A Framework for Component-Based Real-Time Control Applications

Stefan Richter
ABB Corporate Research, Industrial Software Systems
Segelhofstr. 1K, Baden-Dättwil, Switzerland
stefan.richter@ch.abb.com

Michael Wahler
ABB Corporate Research, Industrial Software Systems
Segelhofstr. 1K, Baden-Dättwil, Switzerland
michael.wahler@ch.abb.com

Atul Kumar
ABB Corporate Research, Industrial Software Systems
Whitefield Road, Bangalore, India
atulkumar.d@in.abb.com

Abstract

State-of-the-art real-time control systems execute multiple concurrent control applications using operating system mechanisms such as processes, mutexes, or message queues. Such mechanisms leave a high degree of freedom to developers but are often hard to deal with: they incur runtime overhead, e. g., context switches between threads, and often require tedious and costly fine-tuning, e. g., of process and thread priorities. Reuse is often made more difficult by the tight coupling of software to a given hardware or other software.

In this paper, we present a software architecture and execution framework for cyclic control applications that simplifies the construction of real-time control systems while increasing predictability and reducing runtime overhead and coupling. We present the concepts of this framework as well as implementation details of our RTLinux-based prototype.

1 Introduction

Control systems interact with the physical world through sensors and actuators. Since physical processes typically do not wait for the result of some computation, control systems must meet given deadlines and are therefore real-time systems. Traditionally, there was a separate controller for each control application. These controllers are typically running at control cycles of one millisecond, i.e., they acquire new measurements, process them, and take actions once every millisecond. With the continuous improvement of computer technologies and the introduction of digital communication standards such as IEC 61850 [3], there have been strong trends to combine several such logical controllers into one physical device to reduce cost. Still, a certain degree of independence is required to make sure that a faulty logical controller does not affect other controllers on the same physical device.

Executing multiple control applications on the same controller has significantly increased the complexity of the controller's software. In particular, the concurrent execution of multiple applications requires synchronization between different processes.

The fine-tuning of such synchronization to satisfy all runtime constraints on a given platform can be tedious and error-prone. Furthermore, it is difficult to reuse a given set of processes on a different platform. At runtime, the concurrent execution of multiple processes causes context switches, which may be expensive in comparison to the program logic.

In this paper, we present a component-based software architecture and execution framework for a large class of industrial control systems. These systems comprise one or several periodic real-time tasks (as defined by Liu in Section 3.3.1 [8]). Using a small set of abstraction mechanisms, our approach simplifies system construction and makes large parts of an engineered control system reusable across different platforms.

This introductory section finishes with a motivating example and an overview of related issues. Our component framework and its abstraction mechanisms are presented in Section 2. We give implementation details in Section 3 along with several measurements of the framework's performance in Section 4. We conclude the paper with a summary, discussion, and outlook in Section 5.

1.1 Motivating Example

Throughout this paper we will use an example from the domain of power systems. In this domain, a typical purpose of an embedded controller is protecting primary equipment, e. g., a transformer. To this end, the controller receives measurements, i. e., voltages or currents, from a sensor and determines from these measurements whether the equipment is in a healthy state. If not, the controller sends a command to a circuit breaker that disconnects the equipment from the network to avoid damage. There are controllers for erroneous situations regarding voltage, current, temperature, and many more.

In our example, we want to run an overcurrent protection and an overvoltage protection on the same device. Both protection functions run at 1 kHz (a typical frequency for this kind of controllers) and rely on measurements that are sampled at 1.6 kHz (a typical sampling frequency in IEC 61850). These measurements need to be resampled by the controllers from 1.6 to 1 kHz to feed consistent data into the protection algorithms. Because of the general setup of the environment, all sensor data from one point in time are sent in one Ethernet packet as defined in IEC 61850-9-2. The structure of the software comprising these functionalities is shown in Figure 1.

