

HW/SW Codesign of a Multiple Injection Driver Automotive Subsystem Using a Configurable System-on-Chip

Massimo Baleani^{†‡}

Massimo Conti[†]

Alberto Ferrari[‡]

Alberto Sangiovanni-Vincentelli^{‡§}

[‡]PARADES EEIG
Via San Pantaleo, 66
00186 Rome, Italy

{mbaleani,aferrari}@parades.rm.cnr.it

[†]Department of Electronics
University of Ancona
60131 Ancona, Italy

max@ea.unian.it

[§]Department of EECS
University of California
Berkeley, CA 94709

alberto@eecs.berkeley.edu

Abstract

The increasing complexity of embedded systems and demands for quicker turn-around times require reuse of hardware and software components. Reconfigurable hardware technology opens a new implementation space where software and hardware design cycles might be very close in time and where a broader range of applications can be mapped on. Using a hw/sw co-design methodology quick partitioning of functionality between hardware and software can be obtained leading to very fast system design cycles. In this paper we present the use of a co-design flow and a configurable system-on-chip, whose most important components are a micro-processor and an eFPGA, for an automotive application: the management of electro-mechanical injectors for gasoline and diesel direct injection engines.

1. Introduction

In the design of hardware platforms for embedded systems, the application domain, cost and time-to-market determine whether the hardware architecture must be tailored to the specific problem or if, and to what extent, software programmable and software reconfigurable components should be used. The use of software reconfigurable hardware has been common in rapid prototyping and the introduction of FPGA by Xilinx in the mid 80's spurred a lot of research efforts in the development of reconfigurable FPGA-based systems (a good survey can be found in [9]).

Today, reconfigurable hardware is gaining ground even in final implementations [10], especially in cases where new functionality may be added during the life cycle of the product or where in field error correction has to be provided, and in the past couple of years, there has been a serious attempt at developing platforms that combine micro-processors and reconfigurable logic on the same chip [12, 13, 1, 2]. The

resulting platforms gain efficiency in terms of speed and power consumption (Rabaey has recently reported two orders of magnitude difference in power consumption between a full software implementation and mixed software-reconfigurable hardware one [8]) without giving up much in terms of flexibility. Reconfigurable hardware gives back designers their ability to add value and differentiate systems by post-fabrication selection and integration of components: it was typically done on-board and jeopardized by the advent of system-on-silicon technology.

Two main approaches can be taken to exploit the hardware flexibility of FPGA. The first uses FPGA as hardware accelerator for computing and can be re-configured for different computations during run-time. In the second, on which we focus in this work, the programmable logic is used to customize the integrated circuit, e.g. the CPU core and the I/O sub-system or just the I/O sub-system, for particular applications (Configurable System-on-a-Chip).

An important application domain for platform-based design is automotive electronics, and, in particular, engine and power-train control, where the need of updating functionalities and accommodating last-minute engineering changes in presence of strong real-time constraints is particularly severe. The real-time constraints pose strong demands on both computational resources and I/O subsystems. The common solution to the design problem is to implement computation in software, move as much as possible of the I/O operations to software and, in case the real-time constraints are not met, design custom hardware blocks. In general, the decision on what goes into software and what necessitates hardware components is made based on intuition and experience, often resulting in suboptimal (to say the least) implementations and reduced flexibility. Indeed, moving back and forth from hardware to software is a major endeavor requiring long manual design processes.

The embedded FPGA (eFPGA) architecture is particularly appealing for this application domain, where I/O re-

quirements are key in choosing the appropriate architecture or even in designing a brand new one [6]: it allows moving quickly part of the I/O subsystem from software to hardware and vice-versa without requiring expensive hardware redesign to explore different partitions.

A limiting factor in exploiting the power of reconfigurable platforms is the lack of a unified design flow that could easily move functionalities from software to the eFPGAs and back. For fully exploiting this flexibility in short time and with high degree of reliability, appropriate methodologies and tools for system specification, architecture selection, IP integration and automatic synthesis are essential. At this time, moving software, typically written in C, to hardware, typically written in some hardware description language (HDL), and vice versa is a long and tedious process hampered by the lack of a common abstraction.

