FASA: A Scalable Software Framework for Distributed Control Systems

Manuel Oriol, Robin Steiger, Sascha Stoeter, Egemen Vardar, Michael Wahler
ABB Corporate Research, Industrial Software Systems
Segelhofstrasse 1 K
Baden-Dättwil, Switzerland
{firstname.lastname}@ch.abb.com

Heiko Koziolek
ABB Corporate Research, Industrial Software Systems
Wallstadter Straße 59
Ladenburg, Germany
heiko.koziolek@de.abb.com

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

ABSTRACT

Building a distributed control application is currently performed ad hoc: it consists of building each application part as an independent program and connecting them through a communication layer. With the pervasiveness of multicore chips, future generations of controllers will include a variable number of cores and hosts, making such a static development process obsolete. To solve this issue, the FASA component framework for distributed control systems computes a deployment of the components onto the available computation resources (cores and hosts) and a static schedule for their execution. Depending on the given deployment, FASA chooses a suitable communication protocol for each pair of connected components. This makes FASA a scalable software architecture for heterogeneous distributed control systems. This article presents the framework, as well as a validation of FASA using a case study of a distributed control system comprising both multi-core and single-core processors.

1. INTRODUCTION

Control applications are programs that handle industrial systems. Such applications have strong constraints on their behavior to work in real-time. In industry, such applications are typically coded in an ad-hoc fashion to ensure that resources are used optimally. Such applications are always pushed to their limits to provide more functionalities and to handle the ever growing amount of information to process.

The need for more computing power inevitably leads application developers to consider distributed control systems (DCS) and multicore systems. Writing monolithical programs is then hardly an option and more structured models need to be used to handle the additional complexity of running the application over a network of multicore systems. This however should not come at the expense of a high runtime overhead and should ease the deployment of the system over the available hardware.

The present article details the FASA component infrastructure for the building of distributed real-time systems. Control application developers code FASA components as regular C++ classes. The FASA framework then automatically compiles the code and computes in a static way the initial deployment plan, the schedule, and the best communication protocol to use between components depending on their relative deployment. These abstraction mechanisms separate application *development* and application *deployment*. Furthermore, application distribution is completely transparent for the application developer. Components of the same application can be executed on a single CPU core, on multiple cores, or distributed over several physical controllers. This makes FASA flexible and scalable. The system then runs and meets real-time requirements calculated statically. It can also be updated at runtime by loading new versions of the components and deploying them on the fly, using precalculated new scheduling [20].

We validate our approach using a DCS case-study with multicore and single-core processors. Our experiments reports the execution times reached using FASA.

The contributions of this article are: 1) FASA uses a constraint solver for computing a deployment and schedule for

real-time control applications and arbitrary control system topologies. 2) FASA provides different mechanisms for inter-component communication and chooses the most efficient one for each pair of communicating components. 3) It shows that this can be achieved in a reliable and efficient way.

This article is structured as follows. Section 2 presents our running example. Section 3 explains the general FASA approach. Section 4 presents the implementation. Section 5 evaluates the FASA approach on a real-world example. Section 6 details the related work. We eventually conclude in Section 7 and give an outlook on future work.

2. MOTIVATING EXAMPLE

To make it more concrete, this article uses a cascaded control loop, which controls a physical process, as a running example. In this example, the process comprises a valve and two sensors measuring temperature and pressure. The valve is used to control the pressure and temperature read by the sensors in a tank. This application is a typical example of a feedback loop in industrial control systems. We illustrate the application in Figure 1.

This application is executed cyclically at a given control frequency. Typical control frequencies for software-based control systems range between 1 kHz and 1 Hz (with corresponding cycle durations of 1 ms to 1 s).

In the example, new values for the three input variables (Temperature, Pressure, and Track) are acquired at the beginning of each cycle. The input values of type RealIO are processed by components of type AnalogInCC, which apply low-pass filters to the input signal. The actual control algorithm is implemented by two instances of proportional-integral-derivative (PID) control components [1], which compute their output based on the delta between a given set point and some desired value. The output of the second PID component is fed into an AnalogOutCC component, which prepares a value for output to some I/O interface and allows for plugging-in of error handlers.

The output of the application triggers a valve to open or close. Based on the new setting of the valve, the temperature and/or the pressure of the physical process change. In the subsequent execution cycle, new sensor values are acquired and an updated setting for the valve is computed. A typical plant contains numerous such control applications to control and synchronize actuators such as valves, pumps, conveyor belts, grapplers, or robots.

3. APPROACH

This section presents the different aspects of our approach based on the running example from Section 2.

3.1 Component Framework

The FASA component framework was first introduced [15] focusing on its dynamic updating features. FASA provides a framework to develop, build and run cyclic control applications defined as a set of *components*. Components consists of one of more *blocks*, which are the basic units of execution. *Blocks* can have *ports* to receive and send data objects from/to other blocks. *Channel*s are used to connect an output port of a block to an input port of another block. Channels are one-to-one only, meaning that at most one block can write to it, and at most one block can read from it. All channels are unidirectional. In conjunction with the linearity of blocks this implies that all systems constructed from channels and blocks can be described by a directed graph with blocks as nodes and channels as edges. Channels are responsible for data transport only. They are stateless and do not check the data they transmit. This is the responsibility of the block that reads from a channel. The data type of both input port and output port must be same. The framework allows a block to check whether or not a port is connected through a channel. If connected through a channel, the framework ensures that the data object is written to the input port of the target block before the execution of the target block begins.