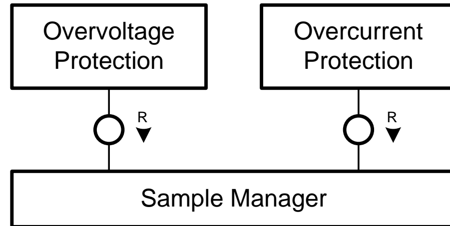

FIGURE 1: *Structure of a typical controller software*

A typical implementation of such a system would follow the client-server pattern and comprise three processes: overvoltage protection, overcurrent protection, and the sample manager. The latter would be responsible for receiving, parsing, resampling, and storing the data packets temporarily. Ultimately, the protection functions would request resampled data at 1 ms intervals.

Both protection processes would contain one thread each, which would consist of an endless loop that waits for a trigger (either as a direct timer interrupt handler, some IPC mechanism such as a semaphore, or by calling a sleep function) and then run its respective protection algorithm. For obtaining data from the sample manager process, some inter-process communication mechanism would be required.

On the other side, the sample manager would provide two threads for answering to the protection functions and one thread for dealing with incoming network packets. These three threads would have to be synchronized using standard mechanisms such as mutexes to protect access to the sample buffer[1].

1.2 Issues

This example exhibits several drawbacks of this kind of implementation.

Predictability Algorithms for analyzing general thread-based systems, e. g., detecting deadlocks, are hard to implement (see, e. g., [10]). By using specific characteristics of the class of systems under consideration here, predictions could be automated.

Implementation effort For every such application, the application engineer has to think thoroughly about the implementation details

[1] Even though there might be a solution without explicit synchronization mechanisms it is not obvious and possibly not easy.

such as priorities, proper synchronization, possible deadlock scenarios etc. Often, application engineers in the power and automation domain lack formal education in computer science, making the task even harder for them.

Reusability While all processes seem to be fairly independent from each other, they do depend on the protocol between sample manager and protection functions. This protocol needs to be implemented by both sides. Further, developers must also decide if the threads are supposed to be run in a light-weight (same process) or in a heavy-weight way (different processes) and must adapt the usage of inter-application communication mechanisms accordingly.

OS overhead If there are fewer CPU cores than threads context switches between threads have to occur several times in each cycle, whenever the flow of execution requires one thread to wait for another thread's output. Because of the short cycle times in many embedded systems, context switches may have a significant impact on the system behavior (see Section 4).

Communication overhead The communication must rely on means of inter-process communication, which is considered to be slow (e. g., message passing) or to potentially compromise data integrity (e. g., shared memory).

1.3 Related Work

Component-based systems have been proposed to overcome some but not all of these issues. Kopetz [4] describes a component system for real-time systems that addresses predictability and implementation effort. While we use a different, i. e., more fine-grained and restrictive, component model, we follow the proposed concept by using static schedules as advocated also by Locke [9].

Kuz et al. [5] present a component framework on top of the L4 microkernel. It strives for defining operating system functionality (e. g., file system) in a component-based way. In contrast, our framework builds on top of existing real-time operating systems such as RTLinux and focuses on the control applications.

Rastofer and Bellosa [12] aim at separating component functionality from platform mechanisms such as concurrency and synchronization to increase predictability of system properties. Earlier, Büchi and Weck [2] pointed out that black-box components are not sufficient for analyzing important properties of systems and that white-box components contain too much detail for this task. The decomposition of components into blocks (see Section 2.1) in our approach follows up on their plea for gray-box components.

Reusability has been addressed in a component-based real-time system by Wang et al. [15]. They propose a component-based resource overlay that isolates the underlying resource management from applications to separate the concerns of application designers and component providers. Complementing the work presented in this paper, they focus on the proper allocation of resources to real-time components.

2 Component Framework

To address the issues in Section 1.2, we designed a component framework with a runtime concept comprising four structural elements: component, function block, port, and channel. In Section 2.1 and Section 2.2, we describe these concepts in greater detail. Our component framework further encompasses a concept for executing fully deterministic static but replaceable schedules. Application schedules are explained in Section 2.3, their execution is presented in Section 2.4. In Section 2.5, we discuss how the concepts introduced in this section address the aforementioned issues.

2.1 Components and Function Blocks

In our component framework, the example above would consist of three *components* representing overvoltage protection, overcurrent protection, and sample manager. Conceptually, components separate pieces of software from each other. Their purpose is to provide for the independence required in Section 1.1, hence to ensure that software defects that could affect system stability (e. g., memory safety violations) do not propagate across component boundaries.

