Towards Parallel Execution of IEC 61131 Industrial Cyber-Physical Systems Applications

Arquimedes Canedo and Mohammad Abdulah Al-Faruque
Siemens Corporate Research, Siemens Corporation, Princeton, USA
Email: {arquimedes.canedo, mohammad.al-faruque}@siemens.com

***Abstract*—In industrial cyber-physical systems (CPS)[1], the ability of a system to react quicker to its inputs by just a few milliseconds can be translated to billions of dollars in additional profit over just a few years of uninterrupted operation. Therefore, it is important to reduce the cycle time of industrial CPS applications not only for the economical benefits but also for waste minimization, energy reduction, and safer working environments. In this paper, we present a novel method to reduce the execution time of CPS applications through a holistic software/hardware method that enables automatic parallelization of standardized industrial automation languages and their execution in multi-core processors. Through a realistic CPS, we demonstrate that parallel execution reduces the cycle time of the application and increases the life-cycle through better utilization of the mechanical, electrical, and computing resources.**

I. Introduction

Industrial automation refers to the use of embedded software for the coordination of high-volume and high-precision manufacturing, food, pharmaceutical, chemical, energy, and mobility industries [1], [2]. These physical processes are typically controlled by industrial CPS comprising of various embedded computing devices capable of sensing, planning complex processes, and actuating thousands of times per second through high-speed cameras, light sensors, collision avoidance and detection, robotic devices, motors, etc. These CPS not only must comply with hard real-time requirements, but also must be able to survive in extreme environments of temperature, pressure, vibration, and humidity and remain operable for decades without interruption or failure [1]. The cost of downtime for the energy, manufacturing, food processing, and other industries is estimated to be of over one million dollars per hour [3]. Therefore, the main objective of industrial CPS is to operate as fast, as accurate, as reliable, as safe, and as cost effective as possible through the selection of suitable embedded **software** and **hardware** technology [1].

Throughout the years, embedded software for industrial CPS has been developed by non-computer experts using domain-specific languages that have been designed and refined by experienced practitioners, manufacturers of automation hardware and software, and independent institutions from different industry sectors. The IEC 61131-3 standard [4] has been widely adopted as the programming standard for industrial CPS [5] since 1993 but its languages have been used in the industry since the early 1970's [6]. It provides a total of 5 different languages: 2 textual (Instruction List or IL, Structured Text or ST), 2 graphical (Ladder Diagram or LD, Function Block Diagram or FBD), and 1 with both textual and graphical representations (Sequence Function Chart or SFC). Different industry sectors use different languages or the combination of them simply because each language has special semantics that facilitate certain automation tasks. These programming languages have been designed to satisfy the needs and increase the productivity of non-computer experts such as electrical, mechanical, and chemical engineers [6]. For example, LD programs strongly resemble relay logic diagrams used by electrical engineers to design circuits. Interestingly, engineers on the same domain but in different countries seem to prefer different languages [7].

Finding the appropriate embedded hardware capable of delivering the required performance for industrial CPS tasks is also very critical. For some applications, custom architectures in ASIC and FPGA are developed because general purpose processors or digital-signal processors are not capable of meeting the performance requirements [8], [9]. For example, high-performance motion control for machine tools (CNC) use specialized hardware and software that is not compatible with the IEC 61131-3 standard. While these custom systems provide the required performance, they are very expensive to buy, maintain, and modify. Flexibility is one of the most important features in industrial CPS because the production requirements change significantly between different products, or different generations of the same product. Therefore, there is an economical and technical motivation to shift from rigid custom architectures and programming languages into flexible off-the-shelf architectures and standardized automation languages. The adoption of multi-core processors appears to be the next evolutionary step in high-performance control systems [10], [11] because they offer better energy efficiency, redundancy, consolidation properties, and scalable performance than existing systems. Unfortunately, as of today, there is a very limited understanding on how to leverage multi-core technology in industrial CPS workloads. Specifically, there is no clear understanding on how to compile IEC 61131-3 languages for execution in multi-core processors.

