bit wide RAM module made of block RAMs where N is the number of cores.

This paper makes two contributions. First, we propose a generic linear state-space calculating co-processor which can support a variety of control algorithms. Second, we explore the design space and identify the trade-offs between combination of resource usage and clock speed. The rest of this paper is organized as follows. We will look into the two architectures in detail and discuss their potential merits in section 5.3. The evaluation method and experimental setup is described in sections 5.4. Our hypothesis will then be compared to the synthesis results of our experiments in section 5.5, followed by the conclusion in section 5.6.

5.2 Related Work

The issue of execution time variance is being addressed by both sides. Control systems are being designed to be robust towards the non-deterministic elements of embedded systems while computer architectures, both software and hardware are being designed to come closer to the original assumptions of control theory. Cervin [16] shows that variance in I/O and computation results in an effect on stability and developed an improved criteria for analyzing input and output variance by realizing that there are certain periods in the execution cycle where the plant runs open-loop. But these methods only reduces the variance whereas [42] shows that simply reducing jitter will not guarantee performance. Depending upon the plant in concern, increasing variance may also improve stability. This is very counter intuitive and means simple reduction of variance is no long blindly applicable to all systems. We can see

in [110] that to make a system stable, the necessary delay that needs to be introduced can be calculated.

In [2] focus have design specifications where update rates and computation per unit time must be high. The trends of the field are described in a survey [76]. We see that controllers evolved from manually tuned single input single output (SISO) controllers to H_∞ which are MIMO controllers. This is a computationally intense algorithm and have large computational latencies when implemented with commercial off-the-shelf (COTS) processing platforms. The author concludes that a co-design approach to implementing H_∞ on FPGAs will allow designers to use them in high-speed systems.

An application of FPGAs in control is given in [99], where a high-speed pan/tilt camera is designed to track objects. A dedicated PC is required to achieve the $3.5ms$ update rate. [53] demonstrates how a Boeing 747 may be flown by running a model predictive controller on an FPGA with hundreds of microsecond execution time.

In [50], the authors develop a state-space controller using a multiply-accumulate unit and interface the custom hardware with a digital signal processor. Some of the challenges all controllers face are rounding errors due to finite word length and knowledge of FPGA programming. [124] and [12] address the latter by proposing an application specific processor for running control algorithms, allowing control designers to work with software instead of repetitively designing custom hardware, but forces them to work with a custom floating-point format and low level programming or at a new language. On the other hand, designing controllers specific to a scenario [77] [68] [7] [118] can be an uphill task even for those familiar with FPGAs.

5.3 Architecture

Before implementing the scalable architectures, the calculator proposed by [83] needed to be modified in two ways; first by creating a clear-cut interface between the calculator to memory and second by placing the FIFO in the adder's feedback path to improve throughput. Figure 5.1 shows the details of the computation core. Say we have a observer-controller represented by $\hat{X}(k+1)_{12\times1} = A_{12\times12} \times \hat{X}(k)_{12\times1} + B_{12\times6} \times U(k)_{6\times1}$. This equation can be rewritten as

$$\begin{bmatrix} \hat{x}_1 \\ \vdots \\ \hat{x}_{12} \end{bmatrix}_{k+1} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,12} & b_{1,1} & \cdots & b_{1,6} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{12,1} & \cdots & a_{12,12} & b_{12,1} & \cdots & b_{12,6} \end{bmatrix} \times \begin{bmatrix} \hat{x}_1 \\ \vdots \\ \hat{x}_{12} \\ u_1 \\ \vdots \\ u_6 \end{bmatrix}_{k} \tag{5.3}$$

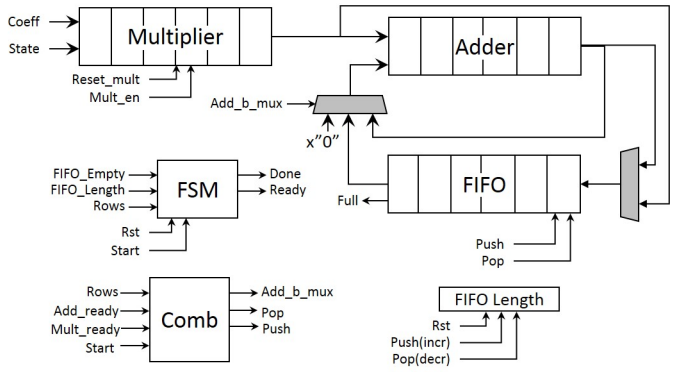


Figure 5.1 Architecture of a single state-space computation core

We have 12 rows and 18 columns and at the end of the calculation, will have twelve dot-products. The multiplier and adder each have 6-stage pipelines, thus to maximize throughput, we first compute all of the products that are of the first state $\hat{x}_1$. Then add the products that are of $\hat{x}_2$ are added to the previous terms and continue until each product-of-sums is computed. The timing diagram in figure 5.2 gives details of the internal working of the core. This will help in deriving an expression of how execution time is dependent upon the number of states of the controller(n), the number of sensor inputs (m) and the number of actuator outputs (p). Triggered by the *Start* signal , the first six clock ticks will fill the multiplier with the first product of the first six dot product terms ($a_{11} * \hat{x}_1 \quad a_{21} * \hat{x}_1 \quad a_{31} * \hat{x}_1 \ldots$) or ($p_{11}\ p_{21}\ p_{31}$). The first product from the multiplier will then be stored in the FIFO. For the next twelve ticks, the first elements of the twelve dot products are stored. At the same time, the first product of the next set of products $a_{12} * \hat{x}_2$ will be available at the multiplier's output. The FIFO's input source switches from the multiplier's output to the adder's first operand input and adding of

terms begins. During the next six ticks, the FIFO will start popping its elements to the adder's second operand input. The dot-products' intermediate results, from the adder, will now be pushed into the FIFO and this loop will continue until all the coefficients and states of $\hat{X}(k)$ are inputted into the multiplier. After six more ticks, the multiplier will be empty, all elements to the adder have been inputted and the FIFO will stop popping elements. Finally, the adder's pipeline will be cleared in the last 6 ticks and the twelve dot-products of the new $\hat{X}$ will be ready in the FIFO. The new $\hat{X}(k+1)$ is then stored in the state *storing memory block*, replacing the previous values of $\hat{X}(k)$. This process is repeated for calculating $Y = C \times \hat{X} + D \times U$. Equation 5.4 gives the total number of clock ticks needed to complete a cycle of computing a single time step of a state-space represented system. Due to the floating-point units' pipelines and to simplify the finite state machine, n and p must both be equal to or greater than six.

Figure 5.2 Timing diagram of a single core calculating the next state of $\hat{X}$

$$clock\ ticks = 12 + (n + p) \times (m + n + 1) \tag{5.4}$$

where

n = number of states

m = number of sensor inputs

p = number of actuator outputs

We now look into the proposed architectures for scaling the single core into a multi-core system which will compute segments of the state-space equation, in parallel. The top-level controller is the same for both implementations. Referring to figure 5.3(a), we see there are

configuration registers to store the dimensions of state-space matrices. n is the number of states, m is the number of sensor inputs that are retrieved from the *Sensor FIFO*, whose input come from a sensor interface module a computer architecture engineer must design. p is the number of outputs that will be pushed onto the *Actuator FIFO* for an actuator interface module to read. These modules will differ according to the plant we control. The T_P *register* stores the time period the controller must wait before starting a computation cycle. The host processor can send commands to the co-processor by writing to the *CMD register*. There are three commands, *soft reset* to clear all of the mentioned configuration registers, *start* enables to the co-processor to start running the controller, *halt* disables the co-processor without resetting the registers. Designers may use this command to update the controller's coefficients when implementing an adaptive controller, without worrying about updating the coefficients during an execution cycle. A single *State memory block* stores the state vector $\hat{X}$ as the states ($\hat{x}_1$, $\hat{x}_2$ ... $\hat{x}_n$) are common to all cores.The difference between the two architectures is the way the coefficient matrices' elements (a_{ij} b_{ij} c_{ij} d_{ij}) are supplied to the cores. Let us use the previous state-space system as above and have an architecture with two cores. We split equation 5.3 such that core0 will be computing equation 5.5and core1 will compute equation 5.6.

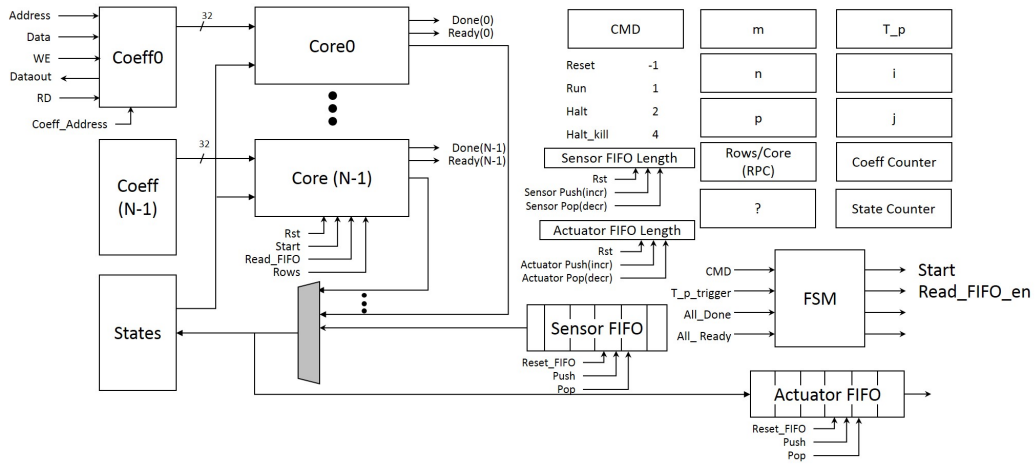
$$\underset{k+1}{\begin{bmatrix} \hat{x}_1 \\ \vdots \\ \hat{x}_6 \end{bmatrix}} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,12} & b_{1,1} & \cdots & b_{1,6} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{6,1} & \cdots & a_{6,12} & b_{6,1} & \cdots & b_{6,6} \end{bmatrix} \times \underset{k}{\begin{bmatrix} \hat{x}_1 \\ \vdots \\ \hat{x}_{12} \\ u_1 \\ \vdots \\ u_6 \end{bmatrix}} \tag{5.5}$$

$$\underbrace{\begin{bmatrix} \hat{x}_7 \\ \vdots \\ \hat{x}_{12} \end{bmatrix}}_{k+1} = \begin{bmatrix} a_{7,1} & \cdots & a_{7,12} & b_{7,1} & \cdots & b_{7,6} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{12,1} & \cdots & a_{12,12} & b_{12,1} & \cdots & b_{12,6} \end{bmatrix} \times \underbrace{\begin{bmatrix} \hat{x}_1 \\ \vdots \\ \hat{x}_{12} \\ u_1 \\ \vdots \\ u_6 \end{bmatrix}}_{k} \tag{5.6}$$

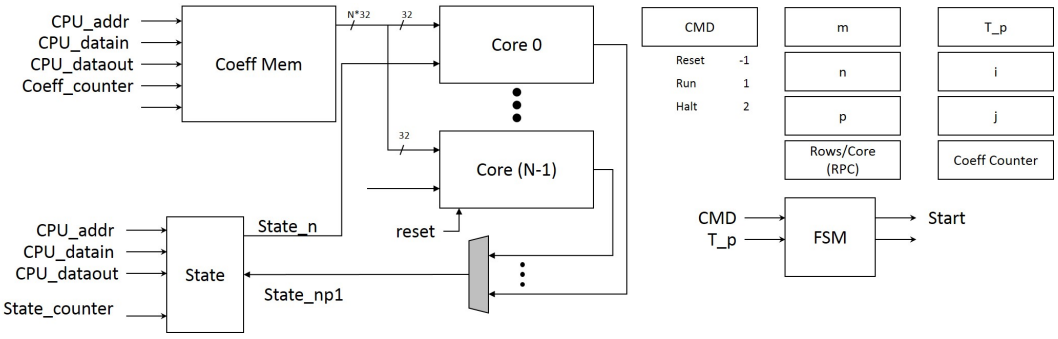
The system waits for the *time period counter* to flag that T_P time has passed. The cores are then initiated with a common *Start* signal and continuously supplied their respective coefficients every clock cycle until the news state elements of $\hat{X}(k+1)$ are calculated and waiting to be sequentially transferred from each cores' FIFO to the *State memory block*. This process is repeated for calculating the output vector Y for the actuator values. While being stored in the State memory block, $y_1 ... y_p$ are also sent to the actuator FIFO for the actuators to act upon.

The *Block RAM* architecture in figure 5.3(b) is our high performance system and can be used when, execution time must be minimal and silicon resources are available. A single memory block, made of one or more block RAMs, stores all of the controller coefficients. The host processor to memory data port is 32-bit, but the data port to the cores is $32N$ bits, where N is the number of cores. The *Distributed RAM* architecture in figure 5.3(a) follows an intuitive extension towards scaling, where each core is given a 32-bit wide memory element to source the coefficients. The memory modules used are implemented in the Look-up tables (LUTs), which are distributed throughout the FPGA chip. This gives the embedded systems engineer the advantage of scaling the memory to use only as many memory cells as needed by the control engineer. The disadvantage is that LUTs require more interconnect resources on the FPGA which causes large signal propagation delay and thus a lower clock speed. Block RAMs on the other hand, are ASIC level memory modules, require fewer routing resources and are thus faster. In both cases, the execution time, in terms of clock cycles is

$$clock\ ticks = 12 + (r+p) \times (m+r+1) \tag{5.7}$$



(a) Distributed RAM implementation



(b) Block RAM Implementation

Figure 5.3 Parallel architecture using 'Look-up tables' (LUTs) as memory cells (a)'Distributed or LUT-based RAM' (b)'Block RAM' as memory to store the coefficients

where

m = number of sensor inputs

p = number of actuator outputs

r = number of states each core must compute

Here we assume that the number of states per core, r is at least six due to the floating-point units' six stage pipelines. For example, if we used a four core architecture for our example, we would either have to wait for the FPU results to pass through the pipeline or fill some segments of the memory with zeros.

We now look at the overall advantage of using these parallelized architectures where exe-

cution time must be kept minimal for large state controllers. Figure 5.4 shows how execution time increases as we increase the number of states n in the controller while keeping the number of input m, output p constant. We calculate the execution time by first computing the number of clock cycles needed to complete a single cycle of the controller, using equation 5.7 and then multiplying that with the arbitrarily selected system clock period, $20ns$. Suppose we have a controller that can accommodate $20\mu s$ of computation delay, marked by the horizontal, dotted line. All controller implementations below this mark are able to execute a single cycle within the constraint. Thus, if our controller has ten states or less, a single core architecture will suffice. If the control engineer decides that a 160 state system is needed then an 8, 16 or 32 core architecture can meet the delay constraint.