Components as such do not directly provide for any executable code; they are in effect comparable to address spaces in typical operating systems. In contrast to a process, which contains threads, a component contains a number of *function blocks*. A function block (or simply *block*) is defined as a sequence of instructions with the following properties:

Sequential It needs to consist of exactly one stream of instructions that can be executed on one processor core, i. e., it cannot distribute its work

on several cores or threads. The stream of instructions can contain branches and loops as long as the other properties below are satisfied.

Terminating Each block needs to guarantee that it finishes its execution within its given deadline for arbitrary inputs if the block is free to run on the CPU.

Non-Blocking A block may not depend on any functionality of the underlying platform that could block its execution. This includes in particular synchronization mechanisms, standard I/O operations, and sleep instructions. It does not mean that blocks cannot be used at all to access file systems but it has to be done in a non-blocking way, i. e., by polling.

Stateless A block may not keep any state, i. e., the output of a block depends only on its inputs in the same execution cycle. [2]

These properties ensure that data flow and control flow are not part of the block's logic. Instead, block execution is orchestrated by the framework according to given data flow and control flow models. This orchestration is described in Section 2.3.

Compared to a thread as a basic entity in operating systems, a function block does not maintain data across cycles. An important consequence is that the stack associated with the execution of a block is empty after a block finished. Moreover, blocks do not require sophisticated synchronization like threads since they do not rely on blocking mechanisms. In particular, blocks cannot wait for other blocks and thus situations involving deadlocks cannot arise.

In Figure 2, we depict a decomposition of our example from Section 1.1 into components and blocks. The sample manager has a block *Sample Receiver*, which receives samples from the network and one block *Sample Provider* for each protection component that provides the required samples. Each protection component has a block that determines which samples have to be requested from the sample manager and a block containing the actual protection algorithm.

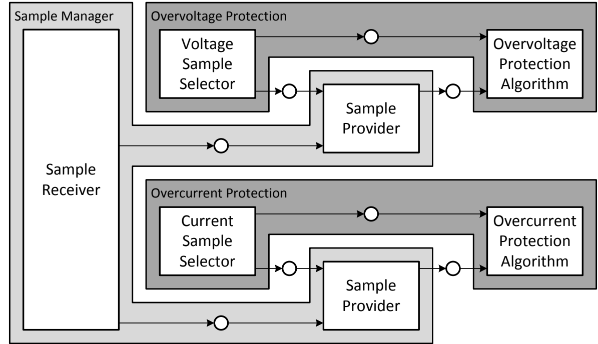

FIGURE 2: *Block diagram of example*

2.2 Ports and Channels

Blocks depend on input and generate output. To this end, they expose interfaces consisting of *input ports* and *output ports*, which have been depicted as inbound and outbound arrows in Figure 2. Blocks are not allowed to write to input ports or to read from output ports.

Ports need to be connected to *channels*, which are represented by circles in Figure 2, such that each port is connected to exactly one channel and each channel is connected to exactly one input port and one output port.

The framework ensures that each block has valid inputs before its execution gets triggered, i. e., all blocks on which a block depends get executed and finished before this block.

There might be different kinds of channels in a system to optimize its performance. For instance, a channel between two blocks in the same component could be implemented as simple as some shared memory. A channel between two components on the same controller could be implemented by some message queue mechanism. This observation yields the essential argument for not encapsulating each block into its own component.

The reader may have noticed that our example requires a buffer for received samples but that a block is explicitly stateless. We consider this buffer a kind of feedback that can be represented by a component-internal channel as depicted with a dashed line in Figure 3. The *Repeater* is a block that distributes the same data from one channel to n others, in this case as feedback to the sample receiver and as input to the sample providers.

[2] Of course, *some* blocks *might* also depend on the state of the underlying system, e. g., the file system. For the sake of simplicity such blocks and their implications to the system shall be out of scope of this paper.

