2014 11th International Conference on Information Technology: New Generations

Task Synthesis for Control Applications on Multicore Platforms

Aneta Vulgarakis*, Rizwin Shooja†, Aurélien Monot‡, Jan Carlson† and Moris Behnam†
*Industrial Software Systems
ABB Corporate Research, Västerås, Sweden
Email: aneta.vulgarakis@se.abb.com
†Mälardalen University
School of Inovation, Design and Engineering
Email: rsa12004@student.mdh.se, jan.carlson@mdh.se, moris.behnam@mdh.se
‡Industrial Software Systems
ABB Corporate Research, Baden–Dättwil
Email: aurelien.monot@ch.abb.com

***Abstract*—Multicore processors promise to improve the performance of systems, by integrating more and more cores onto a single chip. Existing software systems, such as control software from the automation domain, need adjustments to be adapted on multicores. To exploit the concurrency offered by multicore processors, appropriate algorithms have to be used to divide the control application software into tasks, and tailored task partitioning and scheduling approaches are required to increase the overall performance. In this paper we present a model-driven approach for automatic synthesis and deployment of control applications on multicore processors. The approach is centered around a system model, which describes the control applications, the multicore platform, as well as the mapping between the two. We apply the approach on a number of control applications out of which we synthesize tasks and present their run-time behavior in a real-time operating system.**

***Keywords*—task synthesis, control applications, multicore processors, model-based deployment, partitioned scheduling.**

I. Introduction

In automation, control systems are designed to regulate dynamic processes in real-time. They are composed of a set of controllers connected to sensors and actuators. Until now the hardware architectures used in executing control systems were based on single core processor architectures. However, chip manufacturers are reaching the point when they can no longer cost-effectively meet the increasing performance requirements through frequency scaling alone. Hence, the current trend in many industries is to adopt multicore technology.

The shift towards multicore processors leads to new ways of developing software. Multicore processors provide a coarser parallelism for executing tasks that needs to be provided by the programmers, the compilers, or the operating systems. However, legacy user applications are typically written in languages that are unable of expressing task parallelism. Therefore, it is challenging to identify parts of the applications that are independent and that could be potentially executed in parallel.

In the automation domain, the development of control applications typically follows a model-based approach where the application architecture is constructed from interconnected function block components according to, e.g., the IEC 61131-3 [1] or IEC 61499 [2] standard. After their deployment on the run-time architecture of an automation controller, control applications typically execute periodically. Unfortunately, as of today, there is a very limited understanding on how to leverage multicore technology in executing control applications.

When it comes to real-time multicore scheduling, the partitioned multiprocessor scheduling approach can be considered as an interesting approach to be adapted by the industry since it introduces low run-time overhead and predictable execution due to disabling tasks migrations between cores [3]. In addition, single core scheduling algorithms can be used in each core which simplifies the scheduling in multicore. However, the main drawback of this approach is that it suffers from the problem of allocating (partitioning) tasks to cores, which is a bin-packing problem [4]. Therefore heuristic algorithms are usually used to find a near-optimal partition with polynomial time complexity.

In this paper we present a process for automatic deployment of control applications on multicore platforms. Achieving this result requires to go through several steps including finding an efficient deployment (in terms of performance) of the set of control applications onto the multicore platform and generating automatically the corresponding code to be executed by a real-time operating system.

To address the deployment problem, we chose to use the approach of assigning single thread (task) of execution per control application. In addition, we use the partitioned scheduling approach to schedule control applications, thus the parallelism offered by multicore processors is exploited by supporting execution of many control applications in parallel. The worst-fit heuristic approach is used to partition applications on multicore since it balances the load of the control applications over the different cores, leading to shorter response times and jitters of the control applications which is desirable for such applications. To run the control applications on the real-time operating system, we use code generation to extract the control logic of the applications and then generate the corresponding files to set-up the system according to the deployment solution computed previously. To make our

978-1-4799-3187-3/14 $31.00 © 2014 IEEE
DOI 10.1109/ITNG.2014.61

solution general and imposing low run-time overhead, we use the fixed priority scheduling approach on each core which is widespread supported by real-time operating systems (e.g., VxWorks). So our solution requires neither changing the kernel of the operating system nor adding an additional layer between the applications and the operating system to run non-standard scheduling techniques.