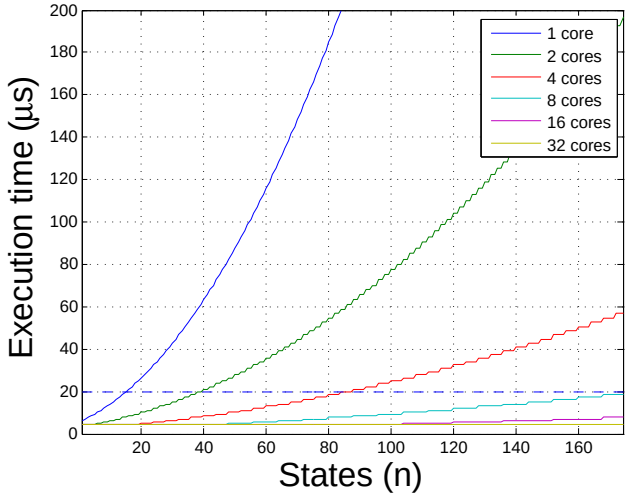


Figure 5.4 An implementation independent analysis of effect of increasing the number of states n of a controller on the execution time assuming we have a 50MHz clock, nine inputs and four outputs. Dotted line indicates maximum permissible execution delay.

5.4 Evaluation Methodology

Our figures of merit must address the concerns of both the control engineer and the embedded systems engineer execution time and resource usage. In an actual implementation, the memory would be connected to a processor's data bus, for example in the Xilinx Zynq the

AXI bus. The sensor and actuator interfaces would be connected to I/O pins or sensor and actuator modules that eventually connected to I/O pins which we've mimicked through the user constraint file. Our independent variable for resource analysis is the number of computation cores instantiated in an architecture, which we increment from 1 to 32 in powers of 2. We record the resource utilization of each increment for both proposed architectures and also determine the maximum clock frequency at which the configuration can run. The number of states that a configuration can handle and the execution time are the important parameters for a control engineer. We can determine the execution time or computational delay for a given architecture with a given number of cores and desired number of states by calculating the total number of clock cycles from equation 5.7 and then multiplying it by the lowest clock period our experiments report. When designing a control system, the number of sensors (m) and actuators p are known and the number of controller states can be easily varied. Thus, the number of sensors and actuators held constant at nine and four, respectively. In summary, we have two independent variables:-

1. cores instantiated in the architecture N
2. number of internal states in the controller n

the dependent variable of interest and are following:-

1. Look-up tables(LUTs)
2. Registers of Flip-flips(FFs)
3. Block RAM
4. Digital Signal Processor Blocks (DSPs)
5. Maximum clock frequency or shortest time period

The same I/O pins are used for the computation cores in order to map the core to the same resources and keep the place and route environment as consistent as possible without heavily optimizing the design for the device we performed our test upon. We analytically compute the number of clock ticks needed for each step in n, from nine states to 173 states (the largest

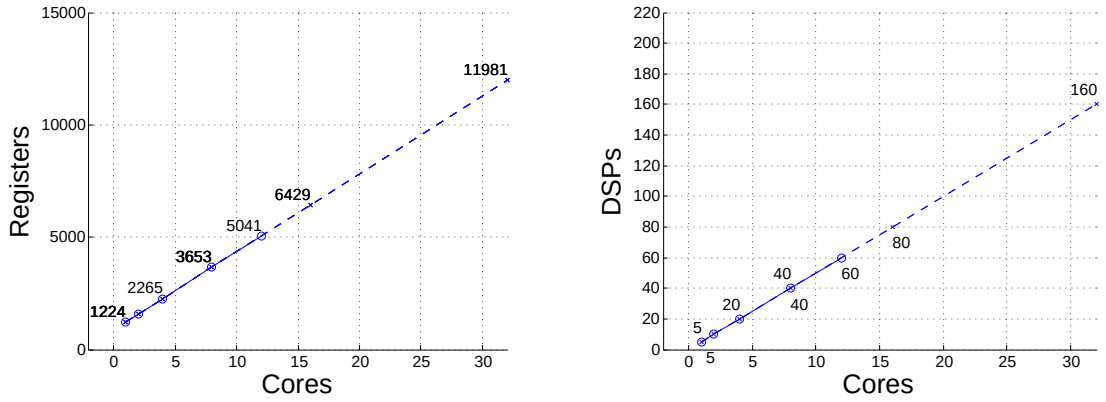
A matrix possible for given m and p) multiply it by the minimum clock period achievable by each configuration to get the execution delay. An example of how this data can be used by the embedded system engineer is given in the end.

5.5 Results

We understand that clock frequency and utilization can be improved by optimizing the location of output pins and locking sections of the design to certain regions of the FPGA. This will make our results specific to the device that we are using and our goal is to give an overall intuition of how to use the proposed architectures. Proficient designers can optimize their performance as they require. We will first look into the resources used by a core without memory modules. This will help in understanding how the architectures utilize resources when scaled into multi-core systems. The resource utilization of a single core is:-

- Look-up Tables 521
- Flip-flop 308
- DSP blocks 5
- FIFO blocks 1

A single core has a maximum clock frequency of $133MHz$ when using the pins similar to the proposed architectures'. Upon looking at the timing reports we observed that the floating-point units, the multiplier and adder, were in the critical paths. Reducing the pipeline length of either, the clock frequency reduced. Clock frequency increased upon increasing the pipeline. This is useful for a designer who knows the number of states in the controller. If the number of states is more than say eight, the pipelines can be increased to eight, thus improving clock frequency. If the number of states is smaller, the pipeline can be reduced to simplify the hardware and minimize memory wastage, but at the cost of reduced clock speed. This is a minor trade-off as such a small controller will require only one core and no parallelism.



(a) (b)

Figure 5.5 Both architectures have the same computation cores and top-level state machine and thus use the same number of (a) registers and (b) DSP blocks

5.5.1 Resource usage

We first look at the results which exactly match expectation. Figure 5.5 (a) gives the number number of registers used by the architectures. We observe that between the architectures, the DSPs blocks scale exactly the same way as the cores do not differ between architectures and are the only sections that use them This can also be seen in figure 5.5 (b). [4]

Tables 5.1 and 5.2 give the summary of resource utilization of both architectures as we increase the number of cores. As expected, the resources scale linearly with the number of cores, but not exact multiples of the core's utilization. This is due to the top-level logic. The distributed RAM architecture fails to place and route after 12 cores resulting in fewer data points in table 5.1 as routing resources as quickly consumed by the distributed RAM modules. The FIFOs also have a similar pattern as they are used in the cores and for the sensor and actuator interfaces. The number of LUTs differ due to the unavoidable differences in implementations at the top-level and the memory modules.

A primary difference between the architectures is how the memory to store the coefficients is implemented on the FPGA and this is highlighted in figure 5.6. Part (a) shows that the total

[4]We have kept the upper limit of all of our resource plots that of the total available resources. This gives an intuition as to how much of the reconfigurable logic has been used by the design

Table 5.1 Resource utilization of Distributed RAM based Architecture

Cores	1	2	4	8	12
Registers	1224	1571	2265	3653	5041
DSPs	5	10	20	40	60
FIFOs	3	4	6	10	14
Look-up Tables	3811	5494	8869	15773	22507
logic	1684	2303	3573	6198	8746
Memory	2104	3151	5245	9432	13621
routing	23	40	51	143	140

Table 5.2 Resource utilization of Block RAM based Architecture

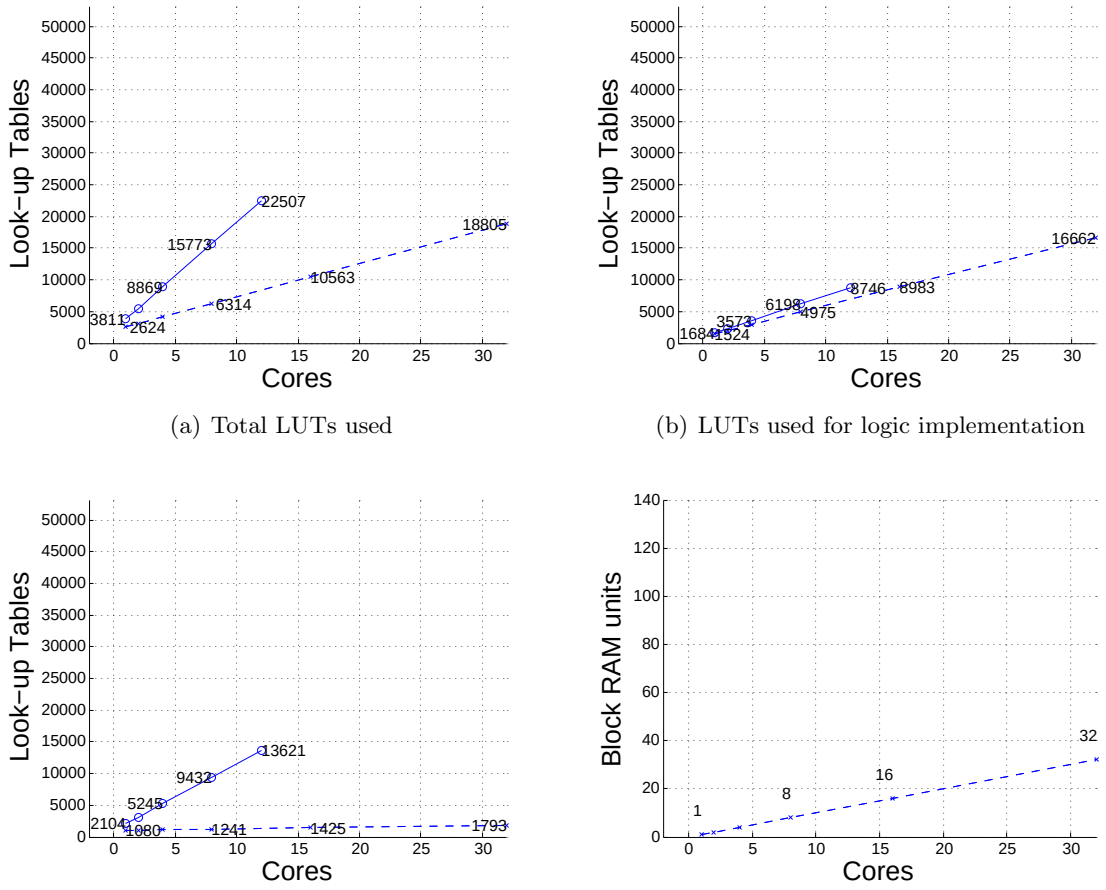
Cores	1	2	4	8	16	32
Registers	1224	1571	2265	3653	6429	11981
DSPs	5	10	20	40	80	160
FIFOs	3	4	6	10	18	34
LUTs	2624	3141	4187	6314	10563	18805
logic	1524	2001	2977	4975	8983	16662
Memory	1080	1103	1149	1241	1425	1793
routing	20	37	61	98	155	350
RAMB36	1	2	4	8	16	32

number of LUTs used by the architectures deviates quickly as the number of cores increases. Routing resources used to connect LUTs is proportional to to the number of LUTs used in the architecture thus explaining why this architecture fails to routing on the device after twelve cores. The LUT usage is high in the distributed memory architecture as the memory cells are implemented in LUTs, as seen in (c). The Block RAM implementation on the other hand uses uses ASIC-type BRAMs for coefficient storage and is able to route up to 32 cores as seen in (d).

5.5.2 Throughput analysis

When analyzing the single core, we showed that the maximum clock speed was proportional to the number of pipeline stages in the floating-point units(FPUs). When scaling the architecture while using six-stage pipelined FPUs, the clock speed reduces as the number of cores increases due to consumption of logic and routing resources. Figure 5.7 shows how the maximum operable clock speed reduces as the number of cores in the architecture is increased. This is very useful as we can now determine which architecture to use when given the number of sensors, states and actuators. It is also interesting to note for both architectures, as we

increase the number of cores, the frequency drops at a rate slower than the reciprocal of cores. This means that as we increase the number of cores, we are increasing throughput.



(a) Total LUTs used

(b) LUTs used for logic implementation

(c) LUTs used for memory

(d) Block RAM units used in BRAM architecture

Figure 5.6 All dashed lines represent the data for BRAM architecture and all solid line plots represent distributed RAM architecture. Number of Look-up Tables (LUTs) used by each architecture as number of cores was increased, (a) in total, (b) as combinatorial logic elements, (c) as memory elements. The drastic increase LUTs are used as memory whereas in the BRAM implementation (d), the block RAMs used to store the coefficients increases linearly with the number of cores.

Let us assume we have a system with nine sensors, six actuators and one hundred states. We need the controller to execute every $1ms$ and complete it's computation in $20\mu s$. Recalling the example in section 5.3, we can now use the clock speed information in equation 5.7 and plot the execution time versus number of states for the architectures, figure 5.8. In our example, if BRAMs are used by another module on the FPGA, we can use the LUT-based architecture

else, either architecture, using four cores, will work. If we increase the number of states to one-hundred and sixty and the allowed execution time to $10\mu s$, then we can use either a twelve core LUT-based architecture to save the memory or an eight core BRAM-based architecture in case the number of states may increase. Further reducing the permissible delay to $2.5\mu s$ shows that only the BRAM's 32 core setup will meet the design criteria.

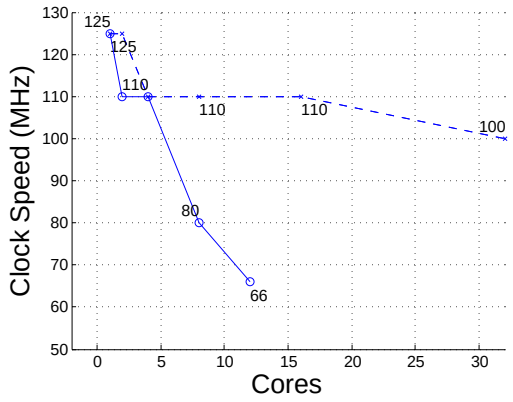
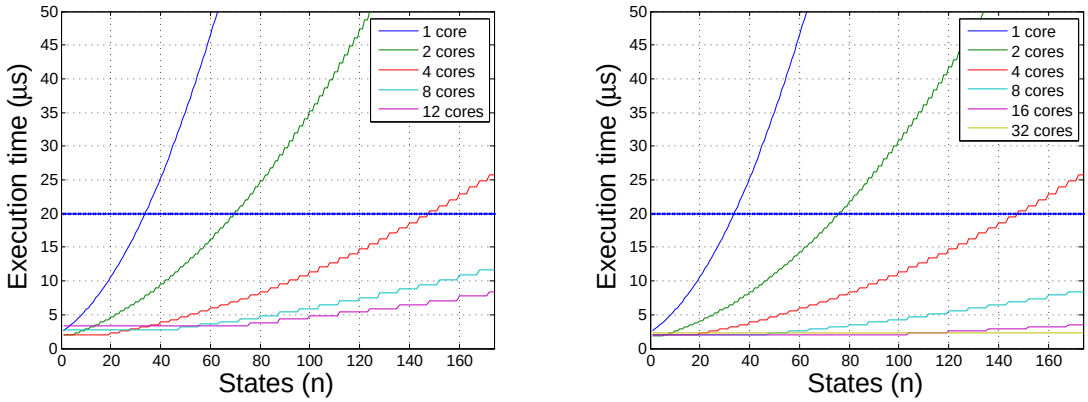


Figure 5.7 Maximum possible clock frequency comparison



(a) Distributed RAM

(b) Block RAM

Figure 5.8 Execution time of (a)LUT-based RAM and (b) BRAM architectures as number of states n is increased. The $20\mu s$ mark indicates the upper bound on execution time.