In the example in Figure 1, all two-dimensional entities such as AnalogInCC are blocks. The input ports of each block are shown on the left side (e.g., AnalogInput) and the output ports are shown on the right side of each block (e.g., Out). Channels are shown as black lines between two ports. Each block is wrapped by a component of the same name. In order to avoid cluttering, components are not displayed in Figure 1.

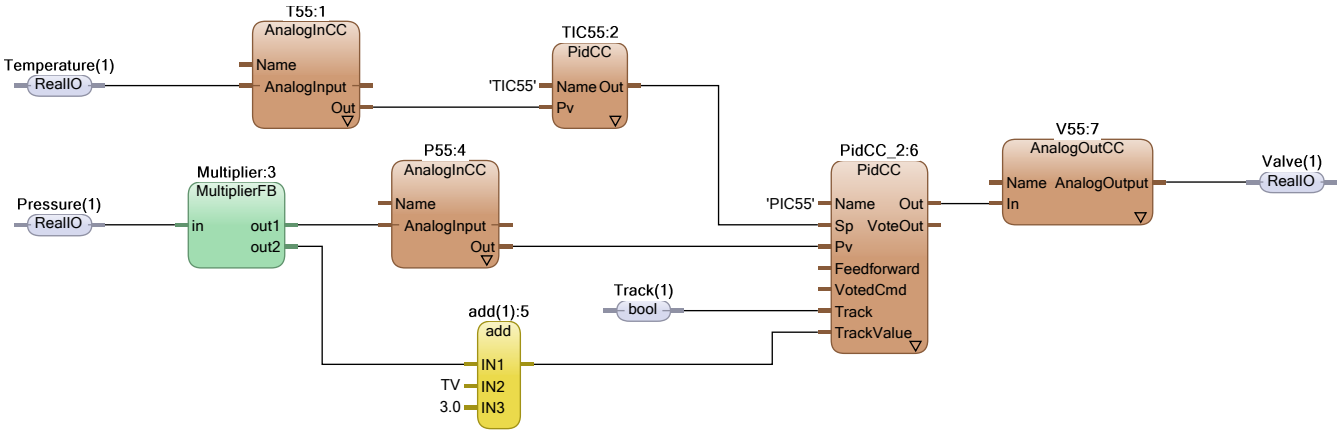

Figure 1: Running example: a cascaded control loop.

3.2 Hosts and Periods

An application can be deployed on one or more hosts. Each FASA host runs one instance of the FASA kernel. Such host is either a computer running a real-time operating system or one of its cores. In a complex system with many hosts running several applications, the blocks of more than one application may run on the same host and the blocks of one application may be distributed across multiple hosts. To achieve this, schedules for each host must be computed based on the real-time requirements of each application. Each of these concepts are described in this subsection.

FASA requires the operating system to support fixed-priority preemptive scheduling. The FASA process runs at the highest priority as a user level process. FASA then executes blocks in an predetermined order determined by a static schedule. Once all the blocks in the schedule complete execution, the execution is repeated. This cycle continues until the FASA process is stopped or a new schedule is provided. FASA uses a fixed time period to run a cycle. If all the blocks complete execution before the cycle time is over, then the scheduler sleeps until the period ends. Figure 2 illustrates periods on a FASA host. Two consecutive execution cycles are shown with period t. Please note that the actual execution time of a block may vary during cycles (within limits) but the next execution sequence starts exactly at t units after its last start.

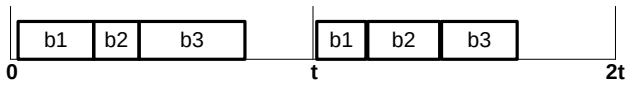

Figure 2: Period on a FASA host.

Blocks on different hosts communicate transparently either through message queues — for hosts on different cores of the same computer – or through network proxies — for hosts on different computers.

3.3 Computing Deployment and Schedule

Deploying a set of FASA applications implies first solving a distributed scheduling problem. This problem is solved offline (i.e., before the system is started) and computes a static schedule for each FASA host. The scheduling problem consists of two parts: (1) Finding an allocation of components to hosts, and (2) assigning time intervals to function blocks. The allocation must take into account the memory requirements of the components and the available memory on the respective hosts. The time intervals must be non-overlapping for blocks which are part of components mapped to the same host. The length of a time interval is always equal to the worst-case execution time (WCET) of a block: it is assumed that blocks always require their WCET. This leads to non-optimal usage of the available time in each cycle, but significantly reduces jitter.

For illustration, we use a system with three hosts and five applications. Host 1 has a period of t, Host 2 has a period of $2t$ and the Host 3 has a period of $4t$. The system has five applications defined as A with three blocks {a1,a2,a3} at a cycle time t, B={b1, b2, b3} at $2t$, C={c1, c2, c3, c4} at $4t$, D={d1, d2, d3} at $2t$, and E={e1, e2, e3} at $4t$. Because of I/O constraints, Block $b3$ (or, to be precise, its component) and Block $c4$ must be deployed on Host 1 whereas Block $c3$ must run on Host 2.

Figure 3 illustrates an example deployment and schedule. In this example, all blocks of Application A are scheduled on Host 1. Since all the blocks of B cannot be scheduled on Host 1 (there is not enough time left in the cycle after scheduling all the A blocks), $b1$ and $b2$ are scheduled on Host 2 and $b3$ is scheduled on Host 1. Application D can be scheduled on either Host 1 or Host 2. Since the worst case execution times of its blocks are too large to be scheduled on Host 1, they are scheduled on Host 2. Since there is not enough time left in the periods of both Host 1 and Host 2 after scheduling other applications, $c1$ and $c2$ are scheduled on Host 3. Block $c3$ is scheduled on Host 2 (repeating every two cycles of the host) and $c4$ is scheduled on Host 1 repeating every four cycles. The blocks of Application E are all scheduled on Host 3.