In brief, the contributions of the paper are twofold: (1) an execution model of control applications on multicore platforms (section II), and (2) a task synthesis process describing the mapping of control applications to tasks, as well the task code generation (section III). In section IV we show the task synthesis process on a number of control applications realized as POSIX processes, which run on a real-time Linux for multicore platforms. Following the case study, we present the related work in section V, then conclude the paper and present a line of future work, in section VI.

II. The System Model

In this section we explain the system model that we consider, which consists of a platform model, an application model and an application execution model. This structure of the system model follows from the partitioning problem (i.e., allocating a set of control applications on a multicore platform), and from the problem of mapping applications or parts of applications on run-time entities (such as tasks implemented as threads and/or processes). The following subsections describe in more details the three models, and their relation.

A. The platform model

In this paper, we use a simple platform model, which assists us in solving the problem of executing a set of control applications on a multicore hardware. Our platform model is composed of identical, m unit-capacity cores with a shared memory. Tasks are assigned to a certain core and they are only scheduled and executed in that core, i.e., migration of tasks between cores is not allowed. The scheduling technique used on each core is the fixed priority preemptive scheduling policy.

B. The application model

The software system that we are considering consists of a set of control applications. The model of the control applications execution is as follows; first a control application reads feedback and user input values from sensors, then computes a control action using a feedback control algorithm (such as proportional-integral-derivative (PID)) and finally sends a control value to actuator(s). Different applications may read from the same sensors, however, applications do not share actuators, i.e., each actuator is dedicated to a certain application.

Since we focus mainly on the execution model of control applications, we use a simple, yet general, model to design our applications. Therefore, even other component models targeting control systems (e.g., ProCom [5], FASA [6]) and industrial standards (e.g., IEC 61131-3 [1]) can be transformed easily to this model. In our model, each application is modeled as a network of small (in terms of code size) function blocks. The function blocks communicate with each other through input/output ports based on the pipes-and-filters paradigm. A signal is used to connect an output port of a function block to an input port of another function block. Applications contain sensor function blocks, a special type of function blocks, for communication with sensors, as well as actuator function blocks for communication with actuators. For simplicity, we assume an already flattened application model i.e., our function block diagrams do not contain composite function blocks.

Each application is characterized by its cycle time, relative priority and offset. The application cycle time specifies the sampling frequency of its associated control loop that the application is designed for. The application priority defines the importance of the application relative to other applications that share the same platform. The offset is the time, relative to the start of the application cycle, when the application will be activated. Offsets are used to decrease the interference between applications.

Figure 1 shows an example of a feedback control application model. At the beginning of each cycle the application reads data from sensors using the three sensor blocks (Temperature, Pressure and Track). In the end of each cycle the application triggers a valve to open or close. According to the new value of the valve, the temperature and/or the pressure in the control process change. Therefore, in the next execution cycle, new sensor values are acquired and a new value for the valve is produced. The control algorithm is implemented by two instances of a PID control function block [7] that computes its output based on the difference between a given set point and some desired value. The output from the second PID function block (i.e., PidCC2) is sent to the AnalogOutCC function block. AnalogOutCC sends data to some I/O interface via the Valve actuator function block.

C. The application execution model

The main goal of the application execution model is to define a mapping between the application model and the platform model i.e., define how applications or parts of the application are mapped to tasks and define the execution model of the tasks. In this paper we map each application to a single periodic task (called control task). The execution parameters of a control task are inherited from its associated application parameters. For instance, the task period equals to the cycle time of the control application, task priority equals to the application priority and task offset equals to the application offset.

