Hardware Architectural Support for Control Systems and Sensor Processing

SUDHANSHU VYAS and ADWAIT GUPTE, Iowa State University
CHRISTOPHER D. GILL and RON K. CYTRON, Washington University in St. Louis
JOSEPH ZAMBRENO and PHILLIP H. JONES, Iowa State University

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

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms: Design, Experimentation, Measurement

Additional Key Words and Phrases: Control systems, real-time systems, sensor processing, application-specific processor, reconfigurable hardware

This work is supported by the National Science Foundation under grant no. CNS-1060337.
Authors' addresses: S. Vyas and A. Gupte, Electrical and Computer Engineering Department, Iowa State University, 2229 Lincoln Way, Ames, IA 50011; C. D. Gill and R. K. Cytron, Computer Science and Engineering Department, Washington University in St. Louis, 1 Brookings Dr., St. Louis, MO 63130; J. Zambreno and P. H. Jones (corresponding author), Electrical and Computer Engineering Department, Iowa State University, 2229 Lincoln Way, Ames, IA 50011; email: phjones@iastate.edu.

DOI: http://dx.doi.org/10.1145/2514641.2514643

ACM Reference Format:
Vyas, S., Gupte, A., Gill, C. D., Cytron, R. K., Zambreno, J., and Jones, P. H. 2013. Hardware architectural support for control systems and sensor processing. ACM Trans. Embedd. Comput. Syst. 13, 2, Article 16 (September 2013), 25 pages.
DOI: http://dx.doi.org/10.1145/2514641.2514643

1. INTRODUCTION

Control systems have a wide spectrum of applications, including aircraft flight control, carbon emission management, and national power grid management. In Astrom [2006] 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 [Ogata 2002]. 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 led some microprocessor architects to explore building architectures centered around MAC computation units, which resulted in the creation of Digital Signal Processors (DSPs) [Frantz 2000]. 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 [Frantz 2000; Nekoogar and Moriarty 1998].

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 [Frantz 2000]. 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 [Frantz 2000]. 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 O'Dwyer [2003] 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 [Penaflor et al. 2006], with tight timing bounds placed on the determinism of the control loop.

Eqs. (1) and (2) give the classic equation of a Proportional-Integral-Derivative (PID) controller in its continuous-time and discrete-time form, respectively.

$$u(t) = K_P e(t) + K_I \int_0^t e(t)\,\mathrm{d}t + K_D \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{1}$$

$$u[n] = K_P e[n] + K_I \sum_{j=0}^{n} e[j] + K_D(e[n] - e[n-1]) \tag{2}$$

Here:

—$u(t), u[n]$ 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 drawback of having a derivative component is that it amplifies system noise (e.g., vibrations, sensor error).

Computing Requirements. The performance requirements of a physical system dictate, 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 [Penaflor et al. 2006]. 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 the determinism of the timing of a control loop. Sources of jitter in an embedded system can stem from 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. Pellizzoni et al. [2008] implement a monitoring device on an FPGA to compensate for these unknown variables. In general, jitter can degrade a control system's performance [Törngren 1998]. One aspect of our work presents experimental results that quantify some sources of jitter within a general-purpose controller, and mitigate 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 onto the underlying computing medium. Some drawbacks with this common approach are 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, nondeterminism still exists in the system. Henriksson et al. [2005] have taken steps to close the gap between the development of control theory and nondeterministic 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 nondeterministic aspects of the computing medium.

Our work, illustrated in Figure 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

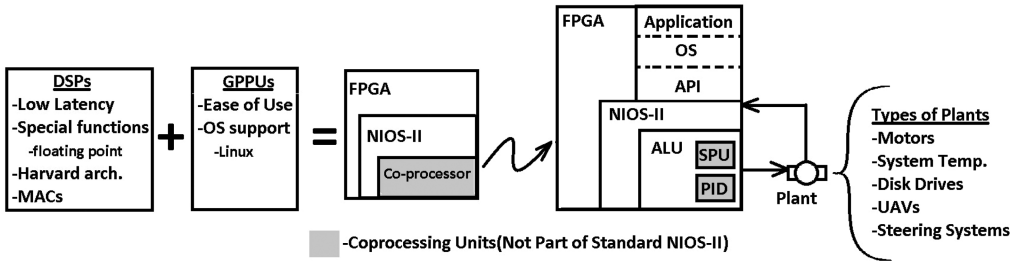


Fig. 1. In this article traits of both DSPs and General-Purpose Processor Units (GPPUs) are encapsulated in an FPGA-based system-on-chip architecture, which can be used to control a wide variety of plants.

Unit (SPU) [Gupte and Jones 2009] 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, nonreal-time tasks can be safely run and efficiently scheduled onto the same device that services the PID controllers. Additionally, our hardware-enforced mitigation of nondeterminism simplifies software development and verification, since a real-time scheduling scheme is not needed for sharing the computing medium between real-time and nonreal-time tasks. Section 4 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 [Gupte and Jones 2009].

Organization. The remainder of this article is organized as follows. Section 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 3, the platform used for deploying and characterizing our architecture is described. Section 4 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 5. Our approach for evaluating the SPU is also discussed in Section 5. Section 6 summarizes the analysis and results of our evaluation experiments. Section 7 concludes this article and suggests avenues of future research.