As explained in Section 3.2, different mechanisms for channel communication have to be used depending on how the applications are distributed. Since channels are written to at the end of the execution of a block and read from at the beginning, the execution times for the required channel mechanism has to be added to the WCET of the block. In other words, the actual value of the WCET for a block depends on the allocation of components to hosts.

The distributed scheduling problem in FASA is close to a typical job shop problem, which makes it NP-hard and impractical to solve manually even on medium-sized systems [12]. An automated solution is required to make FASA scalable. The FASA implementation uses a constraint programming (CP) approach to compute a system schedule (Section 4.3).

4. IMPLEMENTATION

This section presents the implementation of the abstraction mechanisms available to programmers, their runtime implementation, and what input the constraint programming model.

4.1 Abstractions for Component Developers

FASA provides a high level of abstraction for component developers by hiding all the details of the actual communication between components. A component developer only needs to implement the application blocks.

FASA is implemented in C++ and uses regular object-oriented constructs. Most methods are inherited from the superclass representing blocks and developers only need to override two methods to create a fully functional application block:

construct() : Contains the initializer of the block and is called automatically at block instantiation. Ports and class members are initialized in this function.

run() : This function contains the code for the functionality of the block. It is called every time the block is executed, usually once in each cycle.

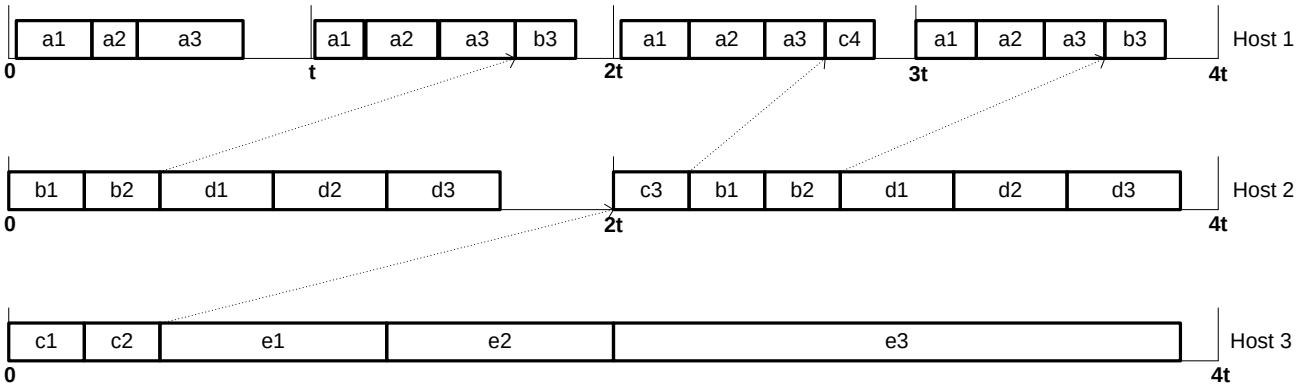

Figure 3: An exemplary distribution and schedule.

Other methods can also be overridden (destructors, handle_incoming_exceptions...) but do not need to be.

As mentioned in Section 3.1, communication between blocks is carried out through ports. Each port provides a pointer to a data area, from/to which a block can read/write data. For each input or output data, the developer needs to create a typed in or out port (for example, an output port of type double), and register it in the construct() function. Read and write operations are performed using the overloaded operator '*'. This operator is used both for reading and writing values as illustrated in the code snippet in Listing 1.

```
{
    In_Port_Typed <double> AnalogInput;
    Out_Port_Typed <double> out;

    void run ()
    {
        //Read the input from the data area
        //pointed to by 'AnalogInput' input port
        double in = *AnalogInput;

        //Write the output to the data area
        //pointed to by 'out' output port
        if (in > 0)
            *out = in * 1.5;
        else
            *out = 0;
    }

    void construct ()
    {
      ISOGRAFT_REGISTER_IN_PORT (AnalogInput);
      ISOGRAFT_REGISTER_OUT_PORT (out);
    }
};
```

Listing 1: Example code of a block.

How the data is sent or received is hidden from the developer, and encapsulated by FASA channels. For the developer, a FASA channel is only a black box operating between two blocks to send data from the sender to the receiver. In the next section we present the main implementations of channels.

4.2 Channel Implementation

Two blocks connected through a channel can either be executed on the same core, on different cores of the same CPU, or on two different hosts. For each channel in an application, the appropriate communication mechanism is determined in the deployment process. This is completely transparent to the application developer. Depending on their relative location, FASA provides three different mechanisms for channel communication.

4.2.1 Blocks on the same core

Blocks residing in the same address space on the same physical host communicate with each other via shared memory. Communication between two blocks is established by making them have pointers to the same memory area. The sender block writes its output data to a memory block, and the receiver block reads its input data from the same memory block. Since FASA channels are unidirectional, only one block has write access to a given memory area, while the corresponding block only has read access (when calling '*', both writing on output channels and reading on input channels are forbidden). Synchronization is provided through the schedule which ensures that no *read* is performed before the correspnding *write* has been performed. Communication with shared memory is depicted in Figure 5, where read and write operations are carried out by using the overloaded operator '*' as shown in Listing 1 .

4.2.2 Blocks on different cores

Message queues are used to send data between blocks running on the same physical host, but on different cores. As shown in Section 3.2, each of the individual cores run an instance of the kernel and components that communicate with each other can reside in different processes. Logically, this is the same as running the application on multiple machines. Since the application is running on the same host, inter-process communication mechanisms can be used to perform communication between the kernel instances running on different cores. Our implementation uses message queues. While shared memory could also be used, using message queues also provides an additional synchronization mechanism between blocks.