Each sensor/actuator is managed by a special dedicated task, called I/O task. The cycle times of control applications are usually between 1ms and 1s, and the I/O tasks run with cycle times smaller than the cycle times of control applications. In order to reduce the context switching between control- and I/O tasks, we decide to execute all I/O tasks on one dedicated core and control tasks are not allocated on that core. The details of the task mapping process are explained in the following Section III.

III. Task Synthesis

Today in single-core control systems (such as the AC 800M controller) it is a common practice to assign one thread of execution per control applications with same cycle time. However, combining many applications in one thread may

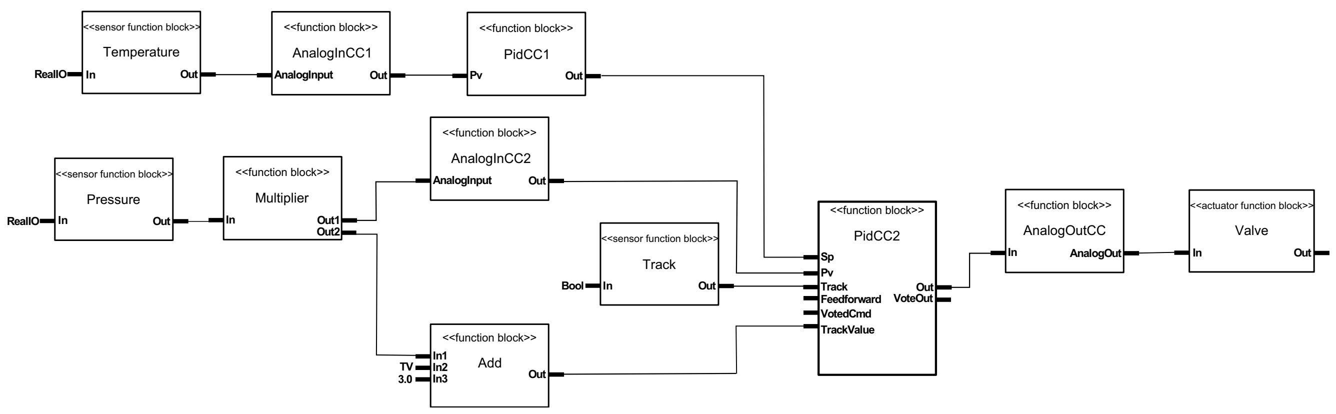


Fig. 1: Example of a control application: a cascaded control loop.

create high utilization tasks that might be difficult to partition on multicore systems. Therefore, we decide to allocate a single thread of execution per control application, and to exploit multicore systems by supporting execution of many small control applications in parallel. In the following we explain our process of mapping applications to control tasks.

A. The task synthesis process

The task synthesis process can be divided into several steps, as presented in Figure 2. First, the component-based design of the control applications and the hardware platform model (such as number of cores, cache size, etc.) of the system where the control applications will be deployed are delivered by means of system descriptions in an XML format[1]. In the application model we assign static priorities to control applications using a rate monotonic policy i.e., the shorter the cycle duration, the higher the application priority. Then, each application is linearized (i.e., a total order of the function blocks is defined) and associated with a single control task. The applications are further sorted by decreasing utilization and partitioned off-line on the cores using the worst-fit heuristic algorithm, i.e., one by one allocated to the least loaded core. We then generate a new XML file that defines the mapping between the control applications and the execution platform (number of cores and control tasks). From the generated XML file containing the mapping between the control applications and the platform we generate C files for all applications, and a C file for the master controller, which is responsible for assigning core affinity and priority to all tasks. In the end we generate a makefile with application and I/O tasks. In the next section we explain the realization of the process in more detail.