2. RELATED WORK

2.1. Implementing PIDs in Control Systems

The first publication of the PID concept was in 1922 [Minorsky 1922], and intensive research on PIDs continues today. In O'Dwyer [2003] it was estimated that 95% of control loops in the world contain PIDs, and it describes various research domains for PIDs, including tuning, cascading of PIDs, implementation techniques, and controller management. Visioli [2001] 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 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 Penaflor et al. [2006]. 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 [Carvalho et al. 2010; Sartori et al. 2009; Gonçalves et al. 2010], 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) [Economakos and Economakos 2008]. According to a study performed by National Instruments [2011], 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 [Economakos and Economakos 2008]. 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 1990's, which was the point at which it was first feasible to use these devices for control applications [Gwaltney et al. 2002]. Examples of research implementing PIDs on FPGAs are Samet et al. [1998] where various architectures of PIDs are tested on earlier FPGAs. In Mic et al. [2008] 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 [Chan et al. 2004, 2007]. Zhao et al. [2005] have 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 trade-offs associated with distributing the architecture across the hardware-software boundary, and characterize the jitter of the system [Cervin et al. 2004]. 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 became available on FPGAs, allowing software and OS's to be incorporated in designs.

2.2. Common Kernel for Sensor Processing

Due to the large and diverse set of domains in which embedded systems are used, past works [EEMBC 2008; Guthaus et al. 2001] 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 [Hellerstein et al. 2003; Goebel and Yan 2008; Petrov et al. 2002; Suh 2007] 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.

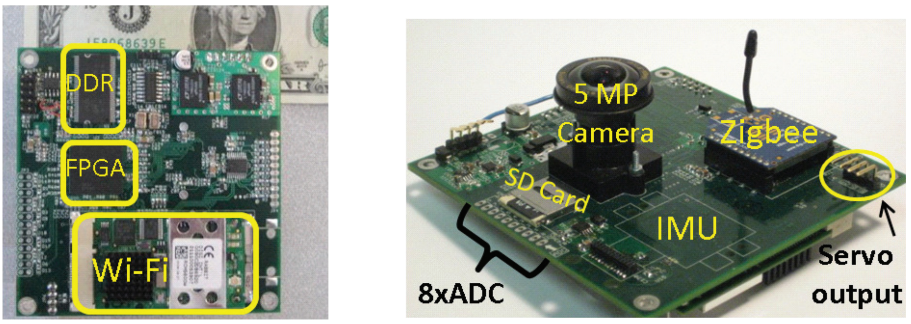
Linear Equations. One of the simplest tasks that can be performed on sensor data is the execution of a linear mathematical operation on a single value or multiple values. In the automotive domain, converting between radians and degrees is identified as a common task in Guthaus et al. [2001]. In the controls system domain the Proportional-Integral-Derivative (PID) controller is one of the most common control algorithms used [Isaksson and Hagglund 2002]. Tang et al. [2001] and Petrov et al. [2002] 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 impacts 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 [Hellerstein et al. 2003; Jeffery et al. 2006; Gu et al. 2005; Liu et al. 2008].

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 and Yan [2008], Ganeriwal et al. [2003], and Nakamura et al. [2006]. In Hellerstein et al. [2003] the authors use 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 Guthaus et al. [2001] includes delta modulation, a process of encoding which may be used to encode the output of sensors before transmission. Suh [2007] presents another example of using the "delta" value in a sensor-based system.

Threshold/Range Check. There are various applications in which the sensor values are required to be within particular range or below/above a certain threshold for the



(a) bottom view of RAVI FPGA-based platform

(b) top view of RAVI FPGA-based platform

Fig. 2. RAVI: FPGA-based board for developing custom architectures for embedded systems.

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 that supports these kernels.

3. DEVELOPMENT PLATFORMS

PID Coprocessor. This work was deployed and evaluated on the Reconfigurable Autonomous Vehicle Infrastructure (RAVI) board, an in-house-developed FPGA develpement 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 softcore 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 crosscut 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 softcore processor and PID coprocessor, while still having a majority of its resources free to implement additional functionality.

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. Gupte and Jones [2009] give further details on some of the specifics of this platform. As indicated in Section 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.

4. ARCHITECTURE OVERVIEW

This section describes the architecture of the time-multiplexed hardware-implemented PID controller, and a hardware-implemented Sensor Processing Unit (SPU).

4.1. Hardware-Implemented Context-Switching PID Controller

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 bottleneck to performance (see Section 6), and interrupts occurring during PID execution will add nondeterminism to the overall control loop. In addition, as stated earlier, the PID controller will have further sources of nondeterminism while competing for CPU time with general-purpose nonperiodic system tasks.

