From Intent to Effect: Tool-based Generation of Time-Triggered Real-Time Systems on Multi-Core Processors

Florian Franzmann*, Tobias Klaus*, Peter Ulbrich*,
Patrick Deinhardt*, Benjamin Steffes*, Fabian Scheler†, Wolfgang Schröder-Preikschat*,

*Chair of Distributed Systems and Operating Systems
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
`{franzman,klaus,ulbrich,deinhardt,steffes,wosch}@cs.fau.de`

†Process Industries and Drives
Siemens AG, Nuremberg, Germany
`fabian.scheler@siemens.com`

***Abstract*—Although the manual creation of time-triggered schedules for multi-core real-time systems can be a daunting task, state-of-the-art scheduling algorithms are far from being widely used. This suggests that the availability of sound algorithms is only one side of the story: real-time systems have to be groomed substantially before they can serve as input to available algorithms. Moreover, systems engineers struggle with the temporal effects of their design decisions, in addition to the intended timing properties. Therefore, we believe that appropriate tools are the other side of the story.**

In this paper, we present the multicore extension of the *Real-Time Systems Compiler*, a compiler-based tool that analyses given event-triggered real-time systems and transforms them into time-triggered equivalents. We focus on the challenges and pitfalls in the transition from theory to practical implementation and present concrete solutions to resolve them. Existing algorithms need to be adapted for performance and, at model level, bound together appropriately to be applicable, for example. Our experiments substantiate the effectiveness and scalability of our approach, even for large tasks sets. Finally, lessons learned give an insight into implementation and hardware details and their impact on schedulability.

I. Introduction

For decades, real-time-system engineering was marked by the two opposing development paradigms: *event* and *time triggered* (not to be confused with clock driven or periodic tasks sets, where online scheduling and preemption *is* mandatory). The latter, although much less common, indisputably has its merits when it comes to safety requirements and hard temporal guarantees. The main reason being the absence of hardware events barring the timer interrupt: external signals are solely polled. Accordingly, the resulting task set is fixed at runtime and precedences as well as mutual exclusions are resolved offline by a feasible job arrangement and schedule. Consequently, the real-time operating system (RTOS) itself becomes minimalistic. It lacks typical facilities for online synchronisation as well as coordination and basically boils down to a simple dispatcher, processing a static schedule table. Hence, the runtime overhead induced by the RTOS is determined. Even more important, the analysis of the job's individual worst case execution times (WCETs) is dramatically simplified as unplanned preemption and interrupts are excluded by design. Consequently, overly pessimistic upper bounds due to cache or pipeline invalidation can be omitted.

However, the time-triggered approach has some major disadvantages, too. First and foremost, the a-priori knowledge necessary for creating a static schedule in the first place. This, for example, includes WCETs and minimal interarrival time of all jobs – a difficult and labour-intensive venture in principle. Moreover, the static schedule has to be revised with every software or hardware change (e. g., number of jobs or their WCET). Consequently, time-triggered systems are harder to evolve and adapt to changing requirements, a key skill of event-triggered systems with online scheduling and task coordination.

Traditionally, safety-striving domains such as the avionics industry rely on the time-triggered paradigm (e. g., ARINC 653), as the costs of damage or loss set off development costs. Here, the ruling paradigm is early compartmentalization in the time domain, assigning slots in the hyperperiod to each application. Contrarily, the automotive domain, as representative of mass industry, faces a tremendous cost and time-to-market pressure as well as fast product cycles. Here, the event-triggered paradigm is predominant, with the time-triggered approach employed only in a few special cases. In the past, each function in a car was rendered by a single Electronic Control Unit (ECU), which made it relatively easy to isolate and satisfy safety requirements. Recently two important factors have changed: On the one hand, autonomous driving is just around the corner, making software even more complex and more safety-critical. On the other hand, multi-core embedded processors have hit the market, supplying an amount of processing power unthinkable in the past.