In this paper, we propose the first algorithm to automatically identify parallelism in the IEC 61131-3 languages and exploit it in multi-core-based industrial CPS. The key insight is that the designer's intentions and knowledge of the physical system under control are captured explicitly in some IEC 61131-3 languages in the form of well-defined functional elements that express parallelism. The contributions of this paper are: (1) computational performance improvement of the industrial CPS systems through automatic application parallelization and multi-core utilization approach, (2) an algorithm for identifying functional parallelism in IEC 61131-3 applications and a one-to-many allocation algorithm for scheduling a *task* into multiple *resources*, (3) a feasibility study of the proposed solution on a realistic CPS application that demonstrates the reduction of the cycle time.

II. Computation Model in IEC 61131-3

IEC 61131-3 *tasks* are executed in embedded processing units called *resources*. Several *tasks* can run on one *resource*. A *task* is an instantiation of a *program*. A *program* is composed of one or more POU (Program Organization Unit). POUs can be of type `Function` or `FUN`, `Function Block` or `FB`, and `Program` or `PROG`. POUs may be written in any of the five IEC 61131-3 languages. While a `PROG` is the top-level entry point of a *program*, POUs may call other POUs. An interesting aspect of *tasks* is that they have explicit synchronous properties. A *task* must have a priority level and may be executed either periodically (cyclically) or driven by interrupts. The *configuration* contains the information about the allocation of *programs* into *tasks* with different synchronous properties, and *tasks* into *resources*. Figure 1 shows a user application consisting of three *programs* (Program A, Program B, Program C). The top level `PROG` in Program A calls a function block "POU 1" written in FBD and "POU 2" written in LAD language. The *configuration* defines five *resources* of type `FAST_CPU`, `SLOW_CPU`, `I/O Module`, `Switch`, and `FPGA` connected by a communication network consisting of three buses. The *configuration* allocates Task 1 and Task 2 to `FAST_CPU`,

[1]A CPS is the tight integration and communication between physical processes and embedded computing elements.

978-3-9810801-8-6/DATE12/©2012 EDAA

and Task 3 and Task 4 to SLOW_CPU. Every task has an associated program. Every task includes a tuple that specifies the synchronous and priority configuration. For example, Task 1 executes periodically every 50ms and has LOW priority, and Task 2 executes only when an interrupt triggered by I/O occurs and has HIGH priority. Once the configuration and the user program are specified, including the mapping of variables to physical addresses (not shown in the Figure), the POUs and the *configuration* are translated into machine code for the target architectures and transfered to the devices for execution. The synchronous and priority information are used by the run-time of the CPS to perform the scheduling and execution. Different CPS vendors implement proprietary scheduling and execution mechanisms to improve the performance, reduce jitter, and response time.

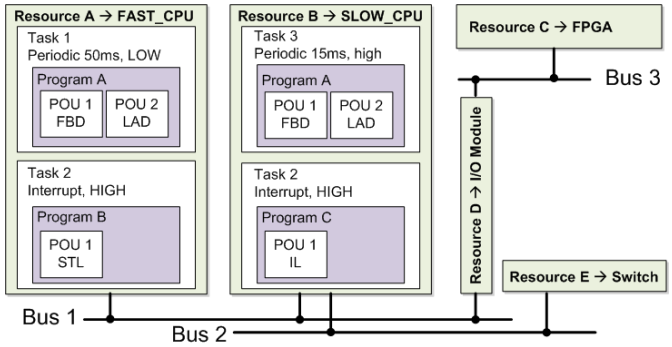

Fig. 1: Organization of an IEC 61131-3 user application and configuration. Programs are instantiated into synchronous and prioritized Tasks. In the current computation model tasks are allocated to a single computing resource enforcing a one-to-one mapping.