5.6 Conclusion

Two architectures of scalable programmable state-space based co-processors have been presented. They can be used for applications where silicon resources are limited yet deterministic execution of digital control loops is needed. The design space and throughput have been analysed and it was seen that LUT-based memory architecture was efficient in using logic resources and minimized wastage at the cost of reduced clock speed when increasing parallelism.

CHAPTER 6. A FAULT-AWARE TOOLCHAIN APPROACH FOR FPGA FAULT TOLERANCE

A paper published in ACM Transactions on Design Automation of Electronic Systems (TODAES), February 2015

Adwait Gupte[1], Sudhanshu Vyas [2] and Phillip Jones[3]

Abstract

As the size and density of silicon chips continue to increase, maintaining acceptable manufacturing yields has become increasingly difficult. Recent works suggest that lithography techniques are reaching their limits with respect to enabling high yield fabrication of small-scale devices, thus there is an increasing need for techniques that can tolerate fabrication-time defects. One candidate technology to help combat these defects is reconfigurable hardware. The flexible nature of reconfigurable devices, such as Field Programmable Gate Arrays (FPGAs), make it possible for them to route around defective areas of a chip after the device has been packaged and deployed into the field.

This work presents a technique that aims to increase the effective yield of FPGA manufacturing by reclaiming a portion of chips that would be ordinarily classified as unusable. In brief, we propose a modification to existing commercial toolchain flows to make them fault-aware. A phase is added to identify faults within the chip. The location of these faults are then used by the toolchain to avoid faults during the placement and routing phase.

Specifically, we have applied our approach to the Xilinx commercial toolchain flow, and evaluated its tolerance to both logic and routing resource faults. Our findings show that

[1]Primary researcher and author, Department of Electrical and Computer Engineering, Iowa State University
[2]Researcher and author
[3]PI and author of correspondence

at a cost of 5 to 10% in device frequency performance that the modified toolchain flow can tolerate up to 30% of logic resources being faulty, and depending on the nature of the target application, can tolerate 1 to 30% of the device's routing resources being faulty. These results provide strong evidence that commercial toolchains, which were not designed for the purpose of tolerating faults, can still be greatly in the presence of faults to place and route circuits in an efficient manner.

6.1 Introduction

The computing industry has been able to leverage the scaling of transistors to ever smaller dimensions in accordance to Moore's law for decades. The amazing ability to keep pace with this law motivated the reprinting of Gordon Moore's original 1965 paper in 1998 [88], and 2006 [89]. However, there is strong evidence that suggests this era of increasing computing performance by packing more transistors on to a device is coming to an end. Typically as the density/size of chips increase, their yields tend to decrease [45]. The 2009 International Technology Road-map for Semiconductors (ITRS) reports that as lithography pushes toward single atom scales, methods currently do not exist to constrain process variations [65]. This will result in large numbers of undesirable structural defects, such as open and short circuits, which in turn will lead to low chip yields. In addition to lowering yields, evidence has been shown that chip's associated with low yield batches have an increased likelihood of having reduced life times due to phenomenon such as oxide puncture [73]. Overall, lower chip yields and reduced device reliabilities (e.g. reduced life times) negatively impacts the computing industry as a whole. From a research perspective, these factors bottleneck the degree to which transistors can be scaled, thus constraining the raw computing power that can be leveraged to solve computationally intensive problems. From a business perspective, lower yields mean more chips discarded, resulting in lower profit margins.

Solutions are being pursued on several fronts to help maintain high yields as fabrication scales continue to decrease. Fabrication engineers are exploring new procedures for fabrication [72], technology developers are experimenting with new materials to replace or enhance the traditional metal-oxide that is primarily used today [69], and design architects are de-

veloping techniques to implement fabrication time redundancies to combat increasing defect rates [121].

Combating defects with reconfigurability Reconfigurable hardware technology, such as Field Programmable Gate Arrays (FPGAs), show promise in complementing some of these solutions. FPGAs are devices that can be used to implement digital hardware quickly and inexpensively. An FPGA can be reprogrammed virtually limitlessly, and in a matter of seconds. This capability makes them suitable for various fields where application functionality is expected to change with time, or in fields where volumes are not large enough to justify the large initial costs associated with producing Application Specific Integrated Circuits (ASICs). FPGAs are also a useful prototyping tool that can be used for high fidelity modeling of an ASIC's functional behavior [9]. Like any other silicon based technology, FPGA suffer from lowing yields and reliability issues as chip sizes and transistor densities increase. However, unlike ASICs, the reconfigurable nature of FPGAs intrinsically support redundancy that can be leveraged to tolerate defects that occur at fabrication time or in the field. Their symmetric architecture and reconfigurability can allow designs to be placed and routed around defective areas of the chip after the device has been fabricated, packaged, and deployed into the market.

Techniques to leverage the reconfigurability of FPGAs to allow them to be used despite the presence of defects could help mitigate increasing defect rates in silicon devices by: 1) encouraging industry to migrate more of their ASIC applications to FPGAs, and 2) integrating aspects of FPGA architectures into their designs. More immediately, such techniques would impact FPGA manufacturers by allowing them to increase their effective chip yields, and extending device lifespans.

Contributions In our work, we evaluate a method that steps toward reclaiming some fraction of FPGAs that would currently be deemed as defective. Our approach introduces lightweight modifications to the end-user FPGA toolchain flow to tolerate both manufacturing defects as well as defects due to aging. The three core contributions of our work are 1) A technique for leveraging existing commercial FPGA toolchains to make them fault-aware, 2)

quantifying to what degree existing tools can tolerate both logic [46] and routing faults, and 3) recognizing and quantifying the trade-off between the tool's tolerance to faults and the frequency performance of circuits it can implement.

Organization The remainder of this article is organized as follows: Section 6.2 gives related work from the areas of defect quantification, fault location and fault tolerance. Section 6.3 introduces our proposed approach. Section 6.4 then describes the evaluation methodology used to quantify the effectiveness of our approach. Section 6.5 discusses the results of our evaluation experiments, and Section 6.6 presents our conclusions and avenues for future work.

6.2 Related Work

Fault location and fault tolerance in FPGAs are closely related to the work presented in this article. This section first discusses previously proposed methods for fault location in FPGAs. Some of the methods introduced are potential candidates for use during the "Test FPGA" phase we propose in our work (see Figure 6.1). The second part of this section reviews fault tolerance techniques for FPGAs, and places our work into context with respect to this existing body of research.

6.2.1 Fault Location Methods

The configurability and inherent parallelism of FPGAs allows for innovative methods of detecting and locating faults, as compared to standard ASICs. For example, an FPGA can be configure with circuits for detecting/locating faults, and then if the FPGA is deemed usable can be reconfigured to implement logic for a target application.

Logic Faults In [127], Wu proposes a method in which faults can be detected on an FPGA by programming test circuits on it. They make use of partial-dynamic-reconfiguration features to reconfigure different portions of the FPGA to act as test circuits. This reduces the amount of time needed to test the entire chip, as opposed to reconfiguring the entire FPGA for each configuration of the test circuits they wish to deploy. Lowing testing time is especially

important in large scale production environments, where each chip needs to pass through a testing phase before being shipped.

In [60], Inoue proposes another method for testing FPGAs that allows faults to be located at the granularity of a single configurable logic block (CLB). The output of each programmed CLB is used as the input to another. This daisy chaining of CLBs allows the relatively small number of FPGA I/O pins to be used to test a large number of CLBs. By sequentially running this procedure on cascaded rows and then on cascaded columns, they can identify individual CLBs that are faulty (e.g., CLBs at the intersection of a given faulty row and column).

Wang [74] presents a method of fault location and suggests that their method could allow for faulty chips to be utilized in the field. A Built-in- self-test (BIST) technique is proposed that uses regions of the FPGA to test other regions of the FPGA. As illustrated in the top portion of Figure 6.1, the FPGA is reconfigured multiple times so that various parts of the chip take turns acting as the testing circuit verses acting as the circuit under test. The classical Preparata, Metze, and Chien (PMC) model [103] is used to account for potential errors in areas configured as test circuits.

In [29], Dutton presents a practical implementation of BIST techniques for locating faults on the Xilinx Virtex-5 FPGA. A total of 17 different configurations were developed that together achieve 100% coverage of the logic faults that can occur. Similarly [30] identifies a practical set of test configurations for identifying I/O tile faults on a Virtex-5 FPGA.

Routing Faults In [14], Campregher discusses the trend of commercial FPGA architectures dedicating increasing amounts of silicon area to routing resources, and indicates a consequent need to develop BIST techniques that concentrate on identifying faulty interconnect resources in addition to faults in logic resources. [115] and [78] present BIST techniques that address this need.

Performance Degradation Worked by Stott in [114] and [113] used experimental data to help quantify the rate at which FPGAs can degrade in performance. This degradation can cause new faults to manifest overtime. In [70], Keane proposes the use of ring oscillators to

act as a means of monitoring silicon device performance degradation in real-time. This type of monitor could be implemented using FPGA resources (e.g. CLBs) and used to trigger a fault detection technique when a given application-specific-performance-degradation threshold is surpassed.

6.2.2 Fault Tolerance Methods

Stott in [111, 112] and Cheatham in [24] present excellent surveys of different FPGA-based fault tolerance techniques. Cheatham classifies the various techniques into two broad types: Device Level and Configuration Level techniques.

6.2.2.1 Device Level

These techniques are incorporated during the manufacturing stage of a device. For example, redundant resources such as routing paths and programmable logic blocks are added. One could further classify these into what this paper will refer to as direct and indirect fault tolerant methods, where in a direct method redundant resources are only used when a fault occurs, and in an indirect method fault tolerance is indirectly achieved through the innate structure of an architecture.

Direct methods In [54], Hatori proposes a technique that tolerates faults discovered at the manufacturing stage by adding redundant rows and selection logic to routing resources. Faulty rows are made invisible to the user by routing around them. Kelly [71] proposes a technique that can mask faults even when they occur in the field. An on board router is added, which on the basis of a stored "fault vector", changes the routing of the design to avoid faulty areas. The advantage of this method is that it remains mostly transparent to the end user software. Durand [28] proposes a method that can tolerate faults at run time. Extra resources are used to continuously test logic resources. When a fault is discovered, spare resources are used to mask the fault.

Indirect methods In [106], routing architectures where evaluated for their intrinsic routablity in the presents of defects. In [59], models of commercial and academic FPGA

switch matrices were developed. Simulations were run with these models to evaluate the effect of routing faults on a circuit's routablity. In [80], various amounts of redundancy were added to an FPGA architecture to quantify the impact of redundancy on a circuit's routablity in the presents of faults. FPGA logic blocks and switch matrices were modeled and Versatile Place and Route (VPR) was run to quantify this relationship. The results of their experiments showed that adding one extra interconnect per switch lane was the most effective amount of redundancy to add. They found this to be true across several generations of fabrication technology.

A common drawback of device-level techniques is the necessity of additional chip-level or board-level resources to support them.

6.2.2.2 Configuration Level

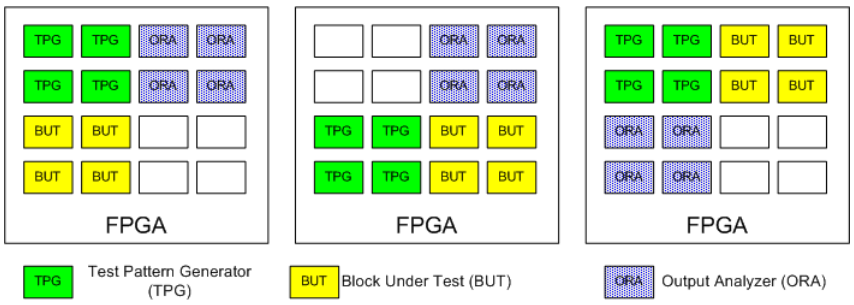
These techniques first identify faulty areas of the chip. Then, by performing simple shifts of configuration memory, use alternative resources to implement the design. Methods such as those described by Narasimhan [92] and Hanchek [52] rely on shifting the configuration memory when faults occur to obtain a fault free implementation. Methods such as the ones described by Emmert [33] apply heuristics to decide the best direction for these shifts. Howard [57] presents a configuration-level approach that works at a higher level of abstraction than shifting bits. Subcircuits of a design are moved from one "block" location to another to avoid faulty areas.

These configuration-level methods have the common drawback of degrading the place and route (PAR) quality of the original circuit, provided by PAR tools. In other words, since these techniques typically only use local information to adjust a design's configuration, there may be more optimal PAR solutions that could be achieved if global information was utilized.

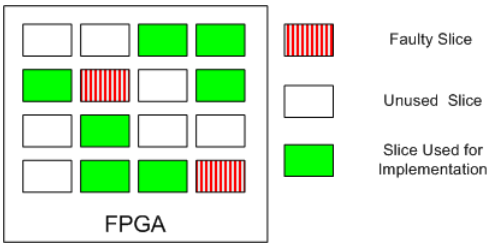
In summary, current FPGA fault tolerance methods try to either mask faults by adding redundancy during the manufacturing stage (device-level) at the cost of resource overhead, or try to rectify them in the field by updating the FPGAs configuration (configuration-level) at the coast of circuit performance. In domains where low latency recover is needed, many of these approaches are a good fit. However, for use-cases where recovery time is not a factor, we propose an approach with zero device overhead that provides a routed circuit of high quality. Current FPGA toolchains do an excellent job of optimizing the way in which a design is implemented.

The solutions they arrive at are typically fairly optimized. This work proposes a method that integrates information about faults into standard FPGA toolchains, thus leveraging their optimization capabilities for the purpose fault tolerance.

It should be noted that Xilinx Corporation provides a service called EasyPath [130] that is closely related to our approach. A fundamental difference between EasyPath and our approach is that our approach allows the toolchain to avoid faults, while EasyPath checks if a given design will work on a given FPGA without knowledge of fault locations. In short with EasyPath, if a given design works on a given FPGA, then it ships. If not, then the FPGA is not used. There is no attempt to re-place and re-route the design to account for errors on the chip.



