2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing

The RTSC: Leveraging the Migration from Event-Triggered to Time-Triggered Systems

Fabian Scheler and Wolfgang Schröder-Preikschat
Department of Computer Science 4 - Distributed Systems and Operating Systems
Friedrich–Alexander University Erlangen–Nuremberg
{scheler,wosch}@cs.fau.de

Abstract

In this paper we present a prototype of the RTSC – the Real-Time System Compiler. *The RTSC is a compiler-based tool that leverages the migration from event-triggered to time-triggered real-time systems. For this purpose, it uses an abstraction called* Atomic Basic Blocks *(ABBs) which is used to capture all relevant dependencies of the event-triggered system in a so-called ABB-graph. This ABB-graph is transformed by the RTSC and finally mapped to a statically computed schedule that could be executed by standard time-triggered real-time operating systems. Moreover, we demonstrate the applicability of our approach and the operation of our prototype by transforming the event-triggered implementation of a real-world embedded system into a trime-triggered equivalent.*

1. Introduction

At the beginning of any real-time systems project one has the choice to either go for an event-triggered or a time-triggered solution. Event-triggered systems have the advantage of being much more flexible and, thus, are much more handy when dealing with changing requirements or *uncertain knowledge* (aperiodic events for instance as their period is not known). According to [1] these were the reasons to select an event-triggered approach when building the space shuttle primary avionics system. On the other hand, time-triggered systems can be verified much easier. This is a significant advantage for any kind of dependable system. Therefore, Gagne and Sheppard made a great effort to port the F18 mission computer software to a time-triggered execution environment [2]. A significant part of this porting process had to be done manually. Thus, this was a work-intensive and also a possibly error-prone undertaking. Often the transition from an event-triggered system to a time-triggered system or vice versa is even deemed too cumbersome to be a viable option. In such cases, a redesign of the system tied with the re-implementation of large parts of the system becomes inevitable.

The reason for this problem could be found at the control flow abstraction that is associated with the event-triggered and time-triggered paradigm, respectively. While this abstraction is relatively feature rich in event-triggered systems and offers various mechanisms for blocking and non-blocking uni- and multilateral synchronisation, it is rather ascetic in time-triggered systems, which offer pure run-to-completion semantics, only. These abstractions tend to infiltrate the structure of the application and the real-time system resulting in implementations that are closely coupled to these abstractions. This problem presumably also abets the dogmatic division of real-time systems into event-triggered systems and their time-triggered counterparts.

In previous work [3], [4] we challenged that dogmatic division. The main distinction among these control flow abstractions is the ability to express dependencies among different activities. So, we proposed an intermediate representation called *Atomic Basic Blocks* (ABBs) to capture these dependencies irrespective of the used control flow abstraction. In [5] we went one step further and advocated for a compiler-based tool working on an intermediate representation based on ABBs. Such a tool would hopefully be able to automate considerable parts of the migration from an event-triggered to a time-triggered implementation of a real-time system and vice versa. In this paper we present our recent efforts to build a prototype of such a tool. For now, we focus on an automated transition from event-triggered to time-triggered systems. The transition from time-triggered to event-triggered systems is subject to future work. So, the contributions presented in this paper are as follows:

- A fine-grained system model that builds on top of ABBs and supports the migration between event-triggered and time-triggered systems and vice versa.
- An automated transformation procedure that maps an ABB-based dependency graph extracted from an event-triggered real-time system to a time-triggered execution environment.
- A prototypical implementation of a compiler-based tool, called *Real-Time System Compiler* (RTSC) to transform an event-triggered into a time-triggered system in a semi-automatic fashion.
- An evaluation of our prototype giving evidence of the validity of our approach by transforming a real-world event-triggered hard real-time system into a time-triggered equivalent.

In the following section 2 we briefly sketch the notion of Atomic Basic Blocks introduced in [5] and in section 3 we give a detailed description of our system model built on top of ABBs. In section 4 we revisit the design of the RTSC and describe the transformation used to migrate an event-triggered to a time-triggered system. Section 5 gives an overview over the prototypical implementation of the RTSC and 6 presents an evaluation of our prototype. The subsequent section 7 discusses related work while section 8 finally concludes the paper.

1555-0885/10 $26.00 © 2010 IEEE
DOI 10.1109/ISORC.2010.11

2. Atomic Basic Blocks

Atomic Basic Blocks are our basic abstraction playing a key role in the migration procedure. All analyses and transformations apply to graphs made up by ABBs. So, ABBs also serve as intermediate representation of the RTSC, comparable to the intermediate code used in any other compiler.

ABBs are designed to carry all relevant dependencies and properties of a real-time system independent of the underlying real-time paradigm. As relevant dependencies we consider directed order relations carrying optional temporal delays and mutual exclusion relations. Relevant properties of ABBs are their WCET and their deadline.