The existing model acknowledges concurrency at the *program* level by allowing different *programs* to run on different *resources* through a **one-to-one** allocation. However, it does not specify how a single program can be executed in multiple *resources* through a **one-to-many** allocation. In this paper, we argue that allowing a *program* to be allocated to many *resources* enables parallel computing and its benefits: reduced execution time [12], faster response time [11], reduced overall power consumption and therefore, efficient thermal management [13] (besides performance, lower thermal stress and longer life-time are strict constraints for CPS applications). This paper shows that the IEC 61131-3 computation model offers not only concurrency at the *program* level (existing model) but also fine grain parallelism at the POU level (FB, FUN, PROG) in the form of data and functional parallelism. The challenge is to identify the fine grain parallelism, partition the program in suitable program fragments, and map them into different *resources* for parallel execution. Existing techniques allow the programmer to do the partitioning manually [14], but this approach may introduce artificial delays that do not exist in the original program. In the following section, we present the algorithm to perform the parallelization automatically.

III. Functional Parallelization of IEC 61131-3

In this section, we present a new methodology to automatically identify, extract, and exploit parallelism in CPS applications. Existing methods rely only on dataflow analysis to extract parallelism and preliminary results show that this approach only contributes with modest performance improvements. In addition, the dissimilarities between CPS workloads make it difficult to develop general dataflow parallelization techniques. However, we take advantage of the fact that IEC 61131-3 high-level languages (i.e. LAD, FBD, and SFC) have domain-specific semantics that facilitate the exploitation of a different type of parallelism that is orthogonal to the parallelism found by dataflow analysis. We call this type of parallelism "functional" because it uses domain-specific semantics that describe "purpose" or "intent"[2].

[2]CPS programmers normally express their intentions explicitly, e.g. concurrency and synchrony in the program through programmatic constructs such as divergence operators and timers.

The main advantage of parallelizing industrial CPS applications is the reduction of the cycle time and therefore the improvement of the response time of the application to the environment. Typically, the faster an application is able to interact with the environment the better it is for the control algorithms. Figure 2(a) shows a periodic *task* composed by 2 POUs (POU 1, POU 2) executed in a single *resource* R1 using the existing one-to-one allocation policy. The cycle time (100ms) represents the time that is configured by the user for the task to complete. Execution time is the time it takes to execute all the POUs in the *task* and it includes the read of inputs, data processing, and write to outputs. The time between the completion of the *tasks* and the beginning of the next cycle is known as sleep time, where the *resource* is idle. Figure 2(b) illustrates the effects of parallelization of IEC 61131-3 applications and their execution using a one-to-many allocation policy. In this case, the POU 1 is partitioned into 3 independent fragments that are allocated to *resources* R1, R2, R3. POU 2 may not be partitioned but since it is independent from POU 1, it can be concurrently executed in R4. Notice that because the POUs are executed in parallel, the execution time is proportional to the execution time of the longest sub-program in R3. In this case, the cycle time can be safely reduced to 50ms to improve the response time of the application and also to reduce the sleep time. Alternatively, shorter execution times and longer sleep times may be used to significantly reduce the energy consumption of the processing devices by idling [15].

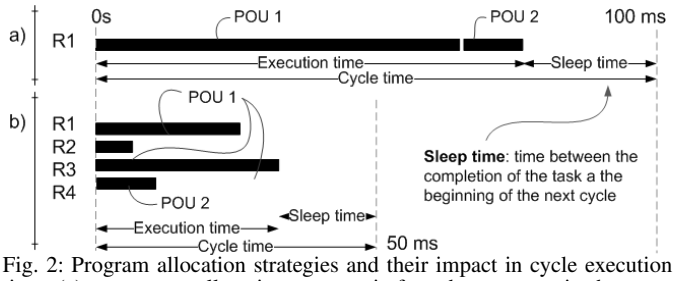

Fig. 2: Program allocation strategies and their impact in cycle execution time: (a) one-to-one allocation strategy is forced to execute in the same resource, (b) one-to-many allocation after program parallelization reduces the execution time and therefore the cycle time can be reduced to improve the response time of the user application.

A. Identifying Data and Functional Parallelism

Many CPS programmers are non-computer experts, they think in terms of their physical domains and the IEC 61131-3 languages reflect this. For example, an electrical engineer that uses LAD language thinks in terms of coils and energized networks rather than variables and program dependencies. These domain-specific constructs help the CPS programmers to express their intentions explicitly. For example, flip-flop and latches store state information over time, asynchronous operators delay signals, simultaneous convergence operators express concurrency, and timers disable the execution of parts of the program until the given time passes. However, the tools that compile these languages are written in conventional computer languages by computer experts that sometimes fail to understand the domain-specific semantics.