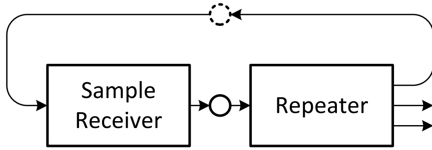

FIGURE 3: *State as feedback*

Because input ports are read-only and output ports are write-only this setup could be easily optimized by a compiler to avoid copying data. The advantage of this approach is its proximity to control theory, in which system state is usually described in terms of some mathematical function that depends on time (see, e. g., [13]). Domain-relevant standards such as IEC 61131 thus consequently model system state as feedback.

2.3 Application Schedules

Application schedules describe the execution order and periodicity of the blocks of an application. In particular, they define the control flow between blocks because applications do not change during execution. Such separation of control logics (defined in the blocks) and how individual algorithms operate together (defined in the schedule) allows for easy reuse of blocks in different contexts.

An application schedule is a tree whose leaf nodes are blocks and whose inner nodes (called *control nodes*) specify details of the execution of their children. Schedule trees can be of arbitrary depth, i. e., control nodes can contain other control nodes. Schedules can be automatically generated from the data flow specified in block diagrams [6] and subsequently be optimized either automatically or manually. The control nodes of our system are:

(S)equential nodes require execution of their child nodes in exactly the sequence specified.

(P)arallel nodes allow for an arbitrary execution order of their child nodes.

(A)lternating nodes alternate through their n child nodes. In each cycle, the next child gets executed. It has to be specified after how many cycles $c \geq n$ execution starts again with the first child. This allows for an efficient implementation of multi-rate systems as each child gets executed with a c times larger period than its parent.

(W)atchdog nodes terminate the execution of their children after a predefined time.

The application schedule for the example of Figure 2 is shown in Figure 4. The root node, which is executed first in each cycle, is a watchdog node that terminates subsequent blocks if they run longer than 850 µs. The next block is a sequential node, which first executes the *Sample Receiver* (SR) block and then a parallel control node. This parallel node specifies that its branches (two sequences of blocks) can be executed in any order.

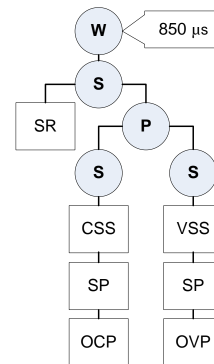

FIGURE 4: *Schedule for example application*

Sometimes the worst-case execution time of a system cannot be statically verified. In this case, the root node of the schedule should be a watchdog control node with a timeout that is less than the cycle time to ensure the timely termination of the whole schedule. As an example, a system running at 1 kHz should be guarded by a watchdog control node that ends a cycle after 850 µs. This leaves a slack time of 150 µs in which asynchronous tasks such as an FTP server can be scheduled by the operating system.

Application schedules as presented in this subsection offer several benefits:

- The information about dependencies between blocks is not in the blocks themselves but in the schedule. Thus, blocks can be independently developed, tested, and verified.
- Interaction between blocks is simplified, which makes it easier to reason about the system.
- Blocks can be guaranteed to get their share of CPU time by using time slot guarantees provided by the scheduling mechanisms of modern real-time operating systems. Conversely, blocks can be checked at runtime whether they stick to their time limits using a watchdog timer as described above.

2.4 Execution

Since multiple applications can be executed on the same controller, a *system schedule* needs to be computed from the individual application schedules. Blocks are executed sequentially in our framework. Therefore, the system schedule defines a total order on the blocks of all applications. This total order must be consistent with the partial orders of the blocks in the individual application schedules.

As an example, assume that the application from Figure 2 is the only application to be scheduled. Since the application schedule of this application (cf. Figure 4) involves parallel control nodes, there are several possible system schedules that satisfy the partial block order. One such system schedule would be the sequence {SR, CSS, SP, OCP, VSS, SP, OVP}.

Although system schedules are static, an active schedule can be replaced with another schedule at runtime. In [14] we describe how this is achieved and explain how control software can be updated at runtime using this mechanism.