The similarity of the terms *Atomic Basic Blocks* and *basic block* – the abstraction widely used in compiler construction – is not at random: an ABB aggregates one or more basic blocks. Like basic blocks, ABBs form graphs. First of all, ABBs reflect the control flow graphs (CFG) described by basic blocks. On the function-level the CFG formed by ABBs is a coarsening of the CFG formed by basic blocks. The same holds for the data flow graph. ABBs extend these graphs and also include inter-function dependencies that cross function boundaries. Such dependencies result from synchronisation mechanisms that are used to explicitly establish dependencies among different flows of control. These dependencies are typically initiated by system calls to the employed real-time operating system (RTOS), for example, locking a mutex or posting a semaphore. In ABB-graphs, interactions between different functions are only allowed at boundaries of ABBs. Thus, ABBs are described by the following rules:

1) ABBs contain one or more basic blocks. On the function-level, ABBs form a coarsening of the control flow graph and data-flow graph of the function.
2) Every ABB has exactly one distinguished *entry basic block* and at most one distinguished *exit basic block*. Except these basic blocks no other basic blocks have preceding or succeeding basic blocks outside that ABB.
3) An ABB always lasts from the end of the preceding ABB to an appropriate ABB termination. ABB terminations mark positions in the CFG that are either origins (so called *joins)* or targets (so called *joinpoints*) of inter-function dependencies.
4) If an ABB termination is located within a basic block the basic block is divided into two subsequent parts.

The program statements that constitute ABB terminations depend on the operating system API that is used to implement the real-time application and its semantics. Typical system calls resulting in *joins* are forking threads, setting flags, leaving critical sections or defining global variables. *Joinpoints* are created by waiting for signals, entering critical sections or reading global variables.

As an example, Figure 1 presents the ABB-graph that was extracted from the function `CSYSTEM_Handler2`. The dashed arrows and boxes show the original basic blocks and the corresponding CFG. The solid arrows and boxes depict the extracted ABBs and the control flow edges mirroring the original CFG. The ABB termination marking the end of ABB `ABB 8` results from forking another flow of control, this in indicated by the label `ABBTerm = Fork`. Similarly ABB `ABB 12` was terminated by a return instruction. All other ABBs are closed by an artificial ABB termination; this is needed to satisfy item 2 of the rules stated above.

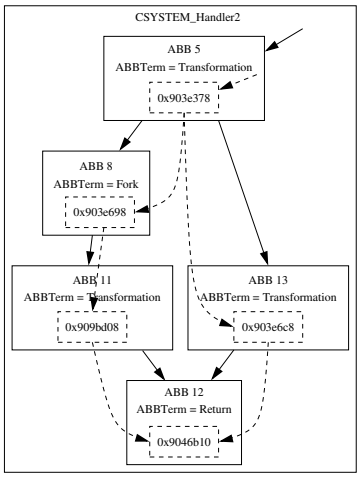

Fig. 1. ABB-graph of function `CSYSTEM_Handler2`

3. System Model

In this section we discuss the elements of the system model in more detail. We describe which elements our system model comprises and how ABB-graphs are used to describe the structure of the overall system. The presented system model intentionally does not make any assumptions on the execution model of ABB-graphs. Our ambition is to map dependency graphs formed by ABBs to an arbitrary execution environment that either works in an event-triggered, a time-triggered or a mixed-mode fashion.

3.1. Events

Every activity in a real-time system is initiated by an *event*. Among events, *periodic* and *non-periodic events* as well as *physical* and *logical events* are further distinguished. Periodic events are characterised by a *period*, a *phase* and a *jitter*, while just a *minimum interarrival time* could be given for non-periodic events. Physical events are created by peripheral hardware components by setting flags or issuing interrupt requests, for instance. Logical events on the other side, relate to changes in the logical state of the application. As an example, consider chunks of a message arriving at the serial port. The arrival of each chunk produces an interrupt; these interrupts are physical events. After a certain amount of chunks the message is complete; this is a logical event. Each logical event is related to a physical event, as a logical event usually can not occur without a preceding physical event. The distinction of physical and logical events enables a much better capturing of the actual temporal properties of an application. Physical events usually tend to occur much more frequently but also less time is needed to handle them, while it is the opposite for logical events.

3.2. Tasks and Subtasks

All activities that are triggered by events are subsumed under the term *task*. A task connects the temporal properties described by the event to its corresponding handler. Each task consists of one distinguished *root subtask* and zero or more additional *subtasks*. A subtask encapsulates the actual implementation of the event handler. The root subtask is executed every time the associated event occurs, the other subtasks are forked by the root subtask or one of its successors. Each subtask could be assigned a *soft*, a *firm* or a *hard deadline*. The deadline marks the latest possible completion time of this subtask relative to the event associated with the task or a related physical event. Tasks containing multiple subtask reflect the situation that the same event could trigger different handlers each combined with a different deadline. Each subtask is implemented by a *handler function*. The control flow structure of the handler functions and all other functions constituting to the implementation of a subtask are described by ABB-graphs as presented in section 2. Thus, a real-time system is represented as forest of ABB-graphs. Each root node in this forest relates to the entry of a handler function implementing a root subtask. All other subtasks are either *forked* or *triggered* by the set of root subtasks.

3.3. Triggering and Forking Subtasks

Forking and *triggering* subtasks relate to special directed dependencies targeting root nodes in the ABB-graph of a subtask. A forked subtask is immediately ready for execution. As a consequence, the temporal properties exposed by the forked subtasks are directly related to those of the forking subtask. In our system model forking subtasks is only supported within the same task. A triggered subtask, on the other side, does not inherit the temporal properties of its predecessor. These are still described by the event tied to the task enclosing that subtask. Furthermore, it is only supported to trigger the root subtask of a task but no other subtasks. Triggering subtasks is useful when a subtask handling a physical event results in a change of the logical state of the system and, thus, in a logical event.