POLIS [3] is a readily available open-domain environment that supports a hardware-software neutral model of computation, enables a fast design space exploration driven by performance estimation, and provides tools to synthesize from this common abstraction to either hardware or software. However, it presents limitations in the synthesis of hardware and interfaces. These limitations are due to the fact that POLIS back end was conceived for rapid prototyping and on-board system integration and so it cannot be used “as is” to target CSOCs. Our original contributions are the specification, validation, design space exploration, automatic synthesis, and implementation of an industrial automotive application onto a CSOC, using an integrated design environment that has been developed by modifying the basic framework offered by POLIS to target reconfigurable platforms.

This paper is organized as follows. In Section 2 the description of the design problem, a multiple injection driver subsystem, is presented. In Section 3, a quick overview of our methodology and tools is introduced. In Section 4, the design process is presented. In Section 5, experimental results are reported.

2. The Application

The cycle of a 4-stroke endothermic direct injection engine consists of four phases. *Intake*: the piston moves downward from the top dead center (TDC) to the bottom dead center (BDC) and air flows from the manifold to the combustion chamber. *Compression*: the piston moves upward from the BDC to the TDC compressing the air/fuel mixture. *Expansion*: due to the combustion of the mixture, the piston moves from the TDC to the BDC and torque is generated. *Exhaust*: the piston moves from the BDC to the TDC, expelling the exhausted gas from the combustion chamber. In modern gasoline or diesel direct injection engines, the fuel is injected multiple times during the en-

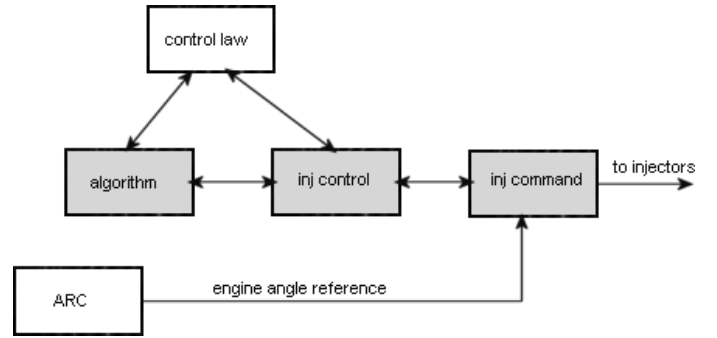


Figure 1. Multi-injection system overview

gine cycle, in order to optimize the output torque minimizing pollution and consumption under any driving condition. Each of these injections is called stroke and can occur in almost any phase of the cycle.

An engine control unit (ECU) that electrically controls the engine, computes the correct amount of fuel, the number of strokes, the position of the strokes in the engine cycles and opens the injectors for the right amount of time. Hence, the ECU must drive the injectors (one for each cylinder) to supply the fuel quantity with a multi-injection profile and perfectly synchronized with the engine position¹. A functional overview of the system is given in Figure 1. The grayed blocks represent the 3 main modules in which the driver can be decomposed. The control law that uses the multiple-injection driver as an actuator, provides, at each exhausted TDC, the opening angle and the fuel quantity of each stroke. The driver computes the injector opening times, executes the injection strokes and provides back to the control law the quantity of fuel actually injected. The injector opening time, computed by the algorithm, depends on the requested fuel quantity, rail pressure and fuel viscosity (i.e. fuel temperature). A maximum of 5 fuel strokes can be performed for each cylinder in a single engine cycle. Each injection stroke IS_i can occur only in a parameterized range of the engine angle: $[\theta^i_{open}, \theta^i_{close}]$. The engine angle reference is provided by the angular clock generator (ARC) that translates signals from the crankshaft and the camshaft sensors into the engine angular position. The control module supervises the generation of injection commands ensuring that all the strokes are performed and that injectors are opened after the opening angle and closed before the closure angle.

The driver must satisfy tight precision constraints: the injection angle requires a precision and resolution of ± 0.2 degrees and the injected quantity, if not cut for angle

¹Engine position can be represented by an angular reference with a 720° period, the *engine angle*, identifying both the angular position of the crankshaft and the working phase of each cylinder.

constraints, requires a resolution/precision of 0.1 mg and stroke-to-stroke dispersion of 0.2 mg. Moreover, the driver must be capable of managing up to two simultaneous injections into two different cylinders at a maximum engine speed of 8000 rpm.

3. Design Methodology

In the POLIS co-design environment, a homogeneous behavioral representation is used to model hardware and software components. This common representation is based on the Co-design Finite State Machine (CFSM) model of computation [3] and several languages with underlying CFSM semantics, such as ESTEREL [5] or State Transition Diagrams, can be used to capture the behavior. System level functional simulation, performed in the PTOLEMY simulation environment [4], allows the designer to validate the functionality of the system without accounting for any implementation detail (e.g. hardware-software partition).