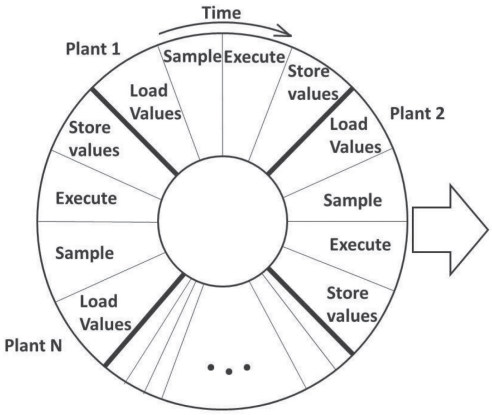
In our design we take advantage of the PIDs' repetitive Sample, Compute, Write actions to develop a resource-efficient architecture that addresses the issues of context-switching overhead, nondeterminism, and competing with nonperiod tasks, when implemented on a general-purpose CPU. Our architecture acts as a coprocessor to the CPU, only relying on the CPU to configure and update each PID that is required for use in the system. Once configured, the PID coprocessor executes each PID in a round-robin fashion, as depicted in Figure 3(a). The the memory and logic resources allocated for the coprocessor completely isolate the timing of its Sample, Compute, Write cycle from the general-purpose CPU's behavior. Figure 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 supported 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.

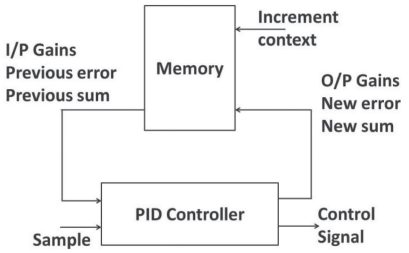
$$N \leq \frac{\tau_{min\_service}}{\tau_{context\_switch} + \tau_{compute}} \tag{3}$$

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 10us, then our architecture can service up to 500 plants (10us/20ns). If a 1ms plant is the fastest, then the architecture can support up to 50,000 plants. As the speed of the fastest plant decreases, memory storage becomes the limiting factor. However, supporting 100's of plants easily satisfies the needs of most systems.

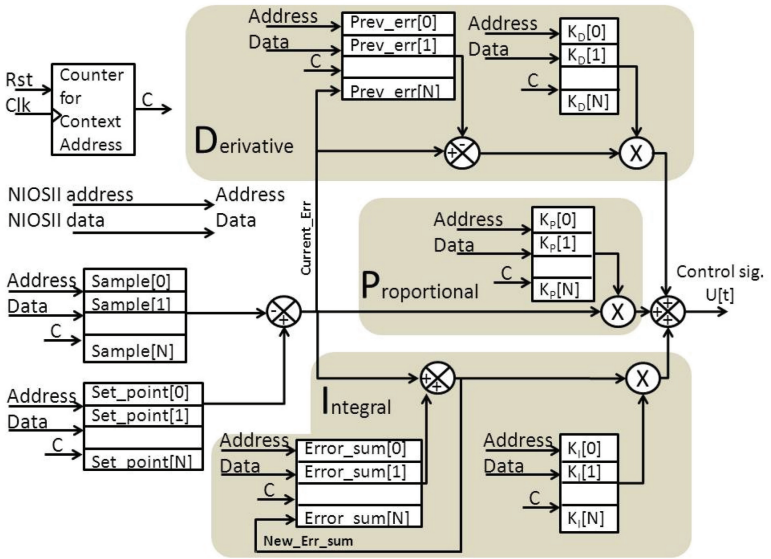
4.1.2. Detailed Architecture. Figure 3(c) illustrates the detailed architecture of the PID execution unit. This architecture represents the custom hardware that is integrated into the NIOS-II; see Figure 4. Except for configuring scaling constants, this execution unit runs independently of the software running on the softcore NIOS-II processor,



(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

Fig. 3. A conceptual to detailed architectural illustration of the hardware-implemented context-switching PID controller.

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.

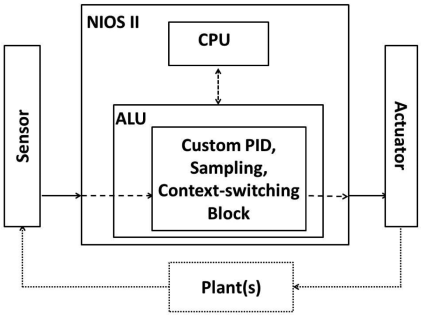


Fig. 4. Hardware-implemented time-multiplexed PID unit integrated into the NIOS processor's ALU using user-defined instructions.

In the upper left-hand corner of Figure 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.

4.2. Sensor Processing Unit (SPU)

4.2.1. Overview. The SPU is designed to be a functional unit within an embedded processor, as shown in Figure 5. It has two main purposes: (1) to efficiently off-load 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.

Sensor processing off-load. 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

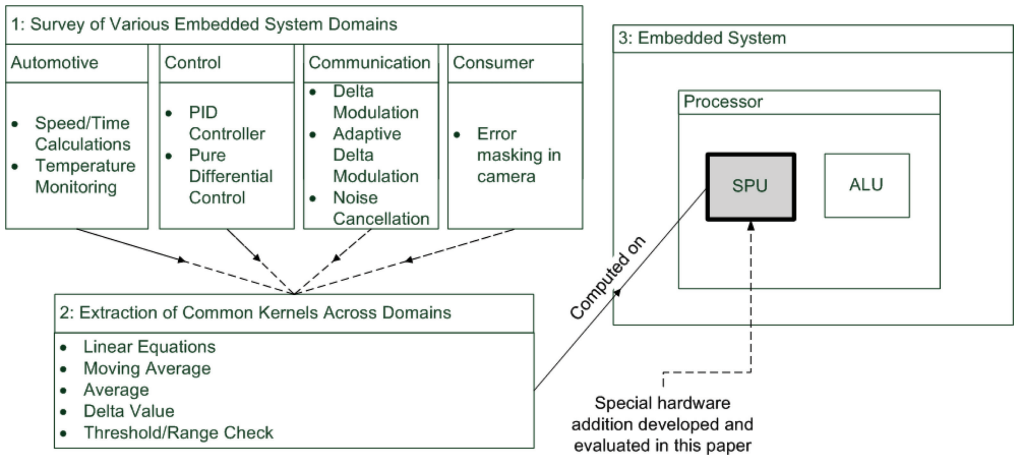


Fig. 5. (1) illustrates some high-level sensor-dependent tasks identified within various embedded domains, (2) highlights common kernels of computation that were identified, (3) shows conceptually how our Sensor Processing Unit (SPU) integrates into an embedded system.