System schedules are executed by a *dispatcher*, which executes a sequence of blocks in a cyclic fashion at a given base frequency. To this end, timer interrupts are generated at regular intervals, each of which triggers one execution of the dispatcher. In Figure 5, timer interrupts are displayed as vertical arrows. The dispatcher then calls some or all blocks according to its schedule, which is represented by the gray bars in Figure 5. After the last block has been executed, the dispatcher waits until the next timer interrupt. During this period of time, which we call *slack time*, other software may run asynchronously, e. g., an FTP server. This software will be preempted by the operating system as soon as the timer interrupt triggers the dispatcher for the execution of the next cycle.

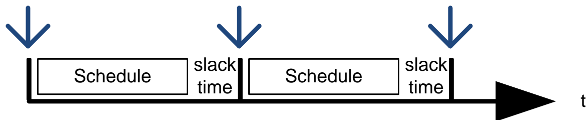

FIGURE 5: *Timeline of schedule execution*

The interval between two timer interrupts is called *base cycle time* of the system; its inverse is called *base frequency*. Alternating nodes allow for the implementation of multi-rate systems as long as the cycle times of all blocks are integer multiples of the base cycle time. In other words, the base cycle time should be chosen such that it represents the least common denominator of the individual cycle times of all applications in the system. As an example, if three applications shall be executed that need to be run at 100 Hz, 500 Hz, and 1000 Hz, the base cycle time should be set to 1 ms. The applications can then be executed in every 10th (100 Hz), in every 5th (500 Hz), and in every single cycle (1000 Hz).

2.5 Benefits

The concepts of the component framework presented in this section reduces the drawbacks listed in Section 1.2. The static scheduling approach of our framework improves the *predictability* of the real-time system compared to dynamic scheduling approaches with priority-based scheduling (cf. Liu, Section 4.4 [8]). Its simplified programming model lets developers focus on the algorithms in the blocks instead of priorities and synchronization mechanisms. It has been shown empirically that the *implementation effort* can be reduced significantly by providing an appropriate programming model [11].

Through the explicit representation of control flow and data flow, blocks become more *reusable* because they can be developed independently of the context in which they are used. Furthermore, programmers can rely on the robust communication protocol between blocks and do not have to implement a protocol themselves.

We have shown that blocks are executed sequentially. As we will show in Section 3, context switches can be avoided when sacrificing component separation. In this case, *OS overhead* will be reduced. See Section 5 for a discussion on how this overhead could be reduced while still maintaining component separation.

Furthermore, system engineers have the freedom to adjust the *communication overhead* depending on their respective priority for performance and safety. This will be discussed in the subsequent section.

3 Implementation

Besides the conceptual separation of concerns by using components on the code level, our framework allows for the physical enforcement of such separation at runtime. By assigning components to distinct address spaces, a defect in one component cannot affect arbitrary other components by corrupting their memory. As an example, on POSIX-based systems, components can run as individual processes and the channels between components can be implemented using message passing.

However, this increased level of safety comes at a price: while blocks in the same address space can be executed without overhead, blocks in different address spaces require additional context and address space switches. Whereas components in the same address space can communicate via light-weight mechanisms such as shared memory, components in different address spaces must employ indirect mechanisms such as message passing or sockets for communicating with each other.[3]

To allow system engineers to make a trade-off between performance and safety, our framework can be configured at compile time to enable or disable physical separation while still having the same conceptual separation. Therefore, systems with a high level of safety can be built by implementing physical separation of all components. For such systems, more powerful and thus more expensive CPUs have to be used. If in contrast the price of a system should be kept as low as possible, engineers can waive increased safety and build a system using a low-cost CPU without physical separation of components.

Consequently, we implemented two versions of our framework based on POSIX. In the *high-performance* (or *fast*) implementation, the framework and all components are executed in the same address space and shared memory is used for inter-component communication. In the *high-safety* (or *safe*) implementation, the framework and each component run as processes in separate address spaces and message queues are used for communication. In Section 4, we compare the performance between these two implementations.

In order to foster reusability of the framework and the components, maximal platform independence has been a major design decision. In fact, both versions of our framework, high-performance and high-safety, share an extensive common code base except for a very thin platform layer. It requires C++ wrapper classes and class templates for only a few platform-specific mechanisms:

- *Timers* allow the framework to perform its execution cycles and watchdog control nodes to terminate blocks.
- *Data Transmission* (e.g., shared memory or message queue) is required for implementing channels.
- *Synchronization* (e.g., mutex, semaphore, or message queue) is used for initiating the execution of blocks and notify the framework of their completion.