At this stage of the design flow, designers can explore the design space mapping behavioral functions onto architectural resources. Hardware-software partitioning (including *functional restructuring* as explained in Section 4.3), scheduling policy selection and communication refinement all take place at this point. All these steps and design decisions are manual in the proposed framework but are no more a result of designers' experience or intuition: performance evaluation can be carried out by simulating system behavior with an abstract timing model of the target architecture and it drives the designer to take the correct decisions. CFSMs mapped to hardware (hw-CFSMs) are supposed to execute in a single clock cycle (this assumption has to be verified with a later timing analysis). Performance estimates of CFSMs mapped to software (sw-CFSMs) are derived using the technique presented in [11]. The number of cycles is estimated on the basis of a set of parameters representing the execution time of basic C constructs (e.g. *if...then, case, assignments, logic/arithmetic operations*) for the selected software domain (i.e. the selected target processor and compiler). These C statements are the ones used by the POLIS automatic software generator to translate sw-CFSMs' description into a C file that can be directly compiled into machine code for the target processor. The accuracy of performance simulation is determined by how well the effects of various hardware and software architectural characteristic, such as the Real Time Operating System (RTOS), buses, hardware-software communication mechanisms (interrupt, polling), peripherals, are modeled.

Even if the final target is reconfigurable hardware, we believe that the use of the co-design methodology enables a faster exploration of the solution space: it provides both hardware-software estimation techniques, which can be used to quickly gather information on the performance of

the target architecture without committing to any actual implementation, and a hardware-software automatic synthesis path, which speeds up the implementation stage. During the exploration phase, CPU load and hardware complexity (e.g. number of latches) are estimated and only when they reach an acceptable level the implementation phase starts.

The automatic synthesis of hardware and of interfaces between the hardware and software partitions and their implementation onto the CSoC represent our main contribution from the method/tool perspective. POLIS automatically generates C code for each sw-CFSM and C code for a real-time OS, which is responsible for task scheduling and provides primitives for software-software, software-hardware and hardware-software communications. Several paths already exist for the synthesis of hw-CFSMs and automatic interface synthesis is also provided, to implement correctly the event-based asynchronous communication mechanism across implementation domains. However, we cannot leverage the entire synthesis flow "as is" as we are targeting CSoCs.

In POLIS, VHDL can be automatically generated from the intermediate CFSM format. It can be used for co-simulation, as shown in [7], but it is inefficient or even plainly not usable if used as a starting point for hardware synthesis. POLIS allows by-passing the generation of VHDL code going directly into the intermediate format supported by synthesis tools, such as the Berkeley Logic Intermediate Format (BLIF) or Xilinx XNF (the latter is used in [3] for rapid prototyping). Unfortunately, not all the tools used to map onto the CSoCs offered on the market support these formats, but all take a VHDL input. We have thus extended POLIS hardware synthesis capabilities to generate, from POLIS internal data representation, a VHDL description of the hardware partition that can be synthesized into the eFPGA, leaving the choice between Mealy-like (I/O unbuffered) and Moore-like (I/O buffered) CFSMs to the designer.

In a synthesis-based flow, the interfaces among the different components of the design should also be automatically (and efficiently) generated. Homogeneous interfaces do not pose a problem since software-to-software interfaces are implemented by the RTOS and so they are completely transparent, and hardware-to-hardware and hardware-to-environment communications are implemented just over a wire without the need for latching or other overhead.

On the other hand, hardware-to-software and vice versa, environment-to-software and vice versa, and finally environment-to-hardware, interfaces are more difficult to handle. POLIS uses tri-state buffers for the connections to the system bus and adds an overhead in terms of configurable logic usage. Since we aim at implementing it onto a single device, which usually provides native multiplexing and decoding logic to access the system bus (refer for ex-

ample to [12, 2]), we want to fully exploit the configurable architectural resources. Rather than synthesizing into the eFPGA the glue logic provided by the POLIS synthesis tool, we extract information on number, size and address of registers, which we use later to configure the built-in logic, while minimizing the use of eFPGA for interface implementation. Built-in logic configuration usually requires an interface to a proprietary CSoC development software. According to which kind of interface specification is provided (text or graphics), this process may or may not be completely automatized. Nevertheless, even in the latter case the extracted information eases the configuration task.