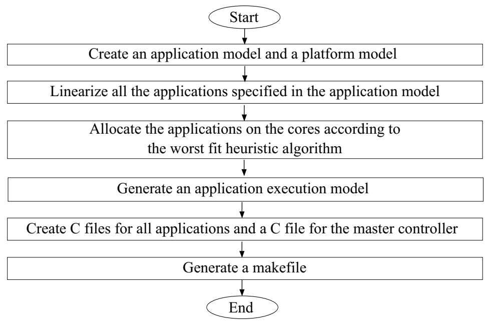
B. Realization of the task synthesis process

In our task synthesis process we use the Java API JAXB [9] to retrieve data from an XML file. JAXB allows generation of a set of Java classes out of an XML schema via a process called unmarshaling[2]. Data represented in an XML file is automatically converted into Java objects.

[1]Note that a tool, such as 4DIAC-IDE, can be used to model the applications and the platform in order to generate a system description in an XML format [8].

[2]Unmarshalling is the process of retrieving the memory representation of an object from a data format suitable for storage or transmission, as for example transforming an XML document into a hierarchy of Java objects.

We linearize the applications in the following way: all the sensor blocks are ordered to be executed in the beginning of each cycle, and all the actuator blocks are ordered to be executed in the end. The rest of the function blocks are grouped into branches, and the function block(s) in a same branch are executed in a cascaded fashion. Branches are formed only by a single function block, or a list of function blocks which are connected, but where each function block has a single successor and a single predecessor block. For example, in Figure 1, the function blocks `AnalogInCC1` and `PidCC1` form a branch. The linearization algorithm takes care of the proper order of the branches in the linearized application. Therefore, there are several possible ways to obtain a linearized application.



Start
Create an application model and a platform model
Linearize all the applications specified in the application model
Allocate the applications on the cores according to the worst fit heuristic algorithm
Generate an application execution model
Create C files for all applications and a C file for the master controller
Generate a makefile
End

Fig. 2: The task synthesis process.

The generated C files for all applications include standard headers (such as stdio.h, stlib.h, etc.) and application specific headers. The application specific headers contain a list of global variables and a POSIX semaphore shared by all control tasks, and header files corresponding to the function blocks that build the application. The semaphore is treated as if each core has its own semaphore, and does not affect the processes executing in other cores. In the body of an application C file we call the function blocks that build the application. Each application is realized as a POSIX process [10] in order to have its own address space.

In the XML files each function block has a *fileReference* tag which points to the function block header file. The functionality of the function blocks is provided in the form of entry functions implemented in C. A POSIX semaphore that is shared among all control tasks in the same core is used to disable preemption between applications during function block execution. The reason for disabling preemptions during the execution of function blocks is that the execution time of function blocks is relatively short, so avoiding context switching during the execution of a function block can decrease the runtime overhead significantly. The semaphore is locked before the execution of every function block and released after the block is done with the execution. Within an application we use shared memory to pass data between function blocks; the sender block writes its data to a memory block, and the receiver block reads the data from that memory block.

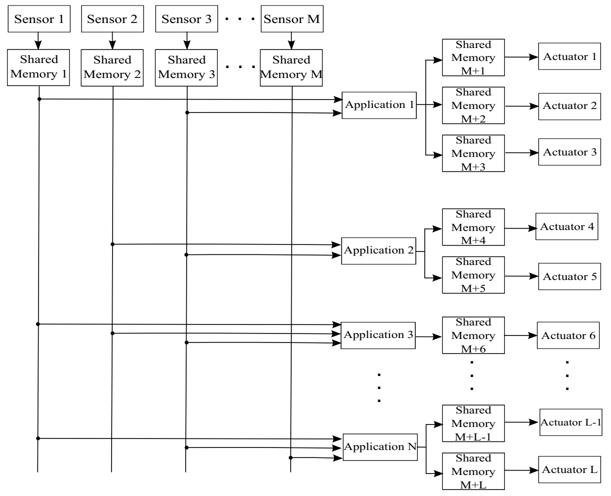


Fig. 3: Communication between I/O tasks and applications.