Figure 3 shows the typical toolchain that translates IEC 61131-3 languages into executable machine code. First, The compiler translates the user program written in any of the five IEC 61131-3 languages into a low-level intermediate representation (IR). This gives the advantage of maintaining simple parsers and a single compiler rather than one specialized compiler for each of the input languages. An optimizer takes the low-level IR and attempts to eliminate redundant and dead code, and it generates the machine code for specific CPS architectures. Finally, the machine code is assigned to the *resources* described in the *configuration* and the CPS application is executed. Functional semantics are typically lost in the toolchain when the low-level IR is produced. Functional semantics mutate into operational semantics necessary for execution in a computer. Recovering functional semantics from operational semantics is as

troublesome as recovering the original high-level program from machine code. Functional parallelism can be identified at the domain-specific languages and therefore the toolchain to support our method must be modified to propagate this information to the low-level IR and the optimizer.

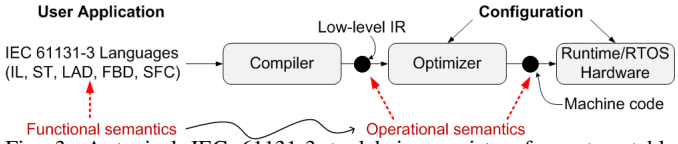

Fig. 3: A typical IEC 61131-3 toolchain consists of a retargetable compiler supporting the five automation languages and producing a low-level intermediate representation that is optimized and translated into machine code. Existing methods only exploit the operational semantics of the application (low-level IR or machine code). Our method exploits the functional semantics that capture the programmer's intentions in the high-level languages.

The "SR" blocks in Figure 4 are flip-flops whose function is to store state information. The operational behavior is to delay the outputs (state) one execution cycle. Therefore, the flip-flop inputs are decoupled from the outputs and this information can be used to break the computation flow of a Network as indicated by the diamonds in Figure 4. While dataflow analysis identifies two parallel regions (Network 1-2 and Network 3-4), functional parallelization identifies an additional opportunity in the flip-flops and creates four parallel regions (Network 1, 2, 3, 4). In combination, the two parallelization techniques partition the program into four concurrent regions in the current execution cycle that can be executed in parallel in different *resources*.

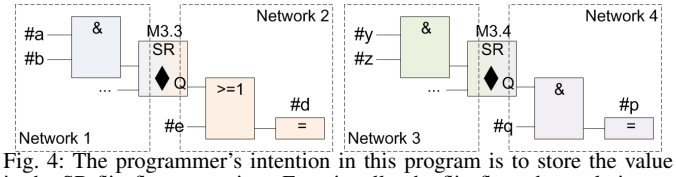

Fig. 4: The programmer's intention in this program is to store the value in the SR flip-flops over time. Functionally, the flip-flops decouple inputs from outputs and therefore create an opportunity to further partition the program into 4 fragments that are data independent and therefore can be executed in multiple processing units.

B. One-to-many Allocation

Multi-core processors and faster interconnection networks open new possibilities for low latency and high-throughput processing that had not been available in previous-generation distributed and parallel computing systems. However, migrating to multi-core processing requires a few changes in the toolchain. Specifically in the IEC 61131-3 context, the new concept of one-to-many allocation of a *program* into multiple *resources* must be implemented in the toolchain. Algorithm 1 proposes a method to accomplish one-to-many allocations. The first step is to find the functional parallelism in the *program* (Line 1) and characterize the performance of the *resources* in the *configuration* in a latency model (Line 3). This model takes into account the topology of the CPS network to calculate communication latencies and computational capacities of its execution nodes. Different heuristics for parallel scheduling can be used in Line 4 to allocate the functional fragments (FR) into multiple resources (R) taking into account the latency model such that the critical path of the application is reduced. To guarantee the hard real-time requirements of CPS systems, this transformation should be only considered when the timing analysis of the parallel schedule is better than the original schedule.