(a) Different parts of the FPGA test each other in order to mark faulty slices in the FPGA.



(b) Tools avoid faulty slices while implementing the circuit.

Figure 6.1 Overview of the proposed method.

6.3 Fault-aware Toolchain

This section provides an overview of our proposed method for integrating fault information into an FPGA implementation toolchain flow. The impact of the proposed approach on the life cycle of an FPGA is discussed, and possible usage models are presented. This is followed by an example nomenclature that manufacturers could use to market FPGAs with faults. The

section concludes with a discussion of some concerns that need to be considered when using our approach in practice.

6.3.1 Overview

Figure 6.1 succinctly summarizes the essence of this article. One of the methods from section 6.2 can be used to identify faulty sections of the the FPGA. The lower part of the figure shows how the identified faults can be avoided by using a modified toolchain that is fault-aware. Proposing such a toolchain and evaluating its effectiveness is the key of this article.

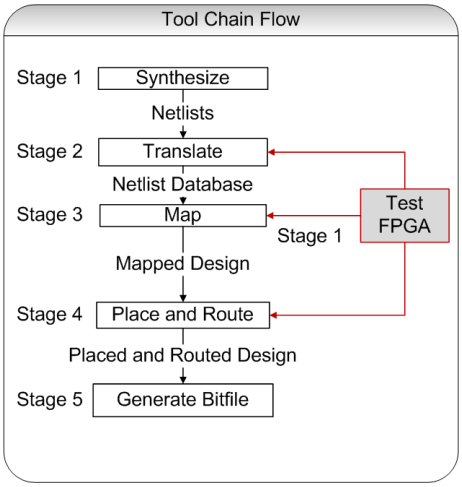


Figure 6.2 A fault-aware toolchain flow that tests the FPGA in parallel with the synthesis stage, and then feeds fault information to the remaining stages.

Figure 6.2 shows a high-level overview of the proposed fault-aware FPGA implementation toolchain flow. A design goes through several standard stages while being implemented on an FPGA. An additional "Test FPGA" stage is proposed that may use any of the fault location techniques discussed in section 6.2. The information gained from this test stage can then be fed back into the rest of the toolchain to implement the design in a way that avoids faults. The Synthesis phase involves translating the HDL design description into netlists. This translation tends to take at least a few minutes for any non-trivial design, so the testing of the FPGA can be done in parallel. Since the test stage is performed in parallel with the synthesis step, the additional time overhead to the implementation process will be small (e.g. a total testing time of 160 seconds is estimated for a XC4025 device [64]). As compared to the previous configuration-

level methods discussed in Section 6.2, our proposed method leverages the intelligence of mature place and route tools to provide an efficient implementation of designs in the presents of faults.

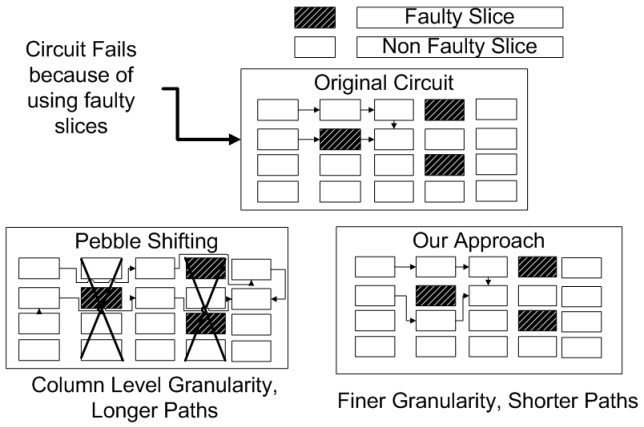
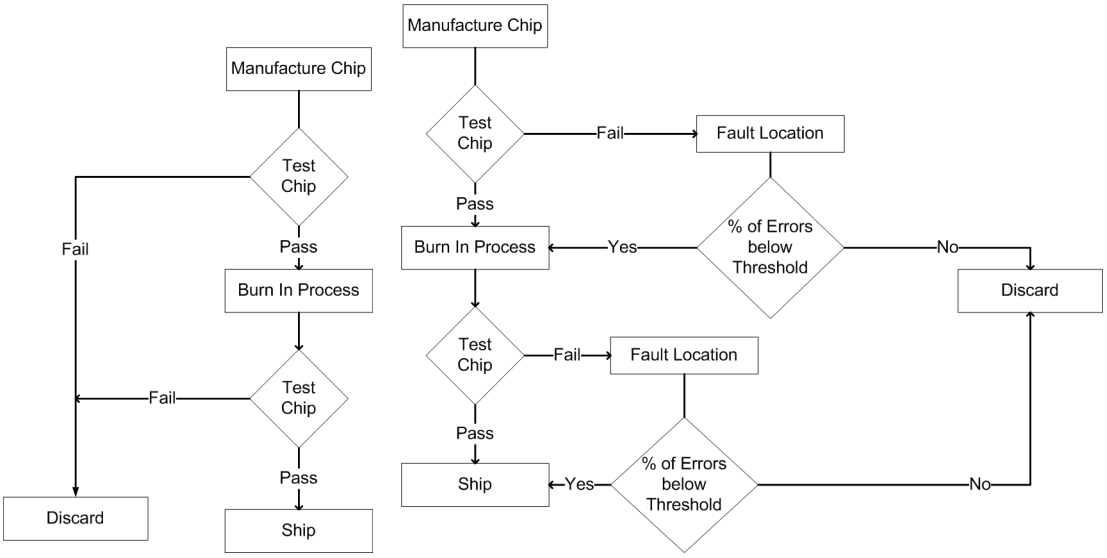


Figure 6.3 Example comparison to an existing fault tolerant technique with respect to a circuit's critical path length. We Illustrate the benefit of using existing place and route tools for fault tolerance as opposed to larger granularity approaches. In this case a conceptual comparison to the Pebble Shifting technique [92] that reconfigures at a column granularity.

Figure 6.3 gives an example of the placement obtained by the proposed method as compared to one obtained with the "Pebble Shifting" method presented by Narsimhan [92]. The "Pebble Shifting" method avoids an entire column even if only a single logic slice in that column is faulty. The resulting circuit has longer paths than the method we propose. To inform the toolchain of slice logic faults, our approach uses the PROHIBIT constraint available in Xilinx tools that allows certain logic sites to be forbidden for the purpose of placing components. For informing the toolchain of routing faults, a more involved approach is needed, described in Section 6.4.4. In short, low-level design mechanisms are used to effectively block the tools from using targeted switch matrices. Although our two techniques for passing fault information to the toolchain is effective, a more integrated method of integrating fault information into the toolchain is desirable. One simple approach could be to have a separate "error file" that would be consulted during the implementation process.



(a) Current manufacturing flow (b) Proposed manufacturing flow

Figure 6.4 Comparison between current manufacturing flow and our proposed flow

6.3.2 Applicability of the Proposed Method

Two main use cases can be considered for the proposed method: 1) using FPGAs with defects in mass production and 2) rectifying FPGAs after a fault occurs. These use cases are discussed below.

Consider a company buying faulty FPGAs in order to implement designs required for a product. As opposed to creating a single bitfile for all the FPGAs, the manufacturer would have to run the toolchain for each FPGA individually since each would have errors in different locations. Although this sounds unreasonable at first glance, it might turn out to be economical depending on the savings from buying faulty chips versus the increase in computing costs. In order to support such a use case, the manufacturer would change the manufacturing flow as shown in Figure 6.4. Curve 2 in Figure 6.5 shows the change in the probability of FPGAs being discarded by the manufacturer, if the customer can use such a method.

In the second use scenario, consider a company using fault-free FPGAs to implement a design. These programmed FPGAs are then used in end user products. When an end user product malfunctions because of an error in the FPGA, rather than discarding the FPGA, a technician could run the fault-aware toolchain on the (now) faulty FPGA. This would create

a new bitfile of the design that would avoid the faulty areas. Curve 3 in Figure 6.5 shows the conceptual change in the probability of FPGAs being discarded if this approach is used.

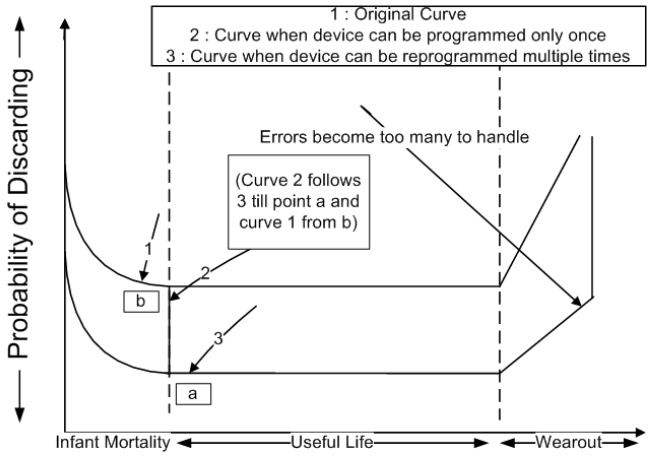


Figure 6.5 Changes to the bath tub curve using our method.

6.3.3 Error Grading

Devices are currently tested after they are manufactured to determine their maximum operating frequency. Due to variances in the manufacturing process, this frequency is not uniform for all chips and hence they are branded with a speed grade to indicate this difference. A similar concept could be used to mark chips with different "Error Grades". For example, the larger the error grade, the more faults in the chip. Also as the error grade increases, the maximum frequency of operation will reduce. This is due to the place and route tools having fewer options from which to choose during implementation (see Section 6.5.2). Thus, either a separate error grade could be constituted or a composite grading system taking both the percentage of errors and the speed into account could be created. Before shipping, the FPGAs could be run through a fault location phase. Then depending on the number of faults detected, the chip could be discarded or branded with an error grade/composite speed-error grade. A consumer could then select a chip based on its error grading, the trade-offs associated with the error grading, and the requirements of the design.

6.3.4 Some Concerns

Implementing fault tolerance in FPGAs requires that certain common obstacles be considered along with issues specific to the nature of the applied approach. In this section both kinds of issues are discussed with respect to our proposed approach.

Other Faulty Resources Many modern FPGAs have other resources such as Block RAMs, DSP Slices, Clock Management Tiles, hardcore microprocessors, etc. Our approach does not evaluate fault tolerance for these resources. In many cases extending this work to examine these other resources is possible by using the same approach we use for emulating logic-slice faults.

Location Constraints In some FPGA designs, certain resources are required to be locked to a specific location (LOC), or must be placed at a location relative to another component (RLOC). In the first case, if the location to which the component is LOCed is faulty, then there is nothing this approach or any other approach can do to allow the FPGA to be used for this design. In the latter case, it might be possible to move the RLOC origin to another point to satisfy the constraints.

Mass Production Viability When FPGAs are a part of a mass-produced product, using faulty FPGAs would require re-running the tools for each individual FPGA. Cost analysis must be performed to determine if the amount of time and resources spent re-running the tools for each FPGA is offset by the reduced bill of materials resulting from using cheaper, faulty FPGAs.

6.4 Evaluation Methodology

In this section we describe the methodology used to evaluate the effectiveness of our proposed fault-aware toolchain. First, the fault model assumed by this work is given. Section 6.4.2 then presents our test flow, and the metrics that were used to evaluate our approach. Sec-

tion 6.4.3 describes the benchmark circuits that were used, and in Section 6.4.4 we discuss how fault emulation was implemented.

6.4.1 Fault Model

Faults occurring in a device can be divided into two main types: those that occur as a result of the fabrication process, and those that occur after the chip is manufactured due to aging. These faults can occur either in the interconnects or the logic elements. Further, when a fault occurs in a logic element it can occur in a LUT, a flip flop, a multiplexer or the combinational carry chain logic. In this work we have abstracted away the details of the exact point of failure of the chip. For the case of logic faults, we consider a slice as a whole to be either working correctly or faulty. For the case of routing faults, we take a pessimistic view that a given switch matrix is as a whole functional or not.

Another common classification of faults is permanent or transient. This work assumes permanent faults. In addition, the positioning and clustering of faults depend on the mechanism causing those faults (e.g. dust particles during lithography, radiation exposure, thermal cycling, electron migration). In this work, we have assumed faults have a uniformly random distribution.

6.4.2 Methodology

Figure 6.6 illustrates our evaluation flow. For each experimental run the first step "Generate Uniformly Distributed Errors" creates a defect-map. This defect-map is generated using a seeded random number generator, allowing us to reproduce fault patterns across benchmark circuits. The fault types and locations are then passed to the Xilinx toolchain via its user-constraint-file (UCF). In general, the UCF file is a mechanism that allows user specific constraints to be defined. In our case, we use this file to keep the tools from using resource that our defect-map indicate are faulty.

From here, the toolchain runs as normal. At the end of each run, the toolchain generates its standard reports that indicate whether it was able to route the design and meet the specified timing requirements.

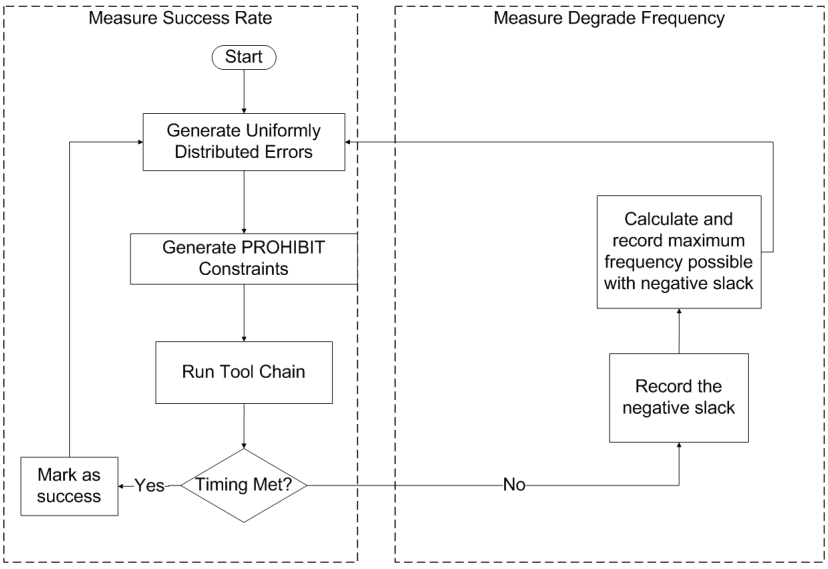


Figure 6.6 The testing flow used to evaluate our proposed approach.

This test procedure was repeated for various fault levels, FPGA utilization levels, and benchmarks (see Section 6.5). Each combination of the parameters tested were run approximately 200 times for different random fault patterns. A set of 200 runs took approximately 9 hours to complete, typically spread over 75 CPU cores. Next we describe the metrics used to evaluate our proposed fault-aware toolchain approach.