The tasks that connect the applications to the outside world are the I/O tasks or more precisely, the sensors and actuators. Sensor function blocks from an application read from shared memory locations to which sensors may write, and actuator function blocks write into shared memory locations from which actuators may read. As we mentioned in Section II, applications are implemented in such a way that they can share access to sensors, but cannot share actuators (see Figure 3). The I/O tasks execute asynchronously and at a higher frequency than the control tasks. We use dedicated System V shared memories protected by System V semaphores for each sensor that can be accessed by both applications and I/O tasks. Also, another set of System V shared memories and semaphores are shared between actuators and the respective applications. The memory footprint of the System V semaphores are higher than that of the POSIX semaphores, since while creating a semaphore object System V creates an array of semaphores while POSIX creates just one. However, the resources that System V IPC uses are available system wide [11] which is essential when it comes to inter-core communication. Since the I/O tasks are allocated on an exclusive core (i.e., no control applications are executed on that core), the shared memory that they write into or read from has to be shared with applications running on other cores. Hence, we chose System V shared memory over POSIX shared memory for inter-core communications among processes.

In the master controller file we use a POSIX semaphore for a synchronized start of all the control tasks once they are created. Every application body is defined within an infinite while loop. We use a high resolution timer [12], and the processes are made periodic according to the system clock CLOCK_MONOTONIC, a non-settable, monotonically increasing clock that measures time since some unspecified point in the past that does not change after system start up. The high resolution timer provides high accuracy even when the cycle times of the tasks (be they control- or I/O tasks) are smaller than the period of the Linux jiffy timer that is used by the Linux scheduler.

IV. Case Study

In this section we present a case study of the task synthesis process consisting of ten instances of the cascaded control loop application introduced in Section II. The hardware platform used in this case study is an Intel Core i7-2620 [13] dual-core processor with hyper-threading enabled, so that it can execute four different threads in parallel. The system runs the 64-bit version of the Ubuntu 12.04 LTS operating system (kernel version 3.6.11.2) patched with the PREEMPT RT patch (version 3.6.11.2-rt33) [14], which turns the stock Linux kernel into a real-time kernel.

We create an XML file containing an application and a platform model. Since we can run four threads in parallel, we declare in the platform model that the target platform supports three cores. We do not include Core0 in the XML description of the platform model as it is dedicated for I/O tasks. Hence, Core1, Core2 and Core3 are used for allocating and running the ten instances of the control loop application. The application model in the XML file consists of ten instances of the cascaded control loop application. Each application contains three sensor blocks, seven function blocks and one actuator function block. The connections between input and output ports of different function blocks are defined as signals in the XML file. The sensor function blocks Temperature, Pressure and Track read data from shared memory locations to which data are writing the respective I/O tasks based on values read from the sensors. The processed data from each cascaded control loop application is fed into an instance of the actuator function block Valve that further writes the data into a dedicated memory location that is not shared among other applications. The data is read from that memory location by I/O tasks for actuators.

After linearizing the applications, and performing worst-fit partitioning we generate a new XML file that contains information about control tasks and their allocation on the cores of the hardware platform. Figure 4 presents one possible linearization of the cascaded control loop that we use in our experiments. Another possible linearization is presented in Figure 5. Table I shows the allocation of the ten instances of the cascaded control loop application on the hardware platform based on the worst-fit heuristic. To demonstrate preemption between control tasks of different priority we run applications 3, 6, 7 and 8 with cycle time 1000 ms, and applications 1, 2, 4, 5, 9 and 10 with cycle time 800 ms. The I/O tasks run with cycle time 10 ms.

TABLE I: Allocation of the ten applications on the cores.

Core	Allocated applications IDs
1	2, 3, 10
2	4, 6, 8, 9
3	1, 5, 7

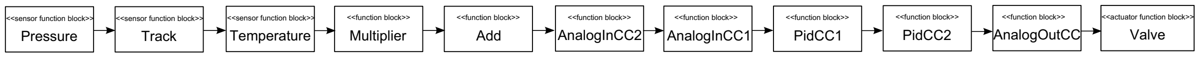


Fig. 4: Linearization of the cascaded control loop application used in the experiments.