3.4. Mutual Exclusion

Mutual exclusion dependencies are expressed by sets of neighbouring ABBs in the ABB-graph that form *critical sections*. Each critical section is assigned one or more *resources*. Critical sections associated with the same resource must not be executed in an overlapping manner.

3.5. Example

Figure 2 shows the ABB-graph of a Task named `Task2`. It is triggered by a non-periodic event with a minimal inter-arrival time of 10 milliseconds. The task contains a subtask `CSYSTEM_Handler2` implemented by a function of the same name. Its ABB-graph was already presented and discussed in section 2. At `ABB 8` the subtask `CSYSTEM_Handler2` forks a subtask called `CSYSTEM_Handler3`, which is also implemented by a function of the same name and consists of a single ABB (`ABB 14`), only.

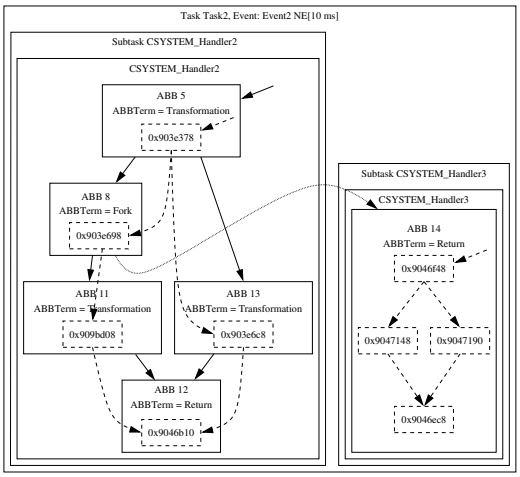

Fig. 2. ABB-graph for task `Event2`

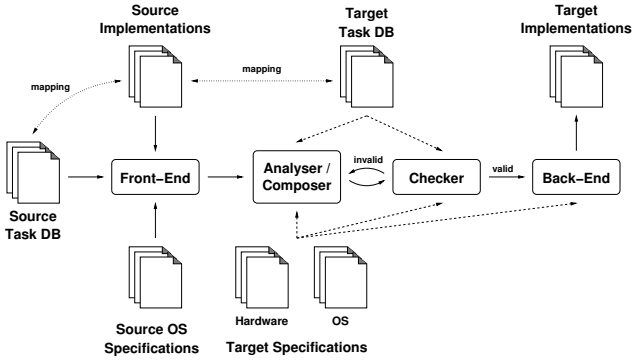

Fig. 3. Design of the RTSC

4. The Transformation Procedure

In this section we revisit and refine the design of our RTSC tool presented in [5]. First, we give an overview of the overall design of the RTSC. Thereafter, we have a closer look on central components of the RTSC and the transformation taking place in these components. Although the RTSC is intended to provide a general operating system (OS) aware compiler tool, we restrict ourself to the steps that are necessary to migrate an event-triggered system to its time-triggered counter part for reasons of space and to maintain focus.

4.1. Overview

Figure 3 depicts the conceptual design of the RTSC. The basic structure of the source and target real-time system are stored in *Task Databases* (Task DBs). These Task DBs describe the source and the target real-time system in terms of the system model presented in section 3. The *Source Task DB* contains all events, tasks and subtasks that are present in the source real-time system, whereas, the *Target Task DB* encompasses all entities that should also be present in the target real-time system. Both, the Source and the Target Task DB, are provided as input to the RTSC. Automatically mapping the Source Task DB to an appropriate Target Task DB is beyond the scope of this paper. First, an OS-dependent *Front-End* extracts the relevant

ABB-graphs from the implementation of the source real-time system. It starts from the handler functions of the subtasks given in the Source Task DB. These ABB-graphs constitute an OS-independent specification of all relevant dependencies in the given source system. In the next step, these ABB-graphs are handed over to a combined *Analyser/Composer*. This component performs the necessary steps to transform the source real-time system targeting a certain RTOS and a certain real-time paradigm into the target real-time system maybe targeting a different RTOS and a different real-time paradigm. If necessary, a *Checker* ensures that certain temporal constraints prescribed in Target Task DB are fulfilled. Finally, an OS-dependent *Back-End* emits the implementation of the transformed real-time system and also creates the required configuration data for the targeted OS.

4.2. Front-End

Succinctly, the front-end extracts an OS-independent ABB-graph from the implementation of the source real-time system. As first step, the functions marked as handler functions of subtasks in the Source Task DB are identified by means of their names. After that, the implementation is scanned for ABB terminations and ABBs are constructed. Subsequently, local ABB-graphs reflecting the function's CFG are built. These local ABB-graphs are then connected by dependencies crossing function boundaries. For this purpose, we search for matching pairs of joins and joinpoints (e.g. setting/awaiting the same flag) and create dependencies between the corresponding ABBs.