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 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 [Intel 2010], the SPU is a lightweight means to generalize the types of events to which 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 [Intel 2006]. 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.

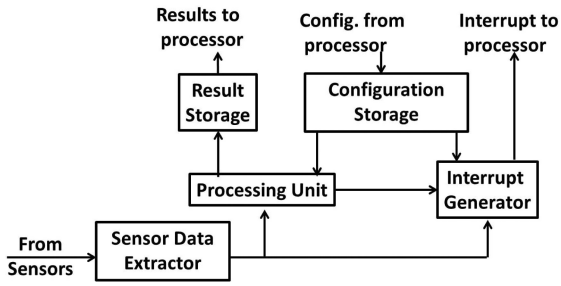


Fig. 6. Architecture of the SPU.

```
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.
```

4.2.2. Architecture. Figure 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.

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

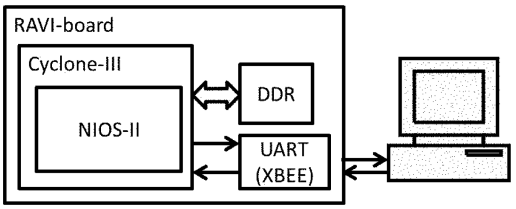


Fig. 7. High-level depiction of experimental setup.

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 execute 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 criterion 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.

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 7.

The FPGA allows all tests to run on a common processor, NIOS-II, and enables implementing tightly processor-coupled hardware blocks. It should be emphasized that while our FPGA infrastructure is well suited for quantifying software/hardware trade-offs, 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.

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 as follows.

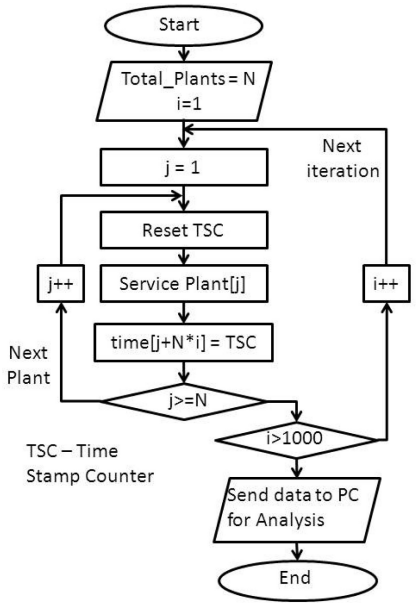


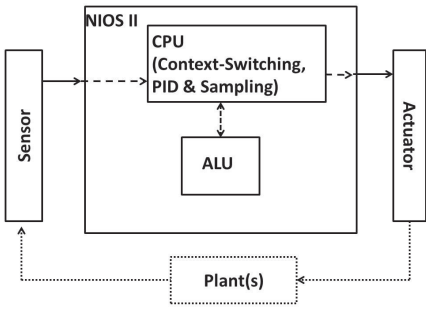
Fig. 8. Test flow for measuring the response time of each architecture.

(1) the architecture (Case 1, 2, 3, 4);
(2) the number of plants controlled (10, 100, 1000);
(3) the processor interrupt timer (1ms, 100ms);
(4) the sensor sampling rate, in Samples Per Second (SPS) (No Delay, 200KSPS, 819SPS);
(5) software-implemented jitter compensation (used or not used) [Wittenmark et al. 2003].

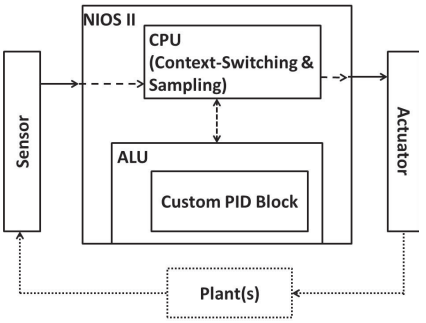
For profiling the system, software and hardware elements were added to take measurements. One advantage of hardware-based monitoring is it does not interfere with the timing of the system under test. For our evaluation a hardware-based Time-Stamp Counter (TSC) was implemented to accurately measure plant response time. Figure 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 collected data is transmitted to a standard PC for analysis. This procedure was run for each configuration evaluated. Section 6.1 discusses the results of these experiments.

Reference Architectures. We created three reference implementations to help clarify the speedup attained from our proposed approach (Case IV, Figure 9(d)). As we move from Case I to Case IV, larger portions of the PID control loop are migrated to hardware. A description of the four cases follows.