IV. Case Study of a Baggage Handling System

Figure 5 shows the top-level control program of a baggage handling system of an airport. The system includes four independent conveyor belts to transport the luggage, a radio-frequency identification (RFID) sensor detects whether the luggage is carry-on (square boxes) or check-in (rectangular boxes), an X-ray machine, and two special

Algorithm 1 One-to-Many Allocation

In: Configuration C
In: Program, $PROG$
Out: One-to-Many Allocation, P

1: FR = FindFunctionalParallism ($PROG$) {Leverage the engineering intentions explicit in IEC 61131-3 programs}
2: R = GetResources (C) {Create a latency model based on the topology and type of *resources* available in the automation network.}
3: LatencyModel = CharacterizeTopology (R) {Use parallel scheduling heuristics to allocate the data and functional parallelism to multiple resources for concurrent execution.}
4: P = ParallelSchedule (FR, R, LatencyModel) {The transformation succeeds if and only if the new schedule is guaranteed to provide a shorter execution time than the original uniprocessor schedule.}
5: **if** Timing (P) >= Timing (C) **then**
6: AbortParallelCompilation ()
7: **end if**

conveyor belts that can be configured to move in different directions (sorting system). The goal of the system is to sort the luggage according to type and send it to the corresponding conveyor belt for X-ray scanning.

Execution begins at the `InitialStep` at the top of the SFC program. The transitions `PowerOn` and `Shutdown` are conditional statements that must be satisfied to initiate the system and to shut it down. In this case, computation is waiting at the initial step until the signal `PowerOn` is received. The parallel bar after `PowerOn` represent simultaneous divergence and allow the CPS designer to express explicitly the intention of having four concurrent tasks: (1) turn on the conveyor belt and set the speed to 1 m/s, (2) turn on and do the sorting of luggage, (3) initiate the RFID sensors and classification, and (4) turn on and do the X-ray processing. The four concurrent actions converge when a stop signal is received and program terminates. Notice that this SFC program is one POU and the *one-to-one* mapping prohibits the four parallel branches to be distributed to different *resources*. Instead, the compiler serializes the concurrency in the chart for execution in a single *resource*.

Serializing a CPS application typically oversubscribes the only available processor (one-to-one mapping) with all the independent real-time tasks and this degrades their overall performance. Figure 6(a) shows a simulation trace of the baggage handler signals executed in a single *resource*. The shaded areas highlight the tasks that are competing during the cycle time (100ms) for the same *resource*. This situation slows down all the algorithms and creates undesired bottlenecks. For example, when the rate of arrival of new luggage is higher than the processing rate of the X-ray machine, the conveyor belt system must be stopped or slowed down. This wears the mechanical components faster and therefore it significantly reduces the lifetime of components such as motors, drives, and actuators.

Figure 6(b) shows the trace of the same program partitioned by our method and executed concurrently in two *resources* (off-the-shelf embedded processor with 2 cores). While this trace still shows oversubscription (shaded areas for *resource*-1, and *resource*-2), the real-time tasks compete less and their execution time is significantly reduced. For example, the X-Ray processing algorithm is reduced from 500ms to 350ms. This speedup is beneficial for the system because the speed of the conveyor belts can be modified to increase the rate of arrival of luggage and thus increase the throughput of the system. Increasing the speed of the conveyor belts and therefore the rate of arrival of new luggage, forces the system to respond quicker to events. Fortunately, the functional parallelization reduces the execution time of the program and its cycle time can be reduced to 50ms. The smaller cycle time allows the system to keep up with a faster rate of arrival of new luggage, without interrupting its normal operation.