These dependencies, however, still are OS dependent, because they carry the explicit semantics of the original system call. Thus, we obtain an OS-independent representation by lowering these dependencies to generic directed dependencies. The original semantics of the system call are reflected by guarding the joinpoint with a logical expression, its *guard*. This expression specifies the set of preceding joins that have to be finished so that the succeeding joinpoint can be executed. As an example, a joinpoint waiting for a flag to be set, would be guarded by a disjunction of all joins setting that flag. At this time, we also extract the sets of ABBs forming critical sections from the ABB-graph. In a last step, we remove the original OS calls from the implementation and gain an OS-independent representation of the source real-time system that still contains all relevant dependencies between its tasks and subtasks.

Our front-end still suffers some restrictions and, thus, implicitly relies on some assumptions. Currently, a flow- and path-insensitive analysis to obtain the global dependency graph is performed. This, of course, restricts the use of OS functions within library routines to a certain extent. We need to know which OS object is manipulated at a certain line of code. We are confident that this problem could be mitigated by a more sophisticated analysis as presented in [6]. Furthermore we expect the application itself and the OS API to be *well-formed*. This mainly refers to a non-ambiguous usage of OS objects and operating system calls. On the application level, for instance, we assume that the same instance of a semaphore is not reused for a different purpose at a different location. On the OS level we assume that the same system call is not used for different purposes, too. This implies that the semantics of a system call are determined by the system call itself. As an example, we require that semaphores are not used for unilateral and multilateral synchronisation. These restrictions and assumptions, however, either do not go beyond those that are also present in similar approaches or are mostly technical and could be relaxed by a more sophisticated implementation.

4.3. Analyser/Composer

In this subsection we describe the mapping of a directed, non-cyclic ABB-graph provided by the Front-End onto a time-triggered execution environment. We suppose that this execution environment provides non-preemptable threads with run-to-completion semantics. Thus, a thread can only be scheduled after the preceding thread is guaranteed to be finished. All transformation steps take place in the target real-time system. So, the ABB-graph extracted by the Front-End first is attached to the tasks and subtasks described by the Target Task DB.

4.3.1. Creating private ABB-graphs. In a first step we create a *private* ABB-graph for each task in the system. This step is necessary as the scheduling algorithm that we use cannot deal with elements that are activated by a multitude of predecessors. Instead, this algorithm assumes that all predecessors have to be finished before a common successor can be executed.

The ABB-graph produced by the front-end only reflects dependencies on the function level. Functions are typically called from more than one location within a program and, hence, can be activated by a bunch of predecessors. The same holds for subtasks, that are forked more than once. We address that problem by creating *private* copies of the ABB-graphs of the handler functions of all subtasks. Then, we recursively clone the ABB-graphs of all functions that are called and all subtasks that are forked by this handler if the corresponding entry ABBs have more than one predecessor. Otherwise, we just assign that ABB-graph to the calling or forking subtask. Note that we solely clone the ABB-graphs, not the actual functions. Also note that trigger dependencies are not covered here.

4.3.2. Triggering Tasks. In a next step, we take care of trigger dependencies introduced in section 3. If a task is triggered by another task, it basically maintains the temporal properties specified by its associated event. Nevertheless, the triggering predecessor needs to be completed before the triggered task can be executed. Hence, in a time-triggered execution environment, a triggered task polls its predecessor for a flag that is inserted by the RTSC. The predecessor sets this flag as soon as it reaches the trigger condition.

In this step, the RTSC also inserts clones of tasks handling logical events that are triggered by tasks handling related physical events. This is required if the temporal distance between the physical and the logical event is too big so that the deadline referring to the physical event cannot be met. By inserting additional logical events this temporal distance can be eliminated to enable a timely execution of the task handling the logical event.

Self-trigger dependencies are special trigger dependencies. In that case a task just triggers itself. Though, we assume that self-triggering tasks do not immediately become ready for execution again. Here, the concerned action could be performed in a loop as often as needed. Instead, we expect that a known amount of time has to pass by before that happens; this delay has to be reproduced as precisely as possible. For this purpose, we adjust the period of the event associated with that task to the greatest common divisor of all possible delays. The RTSC maintains an additional period counter to track the current delay. The period counter is initialised when the task triggers itself and is decremented every time the task is scheduled. The original handler function of this subtask is only executed if the period counter indicates that the delay has passed and the appropriate trigger flag is set.

4.3.3. Spanning the Hyperperiod. Creating a static schedule often is equivalent to ordering a directed acyclic graph (DAG) with respect to temporal constraints. The linearised DAG then is executed cyclically. As a consequence the DAG has to comprise all activities of a complete hyperperiod of the given real-time system. Otherwise, it cannot be executed cyclically in general. So, we now expand the given real-time system to the hyperperiod determined by the target real-time system. This task could be further refined to the following steps:

1) Convert the guards into disjunctive normal form (DNF)
2) Compute the hyperperiod
3) Clone tasks as needed
4) Rematch joins and joinpoints

The very first step is just a preparation step for matching joins and joinpoints again. Here, we compute the DNF for every joinpoint guard present in the system. The next step, determining the hyperperiod, is obvious.