4 Performance Measurement

In the following, we compare the performance of the high-safety implementation and the high-performance implementation.

The hardware used for the measurements is a PC with an Intel Core2 Duo CPU E6550 running at 2.33 GHz. The Linux kernel (2.6.31-9-rt) was instructed to only use one CPU core. The hardware used for the measurements is certainly more powerful than current embedded hardware. However, the measurements will provide valuable information on the factor by which performance decreases in the high-safety implementation.

Figure 6 shows the setup for measuring. In this setup, two blocks b1 and b2, which belong to different components, communicate with each other. In each cycle, b1 gets the current time from the system and sends the timestamp to b2. In Figure 6, two numbered dotted arrows indicate the following two measurements:

1. **Channel transmission time**: The time it takes for data to be sent across a channel. In the high-performance implementation, a channel is simply a location in memory. In the high-safety implementation, a channel is a message queue provided by the OS.

2. **Block control**: The time it takes for the framework to start the execution of a block and regain control after its execution. The high-performance implementation directly calls functions whereas the high-safety implementation relies on message queues.

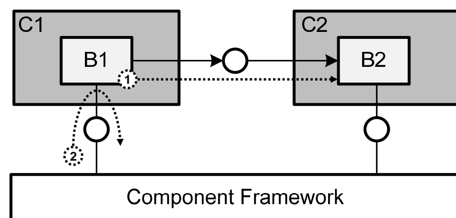

FIGURE 6: *Performance measurement setup*

Table 1 lists the results of our performance measurements. Each measurement is the average result of hundreds of measurements. The tolerance of best case and worst case is about 10% to 15%.

[3] On the other hand, research indicates that message passing can be faster than shared memory on multi-core machines [1].

	“fast”	“safe”
(1) Channel transmission	0.02 µs	1.97 µs
(2) Block control	0.46 µs	6.79 µs
Sum	0.48 µs	8.76 µs

TABLE 1: *Performance measurement results*

In total, it takes the framework about 0.48 µs to schedule a block and a channel in the high-performance implementation and 8.76 µs in the high-safety implementation. In measurements performed by Li et al. [7] on a comparable system, the average context switch was around 3.8 µs. This indicates that most of the overhead of a safe implementation is caused by context switches and not by the feature that is actually required for safety, address space separation.

5 Conclusions

We have presented a component-based software framework that enforces structurization of cyclic real-time control software systems vertically and horizontally. By decoupling different aspects, such as application logic, control flow, or communication, our approach can be expected to simplify the construction of complex control systems and to reduce the implementation effort.

In addition, static scheduling and non-preemptible execution of function blocks increase the system's determinism and thus its predictability compared to interleaved execution of threads and dynamic scheduling. Moreover, our approach allows system engineers to adjust the system to their needs if safety is of lesser concern than cost because the number of context switches and thus system overhead can be reduced.

We have shown how the abstractions offered by the framework can be implemented on RTLinux and provided performance measurements for two different implementations.

5.1 Discussion

In domains such as power and automation systems, components are required to be separated from each other at runtime to prevent the propagation of faults across component boundaries. We have shown that it is feasible to construct systems that are *either* safe *or* fast.

The prevailing concept of processes and threads, however, makes the construction of systems that are safe *and* fast difficult. We argue that a small and easy-to-implement modification of this concept will overcome this limitation: Instead of running a thread in the same process, i. e., address space, we propose to allow threads to change the address space during runtime.

With this modification, we could statically schedule the execution of blocks in one thread per CPU core. During execution of this schedule, the thread would enter and leave address spaces in correspondence to the blocks' components. An obvious advantage is the lack of context switches because blocks always run to completion and because they leave the stack empty after execution.