As is the case for shared memory mechanism, only one com-

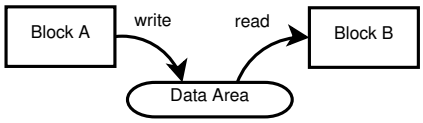

Figure 5: Communication via shared memory.

ponent block has write permissions on a given message queue. The output written to data area is wrapped into a message and sent via the message queue. On the receiver side, that message is received from the queue and stored in the data area pointed to by the corresponding input port. Communication through message queues is illustrated in Figure 6.

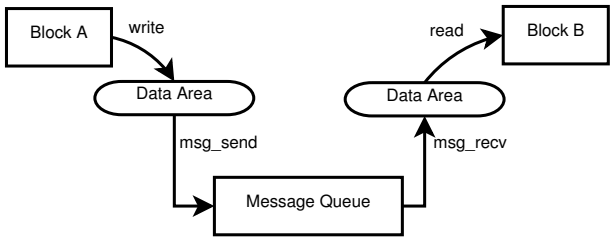

Figure 6: Communication via message queue.

FASA's implementation of message queues uses the System V message queue mechanism. Our message passing model employs a mixed use of synchronous and asynchronous models. Messages are sent asynchronously, meaning that the sender block execution continues after sending a message (unless there is not enough space in the queue for sending a new message, which does not occur in normal operation). Messages are received synchronously: the execution gets blocked while a message is not available in the message queue to make sure that the block does not use uninitialized or stale input data.

4.2.3 *Blocks on different physical hosts*

FASA provides network proxy components to establish communication between blocks running on different physical hosts connected through the network. In order to send an output, a send proxy block is created on the sender side, and a receive proxy block is created on the receiver side. A send proxy block has one input port and a receive proxy block has one output port. The blocks are scheduled to run as any other block in the application. The send proxy receives the data to be sent from its input port, and sends it through TCP or UDP connection to the corresponding receive proxy block, depicted in Figure 4. Upon receiving the data, the receive proxy outputs it to the corresponding application block. The channels also act as a synchronization mechanism.

In Section 5 we will show how these mechanisms are used when our example application (Figure 1) is distributed in different ways and provide detailed performance measurement results.

4.3 Constraint Programming model

We use constraint programming (CP) as a tool for solving the distributed scheduling problem introduced in Section 3.3. This section gives a brief introduction to the CP model for FASA. It focuses on the scheduling aspects of the model and does not enter into further details about peripheral topics such as memory requirements.

A constraint satisfaction problem (CSP) is a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where $\mathcal{X}$ is a finite set of variables, $\mathcal{D}$ is a set of finite domains associated with these variables, and $\mathcal{C}$ is a finite set of constraints. The constraints determine which combinations of value assignments are acceptable and which are not [19]. All domains are finite sets of integers in the model described below. Our implementation uses JaCoP [9] as the underlying CP solver.

Allocation variables. As stated in Section 3.2 a FASA host corresponds to a specific core on a specific physical host. Let N_m be the number of machines and let N_k be the (maximum) number of cores on a machine. For each component c we create the variables $TargetHost_c$, $TargetNode_c$ and $TargetCore_c$, denoting the FASA host, the physical machine and the core on which the component will be deployed.

$\forall c \in$ Components:

$$TargetHost_c \in [0, N_m \times N_k - 1]$$
$$TargetNode_c \in [0, N_m - 1]$$
$$TargetCore_c \in [0, N_k - 1]$$

$$TargetHost_c = TargetCore_c + \quad N_k \times TargetNode_c \tag{1}$$

Because of the constraint in Equation 1 an allocation is com-

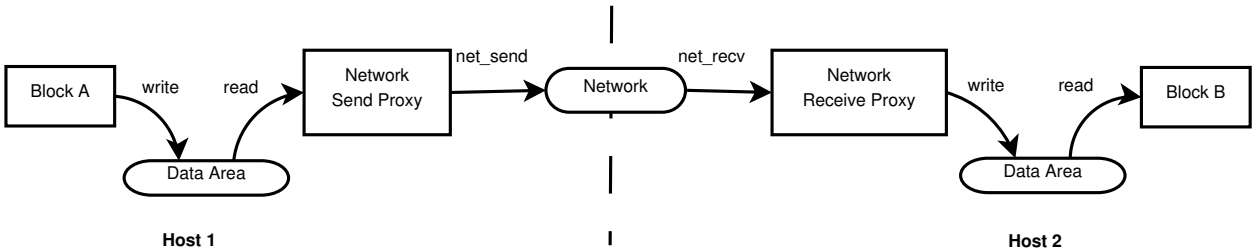

Figure 4: Communication via network proxies.

pletely defined by either assigning a value to $TargetHost_c$ or to both $TargetNode_c$ and $TargetCore_c$, for each component c. This redundancy is useful because in different situations we will be interested in different aspects of the allocation. In practice the allocation is often restricted by a number of constraints: some nodes may have less than N_k cores or some components cannot be mapped to certain nodes. These constraints are easy to express in CP using the variables above.

Scheduling variables. The schedule is defined by assigning a time interval to each block. The simplest way to describe a time interval in CP is to use three variables to denote the starting time, the end time and the duration of each block. Their domains are initialized such that the described time interval is within the bounds $[0, \text{Period(b)}]$, where Period(b) denotes the cycle length of the block. The three variables also have to fulfill the constraint in Equation 2.