After that, the actual expansion of the application is carried out. Every task whose associated event in the target real-time system has a period that is smaller than the hyperperiod is cloned as often as needed. The cloned tasks are triggered by events whose period is set to the hyperperiod and their phase is set to the sum of their original phase and a multiple of their original period. After this step, all events of the system have the same period – the hyperperiod – and their original period is mapped to an according phase.

In the last step, we first rip up all dependencies connecting different tasks. For each joinpoint, we then try to find a set of joins that satisfies the guard of that joinpoint. This is accomplished by a relatively simple list-scheduling like heuristic. The heuristic expects that each join and joinpoint can be replaced by one of its clones and that each join can only be used to satisfy at most one guard. The aim of the heuristic is to find a valid matching and to minimise the temporal distance between the joinpoint and the joins satisfying its guard without unnecessarily delaying the joinpoint. As the implemented heuristic is rather simple, it is by no means guaranteed that a suitable matching is found if such a matching exists. Additionally, more sophisticated matching algorithms could be used. An algorithm targeting a similar problem that is based on linear programming

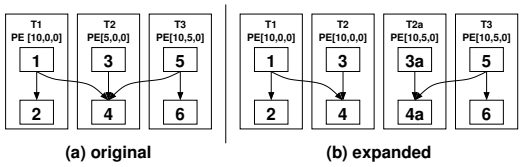

Fig. 4. Rematching Joins and Joinpoints

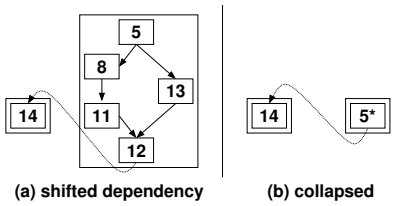

Fig. 5. Serialising the ABB-graph

is presented in [7], for instance. In our previous experiments, however, the heuristic we implemented did not impose any restrictions.

The example in Figure 4 illustrates that transformation step. For reasons of simplicity, all unnecessary information is omitted in this ABB-graph. The original ABB-graph comprises three Tasks `T1`, `T2` and `T3`. All of them are triggered by periodic events; `T1` and `T3` are triggered every 10 time units, `T3` with an additional phase of 5 time units, `T2` is triggered every 5 time units. The joinpoint at ABB `4` is guarded by the logical expression (`1` $\vee$ `5`). Due to its period of 5 time units task `T2` is cloned one time in the expanded system resulting in the task `T2a`. The joins at ABB `1` and `5` are matched with the joinpoints at ABB `4` and `4a`, respectively.

4.3.4. Serialising the ABB-graph. Scheduling ABBs belonging to alternate branches of if-else-statements sequentially is not very helpful because in no case all of them are executed. We try to avoid such situations by serialising the ABB-graph as far as possible. This is accomplished by shifting outgoing dependencies to succeeding ABBs in the ABB-graph and incoming dependencies to preceding ABBs. When shifting dependencies, of course, we must ensure that we do not create any circles within the ABB-graph. As an effect we can collapse all ABBs making up the if-else-statement if we can shift all inter-function dependencies out of the if-else-statement. Hence, there also is no need to temporally order these ABBs anymore.

Figure 5 shows simplified versions of the ABB-graph already presented in Figure 2 and exemplifies the effect of serialising the ABB-graph. If the original dependency between `ABB 8` and `ABB 14` could be shifted to the dependency between `ABB 12` and `ABB 14` as shown in Figure 5 (a), then we can collapse the ABBs `ABB 5`, `ABB 8`, `ABB 11`, `ABB 12` and `ABB 13` into the single ABB `ABB 5`$\star$ as shown in Figure 5 (b).

We assume that the ABB-graph is acyclic, but there still might be patterns that cannot be serialised by shifting incoming and outgoing dependencies. Such patterns typically result from producer-consumer interactions [6]. In such a case, either the producer or the consumer would have to be statically segmented in at least two fragments, via some sort of source

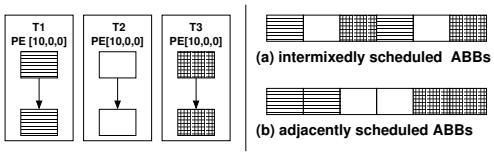

Fig. 6. Scheduling ABBs

code transformation. This is not supported by the RTSC, yet.

4.3.5. WCET-Analysis. A necessary input for a static scheduling algorithm is the WCET of the elements to be scheduled. For this purpose we integrated an automated WCET-analysis in the RTSC. Current WCET-analysis tools consist of two parts: a high-level analysis and a low-level analysis. The high-level analysis examines the CFG of a given function and transforms it into a maximum flow problem that is usually solved via linear programming [8]. The WCET of the basic blocks that serve as input for the high-level analysis are determined by the so called low-level analysis. The low-level part examines the machine code and performs an extensive hardware-dependent analysis to calculate the WCET of the given basic block. We implemented a high-level analysis within the RTSC while we use an external tool for the low-level part. For additional input parameters needed for the high-level analysis that cannot be determined from the source code directly (e.g. loop counts or recursion depths), we rely on annotations.