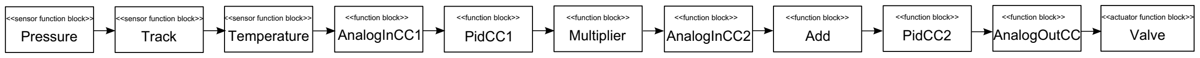


Fig. 5: Another possible linearization of the cascaded control loop application.

We further use the previously generated XML file for creating the C files of every application, which are named after the applications. In the master controller file, the I/O tasks are created first and are assigned to Core0 which is unallocated for applications. Since the I/O tasks have the smallest cycle time we set their priority to 98, which is the highest recommended Linux real-time priority for user processes[3]. A semaphore is locked before the applications are created and unlocked after all the applications are created. Control tasks 3, 6, 7 and 8 hold the same priority 1, and control tasks 1, 2, 4, 5, 9 and 10 have priority 2. All control tasks have an offset 0. We use trace-cmd for recording a trace of the scheduler activities, and KernelShark [16] for visualizing the recorded trace. Figure 6 depicts an example trace of two cycles of the applications execution. The different colors in the figure represent the core on which the processes are running. Observe, for example that the traces of application 4 (i.e., the process Valve_Control4.-3916), application 6 (i.e., the process Valve_Control6.-3922), application 8 (i.e., the process Valve_Control8.-3919) and application 9 (i.e., the process Valve_Control9.-3913) have the same color, meaning they are executing on the same core. During the first depicted cycle application 6 executes until completion. Then application 8 executes for a while, but gets preempted from the higher priority application 9. When application 9 finishes executing, application 4 starts and runs to completion. In the end when there are no more higher priority tasks to be executed, application 8 locks the semaphore and finishes executing.

V. Related Work

Code synthesis for multicore platforms has been proposed for modeling languages such as UML/MARTE [17] and dataflow languages [18], including Simulink [19]. Also, various domain specific languages and meta-models have been proposed that capture detailed information needed to synthesize efficient multicore code, such as the ESE toolset [20].

In the automation domain, there are few approaches targeting multicore, such as the semantic adjustments proposed to better align IEC 61499 with multicore realization [21]. Canedo and Al-Faruque present a mechanism to transform IEC 61131-3 programs into real-time tasks for execution on embedded multicore processors [22]. The FASA component framework specifically targets distributed and multicore embedded systems [6], and includes a constraint programming model to find suitable allocations. Contrasting these approaches that address parallel execution at a lower level of granularity, we consider concurrency at the level of applications, and propose a synthesis process where linearized application are distributed over the cores.

[3]Real time tasks in Linux can be assigned with priorities 1-99 with 99 as the highest priority (reserved for critical kernel threads), and other non-real time processes are assigned priorities 100-139 with 139 as the lowest priority [15].

The general problem of allocating tasks to cores has been addressed using different methods and considering different optimizations criteria. Methods targeting systems with hard real-time requirements typically focus on schedulability (absence of deadline violations in worst-case scenarios) [23], [24], while approaches targeting soft real-time systems tend to focus on optimizing the average behavior [25], [26].

VI. Conclusions and Future Work

In this paper, we have described a process for automatic deployment of automation control applications on multicore platforms. We start the process with a model of the control applications, and of the multicore hardware platform on which the applications should run. Out of these two models we generate an application execution model that defines a mapping between the applications and execution platform i.e., tasks and cores. In the application execution model each application is linearized and mapped to a periodic run-time control task. In addition, the control tasks are partitioned (i.e., allocated) on the cores of the system according to the worst-fit heuristic. The tasks on each core are further scheduled according to the fixed priority preemptive scheduling algorithm. To demonstrate the applicability of our process, we have shown in an example how control applications can be synthesized into tasks and run on a real-time Linux for multicore platforms.