$\forall b \in$ Blocks:

$$StartTime_b, EndTime_b, Duration_b \in [0, \text{Period(b)}]^3$$

$$StartTime_b + Duration_b = EndTime_b \tag{2}$$

Channel variables. It is also necessary to know which channels require network proxies and which channels use message queues. Let (s, r) be a channel where s is the sending and r is the receiving block. For convenience we use s and r to denote the components to which these blocks belong. For each channel (s, r) the boolean variables $UseNetwork_{(s,r)}$ and $UseMessage_{(s,r)}$ are introduced and constrained as follows:

$$UseNetwork_{(s,r)} = \left(TargetNode_s \overset{?}{\neq} TargetNode_r\right) \tag{3}$$

$$UseMessage_{(s,r)} = \neg UseNetwork_{(s,r)} \wedge \left(TargetCore_s \overset{?}{\neq} TargetCore_r\right) \tag{4}$$

Boolean variables have the initial domain $[0, 1]$. The constraint in Equation 3 assigns the truth value of a logical constraint, checking for inequality, to a boolean variable. This constraint is called reification and can be found in most CP solvers. The logic in Equation 4 expresses that message queues are only used when the blocks of the channel are mapped to a different core on the same host.

Proxy counters. The implementation of channels makes use of several network proxy blocks, which send or receive data on behalf of the executing block. There are four types of proxies: Two for sending and receiving data from the network, and two for sending and receiving messages from a message queue. Before we can compute the WCET of a block, we need to know how many proxies of each type will be used by the block. Notationally, we introduce the set Proxies = {net_send, net_recv, msg_send, msg_recv} containing a constant for each type of proxy. The number of proxies $NumProxies_b^p$ can be computed as follows:

$\forall p \in$ Proxies and $\forall b \in$ Blocks

$$NumProxies_b^{\text{NET\_SEND}} = \sum_{(b,r') \in \text{Channels}} UseNetwork_{(b,r')}$$

$$NumProxies_b^{\text{NET\_RECV}} = \sum_{(s',b) \in \text{Channels}} UseNetwork_{(s',b)}$$

$$NumProxies_b^{\text{MSG\_SEND}} = \sum_{(b,r') \in \text{Channels}} UseMessage_{(b,r')}$$

$$NumProxies_b^{\text{MSG\_RECV}} = \sum_{(s',b) \in \text{Channels}} UseMessage_{(s',b)}$$

Block duration. FASA computes the duration of a block in two steps. First, FASA computes the actual WCET, including channel proxies, for a particular block b deployed on a particular host h by adding all WCETs for proxies to the WCET of a block. The WCET for each individual block can be computed using code analysis or experiments. The $WCET(\cdot, \cdot)$ predicate below returns the measured WCETs as constants.

$\forall b \in$ Blocks and $\forall h \in$ Hosts:

$$DeployedWCET_b^h = \text{WCET(b,h)} + \sum_{p' \in \text{Proxies}} \text{WCET(p',h)} \times NumProxies_b^{p'}$$

In a second step, FASA computes the block duration using the WCET per host and the host assignment of the component containing the block. In CP, a partial function (or lookup table) can be modeled using the Element constraint. This constraint expresses the relation $Result = Table[Index]$, where $Result$ and $Index$ are integer variables and $Table$ is an array of integer variables. When all variables are fixed, $Result$ is constrained to be equal to the table entry chosen by $Index$. We constrain the variable $Duration_b$ as follows:

$\forall b \in$ Blocks

$$Duration_b = \text{Element} \begin{cases} TargetHost_b & \text{The index} \\ DeployedWCET_b^h & \text{A table } \forall h \end{cases}$$

Overlap constraint. Finally, the time intervals for blocks should do not overlap. FASA models this using a single instance of the `Diff2` constraint. In JaCoP, `Diff2` is a specialized constraint for solving overlap problems with objects (rectangles) in two dimensions. This is an excellent match to solve the overlap constraints in FASA because blocks are scheduled in two dimensions: time and host. Formally, the `Diff2` constraint is given a list of N rectangles, where each rectangle is defined by four integer variables denoting the length and the origin in both dimensions. A block is scheduled at regular intervals given by its application period. Since a schedule is computed for the hyperperiod, the same block might occur multiple times during this interval. Let the CP variable $RepeatTime_b^i$ denote the start time of the ith occurence of block b. A block must repeat at a regular interval to ensure that the schedule is jitter-free. For every block b and every occurence i of b FASA defines the

rectangle specified in Equation 5.

$$BlockRectangle_b^i = \begin{cases} RepeatTime_b^i & \text{Origin X} \\ Duration_b & \text{Length X} \\ TargetHost_b & \text{Origin Y} \\ 1 & \text{Length Y} \end{cases} \quad (5)$$

$$\text{where } b \in \text{Blocks},\ i \in \left[1, \frac{\text{HyperPeriod}}{\text{Period(b)}}\right]$$

$$\text{and } RepeatTime_b^i = StartTime_b + Period(b) \times (i-1)$$

In Equation 5, the X-axis is the timeline and the Y-axis the host selection. The length in Y-direction is 1 such that a block can only be located on one host. The `Diff2` constraint for FASA is created with a list of all block rectangles given as argument.

Whereas the focus of this paper is on the scalability of the FASA runtime framework we were interested in the performance of our constraint solving solution because combining scheduling and deployment problems are notoriously hard for constraint solvers [12]. To this end we have created a benchmark system consisting of 20 applications (each of which comprising seven blocks organized in three components) and 10 hosts. Thus, the constraint solver needs to compute a mapping for 60 components onto 10 hosts and a schedule for 140 blocks.