Moreover in CSoCs, hardware port address allocation might be delayed to the hardware configuration step. To accommodate this new degree of configurability, we support two main flows: top-down and bottom-up. In a top-down flow, the allocation of the ports for hardware-software communication is decided and fixed in the POLIS environment before interface synthesis. POLIS generates the procedures to access hardware resources for the RTOS and the information for the configuration environment: hardware ports instantiation and configuration. In a bottom-up flow, POLIS generates only symbolic reference for the communication among hardware and software. The hardware configuration tool allocates the port address and generates the binding between the symbolic ports and the physical addresses. The bottom-up flow allows the designer to change the configuration of the system later in the design flow without requiring any redesign.

After the configuration and compilation phases, both the executable and the bit-stream for the eFPGA are downloaded to the target platform and the debug phase might start.

4. Design Process

4.1. Functional Specification and Validation

Starting from an informal, natural-language specification of the system provided by the Power-train division of Magneti Marelli, we have described the multi-injection driver as a network of 12 CFSMs (for a 4-cylinder engine). Each CFSM is captured using ESTEREL, a synchronous reactive language used unbiasedly for both hw-CFSM and sw-CFSM specification, for a total of about 1000 lines of code.

The functional description has been validated by means of extensive simulation reproducing all the possible operating conditions for the system. Figure 2 shows an example of the complex injection profile required to the driver. It is a result of functional simulation in the PTOLEMY environment: as required by the specification, A to E are the five strokes performed for one cylinder in a single engine cycle as shown by the *TETA* diagram representing the engine

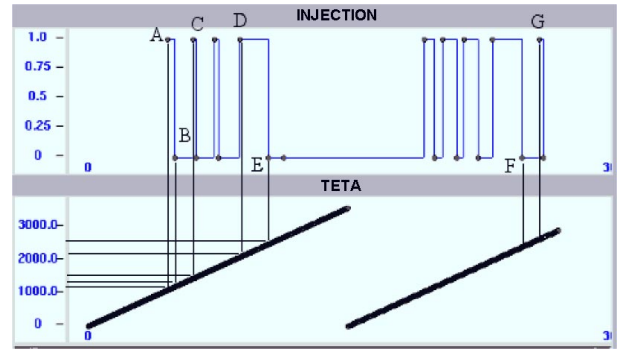


Figure 2. Required injection profile

angle with a resolution of $0.2^\circ/\text{div}$.

4.2. Architecture Selection and Characterization

Specifying correctly the delay values is the key point for achieving a good estimation of software execution time. About 1 man-day has been spent measuring, with an instruction set simulator, 90 benchmark functions for the accurate characterization of the 8032 “Turbo” microprocessor embedded into the Triscend TE520S40-40Q device [12] selected for the implementation of the driver.

The CSoC device from Triscend integrates, on a single chip, a performance-enhanced 8032 “Turbo” microcontroller running at 40MHz, a large block of SRAM, a high-speed dedicated system bus, and configurable logic, intimately connected to the processor and system bus. The embedded SRAM-based Configurable System Logic (CSL) matrix provides “derivative on demand” system customization. The configurable logic architecture consists of a highly interconnected matrix of CSL cells. Resources within the matrix provide easy, seamless access to and from the internal system bus. Each CSL cell performs various potential functions, including combinatorial and sequential logic.

4.3. Mapping on a CSoC Architecture

The complexity of the driver and the tight constraints it has to satisfy (at 8000 rpm roughly one event must be processed every $4\mu\text{s}$) make the choice of the correct hardware-software partition quite critical. This is the reason why we advocate the adoption of a design methodology allowing to evaluate different hardware-software trade-offs very quickly. Extensive architecture exploration has been carried out involving both hardware-software trade-offs and the evaluation of custom rather than standard, off-the-shelf hardware modules. The latter is one of the biggest advantages coming from the availability of configurable hardware, which we made even more effective combining

it with a fast, reliable, semi-automatic hardware generation/configuration flow.

A wide range of architectures, ranging from fully software to fully hardware implementations, have been considered. In the following, two of these experiments are presented in detail. Full-software, full-hardware implementations were immediately rejected. The full-software solution, as performance simulation pointed out, does not even comply with functional requirements while the full-hardware solution does not fit onto the 2048-CSL eFPGA.