We conducted a different experiment to measure the cycle time of the baggage handling application. Figure 7 shows that our parallelization method reduces the execution time of the program when executed in two *resources* (an embedded processor with 2 cores). Although the jitter (variance in the execution time) increases due to the inter-core communication and thread synchronization, a

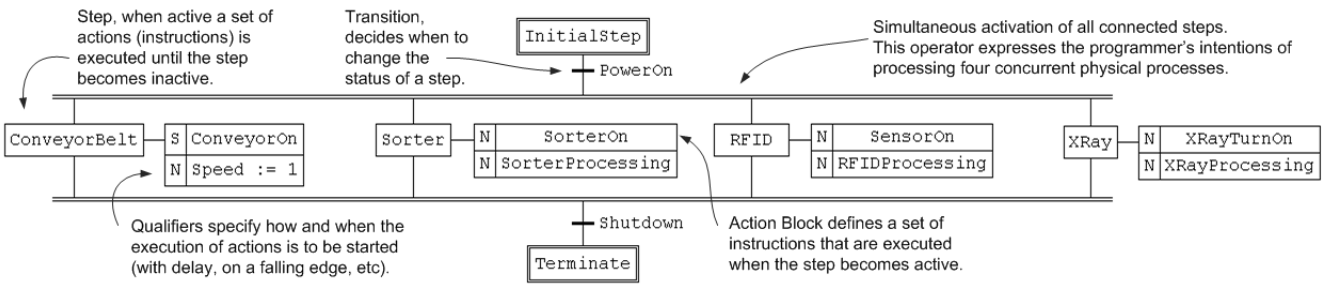

Fig. 5: SFC sample program to control the baggage handling system. The language provides the semantics that allow the programmer to express the concurrent process through a simultaneous divergence operator. Existing approaches serialize these parallel constructs and try to recover them at run-time through preemptive multi-tasking.

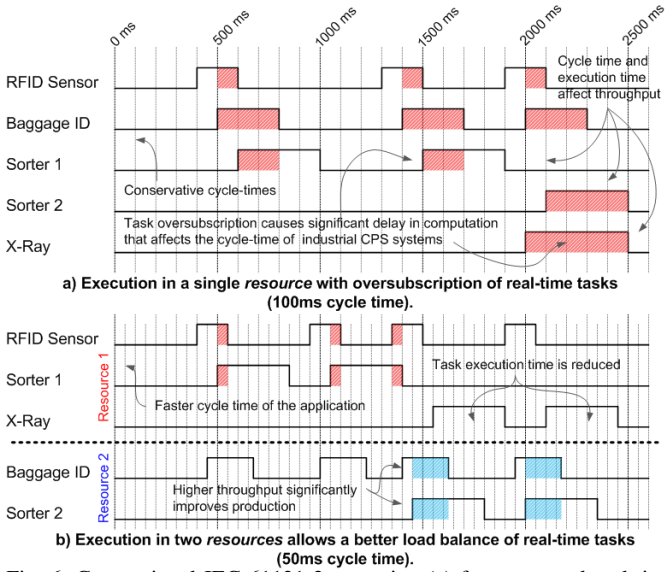

Fig. 6: Conventional IEC 61131-3 execution (a) forces several real-time tasks to compete for a single *resource*. Functional parallelization (b) spreads the load among two *resources* and thus improves the cycle time.

better load balancing is achieved in 2 *resources* and the difference between the MAX and MIN execution times is significantly less than in one *resource*. This allows tighter real-time schedules, shorter sleep times, and better utilization of the computing resources that may lead to better energy efficiency. Reducing cycle time is also beneficial for industries where waste minimization of expensive substances (e.g. coatings) is a priority and dependent on the ability of the control system to react quick enough to the environment. It also allows safer working environments for people who work in proximity to fast moving robots (e.g. assembly and manufacturing) where a response that is faster by just a fraction of a second may make the difference between life and dead.

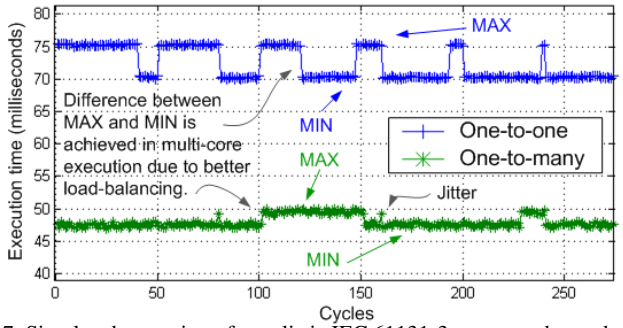

Fig. 7: Simulated execution of a realistic IEC 61131-3 program shows that cycle time can be significantly reduced when executed in two *resources*.