The results of these experiments confirm that the problem we are solving is hard. Independent runs of the constraint solver on the same input either produce a result within 2–4 seconds or do not terminate within five minutes, which suggests that the initial choice of deployment (which is random in our approach) is crucial. Future research in this topic should look into how heuristics for fixing parts of the initial deployment can keep the schedule computation within reasonable boundaries.

5. EVALUATION

This section presents experiments which test the scalability of the FASA architecture. These experiments consist mainly in deploying the example control application (Figure 1) on different architectures. We assess the quality of each deployment by measuring and comparing the individual execution times of each deployment. For our experiments we use four different scenarios with different setups of the control system:

- 1× **(or single-core):** One controller with one single-core CPU.
- 2× **(or dual-core):** One controller with two CPU cores.
- 4× **(or quad-core):** One controller with four CPU Cores.
- 1 + 1 **(or distributed):** Two controllers with one single-core CPU. We simulate this scenario using two cores of the same controller and socket communication.

Depending on the setup, the input for the constraint solver changes as described in Section 3.3. The output of the constraint solver is a deployment of the components of the application on the available (physical or virtual) hosts and a linear schedule for each host.

In single-core (1×), the solver deploys all components on only one core and computes a schedule in which all components are executed linearly. The computed solution is visualized in Figure 7.

In dual-core (2×), the components are deployed on two cores of the same CPU. Some components are executed in parallel as can be seen in Figure 8. For communication between Temperature PID Controller and PID Controller, a message queue (cf. Section 3.2) is used. If the control flow on Core 2 reaches PID Controller before Temperature PID Controller on Core 1 has terminated, the execution on Core 2 is blocked until a message is received.

In quad-core (4×), the components are deployed on four cores of the same CPU. Some components are executed in parallel as can be seen in Figure 9. Whereas this scenario is conceptually similar to Scenario 2×, it will be used in Section 5.1 to measure the benefits versus the overhead caused by an increased level of parallelism.

When distributed (1 + 1), the components are deployed on two cores of CPUs in different systems connected through a network. Some components are executed in parallel. As can be seen in Figure 10, two additional components—network proxies for sending and receiving data—are automatically added to the schedule.

5.1 Performance Measurements and Interpretation

The machine used for the tests contains a QorIQ P4080 processor, which hosts eight PowerPC cores operating at 1.2 GHz. The computer runs under Linux, version 2.6.34.6. The time required for a FASA channel based on shared memory, message queues, and network proxies is 7 µs, 13 µs, and 60 µs respectively.

For each scenario we measure the execution time of the whole application by taking a timestamp at the beginning and at the end of each execution cycle. The difference between these timestamps is the execution time of the application. Essentially, the execution time of the application corresponds to the execution time of one kernel instance in each scenario. For instance, in scenario 2×, the execution time corresponds to the execution time of the kernel instance running on core 2 since the two instances run in parallel and PidCC_2:6 gets blocked if execution reaches there before TIC55:2 sends the output. Similarly, in scenario 4×, execution time of the application corresponds to the execution time of the kernel instance running on core 4 since PidCC_2:6 has to wait for all the inputs from other blocks in order to continue its execution.

We measured the worst, the best, and mean execution times, as well as the standard deviation across 300 cycles. We repeated this process five times, and the measurement results in Table 1 reflect the average over five runs.

Table 2 show the mean execution times of individual blocks in all scenarios. The times shown reflect the average over five runs across 300 cycles, and consist of the execution times for receiving inputs from message queues and writing them to corresponding data areas provided by the in ports, executing

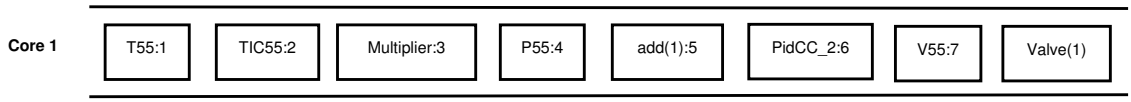

Figure 7: Scenario $1\times$: Single-core deployment.

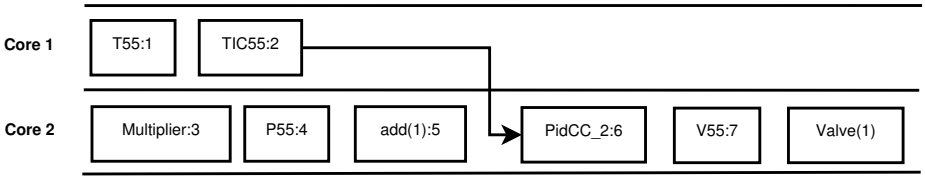

Figure 8: Scenario $2\times$: Dual-core deployment.

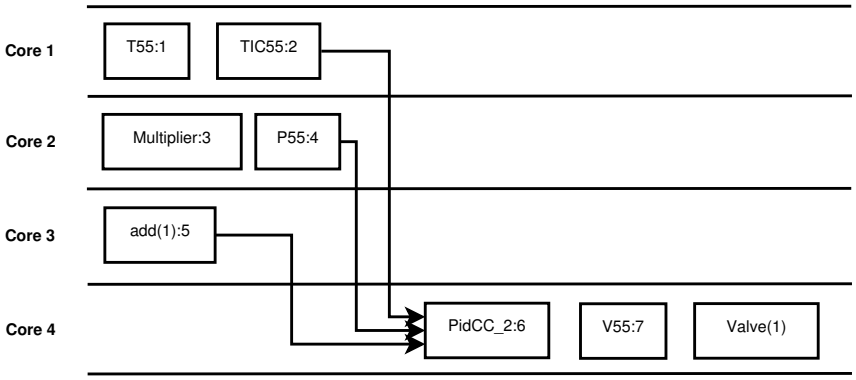

Figure 9: Scenario $4\times$: Quad-core deployment.