Error Tolerance This metric is used to evaluate the degree to which our approach can tolerate faults. In order to measure this metric, various percentages of the FPGA's resources were marked as faulty. A normally distributed random variable was generated to represent the number of faulty resources. These faults were then distributed across the chip as described in Section 6.4.4.

The tools were then run on the synthesized netlists to obtain a placed and routed (PAR) design. The resulting implementation was analyzed to check that it was still able to meet timing. If it did meet timing, then the run was marked as a success for our proposed approach. If timing was not met, then the run was marked as a failure. We discuss the results of these experiments, as well as how this metric relates to chip yields in Sections 6.5.1 and 6.5.2.

Performance Degradation We use this metric to investigate the relation between the percent of faults on the FPGA, and the performance degradation needed by the tools to successfully implement a design. For designs that fail to place and route during the "Error Tolerance" evaluation, the negative slack time in the PAR report was recorded. This slack was then used to calculate the maximum frequency at which that design could have been implemented by the tools (see right side of Figure 6.6).

These results are used to show that even in cases where it is not possible to implement the design with the specified timing constraints, a small performance decrease may be enough to make the design implementable on the faulty chip. Thus, in addition to having fewer resources, a faulty chip may also degrade in performance. Section 6.5.2 evaluates our approach with respect to this metric to determine the error threshold that our approach can tolerate for given performance degradation levels.

Comparison with smaller, fault-free chips In order to understand the trade offs between using a large FPGA with faults, and a smaller fault-free FPGA, we implemented benchmarks circuits targeting smaller fault-free chips and measured the maximum frequency at which they could be implemented. These frequencies were then compared to the ones obtained for the larger faulty FPGA. Our findings are discussed in Section 6.5.4.

6.4.3 Circuit/Device Description

A subset of the benchmark circuits proposed by F. Corno, in [27], were used to evaluate our approach. These benchmarks were originally created to evaluate test pattern generation methods for identifying stuck at faults. They easily scale to occupy various percentages of an FPGA by replicating the design. Since these designs are based on real world circuits (e.g. processor cores), they form good test cases for evaluating our approach. The selected benchmarks were replicated multiple times to obtain utilizations of approximately 25%, 50% and 75%. The benchmarks chosen were:

1. b17: Three subsets of an Intel 80386 processor.

2. b18: Three Viper processors and Six 80386 processors.

3. b20: A Viper processor and a modified version of the Viper processor.

4. b21: Two Viper processors.

5. b22: A Viper processor and two modified versions of the Viper processor.

For evaluating slice-logic faults, all five of the above benchmarks were used. For evaluating the impact of routing faults, only benchmarks b17, b18, and b20 were used. Additionally, it should be noted that for evaluating routing faults that benchmark b20 provided a test case made up of many small and loosely coupled cores, and b17 provided a test case for larger cores (i.e. the 80386 cores are much larger than the Viper cores). The implication of these different characteristics is discussed in our results section, with respect to the tools ability to tolerate routing faults.

The benchmarks did not include a constraint file and hence constraints had to be provided for the evaluation. A feature of these benchmarks is that they use a single clock. This clock was constrained to within 3% of the highest frequency that the tools could successfully place and route a benchmark on a non faulty FPGA. When the benchmarks were replicated, their outputs were ORed together to deal with mapping them to the limited number of I/O ports available on the actual FPGA device.

The test circuits targeted a Xilinx Virtex-5 LXT 110[131]. This device has a total of 17280 slices and was chosen because there are 4 smaller devices available in the same family. This allowed us to compare a large faulty FPGA's performance against smaller fault-free FPGAs (see Section 6.5.4).

6.4.4 Fault Implementation

Here we describe how logic-slice and routing faults were emulated.

Logic-slice faults Emulation of logic-slice faults was a fairly straightforward process. The Xilinx toolchain provides an attribute called PROHIBIT that can be associated with a specified logic-slice. This attribute indicates to the toolchain that a particular logic-slice is not allowed to be used in the design being implemented. Based off of the defect-map generated

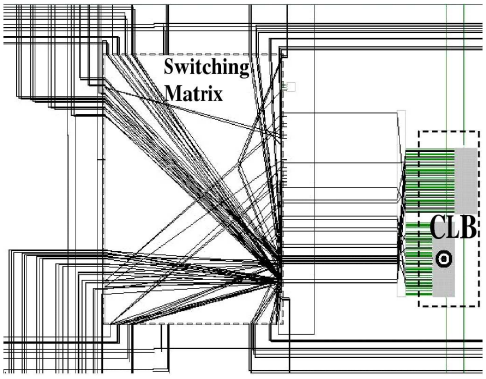


Figure 6.7 Routing fault emulation: Shows the FPGA Editor view of a Virtex-5's switching matrix and associated CLB. The darkened traces indicate the switch matrix output ports, all of which were manually blocked in order to emulated a faulty switch matrix.

by the "Generate Uniformly Distributed Error" step of Figure 6.6, PROHIBIT attributes are appropriately added to the tool's user constraint file (UCF).

Routing faults Emulating routing faults was a much more challenging and involved process. We used a combination of Xilinx's FPGA Editor, XDL (Xilinx Design Language), and DIrect RouTing (DIRT) constraints, to generate a circuit which utilized all of the output ports of a single switching-matrix, Figure 6.7. With all the output port used, the tool was kept from routing through the targeted switch-matrix, thus emulating a faulty switch matrix. We then replicated and instantiated this fault as directed by our generated defect-map.

In addition to defining a routing fault that fully disables a switching matrix, we evaluated the impact of a routing fault that allowed the VLONG and HLONG switch matrix interconnects to be used for routing. These interconnects stretch eighteen CLBs vertically and horizontally. We predict that the toolchain will perform significantly better with the VLONG and HLONG interconnects enable, since it will not be forced to 'hop' over multiple switching-matrices to connect a signal's source and destination when a fault is encountering.

6.5 Results & Analysis

This section presents the findings from the experiments described in Section 6.4. First, we examine the error tolerance of our proposed method. Next, we quantify the percentage of errors that can be tolerated if the frequency of a design is allowed to degrade by a given amount. We then discuss differences between the tools tolerance to logic-slice faults versus routing faults. Section 6.5.4 presents preliminary findings with respect to the idea of using larger faulty FPGAs as equivalent small non-faulty FPGAs. This section concludes with a discussion of the sensitivity of our approach to the tightness of timing constraints.

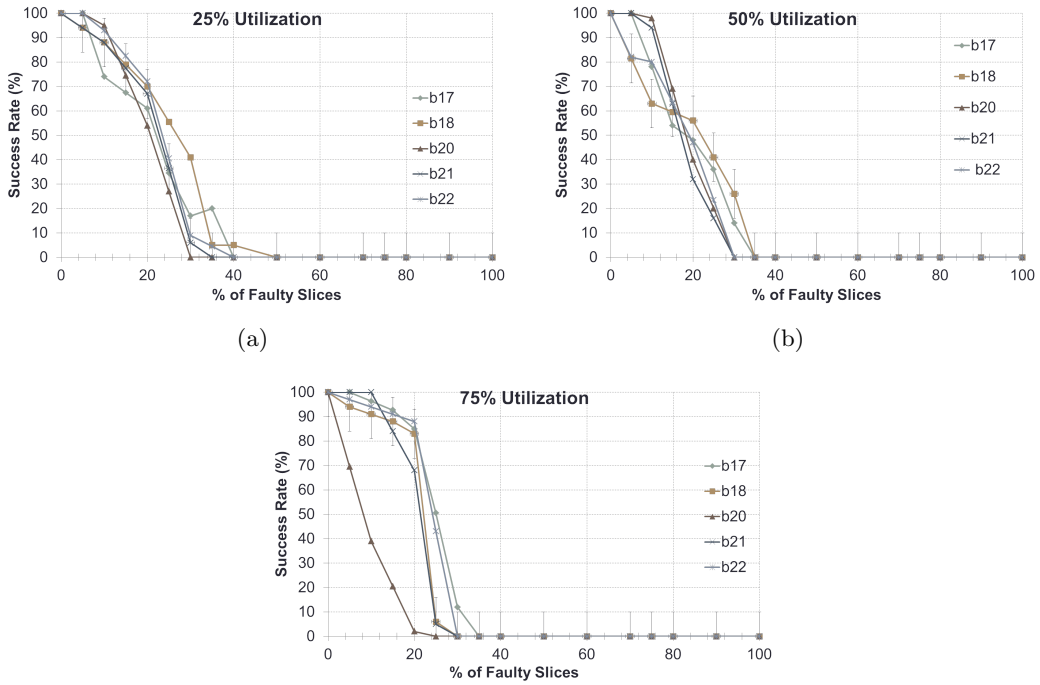


Figure 6.8 Success rate when varying error rates for a device utilization of a) 25%, b) 50% and c) 75%. All data points are within a 10% confidence interval at a confidence level of 90%. For the sake of clarity, error bars are only shown for one benchmark.

6.5.1 Logic-slice Error Tolerance

Figure 6.8(a) shows the success rate of the fault-aware toolchain at 25% utilization of the LX110T FPGA. As the figure shows, even with almost 75% of the chip empty, the designs have a significant chance of not meeting timing if approximately 10% of the logic slices are faulty.

The success rate is 0% when 30% to 60% of the logic slices are faulty. This can be explained by the very tightly constrained clock in the original design. The tools were able to barely meet the timing constraints without any faults, so as faults were introduced it became more difficult for the tools to implement the design while still meeting all constraints.

Figure 6.8(b) shows the success rate when 50% of the FPGA is being utilized. It can be seen that the designs start failing timing much sooner for the higher 50% utilization case as compared to the 25% utilization case. The success rates reach 0% when 20-30% of the logic slices become faulty.

Figure 6.8(c) shows the success rate for 75% utilization of the FPGA. For the b17 benchmark, it was not possible to achieve an exact 75% utilization so it's utilization was kept at 70%. Thus, the b17 benchmark success rate does not fall to 0% at 25% errors, instead it does so at 30% errors.

The designs experimented with were constrained to within 0.2 ns of the smallest possible period, so these numbers represent close to the worst case success rates of the proposed approach. In many practical cases, a design will not be required to be run at the highest achievable frequency. Thus, if a design is more loosely constrained, then it would be expected that the fault-aware toolchain would be able to tolerate more faults.

From the figures it can be seen that the success rates are high until about 10% of the logic slices become faulty. Thus even for designs that cannot compromise on their frequency, it may still be cost effective to buy larger chips with errors present and discard the small percentage of them that cannot meet timing. On the other hand, if frequency degradation is acceptable, then it may be possible to tolerate a given fault level at the cost of performance. The next section examines the amount of frequency reduction necessary to implement the experimental runs that failed at various error levels.

Table 6.1 Percent of faulty logic slices that can be tolerated for a maximum frequency performance cost of 10%.

Utilizations	25%	50%	75%
% of Faulty Logic slices	30%	30%	20%

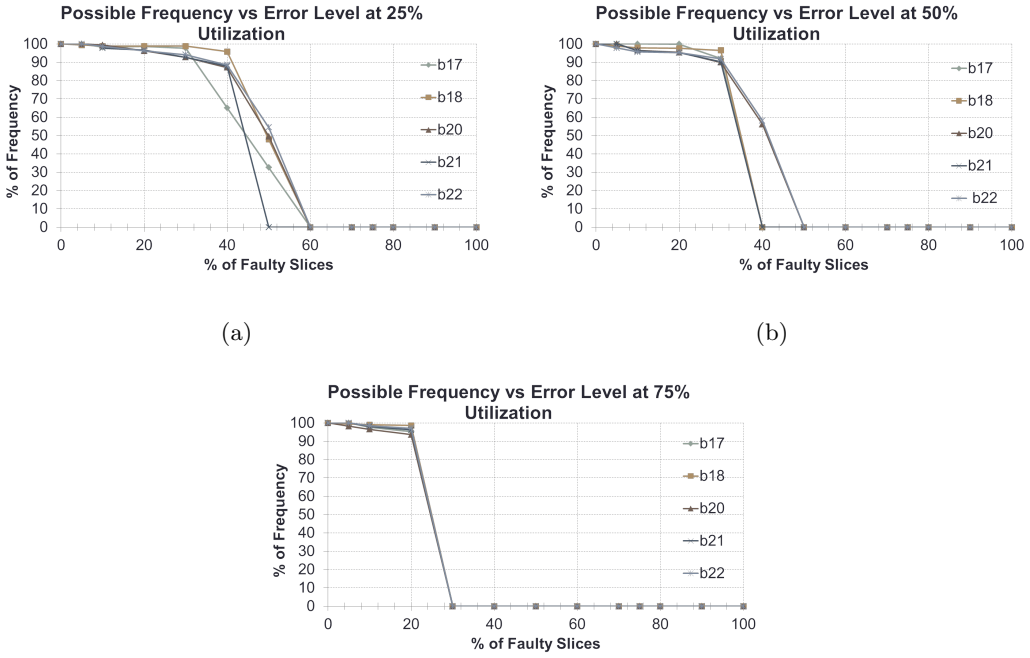


Figure 6.9 Degradation in frequency required for test runs that failed for the circuit's original timing constraint at device utilizations of a) 25%, b) 50%, and c) 75%. All data points are within a 7% confidence interval at a confidence level of 90%.

6.5.2 Logic-slice Fault Tolerance when Degraded Performance is Permitted

When the timing constraints could not be met for a given run of a design, it was still possible to implement the design at a degraded frequency. The negative slack times reported by all designs that failed at various error levels were used to calculate the average percentage of the original frequency for which the designs would still meet timing. This section presents the results of that analysis. Table 6.1 summarizes the key observation obtained from this evaluation that gives evidence that our proposed approach is quite tolerant of FPGA logic-slice faults. It shows that designs using 25%, 50% and 75% of the chip can tolerate 30%, 30% and 20% logic-slice faults respectively.

The success rate of the proposed approach for a given percentage of errors can be seen in Figures 6.8(a), 6.8(b) and 6.8(c). Figures 6.9(a), 6.9(b) and 6.9(c) must be looked at in the context of Figures 6.8(a), 6.8(b) and 6.8(c) respectively. For example, if for a utilization of 25% and 20% logic-slice errors the success rate in Figure 6.8(a) is 70%, then 30% of the runs

fail at the original frequency. It is these runs that require the frequency degradation shown in Figure 6.9(a).

As can be seen from Figure 6.9(a) and Figure 6.8(a), at a 25% utilization and up to 15% errors, the designs can be implemented in 67.5%-82.5% of the cases. In the cases when this is not possible, they still can be implemented after about a 1% degradation in performance, until about 15% of the logic slices are faulty. From 15% to 30% errors, all the designs can be implemented with a frequency degradation of between 2%-7%. Thus, the empirical evidence suggests that for a design that utilizes only 25% of the chip, our fault-aware toolchain approach can tolerate up to 30% of the chip being faulty at a reasonable performance cost (e.g., less than the performance cost typically associated with stepping down by a speed grade).