The goal of the first experiment was to identify the right cut between hardware and software with the twofold objective of satisfying all the constraints imposed by the application while adopting only standard hardware modules. Both the control and data computation part of the driver have been selected for software implementation. We have manually decomposed the functional specification to an equivalent description “mappable” onto the chosen hardware and software partitions (*functional restructuring*). The mapped behavior consists of 24 CFSMs: 4 CFSMs implemented as interrupt service routines (ISR), 8 timers and 12 compare&match modules. The performance estimates provided by POLIS show an overall CPU load of about 20% at the maximum engine revolution speed of 8000 rpm. However, despite the relative low CPU load, performance simulation reveals that the correct execution of all the required injection strokes cannot be guaranteed when two of them are too close in time, as shown in Figure 3: boxes indicate missing strokes due to performance limitation of the hardware-software architecture. If we decided to move our application onto the costlier A7 CSoC family device featuring the very same eFPGA size but a high-performance ARM7TDMI core, then this solution would have been feasible.

In the second experiment, we further decomposed the functional behavior modifying the hardware and software partitions basically moving a significant part of the control module into hardware. This implies the use of custom hardware IPs mapped onto the CSL matrix. In this case, the mapped behavior consists of 4 ISRs, 4 timers, 12 compare&match modules and 8 custom hardware blocks. The estimated CPU load is about 12% and the system exhibits a correct behavior under any operating condition as assessed by the exhaustive simulations we run (see Figure 3).

4.4. System Implementation on Triscend CSoC

C code for sw-CFSM and a RTOS and VHDL code for the hardware partition have been automatically synthesized leveraging POLIS software (native) and hardware (added) synthesis tools. Interface synthesis, on the other hand, is not completely effortless. All the communications towards the

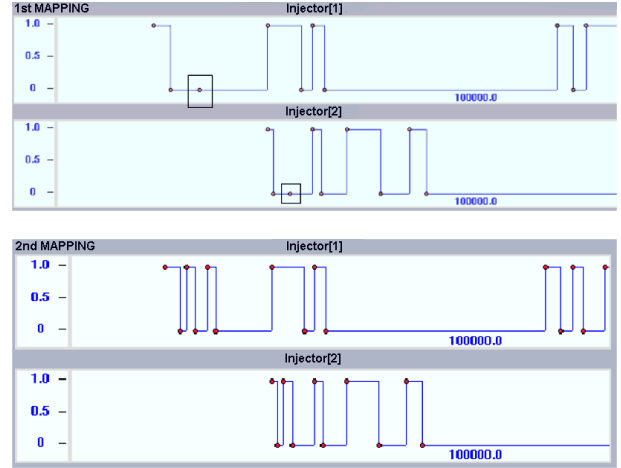


Figure 3. Injection profiles for the first (top) and second mapping (bottom)

software partition are handled by default by POLIS using a polling mechanism and a request-acknowledge protocol. In this case, due to the tight timing constraints on our application, we have to handle these communications as interrupt requests. More in details, we serve the interrupt requests according to a deferred interrupt scheme: each ISR enables all the tasks sensitive to the correspondent event. Enabled tasks are then scheduled according to the selected scheduling policy (round robin in our case). POLIS does synthesize C code for the ISR and so all the designer need to do manually is to install ISRs in the interrupt vector table and connect the interrupt request line to the appropriate microcontroller pin.

The interfaces to the hardware partition synthesized automatically by POLIS use an edge detector to translate pulses, which can potentially last more than one clock cycle, to the one cycle hardware protocol and employ registers to hold values. In addition, the necessary glue logic, i.e. registers, multiplexing and decoder logic, is generated to interface the hardware partition to the system bus. In our case, the hardware modules are built onto the CSL matrix that is provided with a native interface to the system bus. Thus we did not use the POLIS interface synthesis flow but we have automatically extracted from the POLIS internal data structure all the information on the required interfaces such as the number of registers, registers’ width, registers’ address, in case of early binding (top-down flow). Interfaces are actually added and configured providing all the information to Triscend FastChip software, which manages the entire CSoC design process, from CSoC programmable logic configuration to system level debugging. At present, this is done by means of a set of script files which read the

information we extract from POLIS and use the FastChip command-line interface to instantiate and configure off-the-shelf IPs, such as command and status registers, as well as imported user-IPs. For the Triscend platform, we needed to provide just a few hand-written lines of VHDL code to create a user-IP acting as edge detector.