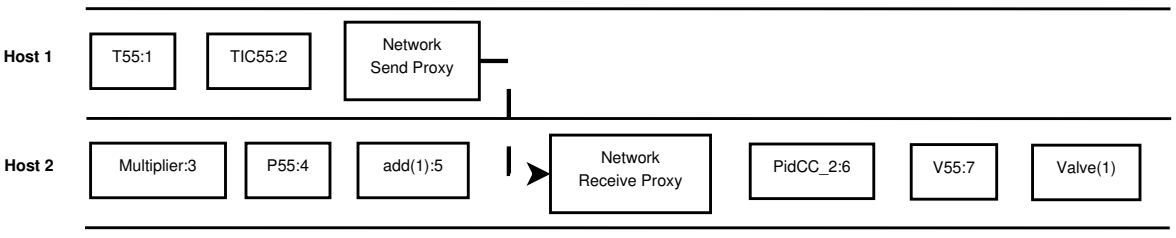

Figure 10: Scenario $1 + 1$: Physically distributed deployment.

Table 1: Application execution times in µs.

Scenario	Mean	Worst	Best	Std. Deviation
1×	182.79	198.53	175.39	4.39
2×	161.16	190.83	151.97	6.53
4×	149.74	215.82	116.82	24.69
1 + 1	180.30	243.79	167.63	14.04

the block[1], and writing the outputs to corresponding message queues. If the block has no communication with any other block via message queue, the execution time essentially corresponds to the execution time of the run() function. In Table 2, values are in bold font when they differ from the base scenario (1×). This explains the difference between the execution times of some blocks in different scenarios. For instance, PidCC_2:6 receives the inputs via shared memory in scenario 1×, whereas in scenario 2×, it receives the input from TIC55:2 block via message queue, which introduces a small overhead. In scenario 4×, since the kernel instances on all four cores are run in parallel PidCC_2:6 blocks until all inputs are available, which explains the longer execution time.

Table 2: Mean execution times of individual blocks for 1×, 2×, 4×, and 1 + 1.

Block	1×	2×	4×	1 + 1
T55:1	13.82	13.95	13.82	13.42
TIC55:2	6.22	**8.67**	8.67	5.10
P55:4	8.65	8.65	**13.37**	7.55
Multiplier:3	16.86	16.86	**21.15**	15.15
add(1):5	8.05	8.05	**35.45**	6.96
PidCC_2:6	8.97	**11.32**	**59.24**	8.47
V55:7	5.85	5.70	5.85	4.65
Valve(1)	29.47	31.32	29.47	28.48
Net_Rec				**16.33**
Net_Send				**27.93**

These measurements show that not all possible deployments are beneficial for the performance. Whereas our example application benefits from a dual-core deployment (2×) compared to single-core deployment (1×), a deployment on four cores (4×) introduces a higher communication overhead than what is saved due to parallel block execution. Several applications exist where such an overhead is acceptable. For example, such an overhead is acceptable in a control application where the calculation time is much higher than the induced overhead, or when safety depends on redundant components executing on different hosts.

6. RELATED WORK

FASA uses a constraint-solving tool to calculate schedules and assign components to hosts. A large body of knowledge exist on scheduling problems [12]. Real-time scheduling traditionally consists of heuristics-based schedulers [2, 7]. Using a constraint solver for real-time software systems is more uncommon [16, 5, 6]. The scheduling part of FASA is however more complex due to the choice of different possible communication mechanisms introduced by the multicore and distributed cases.

FASA is a component framework for networked, embedded systems that is scalable and allows for dynamic updates. To date, no other component framework contains all these characteristics. Fore example, RUNES [3] and DREAM [11] are reconfigurable, component-based middleware systems. They do not deal with scalability. Another example is CAPSULE [14], an environment to find an appropriate parallelism granularity and to map tasks with complex control and data flow to threads. This approach is on a lower abstraction level than the FASA framework and does not support runtime component updates.

Crnkovic et al. [4] provided a classification framework for software component models. Their survey also includes multiple component models for embedded systems, which for example support interface specifications with resource usages and deadlines and allow to verify safety and timing properties. Among the related component models are BlueArX, CAmkES, COMDES II, CorbaCM, IEC61131, IEC61499, PECOS, PIN, ProCom, Robocop, Rubus, and SaveCCM. None of these support dynamic updates.

We distinguish FASA from most similar approaches: CAmkES [10] builds on top of the L4 microkernel and strives for defining operating system functionality (such as file system) in a component-based way. In contrast, FASA builds on top of existing real-time operating systems and focuses on the control applications. PECOS [13] allows the specification of component-based embedded systems and provides a composition language (CoCo). It could be beneficial for FASA to use CoCo to check composition consistency rules. The PIN component model [8] allows the specification of timing properties in order to predict latencies and verify temporal properties. ProCom [17] enables component specification on multiple layers, a rich set of connectors, and resource consumption specifications. None of the component models provides special concepts or implementation aids to scale seamlessly on multi-core processors.

Finally, there are special methods to enable running embedded programs in parallel on multi-core processors. For example, Shih et al. [18] proposed a model-driven multi-core software development environment for embedded systems. Based on SysML specifications, developers can generated pattern-optimized code that can be mapped to different hardware platforms. However, these approaches usually do not target networked, component-based systems.

Our own previous work was mainly concerned with FASA's component model [15] and FASA's support for runtime updates [21, 20]. In contract, this paper focuses on FASA's support for scalable distributed and multi-core systems.