Similarly, from Figure 6.8(b) and Figure 6.9(b) it can be seen that between 32%-56% of the designs successfully meet timing until about 20% of the logic slices are faulty. Those that fail can meet timing with a 8%-10% frequency degradation, until about 30% of the logic slices are faulty. Thus, the experimental results suggest that for a design that utilizes 50% of the chip, the proposed approach can tolerate up to 30% of the chip being faulty at a reasonable performance cost.

From Figure 6.8(c) and 6.9(c) it can be seen that there is a knee point at an error level of 20%-25%, below which approximately 90% of the time the designs are successfully implemented. For the remaining 10% of the cases that the designs fail timing, a 3%-5% frequency degradation allows them to be implemented. This knee point varies with utilization. In these graphs the knee point occurs when about 20% of the FPGA is made faulty in the worst case (i.e., at 75% utilization). If the design being implemented is smaller, then up to 30% of the FPGA logic slices being faulty can be tolerated by the proposed approach for a maximum performance cost of 10% (see Table 6.1).

6.5.3 Routing Faults Compared to Logic-slice Faults

In [107], Dehon states that 80-90% of an FPGA's area is allocated to routing resources. Thus, it is important to evaluate our proposed approach in the presents of routing faults in addition to logic-slice faults. Figures 6.10(a) and 6.10(b) show that routing faults can have

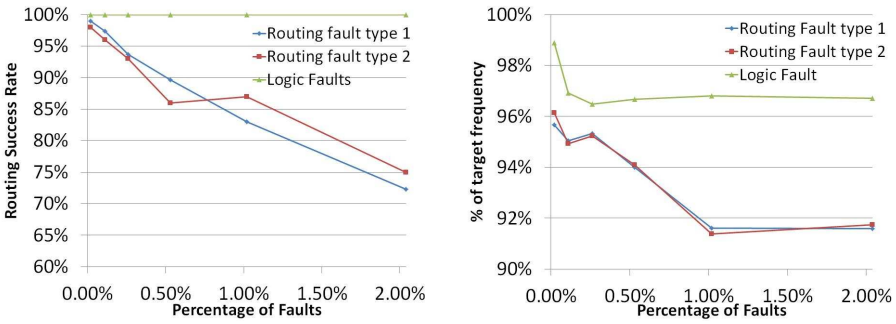


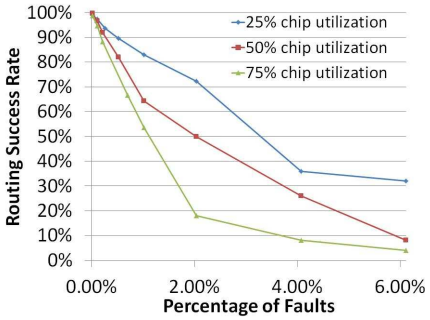
Figure 6.10 Comparing Logic-slice & routing faults, a) connectivity versus percent of faults, and b) Percent degradation of circuit frequency, for b17 using 25% of the FPGA.

a much larger impact on the toolschain's ability to tolerate faults as compared to logic-slice faults.

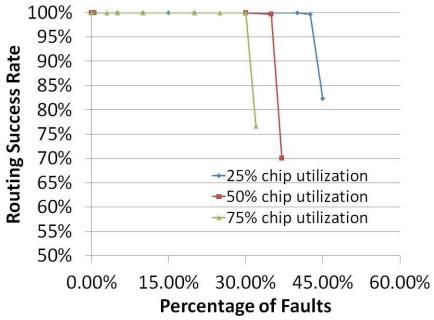
Figure 6.10(a) quantifies at what fault-level the toolchain can no longer route (i.e. fully connect) the design. This concept did not exist when only logic-slice faults were emulated, since 100% of the routing resources were available for constructing any given signal's path. However, now that routing resources are being reduced (i.e. removing available switch matrices), the tools quickly run into issues fully connecting (i.e. routing) the design. In this case, even with only 25% of the FPGA is being utilized and when just over 1% routing faults are present, approximately 15% of the designs are not connectable by the toolchain. Where as for fault scenarios made up only of logic-slice faults, the toolchain could route designs up until the point that there were not enough fault-free logic slices to implement the design.

Figure 6.10(b) illustrates that routing faults tend to place and route at lower frequencies, as compared to when only logic-slice faults are present. However, this difference in operating frequency is overshadowed by the routing-fault's impact on circuit connectivity, shown in Figure 6.10(a)

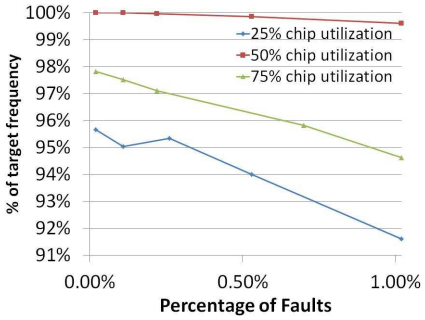
Loosely coupled vs. tightly coupled cores Figures 6.11(a)-6.11(d) show that characteristics of a design play a strong role in the toolchain's ability to successfully and efficiently place and route that design. For example, in Figure 6.11(a), in most cases before the percent of routing faults reaches 2%, over 30% of the designs are not even connectable by the toolchain.



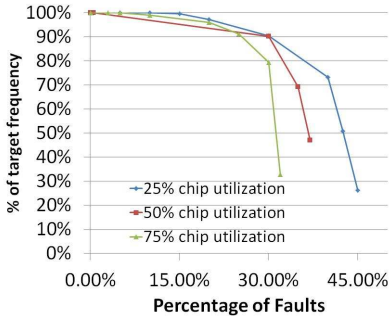
(a) Routability for b17: dense 80386 cores



(b) Routability for b20: small loosely coupled Viper cores



(c) Performance degradation for b17: dense 80386 cores



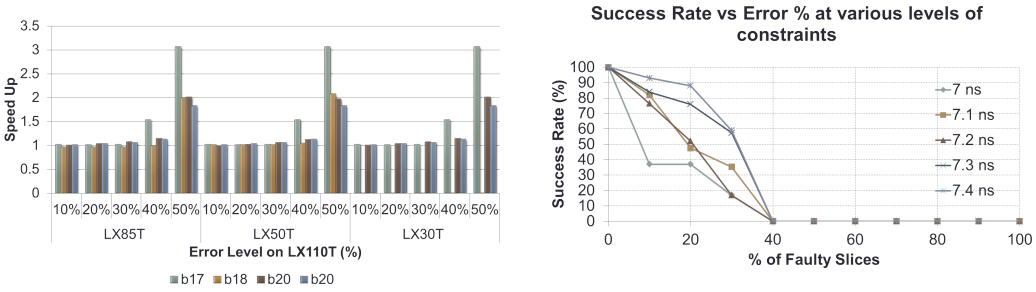
(d) Performance degradation for b20: small loosely coupled Viper cores

Figure 6.11 It can be seen that designs made up of small loosely coupled cores allow the toolchain to tolerate routing faults significantly better than designs composed of large densely routed cores.

Further more Figure 6.11(c) shows that even for just 1% routing faults, the designs for these same fault emulation experiments are already quickly degrading in operating frequency performance. These two figures illustrate the results for benchmark b17, which is made up of large densely routed cores.

However, benchmark b20, which is composed of smaller loosely coupled cores, shows a very different behavior. Figure 6.11(b) shows that the toolchain can fully connect the design, even as the number of routing faults reaches 30% or more. In addition, Figure 6.11(d) shows that for b20 the toolchain can tolerate up to 25% routing faults before degrading the operating frequency by more than 10%.

These results clearly show that the toolchain can better tolerate routing faults when implementing designs made up of small loosely coupled cores, as opposed to designs composed of large densely routed cores.



(a) (b)

Figure 6.12 a) Shows the ratio of the frequency at which different benchmarks could have been implemented on smaller fault-free chips as compared to the maximum frequencies possible at various errors level on the larger LX110T FPGA. These tests were run for benchmarks that utilized 25% of the LX110T, b) Sensitivity Analysis: success rate at various error percentages with different timing constraints in nanoseconds (benchmark b17, 25% utilization).

6.5.4 Comparison with Smaller, Fault-free Chips

Figure 6.12(a) shows the ratio of the maximum frequency at each error level in the larger chip (LX110T) with respect to the frequency possible in smaller fault-free FPGAs. A ratio of 1 indicates the same performance. As can be seen in Figure 6.12(a), having up to 30% logic-slice errors in the larger chip results in approximately the same performance as a smaller fault free chip. Thus a design that fits a smaller fault-free chip can be efficiently implemented on a larger faulty chip having the same number of fault free logic slices. This observation could be used by manufacturers to help market chips with faults at a reduced price.

If a manufacturer were to choose an error threshold for which any chip below this threshold was deemed appropriate for market, then the manufacturer's effective yields would increase to that given in Equation 6.1. Where RBT (Rejected Below Threshold) is the % of originally rejected chips that have an error level below a set threshold.

$$Effective\_Yield = Old\_Yield + \frac{(100 - Old\_Yield) * (RBT)}{100} \tag{6.1}$$

6.5.5 Timing Constraint Sensitivity

All benchmark designs were constrained to within approximately 0.2ns of their minimum fault-free periods. In order to get an idea of how sensitive success rate was to how tightly

the design was constrained, we ran tests to constrain benchmark b17 from 7ns (∼minimum fault-free period) to 7.4ns. Figure 6.12(b) shows the findings for this experiment. As can be seen in the figure, the tools are quite sensitive to how tightly the design is constrained. Thus, it seems reasonable that relaxing the timing constraints by just 10% (∼.7ns) allows the tools to tolerate a significant number of faults.

6.6 Conclusion

We have put forth and evaluated an approach for developing an FPGA implementation toolchain that is fault-aware. Our results have shown that this approach can tolerate a significant number of logic-slice faults (up to 30%) with a modest decrease in circuit frequency performance (10% or less) for the Virtex 5 LX110T. With respect to routing faults, it was found that designs made up of small loosely coupled modules could tolerate up to approximately 30% faults as well. However, for designs composed of large and tightly coupled cores, only about 1% routing faults could be tolerated (a factor of 30 less).

Even for designs in which only 1% of routing faults could be tolerated (which is still a significant number of faults), it is worth recalling that for our experiments we defined a single routing fault to be an entire switch matrix becoming unusable. This is an extremely pessimistic model of a routing fault. It is expected that the tools would perform much better if only a couple of the routes within a given switch matrix were marked faulty.

We have also proposed the idea of establishing an equivalence between larger, faulty FPGAs and smaller, non-faulty FPGAs. Our results show that for logic-slice faults such an equivalence is feasible.

It has been clearly shown that while FPGA toolchains were not designed to avoid faults, they show an impressive capability to do so. If fault information could be coupled more tightly with the tool's underlying routing algorithm, it is expected that even more impressive performance could be obtained. Our results provide strong motivation for further exploration of such research directions.

CHAPTER 7. CONCLUSION AND FUTURE WORK

A co-processor to mitigate the affect of delay on control loops was developed. Its software configurability allows little to no hardware knowledge to use while maintaining the original assumption by control engineers of constant, non-varying delay and thus assuring stability. There are methods in scheduling and control theory that mitigate the effects of delay variations, but such methods consume processor resources which may be limited for systems of light weight and low power. Two scalable versions of state-space based co-processors were also designed and evaluated. It was discovered that the LUT-based memory architecture was efficient in using logic resources and minimized wastage at the cost of reduced clock speed when increasing parallelism. The BRAM architecture is useful when implementing large systems that have tight delay constraints.

The architecture was tested in an emulated environment. In the future, it would be nice to test the system with an actual plant. Run time software reconfiguring could be taken advantage of to implement piece-wise linear controllers and other forms of hybrid control.

APPENDIX A. FIXED-POINT MATH

Our PoC uses fixed-point math, which is prone to overflow and underflow issues if the radix-point is not chosen correctly [26]. We use Matlab's fixed-point math functions as seen in Listing A.1, to determine an appropriate radix-point[1] for each variable and coefficient. Our PoC implementation performs calculations and represents the emulator's internal state using a 64-bit fixed-point format with a 48-bit fractional part (line 27) The 32-bit controller used two radix-point configurations; 16-bit fraction for large intermediate values (lines 44 and 45) and 20-bit fraction for calculations requiring more accurate and smaller values (line 37 and 57)[2]. We used 32-bit based math for the controller since many embedded processors, like NIOS II are 32-bit. Also commercial analog-to-digital (ADC) converters and digital-to-analog (DAC) converters usually do not have resolutions higher than 32-bits. Lines 36 to 59 implements the control loop, where reference-point r is set (line 38), the plant's state is updated every iteration (line 56) and the controller is triggered every sample-period (line 39 to 54). Lines 48 to 52 of the code limit the magnitude of the control signal to 40 Newtons adding non-linearity to the actuator and making the system model more realistic.

```
1 %Inverted_Pendulum_fixedpoint_model.m
2 %A,B,C,D from Ogata
3
4 %fixed point setting => truncate numbers less than LSbit
5 F2= fimath('RoundMode','floor','MaxSumWordLength',200);
6
7 %desired pole positions for closed-loop system
```

[1]line 5 contains the rules of rounding for fixed-point arithmetic results. These rules match those that are followed by the PoC. The Matlab code for calculating the coefficients for the controller and plant's state-space model can be found in [97]

[2]Though fewer bits for fraction would suffice to prevent rounding errors (e.g.41-bits instead of 48 for the plant), we decided to use conventional formats which are multiples of 8

```
8  J = [-1+02*1i -1-2*1i -19 -19 -19];
9
10 % controller gain calculation
11 Khat = acker(Ahat,Bhat,J);%Ahat Bhat from Ogata
12 %controller coefficients
13 K  = fi([Khat(1) Khat(2) Khat(3) Khat(4)],1,32,20,F2);
14 %integrator gain
15 KI     = fi(-Khat(5), 1,  32,    20,       F2);
16 %'fi' ref => value,signed|word  |fraction|options
17 %                        |length|length
18
19 %digitizing plant
20   % creating state-space model of plant
21   sysC      = ss(A,B,C,0);%creating state-space model
22   % simulation time step
23   time_step = 0.0001;
24   % creating digital version of plant
25   sysD      = c2d(sysC,time_step,'zoh');%digitizing system
26
27   %digitized Plant's state-space
28   A_D = fi(sysD.a,1,64,48,F2);%64.48 fixed point to eliminate
29                               %'rounding to zero' errors
30   % similar steps for B,C and D of plant
31
32 sim_time = 1;                 %simulation time = 1 second
33 steps    = sim_time/time_step;%time between calculations
34 time_step_intgrtr = 0.002; %time between samples
35
36 for n = 1:1:steps;            % run from t=0 till 1 second
37   t = n*time_step;
38   r = fi(0.5,1,32,20,F2);% ref. signal of 0.5meter step at t=0
39   if(mod(t,time_step_integrator)==0)%update cntrler each 2ms
```

```
40      %calculating error between r and actual position
41      Error               = fi(r - y,1,32,20,F2);
42      error_scaled= fi(Error*time_step_intgrtr,1,32,20,F2);
43      integrator      = fi(integrator+error_scaled,1,32,20,F2);
44   %NOTE:32.16 fixed point format for large intermediate values
45      KX                  = -fi(K*X,1,32,16,F2);
46      IKI                 =  fi(integrator*KI,1,32,16,F2);
47      u                   = fi(KX+IKI,1,32,20,F2);%control signal
48
49      if(u>fi(40,1,32,20))      %saturate to 40N force
50        u                 = fi(40,1,32,20);
51      elseif(u<fi(-40,1,32,20))%saturate to -40N force
52        u                 = fi(-40,1,32,20);
53      end
54    end
55    %Calculating new state of the plant
56    X = fi(A_D*X + B_D*(u),1,64,48,F2);
57    %extracting plant position(sampling sensors)
58    y = fi(C_D*X,1,32,20,F2);
59  end
```