In control systems, it is desired to decrease the response times and jitters of control tasks to maintain the performance of the control loop. However, the interference from other control tasks that share the core and blocking due to sharing resources with I/O tasks can not be avoided, which increases both response times and jitters of control tasks. In our solution, task preemption is only allowed between function blocks. To decrease response times of control tasks, in the future, we can define larger non-preemtable sequence of blocks by analyzing their execution times as well as the structure of the original function block diagram before performing the linearization. Another method to decrease response times is by controlling tasks offsets to minimize the interference between tasks. In the future, we could also distribute the I/O tasks on different cores and try to group the applications that use them in the same core to improve the performance and decrease the overhead of inter-core communication.

As a future work, we can tune our implementation by using APIs targeting specific real-time operating systems. Concerning the hardware platform architecture, instead of symmetric multicore architecture, we can investigate the performance of our system in more complex architectures, such as multi-cluster architectures or heterogeneous cores. Finally, we want to exercise the scalability of our task synthesis approach.

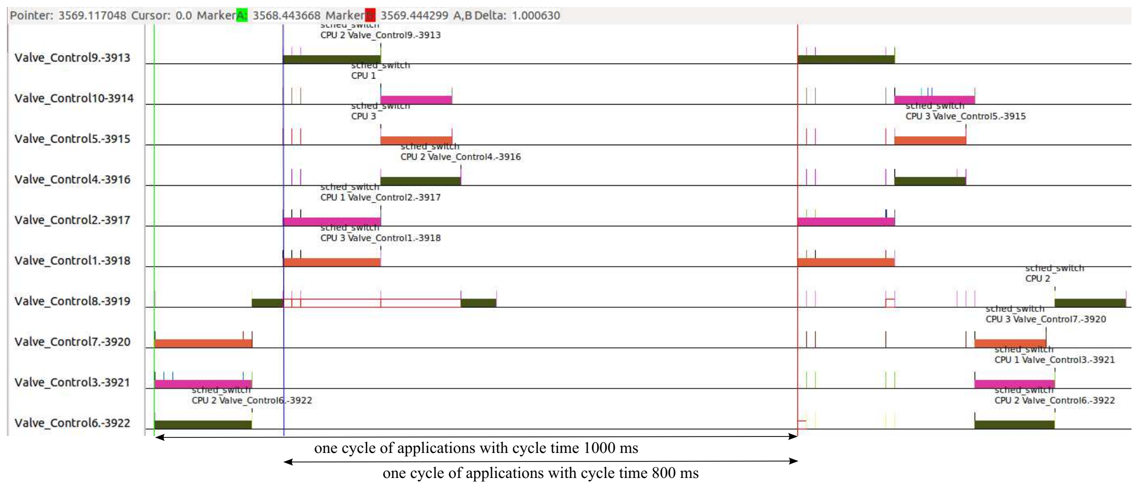


Fig. 6: Recorded trace of two execution cycles of the ten applications.

Acknowledgment

This work was supported by Industrial Software Systems at ABB Corporate Research, by the Swedish Research Council project Contesse (2010-4276) and by the Knowledge Foundation (KKS). We would like to thank Manuel Oriol and Michael Wahler (ABB Corporate Research, Baden–Dättwil) for their valuable comments on this paper.

References

[1] K.-H. John and M. Tiegelkamp, *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*. Springer Berlin Heidelberg, 2010.

[2] International Electrotechnical Commission, "IEC 61499-1: Function Blocks-Part 1 Architecture," Geneve, 2005.

[3] D. Potop-Butucaru, R. De Simone, and Y. Sorel, "From Synchronous Specifications to Statically-Scheduled Hard Real-Time Implementations," in *Synthesis of Embedded Software: Frameworks and Methodologies for Correctness by Construction*, S. K. Shukla and J.-P. Talpin, Eds. Springer, 2010, p. 34, chapter 8.

[4] D. S. Johnson, "Near-optimal bin packing algorithms," Ph.D. dissertation, Massachusetts Institute of Technology, 1973.

[5] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnković, "A Component Model for Control-Intensive Distributed Embedded Systems." Springer, October 2008, pp. 310–317.