4.3.6. Scheduling. The final step of the transformation process is the computation of a static schedule that arranges all ABBs in an absolute temporal order. To successfully schedule an ABB-graph, the scheduling algorithm needs to deal with order dependencies, mutual exclusion dependencies, release times and deadlines. Hence, we chose the algorithm by Abdelzaher and Shin [9] as it meets all our requirements. Furthermore, it offers the possibility to target multiprocessor systems and to incorporate an additional message scheduling algorithm later on. Currently, we target systems with a single processor, only. The order and mutual exclusion dependencies needed as input can easily be extracted from the ABB-graph while the release times and deadlines are stored in the Target Task DB.

Nevertheless, we had to modify that algorithm slightly. Originally, it was supposed to schedule tasks, each consisting of several *modules*. Implicitly, implementation elements do not cross module boundaries, i.e. a function is never split across two modules. This is inherently different with ABB-graphs. The ABB-graph for a single function could easily comprise several ABBs. If all ABBs were handled equivalently that might lead to sequences of ABBs where ABBs of different functions are intermixed without caution (see Figure 6 (a)). This is not desired as such a schedule might contain lots of non-local jumps between different functions that would induce significant runtime overhead. For this reason, we try to schedule ABBs belonging to the same function adjacently (see Figure 6 (b)). We achieved this by modifying the EDF-algorithm that is used inside of [9]. EDF schedules modules according to their deadlines that are normally specified on a per task base. The deadline for each module is determined with respect to the deadline of its successors and its own WCET. So, ABBs belonging to the same function may be allotted different deadlines favouring a inter-mixture of ABBs belonging to various functions. We avoid this by assigning the same deadline to all ABBs of a function. We break ties in the EDF-algorithm in favour of that ABB that belongs to the same function like the ABB that was scheduled at last.

4.4. Checker

The duty of the checker is to explicitly ensure that the temporal properties that are specified in the Target Task DB are met. As this paper considers only targeting time-triggered operating systems such properties, primarily the schedulability, are guaranteed by construction in the generated system.

4.5. Back-End

The task of the Back-End is to generate code that can be execute by the targeted RTOS. In the specific case, we generate configuration data and an application skeleton for the targeted operating system. The application skeleton just calls the event-handlers at predefined points in time that are specified by the previously computed static schedule. The original application (i.e. all the implemented event handlers and libraries) is directly emitted as assembly code for the target system.

5. Implementation

The RTSC is implemented as a set of passes for the LLVM compiler framework [10]. All our transformations work on the virtual instruction set used as intermediate representation within the LLVM that also serves as basis for ABBs. So, all of them are basically implemented in a programming language and OS independent manner. Currently, we use the GCC-based front-end of the LLVM to compile existing real-time applications written in the C programming language into the LLVM representation. We also make use of standard transformations and analyses provided by the LLVM for standard compiler optimisations and the code generation framework to emit assembly code for the target processor.

While we implemented the high-level part of the WCET-analysis on our own, we rely on an external tool for the low-level part. Therefore, we integrated the static WCET analysis tool aiT[1] into the RTSC. We use it to compute the WCET for every single basic block in the system.

The RTSC currently encompasses a Front-End for an artificial event-triggered OS API. The API comprises services for forking and triggering subtasks, lock variables, message passing and flags. All in all, the expressiveness of this API is very similar to the API specified by the OSEK OS [11] and we intend to implement support for the OSEK OS API in the near future. On the Back-End side, we currently support time-triggered systems only. Here, we target operating systems that are compliant with OSEKtime specification [12]. The processor

1. http://www.absint.com/ait

we currently target is the TriCore processor [13] by Infineon, as we already have extensive experience using it and there are several RTOS available for that processor. Last but not least, the WCET analysis tool aiT is available for exactly this processor. As the LLVM did initially not support it we also implemented a LLVM Back-End for TriCore processor.

6. Evaluation

We have evaluated our the RTSC tool with a real-time system that mimics a highstriker, the well-known attraction on fairs. This system constitutes a fairly challenging evaluation scenario as it comprises non-periodic events only and, thus, could serve as prime example for an event-triggered system. The control application also was developed in an event-triggered fashion and we transformed it into an equivalent time-triggered one. In our quantitative evaluation we mainly pay attention to temporal parameters that are relevant for a successful operation of this system. For a better understanding of the observed parameters we first give a more elaborate explanation of the evaluation scenario and then take a look at the measured parameters.

6.1. Evaluation Scenario

Our highstriker is equipped with a Plexiglas tube that houses an iron projectile. The projectile is controlled by switching coils attached to the tube in an equispaced manner. Hereby, the projectile can be guided almost arbitrarily between the different coils. Right above each coil there is a light barrier that is used to track the position of the projectile.

The control application itself is state machine-based. A step in the state machine is either triggered by the projectile leaving or entering a light barrier or a countdown indicating that the next step must be taken. The countdown is set up within a step of the state machine itself, as some actions require the next step to be carried out with a certain temporal distance. All these occasions are subsumed by the logical event *SMStep*. If several consecutive steps in the state machine are necessary, all these steps are executed in a row within a loop. So, the task handling *SMStep* is a self-triggering task as described in section 4.3. The task is also triggered by the task handling the physical *LightBarrier* event. It is not necessary to perform a state machine step every time a light barrier is entered or left. For this reason, the *SMStep* task is triggered but not forked by the *LightBarrier* task.