Listing A.1 Fixed-point math model of digitally controlled PoC

APPENDIX B. JITTERBUG SUPPORT

Details on using JitterBug can be found in the manual [19]. Here, we give an example of how we used JitterBug for our research. In JitterBug, the metric of robustness is a cost function (the square of error with respect to zero) which the designer can select. In our case we inject a noise or disturbance in the cart position through R_1 (line 4) and keep the sensors noise-free through R_2 (line 5). The delay information is passed through probability density functions[1], one for each system node in a timing-node network (lines 37-39) that is mapped to system system block (lines 42-44). Our cost function, defined by Q (line 6), is equal to ${e_\theta}^2 + {e_{\dot{\theta}}}^2 + {e_x}^2$. We have two setups for JitterBug, one in which we have complete control over the sample period and delay, as we do in the Simulink setup. The other setup takes the probability distribution of the computational delay observed in the PoC experiments.

```
1  % Jitterbug: Inverted_Pendulum_PoC.m
2  % ================================
3  G = ss(A,B,C,D);
4  R1 = diag([0 0 1 0]);  % Input noise on position i.e. x3
5  R2 = diag([0 0 0 0]);  % No Output noise
6  Q = [1 0 0 0 0;...
7       0 1 0 0 0;...
8       0 0 1 0 0;...% cost  = x1^2 + x2^2 + x3^2
9       0 0 0 0 0;...%       = theta^2 + theta_dot^2 + position^2
10      0 0 0 0 0];
11 use_OS_pdf = 1;
```

[1]The developers of JitterBug have also made TrueTime, a processor simulator which generates pdf's for JitterBug according to the information passed to it like the scheduling and control algorithms used and behavior of interrupts. It is difficult to mimic and compare multiple commercial processors in such a manner as the information required is difficult to feed to the tool. Instead of feeding information about NIOS II to TrueTime, we found it easier to take the timing information directly from NIOS II through our *System Profiler* and pass the pdf's to JitterBug

```
%range of sample periods to explore
hvec = (Start_period:Steps_period:End_period)/1000;
h_it = 0;
for h = hvec %moving through sample period points
h_it = h_it+1;
  dt = h/40;            %delay step size
  taumaxvec = 0:2*dt:h;%delay array
  tau_it = 0;
  for taumax=taumaxvec % moving through delay points
      tau_it = tau_it + 1;
  %chosing delay probability density function for controller
    if(use_OS_pdf==0)
      %use fixed delay, no distribution
      Ptau         = zeros(1,round(h/dt)+1);
      Ptau(tau_it) = 1;
    else
      %use OS derived delay distribution
      Ptau = myPtau_surf(tau_it,:,h_it);
    end
    H1 = eye(4);                          % Sampler
    H2 = c2d(ss(1,[0 0 -1 0],KI,-K),h); % Controller
    H3 = 1;                               % Actuator

    N = initjitterbug(dt,h);           % Initialize Jitterbug

    N = addtimingnode(N,1,2); % Add node 1  sampler no delay
    N = addtimingnode(N,2,Ptau,3);% Add node 2 controller
    N = addtimingnode(N,3);   % Add node 3 actuator no delay

    N = addcontsys(N,1,G,4,Q,R1,R2);% Add sys 1 (G) plant
    N = adddiscsys(N,2,H1,1,1); % Add sys 2 (H1) to node 1
    N = adddiscsys(N,3,H2,2,2); % Add sys 3 (H2) to node 2
```

```
44      N = adddiscsys(N,4,H3,3,3); % Add sys 4 (H3) to node 3
45
46      N = calcdynamics(N);  % Calculate the internal dynamics
47      J = calccost(N)       % Calculate the cost
48      Jmat(find(h==hvec),find(taumax==taumaxvec)) = J;%store
49    end
50  end
```

Listing B.1 JitterBug for fixed delay of digitally controlled PoC

BIBLIOGRAPHY

[1] A. Aminifar, E. Bini, P. Eles, and Zebo Peng. Bandwidth-efficient controller-server co-design with stability guarantees. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–6, March 2014.

[2] Amir Aminifar, Soheil Samii, Petru Eles, Zebo Peng, and Anton Cervin. Designing high-quality embedded control systems with guaranteed stability. In *33rd IEEE Real-Time Systems Symposium*, 2012.

[3] Karl-Erik Arzen and Anton Cervin. Control and embedded computing: Survey of research directions. In *Proceedings of the 16th IFAC World Congress*. Elsevier, 2005.

[4] Karl J. Astrom. Challenges in control education. In *7th IFAC Symposium on Advances in Control Education (ACE)*, Madrid, Spain, June 2006.

[5] Karl J. Åström and Bjorn Wittenmark. *Computer-controlled systems*. Prentice-Hall Inc., third edition, 1997.

[6] Karl Johan Astrom and Bjorn Wittenmark. *Adaptive Control*. Addison-Wesley Longman Publishing Co., 2nd edition, 1994.

[7] T.O. Bachir and J.-P. David. Fpga-based real-time simulation of state-space models using floating-point cores. In *14th International Power Electronics and Motion Control Conference*, 2010.

[8] K. Basterretxea and K. Benkrid. Embedded high-speed model predictive controller on a FPGA. In *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2011.

[9] Xiong Bing and C. Charoensak. Rapid FPGA prototyping of Gabor-wavelet transform for applications in motion detection. In *Proceedings of the 7th International Conference on Control, Automation, Robotics and Vision*, volume 3, pages 1653 – 1657, Singapore, 2002.

[10] E. Bini and A. Cervin. Delay-Aware Period Assignment in Control Systems. In *Real-Time Systems Symposium*, 2008.

[11] L.G. Bleris, P.D. Vouzis, M.G. Arnold, and M.V. Kothare. A co-processor FPGA platform for the implementation of real-time model predictive control. In *American Control Conference*, June 2006.

[12] L.G. Bleris, P.D. Vouzis, M.G. Arnold, and M.V. Kothare. A co-processor fpga platform for the implementation of real-time model predictive control. In *American Control Conference*, 2006.

[13] Giorgio Buttazzo and Anton Cervin. Comparative Assessment and Evaluation of Jitter Control Methods. In *Proceedings of the 15th International Conference on Real-Time and Network Systems*, 2007.

[14] N. Campregher. FPGA interconnect fault tolerance. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, Copenhagen, Denmark, 2005.

[15] B.B. Carvalho, A.J.N. Batista, M. Correia, A. Neto, H. Fernandes, B. Gonalves, and J. Sousa. Reconfigurable atca hardware for plasma control and data acquisition. *Fusion Engineering and Design*, 85(3-4):298 – 302, 2010. Proceedings of the 7th IAEA Technical Meeting on Control, Data Acquisition, and Remote Participation for Fusion Research.

[16] A. Cervin. Stability and worst-case performance analysis of sampled-data control systems with input and output jitter. In *American Control Conference (ACC), 2012*, 2012.

[17] A. Cervin, K.-E. Arzen, D. Henriksson, M. Lluesma, P. Balbastre, I. Ripoll, and A. Crespo. Control Loop Timing Analysis using TrueTime and JitterBug. In *IEEE International Conference on Control Applications*, 2006.

[18] Anton Cervin. Stability and Worst-Case Performance Analysis of Sampled-Data Control Systems with Input and Output Jitter. In *American Control Conference*, 2012.

[19] Anton Cervin and Bo Lincoln. Jitterbug 1.23—Reference Manual. Technical report, Department of Automatic Control, Lund University, Sweden, July 2010.

[20] Anton Cervin, Bo Lincoln, Johan Eker, Karl-Erik Årzén, and Giorgio Buttazzo. The jitter margin and its application in the design of real-time control systems [elektronisk resurs]. *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications*, 2004.

[21] Bo Cervin, Anton ans Lincoln, Johan Eker, Karl Arzen, and Giorgio Buttazzo. The Jitter Margin and Its Application in the Design of Real-Time Control Systems. In *10th International Conference on Real-Time and Embedded Computing Systems and Applications*, 2004.

[22] Y.F. Chan, M. Moallem, and W. Wang. Efficient implementation of pid control algorithm using fpga technology. 2004.

[23] Yuen Fong Chan, M. Moallem, and Wei Wang. Design and implementation of modular fpga-based pid controllers. *Industrial Electronics, IEEE Transactions on*, 2007.

[24] Jason A. Cheatham, John M. Emmert, and Stan Baumgart. A survey of fault tolerant methodologies for FPGAs. *ACM Transaction on Design Automation of Electronic Systems*, 11:501–533, April 2006.

[25] Tongwen Chen, Bruce Francis, and T Hagiwara. Optimal sampled-data control systems. *Proceedings of the IEEE*, 86(4):741–741, 1998.

[26] George A. Constantinides. *Choose-your-own-adventure routing: lightweight load-time defect avoidance*, chapter Precision Analysis of Fixed-Point Computation, pages 23–32. In [107], 2009.

[27] F. Corno, M.S. Reorda, and G. Squillero. RT-level ITC'99 benchmarks and first ATPG results. *IEEE Design and Test of Computers*, 2000.

[28] S. Durand and C. Piguet. FPGA with self-repair capabilities. In *Proceedings of the ACM International Workshop on Field Programmable Gate Arrays.*, New York, 1994.

[29] B.F. Dutton and C.E. Stroud. Built-In Self-Test of configurable logic blocks in Virtex-5 FPGAs. In *Proceedings of the 41st Southeastern Symposium on System Theory*, Tullahoma, TN, 2009.

[30] B.F. Dutton and C.E. Stroud. Built-In Self-Test of programmable input/output tiles in Virtex-5 FPGAs. In *Proceedings of the 41st Southeastern Symposium on System Theory*, 2009.

[31] C. Economakos and G. Economakos. Fpga implementation of plc programs using automated high-level synthesis tools. In *Industrial Electronics, 2008. ISIE 2008. IEEE International Symposium on*, pages 1908 –1913, 30 2008-july 2 2008.

[32] Embedded Microprocessor Benchmark Consortium (EEMBC) homepage, 2008. http://www.eembc.org.

[33] John M. Emmert and Dinesh Bhatia. Partial reconfiguration of FPGA mapped designs with applications to fault tolerance and yield enhancement. In *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, pages 141–150, London, UK, 1997.

[34] J. Engblom. Analysis of the Execution Time Unpredictability Caused by Dynamic Branch Prediction. In *Proceedings of The 9th Real-Time and Embedded Technology and Applications Symposium*, 2003.

[35] G. Frantz. Digital signal processor trends. *Micro, IEEE*, 20(6):52 –59, 2000.

[36] Hisaya Fujioka. Stability analysis of systems with aperiodic sample-and-hold devices. *Automatica*, 2009.

[37] S. Ganeriwal, C. Han, M. B., and Srivastava. Going beyond nodal aggregates: Spatial average of a continuous physical process in sensor networks. In *Poster in Sensys*, 2003.

[38] B. Garbergs and B. Sohlberg. Specialised hardware for state space control of a dynamic process. In *TENCON '96. Proceedings., 1996 IEEE TENCON. Digital Signal Processing Applications*, volume 2, pages 895–899 vol.2, Nov 1996.

[39] B. Garbergs and B. Sohlberg. Implementation of a state space controller in a fpga. In *Electrotechnical Conference, 1998. MELECON 98., 9th Mediterranean*, volume 1, pages 566–569 vol.1, May 1998.

[40] Kai Goebel and Weizhong Yan. Correcting sensor drift and intermittency faults with data fusion and automated learning. In *IEEE Systems Journal*, volume 2, June 2008.

[41] Marcella M. Gomez and Richard M. Murray. Stabilization of feedback systems via distribution of delays. In *10th IFAC Workshop on Time Delay Systems*, 2012.

[42] Marcella M Gomez and Richard M Murray. Stabilization of feedback systems via distribution of delays. In *10th IFAC Workshop on Time Delay Systems*, 2012.

[43] B. Gonalves, J. Sousa, and C.A.F. Varandas. Real-time control of fusion reactors. *Energy Conversion and Management*, 51(9):1751 – 1757, 2010. 14th International Conference on Emerging Nuclear Systems (ICENES 2009).

[44] Lin Gu, Dong Jia, Pascal Vicaire, Ting Yan, Liqian Luo, Ajay Tirumala, Qing Cao, Tian He, John A. Stankovic, Tarek Abdelzaher, and Bruce H. Krogh. Lightweight detection and classification for wireless sensor networks in realistic environments. In *Proceedings of SenSys*, San Diego, California, 2005.

[45] A. Gupta and J.W. Lathrop. Yield analysis of large integrated-circuit chips. *IEEE Journal of Solid-State Circuits*, 7(5):389 – 395, October 1972.

[46] A. Gupte and P. Jones. An evaluation of a slice fault aware tool chain. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, 2010.

[47] A. Gupte and P. H. Jones. Towards hardware support for common sensor processing tasks. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference on*, 2009.

[48] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.

[49] David A. Gwaltney, Kenneth D. King, and Keary J. Smith. Implementation of adaptive digital controllers on programmable logic devices, 2002.

[50] David A. Gwaltney, Kenneth D. King, and Keary J. Smith. Implementation of adaptive digital controllers on programmable logic devices, 2002.

[51] David A Gwaltney, Kenneth D King, Keary J Smith, and Justino Montenegro. Implementation of Adaptive Digital Controllers on Programmable Logic Devices. *Military and Aerospace Programmable Logic Devices (MAPLD)*, Sept 2002.

[52] F. Hanchek and S. Dutt. Design methodologies for tolerating cell and interconnect faults in FPGAs. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 326 –331, Austin, TX, October 1996.