[6] M. Oriol, M. Wahler, R. Steiger, S. Stoeter, E. Vardar, H. Koziolek, and A. Kumar, "FASA: a scalable software framework for distributed control systems," in *Proceedings of the 3rd International ACM SIGSOFT Symposium on Architecting Critical Systems (ISARCS'12)*. ACM, June 2012, pp. 51–60.

[7] K. H. Ang, G. Chong, and Y. Li, "PID control system analysis, design, and technology," *IEEE transaction on control system technology*, vol. 13, no. 4, pp. 559–576.

[8] V. Domova, E. Ferranti, T. de Gooijer, and A. Vulgarakis, "4DIAC integration into the FASA project: a success story of increased maintainability and modularity," in *4DIAC Workshop, 18th IEEE International Conference on Emerging Technologies and Factory Automation*, 2013.

[9] "Java Architecture for XML binding," https://jaxb.java.net/, (Last Accessed: 2013-10-07).

[10] "The POSIX standard," http://pubs.opengroup.org/onlinepubs/9699919799/, (Last Accessed: 2013-10-07).

[11] "Comparison between POSIX and System V semaphores," http://www.linuxdevcenter.com/pub/a/linux/2007/05/24/semaphores-in-linux.html?page=4, (Last Accessed: 2013-10-07).

[12] "High Resolution Timers," http://elinux.org/High_Resolution_Timers, (Last Accessed: 2013-10-07).

[13] "Intel® Core^{TM} i7-2620M," http://ark.intel.com/products/52231, (Last Accessed: 2013-10-23).

[14] "PREEMPT RT patch," https://rt.wiki.kernel.org/index.php, (Last Accessed: 2013-10-07).

[15] "Real-time task priorities in Linux," www.linuxjournal.com/magazine/real-time-linux-kernel-scheduler, (Last Accessed: 2013-10-23).

[16] "KernelShark," http://people.redhat.com/srostedt/kernelshark/HTML/, (Last Accessed: 2013-10-07).

[17] H. Posadas, P. P. nil, A. Nicolás, and E. Villar, "System synthesis from UML/MARTE models: The PHARAON approach," in *Electronic System Level Synthesis Conference (ESLsyn)*, 2013.

[18] J. Piat, S. S. Bhattacharyya, M. Pelcat, and M. Raulet, "Multi-core code generation from interface based hierarchy," in *Proceedings of the 2009 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2009.

[19] M. Cha, K. H. Kim, C.-J. Lee, D. Ha, and B. S. Kim, "Deriving high-performance real-time multicore systems based on simulink applications," in *IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, 2011, pp. 267–274.

[20] S. Abdi, G. Schirner, I. Viskic, H. Cho, Y. Hwang, L. Yu, and D. Gajski, "Hardware-dependent software synthesis for many-core embedded systems," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*. IEEE Press, 2009, pp. 304–310.

[21] V. Vyatkin, V. Dubinin, C. Veber, and L. Ferrarini, "Alternatives for execution semantics of IEC6149," in *The 5th IEEE Conference on Industrial Informatics*, 2007.

[22] A. Canedo and M. A. Al-Faruque, "Towards parallel execution of IEC 61131 industrial cyber-physical systems applications," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Los Alamitos, CA, USA: IEEE Computer Society, 2012, pp. 554–557.

[23] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35:1–35:44, Oct. 2011.

[24] A. Monot, N. Navet, B. Bavoux, and F. Simonot-Lion, "Multisource software on multicore automotive ECUs - combining runnable sequencing with task scheduling," *IEEE Transactions on Industrial Electronics*, vol. 59, no. 10, pp. 3934–3942, 2012.

[25] J. M. Calandrino, J. H. Anderson, and D. P. Baumberger, "A hybrid real-time scheduling approach for large-scale multicore platforms," in *19th Euromicro Conference on Real-Time Systems*. IEEE Computer Society, 2007, pp. 247–258.

[26] J. Feljan, J. Carlson, and T. Seceleanu, "Towards a model-based approach for allocating tasks to multicore processors," in *38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, September 2012.