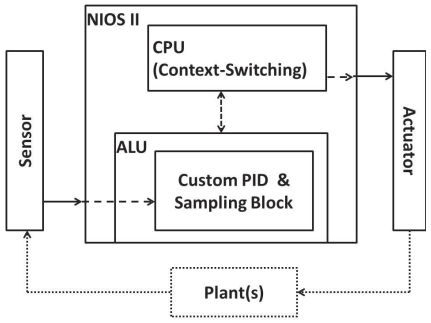
—*Case I.* Case I is a fully-software-implemented solution on the NIOS softcore processor (Figure 9(a)). This setup uses the system's standard software interface for sampling sensor data. This case's performance is expected to suffer high bus latencies and having to execute the PID functionality in software.



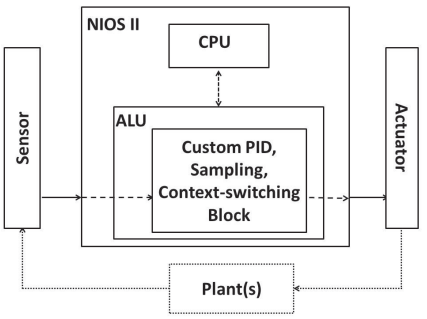
(a) Case I: software-implemented PID controllers



(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)

Fig. 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 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 high bus latency (Figure 9(b)).
—*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 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 through III (Figure 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 fire. This allows us to isolate jitter caused by interrupts from