Both events are obviously non-periodic. The minimum interarrival times are achieved when the projectile crosses two subsequent light barriers at its maximum speed of roughly 6.14 m/s. The length of the projectile of 8.2 cm leads to a minimum interarrival time of about 12.8 ms for the physical event *LightBarrier*. The distance of 23 cm between two light barriers leads to an minimum interarrival time of 37 ms for the event *SMStep*. The physical event *LightBarrier* is handled by the task `CSYSTEM_P_ISR` and the logical event *SMStep* by the task `CSYSTEM_FSM_Task`. The deadlines of 1 ms are empirical values we gained from the operation of our highstriker experiment. The deadline of the task `CSYSTEM_FSM_Task` handling the logical event *SMStep* refers to the physical event *LightBarrier*. This information is summarised in the Source Task DB that is depicted in Table 1. The Target Task DB that is also provided as input for the RTSC is shown in Table 2; in this case, it could easily be derived from the Source Task DB. The events are converted into periodical events and their deadlines and the period of the physical event are adjusted according to the well known theorem by Nyquist and Shannon. Both events are handled by the same tasks as in the source real-time system. Note that the period of the event *SMStep* is not adjusted, as it will be completely handled by the RTSC.

Event	Interarrival time	Task	Deadline
LightBarrier	12.8 ms	`CSYSTEM_P_ISR`	1ms
SMStep	37 ms	`CSYSTEM_FSM_Task`	1ms

TABLE 1. Source Task DB

Event	Period	Task	Deadline
LightBarrier	500 μs	`CSYSTEM_P_ISR`	500 μs
SMStep	37 ms	`CSYSTEM_FSM_Task`	500 μs

TABLE 2. Target Task DB

6.2. Evaluation Results

The schedule table generated by the RTSC is depicted in Table 3. The schedule table is obviously not optimal; it could be shortened to a period of 500 μs. At the moment, we do not make any effort to produce a compressed schedule table. The initial offset of 10 μs is owed to a restriction in the targeted RTOS. The RTSC automatically adjusted the period of the task `CSYSTEM_FSM_Task` so its deadline related to the physical *LightBarrier* event could be met. There was no need to explicitly convert the former interrupt service routine `CSYSTEM_P_ISR` into a polling variant. The original system suffered from bouncing light barriers, thus, `CSYSTEM_P_ISR` already had to filter relevant interrupts.

Using that schedule table and the transformed implementation generated by the RTSC, we were able to operate our highstriker experiment the same way we already did it using the original event-triggered implementation. Besides the successful operation of the highstriker, we also performed a quantitative comparison of the generated and the original system regarding the following aspects: the event handler latencies and their response times (Table 4), the accuracy of the countdowns (Table 5) and the overall CPU utilisation. [2]

It is not astonishing that the original event-triggered system outperforms its time-triggered counterpart generated by the RTSC with respect to the event handler latencies and response

2. All benchmarks have been performed on an Infineon TriCore TC1796 processor running at a CPU clock speed of 100 MHz and a system clock speed of 50 MHz. The STM-timer of the TC1796 was used for time measurement. The CPU utilisation was determined by directly inspecting the CPU utilisation of an idle function by the Trace32 debugger by Lauterbach. Data and code were completely located in the internal RAM of the TC1796.

Start time	Task	WCET
10 μs	`CSYSTEM_P_ISR`	19,25 μs
30 μs	`CSYSTEM_FSM_Task`	75,87 μs
510 μs	`CSYSTEM_P_ISR`	19,25 μs
530 μs	`CSYSTEM_FSM_Task`	75,87 μs

TABLE 3. Schedule Table generated by the RTSC

	Latency			Response Time		
	min	avg	max	min	avg	max
Source	6 μs	8 μs	10 μs	9 μs	12 μs	18 μs
Target	13 μs	268 μs	507 μs	24 μs	276 μs	518 μs

TABLE 4. Event handler latencies and response times

	Source			Target		
Countdown	min	avg	max	min	avg	max
4	3,05	3,63	4,01	4,00	4,00	4,00
9	8,01	8,46	8,98	9,00	9,00	9,00
14	14,00	14,00	14,00	14,00	14,00	14,00
18	17,05	17,50	17,99	18,00	18,00	18,00
86	86,00	86,00	86,00	86,00	86,00	86,00
950	949,07	949,49	950,00	950,00	950,00	950,00

TABLE 5. Countdown accuracy (in milliseconds)

times. The maximum latencies in the target system result from the fact that the task `CSYSTEM_FSM_Task` is scheduled 20 μs after the task `CSYSTEM_P_ISR`, summing up to the maximum latency of 520 μs. Interestingly, the generated target system performs very well regarding the accuracy of the countdowns. The fast response times and the precise countdown are facilitated by a low overall CPU utilisation of only about 0.5% in the original system. Due to overhead induced by polling the generated system exposed a considerably increased overall CPU utilisation of 4.1%. This, however, is a general problem of time-triggered systems and not specific to the system generated by the RTSC.

7. Related Work

Using compiler techniques to improve and automate the construction and analysis of real-time systems has already been done before. Program slicing was employed in [14], [15] to improve the schedulability of real-time system. In [16] a method is given to analyse the schedulability of real-time systems based on hierarchical event streams that could be extracted from the CFG semi-automatically. Others created domain-specific languages to describe the temporal structure of real-time systems [17], [18], [19].