5. Results and Concluding Remarks

In this paper we presented our experience in applying a design methodology based on hardware-software co-design to the implementation of an automotive industrial case study onto a CSoC. POLIS confirmed to be an efficient basis for the fast exploration of hardware-software trade-offs. However, it did have problems, for our chosen architecture and application, in the synthesis process for hardware and for interfaces. The case study has eventually led to the extension of POLIS hardware and interface synthesis flow, thus easing the design of CSoCs making it faster and more reliable. In fact, our approach allowed to evaluate a number of partitions in a few hours yielding what we believe to be an effective implementation of the functionality of the automotive subsystem.

The full design flow from system specification down to the final implementation took about 60 days accounting also for the extensive architecture exploration. The design was carried out by one person without requiring any specific skills in embedded software and hardware synthesis. Most important is the fact that combining the function-architecture co-design methodology supported by the (extended) POLIS framework with reconfigurable hardware, we were able to change the implementation of the system in a really short amount of time. We used less than an hour to move from a solution to another during architectural exploration, dropping all those alternatives which did not comply with our functional/performance requirements. The implementation design step, whose time is completely dominated by hardware synthesis, took no more than 8 hours to move from a first candidate solution to a second one.

Considering the relative complexity and tight real-time constraints of our application, we believe that the same approach can also be applied to other real-time control-oriented applications and to target other implementation platforms. Some problems for our ideal design flow remain open as it pertains to interface synthesis. In fact, in the present version of our environment, we still have some, albeit minimal, manual steps. We are currently working to make the interface synthesis flow completely automatic and to extend it to handle also specific CSoC communication resources. We believe that the resulting system will be able to make the use of CSoC quite appealing for a number of industrial designs.

6. Acknowledgments

This work has been partially funded by CNR-MADES II project, code 2.1.3. We wish to acknowledge the contributions of Marco Graciotti to the development of the case study, of Valerio Frascolla and Giampiero Spugni for the extension of the POLIS synthesis flow, Giorgio Bombarda from Magneti Marelli for the application example and support, Chris Balough from Triscend Corp. for his valuable support, Luciano Lavagno from Cadence Berkeley Labs for his suggestions and original contribution to POLIS.

References

- [1] Altera Corporation. Altera Excalibur embedded processor solutions. [On-line] <http://www.altera.com>.
- [2] Atmel Corporation. Atmel FPSLIC devices. [On-line] <http://www.atmel.com>.
- [3] F. Balarin, E. Sentovich, M. Chiodo, P. Giusto, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli. *Hardware-Software Co-design of Embedded Systems – The POLIS Approach*. Kluwer Academic Publishers, 1997.
- [4] J. Buck, S. Ha, E. Lee, and D. Masserschmitt. PTOLEMY: a framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, Special Issue on Simulation Software Development, January 1990.
- [5] The ESTEREL language. [On-line] <http://www.esterel.org>.
- [6] A. Ferrari, S. Garue, M. Peri, S. Pezzini, L. Valsecchi, F. Andretta, and W. Nesci. Design and implementation of a dual processor platform for power-train systems. In *Proceedings of Convergence Conference*, October 2000.
- [7] E. Filippi, L. Lavagno, L. Licciardi, A. Montanaro, M. Paolini, R. Passerone, M. Sgroi, and A. Sangiovanni-Vincentelli. Intellectual property re-use in embedded system co-design: An industrial case study. In *Proceedings of International Symposium on System Synthesis*, 1998.
- [8] V. George, H. Zhang, and J. Rabaey. The design of a low energy FPGA. In *International Symposium on Low Power Electronics and Design*, pages 188–193, August 1999.
- [9] R. Hartenstein. A decade of reconfigurable computing: A visionary perspective. In *Proceedings of the European Design, Automation and Test Conference (DATE)*, March 2000.
- [10] P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh. A quick safari in the reconfiguration jungle. In *Proceedings of the Design Automation Conference*, June 2001.
- [11] K. Suzuki and A. Sangiovanni-Vincentelli. Efficient software performance estimation methods for hardware/software codesign. In *Proceedings of the Design Automation Conference*, June 1996.
- [12] Triscend Corporation. Triscend E5 and A7 CSoC families. [On-line] <http://www.triscend.com>.
- [13] Xilinx Corporation. Virtex Platform FPGAs. [On-line] <http://www.xilinx.com>.