V. Conclusion

In a necessary effort to increase the computing performance of the industrial CPS [1], [16], we propose a novel holistic software/hardware approach that embraces the functional parallelism in control applications to transform IEC 61131-3 programs into real-time tasks that can be executed in truly parallel manner in embedded multi-core processors. This paper presents an algorithm to identify functional parallelism in IEC 61131-3 languages by exploiting concurrent operators and delays introduced by synchronous operators. In addition, we introduce the concept of one-to-many mapping of IEC 61131-3 *tasks* into *resources*. Through the case study of a realistic baggage handling CPS application, we have demonstrated the feasibility of our solution by spreading the load among several *resources*, increasing the throughput of the application, and ultimately reducing the cycle time.

References

[1] H. Koziolek, R. Weiss, Z. Durdik, J. Stammel, and K. Krogmann, "Towards software sustainability guidelines for long-living industrial systems," in *3rd Workshop of GI Working Group 'Long-living Software Systems (L2S2): Design for Future 2011 (DFF'11)*, 2011.

[2] G. Mustapic, A. Wall, C. Norström, I. Crnkovic, K. Sandström, J. Fröberg, and J. Andersson, "Real world influences on software architecture - interviews with industrial system experts," *IEEE/IFIP Conference on Software Architecture*, pp. 101–112, 2004.

[3] Meta Group, "IT Performance Engineering and Measurement Strategies: Quantifying Performance and Loss," 2000.

[4] IEC, "IEC 61131-3 Edition 2.0, Programmable controllers - Part 3: Programming Languages," http://www.iec.ch, 2011.

[5] Siemens Corporation, "Automation Technology," http://www.automation.siemens.com, 2011.

[6] K.-H. John and M. Tiegelkamp, *IEC 61131-3: Programming Industrial Automation Systems*. Springer, 2010.

[7] Control.com:, "Online discussion: SFC programming," http://www.control.com/thread/1013784893, 2011.

[8] J. U. Cho, Q. N. Le, and J. W. Jeon, "An FPGA-based multiple-axis motion control chip," *IEEE Transactions on Industrial Electronics*, vol. 56, no. 3, pp. 856 –870, 2009.

[9] D. O'Sullivan and D. Heffernan, "VHDL architecture for IEC 61499 function blocks," *IET Computers & Digital Techniques*, vol. 4, no. 6, pp. 515–524, 2010.

[10] Control Engineering, "Computing Power: Multi-Core Processors Help Industrial Automation," http://www.controleng.com/single-article/computing-power-multi-core-processors-help-industrial-automation/0266edc931.html, 2011.

[11] Intel Corporation, "Migrating Industrial Applications to Multi-core Processors," http://download.intel.com/platforms/applied/indpc/319580.pdf, 2011.

[12] A. Canedo, T. Yoshizawa, and H. Komatsu, "Skewed pipelining for parallel simulink simulations," in *Design Automation and Test in Europe (DATE)*, 2010, pp. 891–896.

[13] M. Al Faruque, J. Jahn, T. Ebi, and J. Henkel, "Runtime thermal management using software agents for multi- and many-core architectures," *IEEE Design Test of Computers*, vol. 27, no. 6, pp. 58 –68, 2010.

[14] National Instruments, "Multicore Programming with LabVIEW," http://zone.ni.com/devzone/cda/tut/p/id/6099, 2011.

[15] A. Bilgic, V. Pichot, M. Gerding, and F. Bruns, "Low-power smart industrial control," in *Design, Automation Test in Europe Conference (DATE)*, 2011, pp. 1 –5.

[16] R. Rajkumar, I. Lee, L. Sha, and J. A. Stankovic, "Cyber-physical systems: the next computing revolution," in *47th Design Automation Conference (DAC)*, 2010, pp. 731–736.