Considerable work has also been done regarding the provision of integrated tools and tool chains that support the development of software for real-time systems. Among the numerous examples are [20], [17], [18] or well-known commercial products like TargetLink. Approaches comparable to ours are implemented by Anvil [21] or PORTOS [22].

The authors of this paper, however, are not aware of any tool or tool-chain that explicitly aids the migration between different real-time paradigms. Most of these tools assume some kind of abstract, model-based input. Thus, such tools automatically gain independence of the employed real-time paradigm. The RTSC, in contrast, works on the much lower level of an existing implementation. So, the RTSC is also able to deal with existing legacy software.

8. Conclusion and Future Work

In this paper we presented the first prototype of the RTSC, a tool that assists in migrating from event-triggered to time-triggered systems. It can already handle a certain range of real-world real-time systems and, thus, provides a profound and comfortable alternative to ad-hoc techniques that are still widely used in the development of time-triggered real-time systems. Nonetheless, this prototype was just a first step and there are more challenges that need to be tackled. As an answer to the dawn of the multi-core era also in the field of embedded real-time systems, we plan to target distributed and multi-processor systems in the near future. Furthermore, we want to add advanced analysis techniques and source code transformation techniques to the RTSC. Thereby, more complex real-time systems resulting in non-serialise-able ABB-graphs shall be transformed into time-triggered equivalents.

References

[1] G. D. Carlow, "Architecture of the space shuttle primary avionics software system," *CACM*, 1984.

[2] T. Shepard and M. Gagne, "A model of the f18 mission computer software for pre-run-time scheduling," in *ICDCS '90*. IEEE, May 1990.

[3] F. Scheler and W. Schröder-Preikschat, "Synthesising real-time systems from atomic basic blocks," *RTAS'06*, Apr. 2006, wiP presentation.

[4] ——, "Time-triggered vs. event-triggered: A matter of configuration?" in *MMB-NFPES*. VDE, Mar. 2006.

[5] F. Scheler, M. Mitzlaff, W. Schröder-Preikschat, and H. Schirmeier, "Towards a real-time systems compiler," in *WISES '07*. IEEE, Jun. 2007.

[6] S. Mohan and J. Helander, "Temporal analysis for adapting concurrent applications to embedded systems," in *ECRTS '08*. IEEE, 2008.

[7] J. E. Cuny and L. Snyder, "Conversion from data-driven to synchronous execution in loop programs," *ACM TOPLAS*, vol. 9, no. 4, pp. 599–617, 1987.

[8] P. Puschner, "Zeitanalyse von echtzeitprogrammen," Ph.D. dissertation, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 1993.

[9] T. F. Abdelzaher and K. G. Shin, "Combined task and message scheduling in distributed real-time systems," *IEEE TPDS*, vol. 10, no. 11, pp. 1179–1191, 1999.

[10] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO'04*, Mar 2004.

[11] OSEK/VDX Group, "Operating system specification 2.2.3," OSEK/VDX Group, Tech. Rep., Feb. 2005, http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2009-09-09.

[12] ——, "Time triggered operating system specification 1.0," OSEK/VDX Group, Tech. Rep., Jul. 2001, http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf.

[13] *TriCore 1 User's Manual (V1.3.5), Volume 1: Core Architecture*, Infineon Technologies AG, St.-Martin-Str. 53, 81669 München, Germany, Feb. 2005.

[14] R. Gerber and S. Hong, "Slicing real-time programs for enhanced schedulability," *ACM TOPLAS*, vol. 19, no. 3, pp. 525–555, 1997.

[15] P. Gopinath and R. Gupta, "Applying compiler techniques to scheduling in real-time systems," in *RTSS '90*. IEEE, Dec. 1990.

[16] K. Albers, F. Bodmann, and F. Slomka, "Hierarchical event streams and event dependency graphs: A new computational model for embedded real-time systems," in *ECRTS '06*. IEEE, 2006.

[17] T. A. Henzinger, C. M. Kirsch, E. R. Marques, and A. Sokolova, "Distributed, modular HTL," in *RTSS '09*. IEEE, Dec. 2009.

[18] E. Farcas, C. Farcas, W. Pree, and J. Templ, "Transparent distribution of real-time components based on logical execution time," in *LCTES '05*. ACM, 2005, pp. 31–39.

[19] V. F. Wolfe, S. Davidson, and I. Lee, "RTC: language support for real-time concurrency," in *RTSS '91)*. IEEE, Dec. 1991.

[20] D. de Niz, G. Bhatia, and R. Rajkumar, "Model-based development of embedded systems: The sysweaver approach," in *RTAS '06*. IEEE, 2006.

[21] I. Gray and N. C. Audsley, "Exposing non-standard architectures to embedded software using compile-time virtualisation," in *CASES '09*. ACM, 2009.

[22] M. Krause, O. Bringmann, and W. Rosenstiel, "Target software generation: an approach for automatic mapping of SystemC specifications onto real-time operating system," *Des Autom Embed Syst*, vol. 19, no. 5, pp. 229–251, May 2006.