7. CONCLUSIONS

This article presents FASA, a scalable component framework for distributed control systems. FASA allows developers to write control applications with full execution transparency. FASA automatically deploys a set of control applications on the control hardware by using a constraint solver approach

[1]This corresponds to executing the run() function of the block, as explained in Section 4.1.

to define the schedule and assignment to available hosts. It supports heterogeneous environments with single-core or multi-core hosts and automatically choses the best communication protocol between blocks.

We envision several avenues for future work. An obvious extension of our approach is the inclusion of optimality constraints in the constraint solving approach. Whereas our current solution computes one or more valid deployments and schedules, additional constraints could be added that optimize the solution for criteria such as the execution time of the whole application or an evenly balanced load across all available hosts.

In our current solution the schedule for the control system is calculated based on the worst-case execution time of the blocks. This has the advantage that jitter is reduced to a minimum because each block starts at the same time in every cycle. A disadvantage of this approach is however that it is pessimistic and time is wasted in each cycle if the average execution time of the blocks executed is significantly lower than their worst-case execution time. As a solution the constraint programming model could be extended to take into account average-case execution time.

8. REFERENCES

[1] K. Ang, G. Chong, and Y. Li. PID Control System Analysis, Design, and Technology. *IEEE Transactions on Control Systems Technology*, 13(4):559–576, 2005.

[2] N. Audsley and A. Burns. Real-Time System Scheduling. Technical report, University of York, 1990.

[3] P. Costa, G. Coulson, C. Mascolo, L. Mottola, G. Picco, and S. Zachariadis. Reconfigurable Component-based Middleware for Networked Embedded Systems. *International Journal of Wireless Information Networks*, 14:149–162, 2007. 10.1007/s10776-007-0057-2.

[4] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. Chaudron. A Classification Framework for Software Component Models. *Software Engineering, IEEE Transactions on*, 37(5):593 –615, 2011.

[5] C. Ekelin and J. Jonsson. Real-Time System Constraints: Where do They Come From and Where do They Go? In *International Workshop on Real-time Constraints*, pages 53–57, 1999.

[6] C. Ekelin and J. Jonsson. Solving Embedded System Scheduling Problems Using Constraint Programming. Technical report, Dept. of Computer Engineering, Chalmers University of Technology, 2000.

[7] C. Eriksson, J. Mäki-Turja, K. Post, M. Gustafsson, J. Gustafsson, K. Sandström, and E. Brorsson. An Overview of RealTimeTalk, a Design Framework for Real-time Systems. *J. Parallel Distrib. Comput.*, 36(1):66–80, July 1996.

[8] S. Hissam. Pin Component Technology (V1. 0) and its C Interface. Technical report, DTIC Document, 2005.

[9] K. Kuchcinski and R. Szymanek. Java Constraint Programming Solver. http://jacop.osolpro.com/, 2001-2012.

[10] I. Kuz, Y. Liu, I. Gorton, and G. Heiser. CAmkES: A Component Model for Secure Microkernel-based Embedded Systems. *Journal of Systems and Software*, 80(5):687 – 699, 2007.

[11] M. Leclercq, V. Quema, and J.-B. Stefani. DREAM: A Component Framework for Constructing Resource-Aware, Configurable Middleware. *Distributed Systems Online, IEEE*, 6(9):1, 2005.

[12] M. Lombardi and M. Milano. Optimal Methods for Resource Allocation and Scheduling: a Cross-disciplinary Survey. *Constraints*, 17(1):51–85, 2012.

[13] O. Nierstrasz, G. Arevalo, S. Ducasse, R. Wuyts, A. Black, P. Mueller, C. Zeidler, T. Genssler, and R. van den Born. A Component Model for Field Devices. In J. Bishop, editor, *Component Deployment*, volume 2370 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin / Heidelberg, 2002.

[14] P. Palatin, Y. Lhuillier, and O. Temam. CAPSULE: Hardware-Assisted Parallel Execution of Component-Based Programs. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 247 –258, 2006.

[15] S. Richter, M. Wahler, and A. Kumar. A Framework for Component-Based Real-Time Control Applications. In *13th Real-Time Linux Workshop*, 2011.

[16] K. Schild and J. Würtz. Off-line Scheduling of a Real-time System. In *SAC*, pages 29–38, 1998.

[17] S. Sentilles, A. Vulgarakis, T. Bureą, J. Carlson, and I. Crnkovic. A Component Model for Control-Intensive Distributed Embedded Systems. In M. Chaudron, C. Szyperski, and R. Reussner, editors, *Component-Based Software Engineering*, volume 5282 of *Lecture Notes in Computer Science*, pages 310–317. Springer Berlin / Heidelberg, 2008.

[18] C. Shih, C.-T. Wu, C.-Y. Lin, P.-A. Hsiung, N.-L. Hsueh, C.-H. Chang, C.-S. Koong, and W. Chu. A Model-Driven Multicore Software Development Environment for Embedded System. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 2, pages 261 –268, 2009.

[19] P. Vilím. *Global Constraints in Scheduling*. PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Mathematical Logic, KTIML MFF, Universita Karlova, Malostranské náměstí 2/25, 118 00 Praha 1, Czech Republic, 2007.

[20] M. Wahler, S. Richter, S. Kumar, and M. Oriol. Non-disruptive Large-scale Component Updates for Real-time Controllers. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pages 174 –178, 2011.

[21] M. Wahler, S. Richter, and M. Oriol. Dynamic Software Updates for Real-time Systems. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, HotSWUp '09, pages 2:1–2:6, New York, NY, USA, 2009.