Moreover, inter-process communication could be implemented efficiently by using shared memory without synchronization mechanisms. Imagine two blocks A and B in different components that are connected by a channel such that A sends data to B. Since B only gets started after A finished its execution, B will always read consistent data from the channel/shared memory. This is still true if the same thread is executing A and B, or if A and B are executed on different cores.

Note that it is also possible to implement an operating system based on our component/block paradigm instead of the process/thread paradigm. We currently consider this idea less feasible. The main reason is that in cyclic control systems there are also sporadic tasks without real-time requirements, e. g., an FTP server. Such low-priority tasks run in the slack time of the real-time cycle, and typically their execution requires the slack time of more than one cycle. They therefore need to be preemptible and the operating system would have to provide some preemption mechanism similar to the one in the process/thread paradigm.

5.2 Future Work

We see considerable potential for automatic tools that can assist in system design. For instance, component boundaries do not have to be drawn arbitrarily. Instead, an automatic tool can formally derive component boundaries according to logical constraints. In our example, the three components have to exist because the two protection functions have to be separated: If one of them fails the other one is not affected. In addition, the sample manager needs to be separate because it must not be affected by faults from either protection function in order to still be

able to serve the other one. However, if there is only one protection function it can be merged with the sample manager because a fault in any block renders the whole system dysfunctional.

In Section 2.4, we showed how the system schedule maps all blocks to be executed onto the same CPU core. This approach can be extended to *multiple cores* by providing one mapping for each core such that every block is statically assigned to one core. This allows for executing our framework in multi-core, multi-CPU, or even distributed scenarios. Synchronization between the different cores is simplified to well-defined synchronization points because the dependencies between the blocks are explicitly specified in the application schedules.

The dispatcher in our framework can be extended such that blocks are not only executed in sequence, but at precise points in time. This is an effective means for reducing the system's *jitter.*

References

[1] A. Baumann, P. Barham, P. E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A New OS Architecture for Scalable Multicore Systems," in *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles.* New York, NY, USA: ACM, 2009, pp. 29–44.

[2] M. Büchi and W. Weck, "A Plea for Grey-Box Components", in *Workshop on Foundations of Component-Based Systems, (FoCBS '97)*, Zrich, 1997.

[3] IEC 61850. Communication networks and systems in substations. International Eletrotechnical Commission Standard.

[4] H. Kopetz, "The component-based design of large distributed real-time systems," *Control Engineering Practice*, vol. 6, no. 1, pp. 53–60, 1998.

[5] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, "CAmkES: A Component Model for Secure Microkernel-based Embedded Systems," *Journal of Systems and Software*, vol. 80, no. 5, pp. 687–699, 2007.

[6] E. Lee and D. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, vol. 36, no. 1, pp. 24–35, 1987.

[7] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch", *Proceedings of the 2007 workshop on Experimental computer science*, Article 2, San Diego.

[8] J. W. S. Liu, "Real-Time Systems", Prentice-Hall, New Jersey, 2000, ISBN 0-13-099651-3.

[9] C. D. Locke, "Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives," *Journal of Real-Time Systems*, no. 4, pp. 37–53, 1992.

[10] M. Naik, C.-S. Park, K. Sen, and D. Gay, "Effective Static Deadlock Detection", ICSE'09.

[11] L. Prechelt, "An Empirical Comparison of Seven Programming Languages", IEEE Computer vol. 33, no. 10, pp. 23–29, 2000.

[12] U. Rastofer and F. Bellosa, "Distributed component-based software engineering for distributed embedded real-time systems," *IEE Proceedings Software*, vol. 148, no. 3, pp. 99–103, 2001.

[13] C. B. Speedy, R. F. Brown, and G. C. Goodwin, "Control Theory: Identification and Optimal Control", Oliver & Boyd, Edinburgh, 1970.

[14] M. Wahler, S. Richter, S. Kumar, and M. Oriol, "Non-disruptive Large-scale Component Updates for Real-Time Controllers," in *Third Workshop on Hot Topics in Software Upgrades (HotSWUp'11)*, 2011.

[15] S. Wang, S. Rho, Z. Mai, R. Bettati, and W. Zhao, "Real-time component-based systems," in *Proceedings IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, San Francisco, CA, USA, 7-10 March 2005.