jitter associated with the nature of an implementation approach (i.e., Cases I through IV). We examine the impact of setting the interrupt period to be 1ms versus 100ms. 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 Wittenmark et al. [2003]. The pseudocode 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 5us, and the current servicing of the plant finishes in 3us, then the processor will wait for another 2us. This gives the plant an effective service time of 5us, 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.
```

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 pseudosensor. Evaluation experiments were performed for the following three setups.

(1) *PLB.* A pseudosensor 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 proposed 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. 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 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

Fig. 10. Summary of response time across Cases I through 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

Fig. 11. Illustration of movement of functionality from software to hardware.

(1) *Execution Time.* This is the amount of time required to complete a sensor processing operation.
(2) *Code Density.* This is the size of the compiled programs.
(3) *Response Latency.* This is the latency between an event occurring and the processor responding (used for the interrupt generation feature of the SPU).

6. RESULTS AND ANALYSIS

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

6.1. Hardware-Based Context-Switching PID

Figure 10 summarizes the average plant response time, in tabular form, when sensor delay and the number of plants are varied. Figure 11 shows how functionality is migrated from software to hardware when moving from Case I to Case IV.

Figures 12 through 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 noncompensation 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

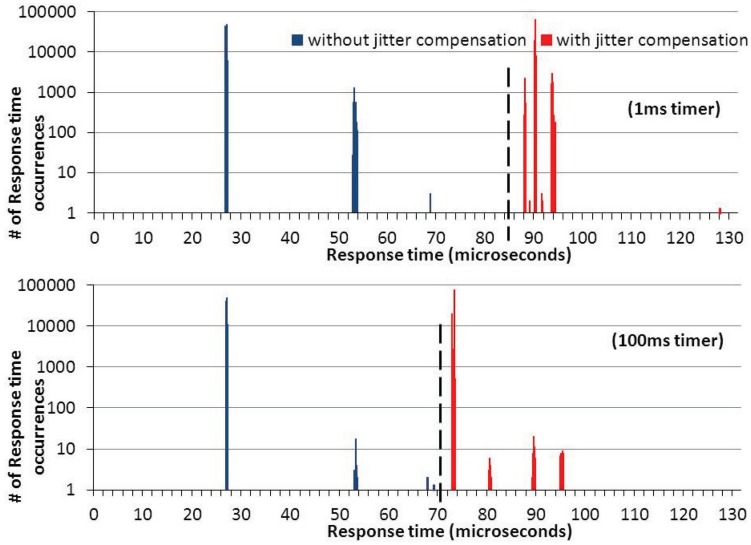


Fig. 12. Response time and jitter: This plot shows the number of times plants experienced a given amount of response time. The distribution of these response times is a measure of how much jitter a given system configuration experienced. The parameter settings for this experiment were: # of plants = 100, sample rate = No Delay, architecture = Case I.

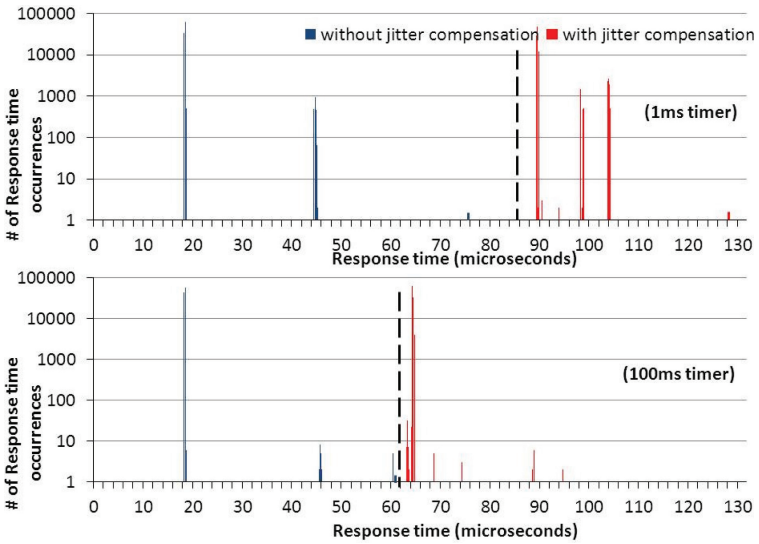


Fig. 13. Response time and jitter: This plot shows the number of times plants experienced a given amount of response time. The distribution of these response times is a measure of how much jitter a given system configuration experienced. The parameter settings for this experiment were: # of plants = 100, sample rate = No Delay, architecture = Case II.

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.

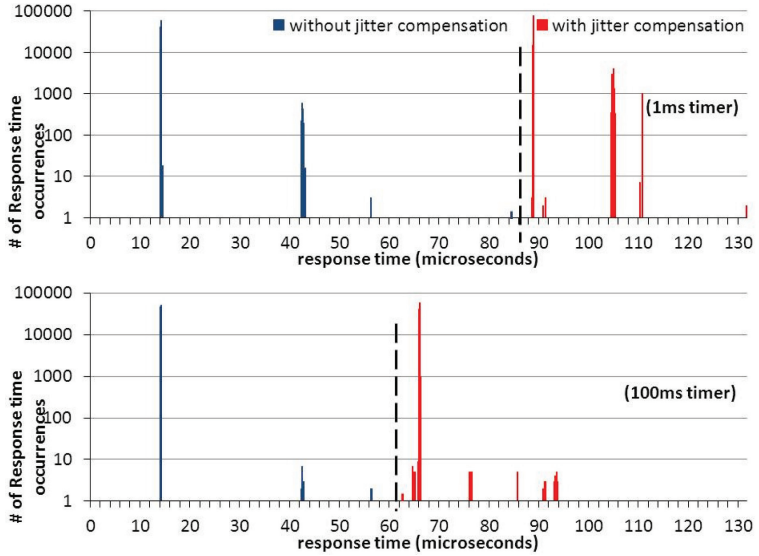


Fig. 14. Response time and jitter: This plot shows the number of times plants experienced a given amount of response time. The distribution of these response times is a measure of how much jitter a given system configuration experienced. The parameter settings for this experiment were: # of plants = 100, sample rate = No Delay, architecture = Case III.

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.

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 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 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 from Case III to Case IV. This indicates the vast majority of the main processor's time is spent 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

	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)

Fig. 15. Table showing the device utilization of our context-switched PID system.

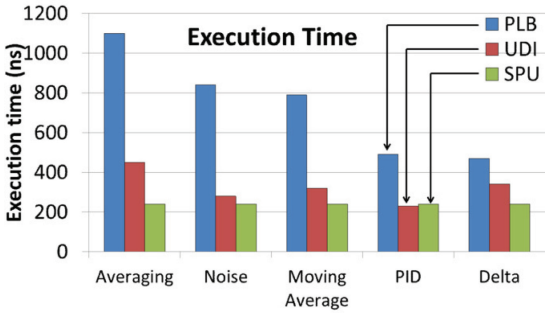


Fig. 16. Comparing the execution time of various common kernels on all three experimental setups. As seen here, the SPU is typically fastest for most kernels.

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 (see Figures 12 through 14). Digital control theory assumes the response time and sampling time are periodic, thus unbounded jitter can cause serious instability issues [Wittenmark et al. 2003]. Using jitter compensation methods helps to 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 15, the resource utilization of the PID coprocessor is negligible with respect to the NIOS-II processor. Even the combination of the two utilizes 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.

6.2. Sensor Processing Unit

This section presents the results of the SPU experimental setup described earlier in Section 5.2.

Execution Time. As can be seen from Figure 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 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 increase. Across all cases, the average speedup obtained using the SPU is 2.48fold faster than the PLB setup. Compared to the UDI setup, the SPU shows an average speedup factor of 1.38. This shows the speedups obtained over the PLB setup are not solely due to the SPU using the faster coprocessor interface.