[53] Edward N Hartley, Juan L Jerez, Andrea Suardi, Jan M Maciejowski, Eric C Kerrigan, and George A Constantinides. Predictive control of a boeing 747 aircraft using an fpga. 2012.

[54] F. Hatori, T. Sakurai, K. Nogami, K. Sawada, M. Takahashi, M. Ichida, M. Uchida, I. Yoshii, Y. Kawahara, T. Hibi, Y. Saeki, H. Muroga, A. Tanaka, and K. Kanzaki. Introducing redundancy in Field Programmable Gate Arrays. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 7.1.1 –7.1.4, San Diego, CA, May 1993.

[55] J. Hellerstein, W. Hong, S. Madden, and K. Stanek. Beyond average: Towards sophisticated sensing with queries. In *Proceedings of IPSN*, 2003.

[56] Henriksson, Dan, Redell, Ola, El Khoury, Jad, Törngren, Martin, Årzén, and Karl Erik. Tools for real-time control systems co-design — a survey. Technical Report ISRN

LUTFD2/TFRT--7612--SE, Department of Automatic Control, Lund Institute of Technology, Sweden, 2005.

[57] N.J. Howard, A.M. Tyrrell, and N.M. Allinson. The yield enhancement of Field Programmable Gate Arrays. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(1):115 –123, March 1994.

[58] Chenglin Hu and Feng Wan. Parameter identification of a model with Coulomb friction for a real Inverted Pendulum System. In *Control and Decision Conference (CCDC)*, pages 2869–2874, June 2009.

[59] Jing Huang, M.B. Tahoori, and F. Lombardi. Fault tolerance of switch blocks and switch block arrays in FPGA. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 13(7):794 –807, july 2005.

[60] T. Inoue, S. Miyazaki, and H. Fujiwara. Universal fault diagnosis for lookup table FPGAs. *IEEE Design and Test of Computers*, 1998.

[61] Intel Inc. *Intel Centrino Mobile Technology, Wake on Wireless LAN Feature*, 2006.

[62] Intel Inc. *Intel Atom Processor Z5XX Series Datasheet, Section 5.1.2 Intel Thermal Monitor*, 2010.

[63] Alf Isaksson and Tore Hagglund. Editorial. In *IEE Proceedings of Control Theory Applications*, volume 149, January 2002.

[64] N. Itazaki, F. Matsuki, Y. Matsumoto, and K. Kinoshita. Built-In Self-Test for multiple CLB faults of a LUT type FPGA. In *Proceedings of Seventh Asian Test Symposium*, pages 272 –277, Singapore City, Singapore, December 1998.

[65] ITRS. The International Technology Roadmap for Semiconductors (ITRS), Lithography. http://www.itrs.net/, 2009.

[66] S. R. Jeffery, G. Alonso, M. J. Franklin, W. Hong, and J. Widom. Declarative support for sensor data cleaning. In *Proceedings of PerCom*, 2006.

[67] J.L Jerez, G.A Constantinides, and E.C. Kerrigan. An FPGA implementation of a sparse quadratic programming solver for constrained predictive control. In *In Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 209–218, June 2011.

[68] Juan Luis Jerez, George Anthony Constantinides, and Eric C. Kerrigan. An fpga implementation of a sparse quadratic programming solver for constrained predictive control. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011.

[69] John and Robertson. Growth of nanotubes for electronics. *Materials Today*, 10(12):36 – 43, 2007.

[70] J. Keane, Xiaofei Wang, D. Persaud, and C.H. Kim. An all-in-one silicon odometer for separately monitoring HCI, BTI, and TDDB. *Solid-State Circuits, IEEE Journal of*, 45(4):817 –829, april 2010.

[71] J.L. Kelly and P.A. Ivey. Defect tolerant SRAM based FPGAs. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 479 –482, Cambridge, MA, October 1994.

[72] H.R. Khan, D. Mamaluy, and D. Vasileska. Simulation of the impact of process variation on the optimized 10-nm FinFET. *Electron Devices, IEEE Transactions on*, 2008.

[73] K.O. Kim, M.J. Zuo, and W. Kuo. On the relationship of semiconductor yield and reliability. *Semiconductor Manufacturing, IEEE Transactions on*, 2005.

[74] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 2nd edition, 10 January 1981.

[75] S. Kozak. Advanced control engineering methods in modern technological applications. In *Carpathian Control Conference (ICCC)*, pages 392–397, May 2012.

[76] S. Kozak. Advanced control engineering methods in modern technological applications. In *13th International Carpathian Control Conference (ICCC)*, 2012.

[77] K. Ling, B. Wu, and J. Maciejowski. Embedded model predictive control (mpc) using fpga. *Proc. of the 17th IFAC World Congr.*, 2008.

[78] J. Liu and S. Simmons. BIST-diagnosis of interconnect fault locations in FPGAs. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, volume 1, pages 207 – 210 vol.1, Montreal, Canada, May 2003.

[79] Kebin Liu, Lei Chen, Yunhao Liu, and Minglu Li. Robust and efficient aggregate query processing in wireless sensor networks. In *Springer Science + Business Media*, 2008.

[80] Pongstorn Maidee and Kia Bazargan. Defect-tolerant FPGA architecture exploration. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, 2006.

[81] Bill Messner and Dawn Tilbury. Inverted Pendulum: System Modeling. http://ctms.engin.umich.edu/CTMS, 2012.

[82] D. Mic, S. Oniga, E. Micu, and C. Lung. Complete hardware / software solution for implementing the control of the electrical machines with programmable logic circuits. 2008.

[83] Aaron Mills, Pei Zhang, Sudhanshu Vyas, Joseph Zambreno, and Phillip Jones. A software configurable coprocessor-based state-space controller. In *Proceedings of the International Symposium on Field-Programmable Logic and Applications (FPL)*, 2015.

[84] Minorsky. Directional stability of automatically steered bodies. *Journal of the American Society for Naval Engineers*, 34(2):280–309, 1922.

[85] E. Monmasson and M. Cirstea. Guest editorial special section on industrial control applications of fpgas. *Industrial Informatics, IEEE Transactions on*, 9(3):1250–1252, Aug 2013.

[86] E. Monmasson, L. Idkhajine, and M. W Naouar. FPGA-based Controllers. *Industrial Electronics Magazine, IEEE*, 2011.

[87] E. Monmasson, L. Idkhajine, and M-w Naouar. Fpga-based controllers. *IEEE Industrial Electronics Magazine*, 2011.

[88] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82 –85, jan 1998.

[89] Gordon E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *Solid-State Circuits Newsletter, IEEE*, 2006.

[90] B.R. Mutlu and M. Dolen. Implementations of state-space controllers using Field Programmable Gate Arrays. In *International Symposium on Power Electronics Electrical Drives Automation and Motion (SPEEDAM)*, pages 1436–1441, June 2010.

[91] Masayuki Nakamura, Atsushi Sakurai, Toshio Watanab, Jiro Nakamura, and Hiroshi Ban. Improved collaborative environment control using mote-based sensor/actuator networks. In *Proceedings of Conference on Local Computer Networks*, 2006.

[92] J. Narasimham, K. Nakajima, C.S. Rim, and A.T. Dahbura. Yield enhancement of programmable ASIC arrays by reconfiguration of circuit placements. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(8):976 –986, August 1994.

[93] National Instruments. Pacs for industrial control, the future of control. White paper, National Instruments, Aug 2011. Available online (3 pages).

[94] Farzad Nekoogar and Gene Moriarty. *Digital Control Using Digital Signal Processing (1st ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[95] A. O'Dwyer. Pid compensation of time delayed processes 1998-2002: a survey. 2003.

[96] Katsuhiko Ogata. *Modern Control Engineering (4th ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2002.

[97] Katsuhiko Ogata. *Modern Control Engineering*. Prentice Hall, fifth edition, 2009.

[98] K. Okumura, H. Oku, and M. Ishikawa. High-Speed Gaze Controller for Millisecond-Order Pan/Tilt Camera. In *IEEE International Conference on Robotics and Automation*, pages 6186–6191, May 2011.

[99] K. Okumura, H. Oku, and M. Ishikawa. High-speed gaze controller for millisecond-order pan/tilt camera. In *IEEE International Conference on Robotics and Automation (ICRA),*, 2011.

[100] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Real-Time Systems Symposium, 2008*, pages 481 –491, December 2008.

[101] B.G. Penaflor, J.R. Ferron, D.A. Piglowski, R.D. Johnson, and M.L. Walker. Real-time data acquisition and feedback control using linux intel computers. *Fusion Engineering and Design*, 81(15-17):1923 – 1926, 2006. 5th IAEA TM on Control, Data Acquisition, and Remote Participation for Fusion Research - 5th IAEA TM.

[102] M Petrov, I Ganchev, and A Taneva. Fuzzy pid control of nonlinear plants. In *Proceedings of Intelligent Systems*, 2002.

[103] Franco P. Preparata, Gernot Metze, and Robert T. Chien. On the connection assignment problem of diagnosable systems. *IEEE Transactions on Electronic Computers*, 1967.

[104] Frederick M. Proctor and William P. Shackleford. Real-time Operating System Timing Jitter and its Impact on Motor Control. In *SPIE Conference on Sensors and Control for Intelligent Manufacturing*, 2001.

[105] Frederick M. Proctor and William P. Shackleford. Timing Studies of Real-Time Linux for Control. In *Proceedings of Design Engineering Technical Conferences*, 2001.

[106] K. Roy and S. Nag. On routability for FPGAs under faulty conditions. *Computers, IEEE Transactions on*, 44(11):1296 –1305, nov 1995.

[107] Raphael Rubin and André DeHon. Choose-your-own-adventure routing: lightweight load-time defect avoidance. In *FPGA '09: Proceeding of the ACM/SIGDA international*

symposium on Field programmable gate arrays, pages 23–32, New York, NY, USA, 2009. ACM.

[108] L. Samet, N. Masmoudi, M.W. Kharrat, and L. Kamoun. A digital pid controller for real time and multi loop control: a comparative study. volume 1, pages 291 –296 vol.1, 1998.

[109] F. Sartori, T. Budd, P. Card, R. Felton, P. Lomas, P. McCullen, F. Piccolo, L. Zabeo, R. Albanese, G. Ambrosino, G. De Tommasi, and A. Pironti. Jet operations and plasma control: A plasma control system that is safe and flexible in a manageable way. 2009.

[110] R. Sipahi, S.-I. Niculescu, C.T. Abdallah, W. Michiels, and Keqin Gu. Stability and stabilization of systems with time delay. *Control Systems Magazine, IEEE*, 2011.

[111] E. Stott, P. Sedcole, and P. Cheung. Fault tolerant methods for reliability in FPGAs. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 415 –420, sept. 2008.

[112] E. Stott, P. Sedcole, and P. Cheung. Fault tolerance and reliability in field-programmable gate arrays. *Computers Digital Techniques, IET*, 4(3):196 –210, may 2010.

[113] E. Stott, J.S.J. Wong, and P.Y.K. Cheung. Degradation analysis and mitigation in FPGAs. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 428 –433, 31 2010-sept. 2 2010.

[114] Edward A. Stott, Justin S.J. Wong, Pete Sedcole, and Peter Y.K. Cheung. Degradation in FPGAs: measurement and modelling. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '10, pages 229–238, New York, NY, USA, 2010. ACM.

[115] C. Stroud, J. Nall, M. Lashinsky, and M. Abramovici. BIST-based diagnosis of FPGA interconnect. In *Proceedings of the International Test Conference*, pages 618 – 627, Baltimore, MD, 2002.

[116] Young Soo Suh. Send-on-delta sensor data transmission with a linear predictor. In *MDPI Sensors*, 2007.

[117] K. S. Tang, Kim Fung Man, Guanrong Chen, and Sam Kwong. An optimal fuzzy pid controller. In *IEEE Transactions on Industrial Electronics*, volume 48, August 2001.

[118] Yi-Wei Tu and Ming-Tzu Ho. Design and implementation of robust visual servoing control of an inverted pendulum with an fpga-based image co-processor. *Mechatronics*, 2011.

[119] YiWei Tu and MingTzu Ho. Design and implementation of robust visual servoing control of an inverted pendulum with an FPGA-based image co-processor. *Mechatronics*, 21(7):1170 – 1182, 2011.

[120] Martin Trngren. Fundamentals of implementing real-time control applications in distributed computer systems. *J. of Real-Time Systems*, 14:219–250, 1998.

[121] J. Vial, A. Bosio, P. Girard, C. Landrault, S. Pravossoudovitch, and A. Virazel. Using TMR architectures for yield improvement. In *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS '08. IEEE International Symposium on*, pages 7 –15, oct. 2008.

[122] A. Visioli. Tuning of pid controllers with fuzzy logic. *Control Theory and Applications, IEE Proceedings* -, 148(1):1 –8, jan. 2001.

[123] P.D. Vouzis, M.V. Kothare, L.G. Bleris, and M.G. Arnold. A System-on-a-Chip Implementation for Embedded Real-Time Model Predictive Control. *IEEE Transactions on Control Systems Technology*, 17(5):1006–1017, Sept 2009.

[124] P.D. Vouzis, M.V. Kothare, L.G. Bleris, and M.G. Arnold. A system-on-a-chip implementation for embedded real-time model predictive control. *IEEE Transactions on Control Systems Technology*, 2009.

[125] S. Vyas, N.G. Chetan Kumar, J. Zambreno, C. Gill, R. Cytron, and P. Jones. An FPGA-Based Plant-on-Chip Platform for Cyber-Physical System Analysis. *IEEE Embedded Systems Letters*, 6(1):4–7, March 2014.

[126] Bjrn Wittenmark, Karl Johan strm, and Karl erik rzn. Computer control: An overview. Technical report, 2003.

[127] Chi-Feng Wu and Cheng-Wen Wu. Fault detection and location of dynamic reconfigurable FPGAs. In *Proceedings of the International Symposium on VLSI Technology, Systems, and Applications*, pages 215 –218, Taipei, Taiwan, 1999.

[128] Yifan Wu, G. Buttazzo, E. Bini, and A. Cervin. Parameter Selection for Real-Time Controllers in Resource-Constrained Systems. *IEEE Transactions on Industrial Informatics*, 2010.

[129] Peter Petrov Xiangrong Zhou. Rapid and low-cost context-switch through embedded processor customization for real-time and control applications. In *Proceedings of Design Automation Conference*, 2006.

[130] Xilinx. *EasyPath-6 FPGA Product Brief*, 2011.

[131] Xilinx Inc. *Virtex-5 FPGA Datasheet*.

[132] Wei Zhao, Byung Hwa Kim, Amy C. Larson, and Richard M. Voyles. Fpga implementation of closed-loop control system for small-scale robot. In *in Proceedings of the 2005 International Conference on Advanced Robotics*, page 70, 2005.