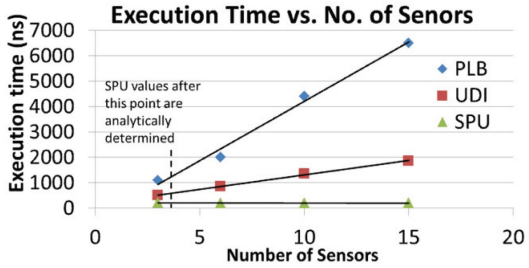


Fig. 17. The effect of increasing the number of sensors on the execution time of the "Average" kernel. As seen here, regardless of the number of sensors, the time taken for reading the result from the SPU remains constant.

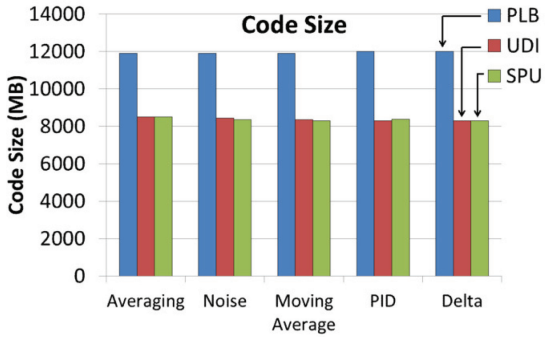


Fig. 18. The executable binary size for various kernels for all three experimental setups. As seen here, the SPU case is the smallest in most cases.

Code Density. As shown in Figure 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 use the SPU are 68.6% smaller than programs that use the PLB. The difference in code size between the SPU and UDI cases is 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 clearly seen in Figure 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 interrupt handler. However, we have considered only the simplest of cases wherein the PLB and UDI setups continuously poll 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 Zhou [2006]) would be beneficial for such use cases.

Resource utilization. Figure 20 shows that the resource utilization of the SPU is well under 1% of the resources available on the Virtex5 FPGA (FX70).

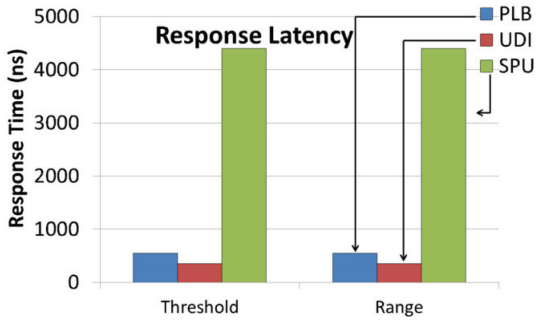


Fig. 19. Comparing the response across setups. The PLB and UDI setups use polling, and the SPU generates interrupts. Although the SPU case has a large latency, off-loading event monitoring to the SPU could relieve the main processor of a major computing burden when monitoring rare events for a high sample rate sensor.

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

Fig. 20. Table showing the device utilization of our sensor processing unit.

7. CONCLUSION AND 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 softcore-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 nonreal-time tasks to execute on the softcore 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 off-loading common sensor processing kernels of computation.

Future directions for this work are: (1) deploying the Linux operating system onto the softcore 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, (4) extending our SPU to support more advanced sensor processing tasks (e.g., sensor fusion using Kalman filtering), (5) potentially making these modules dynamically swappable at runtime, (6) implementing more advanced in hardware scheduling schemes (e.g., rate-monotonic, earliest-deadline-first) for scheduling plant servicing, and (7) using the SPU to condition sensor data and forward results directly to the PID coprocessor.

REFERENCES

ASTROM, K. J. 2006. Challenges in control education. In *Proceedings of the 7th IFAC Symposium on Advances in Control Education (ACE'06).*

CARVALHO, B., BATISTA, A., CORREIA, M., NETO, A., FERNANDES, H., GONÇALVES, B., AND SOUSA, J. 2010. Reconfigurable atca hardware for plasma control and data acquisition. *Fusion Engin. Des. 85,* 3–4, 298–302.

CERVIN, A., LINCOLN, B., EKER, J., ARZEN, K.-E., AND BUTTAZZO, G. 2004. The jitter margin and its application in the design of real-time control systems [elektronisk resurs]. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications*.

CHAN, Y., MOALLEM, M., AND WANG, W. 2004. Efficient implementation of pid control algorithm using fpga technology. In *Proceedings of the 43rd IEEE Conference on Decision and Control (CDC'04)*. Vol. 5, 4885–4890.

CHAN, Y. F., MOALLEM, M., AND WANG, W. 2007. Design and implementation of modular fpga-based pid controllers. *IEEE Trans. Indust. Electron. 54,* 4, 1898–1906.

ECONOMAKOS, C. AND ECONOMAKOS, G. 2008. Fpga implementation of plc programs using automated high-level synthesis tools. In *Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE'08)*. 1908–1913.

EEMBC. 2008. Embedded microprocessor benchmark consortium (EEMBC) homepage. http://www.eembc.org.

FRANTZ, G. 2000. Digital signal processor trends. *IEEE Micro 20,* 6, 52–59.

GANERIWAL, S., HAN, C., B., M., AND SRIVASTAVA. 2003. Going beyond nodal aggregates: Spatial average of a continuous physical process in sensor networks. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (Sensys'03)*, Poster.

GOEBEL, K. AND YAN, W. 2008. Correcting sensor drift and intermittency faults with data fusion and automated learning. *IEEE Syst. J. 2,* 2, 189–197.

GONÇALVES, B., SOUSA, J., AND VARANDAS, C. 2010. Real-time control of fusion reactors. *Energy Convers. Manag. 51,* 9, 1751–1757.

GU, L., JIA, D., VICAIRE, P., YAN, T., LUO, L., TIRUMALA, A., CAO, Q., HE, T., STANKOVIC, J. A., ABDELZAHER, T., AND KROGH, B. H. 2005. Lightweight detection and classification for wireless sensor networks in realistic environments. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SenSys'05)*. 205–217.

GUPTE, A. AND JONES, P. H. 2009. Towards hardware support for common sensor processing tasks. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'09)*. 85–90.

GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th IEEE Annual Workshop on Workload Characterization.*

GWALTNEY, D. A., KING, K. D., AND SMITH, K. J. 2002. Implementation of adaptive digital controllers on programmable logic devices. http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20020094342_2002153594.pdf.

HELLERSTEIN, J., HONG, W., MADDEN, S., AND STANEK, K. 2003. Beyond average: Towards sophisticated sensing with queries. In *Proceedings of the 2nd International Conference on Information Processing in Sensor Networks (IPSN'03)*. 63–79.

HENRIKSSON, D., REDELL, O., EL-KHOURY, J., TÖRNGREN, M., AND ARZEN, K.-E. 2005. Tools for real-time control systems co-design—A survey. Tech. rep. ISRN LUTFD2/TFRT- 7612- SE. Department of Automatic Control, Lund Institute of Technology, Sweden.

INTEL INC. 2006. Intel Centrino mobile technology, Wake on wireless LAN feature. Intel Inc.

INTEL INC. 2010. Intel Atom Processor Z5XX series datasheet, section 5.1.2 Intel thermal monitor. Intel Inc.

ISAKSSON, A. AND HAGGLUND, T. 2002. Editorial. *IEE Proc. Control Theory Appl. 149*, 1, 1–2.

JEFFERY, S. R., ALONSO, G., FRANKLIN, M. J., HONG, W., AND WIDOM, J. 2006. Declarative support for sensor data cleaning. In *Proceedings of the 4th International Conference on Pervasive Computing (PERVASIVE'06)*. 83–100.

LIU, K., CHEN, L., LIU, Y., AND LI, M. 2008. Robust and efficient aggregate query processing in wireless sensor networks. *Mobile Netw. Appl. 13*, 2, 212–227.

MIC, D., ONIGA, S., MICU, E., AND LUNG, C. 2008. Complete hardware/software solution for implementing the control of the electrical machines with programmable logic circuits. In *Proceedings of the 11th International Conference on Optimization of Electrical and Electronic Equipment (OPTIM'08)*. 107–114.

MINORSKY. 1922. Directional stability of automatically steered bodies. *J. Amer. Soc. Naval Engin. 34,* 2, 280–309.

NAKAMURA, M., SAKURAI, A., WATANAB, T., NAKAMURA, J., AND BAN, H. 2006. Improved collaborative environment control using mote-based sensor/actuator networks. In *Proceedings of the Conference on Local Computer Networks*.

NATIONAL INSTRUMENTS. 2011. Pacs for industrial control, the future of control. White paper, National Instruments. http://www.ni.com/white-paper/3755/en/.

NEKOOGAR, F. AND MORIARTY, G. 1998. *Digital Control Using Digital Signal Processing* 1st Ed. Prentice-Hall, Upper Saddle River, NJ.

O'DWYER, A. 2003. Pid compensation of time delayed processes 1998–2002: A survey. In *Proceedings of the American Control Conference.* 1494–1499.

OGATA, K. 2002. *Modern Control Engineering* 4th Ed. Prentice-Hall, Upper Saddle River, NJ.

PELLIZZONI, R., MEREDITH, P., CACCAMO, M., AND ROSU, G. 2008. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Proceedings of the Real-Time Systems Symposium.* 481–491.

PENAFLOR, B., FERRON, J., PIGLOWSKI, D., JOHNSON, R., AND WALKER, M. 2006. Real-time data acquisition and feedback control using linux intel computers. In *Proceedings of the 5th IAEA Technical Meeting on Control, Data Acquisition, and Remote Participation for Fusion Research.*

PETROV, M., GANCHEV, I., AND TANEVA, A. 2002. Fuzzy pid control of nonlinear plants. In *Proceedings of the Intelligent Systems Conference*.

SAMET, L., MASMOUDI, N., KHARRAT, M., AND KAMOUN, L. 1998. A digital pid controller for real time and multi loop control: A comparative study. In *Proceedings of the IEEE International Conference on Electronics, Circuits and Systems.* Vol. 1, 291–296.

SARTORI, F., BUDD, T., CARD, P., FELTON, R., LOMAS, P., MCCULLEN, P., PICCOLO, F., ZABEO, L., ALBANESE, R., AMBROSINO, G., DE TOMMASI, G., AND PIRONTI, A. 2009. Jet operations and plasma control: A plasma control system that is safe and flexible in a manageable way. In *Proceedings of the 23rd IEEE/NPSS Symposium on Fusion Engineering (SOFE'09)*. 1–6.

SUH, Y. S. 2007. Send-on-delta sensor data transmission with a linear predictor. *MDPI Sensors 7,* 4, 537–547.

TANG, K. S., MAN, K. F., CHEN, G., AND KWONG, S. 2001. An optimal fuzzy pid controller. *IEEE Trans. Indust. Electron. 48*, 4.

TÖRNGREN, M. 1998. Fundamentals of implementing real-time control applications in distributed computer systems. *J. Real-Time Syst. 14*, 219–250.

VISIOLI, A. 2001. Tuning of pid controllers with fuzzy logic. *IEE Proc. Control Theory Appl. 148*, 1, 1–8.

WITTENMARK, B., ASTRM, K. J., AND ARZEN, K.-E. 2003. Computer control: An overview. Tech. rep. http://cepac.cheme.cmu.edu/pasilectures/crisalle/%5BWiAA%5D%20Computer_Control.pdf.

ZHAO, W., KIM, B. H., LARSON, A. C., AND VOYLES, R. M. 2005. Fpga implementation of closed-loop control system for small-scale robot. In *Proceedings of the International Conference on Advanced Robotics*. 70.

ZHOU, X. AND PETROV, P. 2006. Rapid and low-cost context-switch through embedded processor customization for real-time and control applications. In *Proceedings of the Design Automation Conference*.

Received March 2011; revised February 2012; accepted May 2012