Market pressure forces the industry to seize this opportunity by consolidating ECUs. Therefore, software of lower safety requirements now co-resides with applications of high safety demands, thus increasing interference, complexity and the

[1]This work was supported by the German Research Foundation (DFG) under grant no. SCHR 603/9-1 and the Bavarian Ministry of State for Economics under the (EU EFRE funds) grant no. 0704/883 25.

certification hurdle at the same time. Yet, multi-core real-time systems still pose a challenge to systems engineers and researchers. Here, there seems to be a strong focus on event-triggered systems and constructive means, such as resource protocols and scheduling algorithms. Advantageously, this preserves development freedom and leaves the developers in their natural (event-driven) habitat.

In spite of it's inflexibility, the time-triggered paradigm offers beneficial properties especially in multi-core settings: Offline scheduling allows for an optimal partitioning of jobs. Furthermore, synchronising schedule tables among cores is almost trivial in systems with a common clock. Hence, inter-core synchronisation and resource contention can be excluded once again. This also eliminates the need for complex RTOS facilities, shared data structures and critical sections.

Neither do we envision further constructive methods for the event-triggered approach, nor are we proposing to switch to the time-triggered development paradigm. To the contrary, we think that it makes sense to continue to use the *much easier to develop* event-triggered approach. Instead, we posit a third way, an approach relying on an *analytical, automated, compiler-based, transformative process converting event-triggered real-time systems into multi-core time-triggered ones*, which, by delaying design decisions preserves degrees of freedom for the developer. Such a compiler-based tool would allow systems integrators to forgo the inflexible time-triggered scheme and instead perform the – now automated – compartmentalization step later in the development cycle.

This idea, which we have implemented for single-core processors in our prototypical Real-Time Systems Compiler (RTSC) [1], comes with an abundance of advantages: First, and most importantly, engineers can continue to use their accustomed development process, regardless of whether this process is model-based or results in a hand-written real-time application. The RTSC accepts the source code of an application as input. Therefore, it is relatively easy to consolidate legacy-real-time applications for which the source code is available. The resulting time-triggered systems are verifiable by construction, which means that once the necessary real-time schedule has been determined, it has also been proven that it will observe all deadlines. Furthermore, the absence of explicit synchronization, interrupts (except the timer), and online scheduling, makes it feasible to give tight upper bounds for the WCET. Finally, the small footprint of a time-triggered RTOS makes it relatively easy to perform proofs of correctness of the execution environment itself.

A. Problem Statement

The development processes in industry share some striking properties: Certain design decisions like the choice of target hardware and the real-time paradigm employed are made early on and from then on are almost irreversible. Vast amounts of legacy real-time software exist that have been written following the event-triggered paradigm, and, since event-triggered real-time systems are not trivially composable, this software has to be rewritten if it is to be consolidated with software of higher safety requirements. Therefore, supporting tools are required that remove the need to completely rewrite real-time systems and instead are capable of proving real-time properties and generating time-triggered multi-core systems that are equivalent in functionality to the single-core legacy software. Appropriate algorithms for solving the problems of assigning jobs to processing nodes and scheduling them have been around for at least the past two decades. However, no wide-spread adoption of these algorithms in the systems community has happened, which is why the state of the art still is hand porting application software as stronger safety requirements arise. This suggests that availability of algorithms is only part part of the solution – one missing piece here seems to be proper tool support. Our contributions in this paper therefore are:

- The extended compiler-based RTSC transformation tool which creates time-triggered systems for multi-core processors from event-triggered input using an intermediate representation (IR) that captures all relevant real-time properties.
- An implementation of the assignment algorithm by Peng et al. [2] and the task and message scheduling algorithm by Abdelzaher et al. [3]. Disclosure of effects and pitfalls when deploying these theoretically sound algorithms in the field of real-time system engineering.
- Grooming these algorithms for efficiency and a performance evaluation for a large number of realistic task sets.
- Practical measurements conducted on the Infineon Aurix multi-core target hardware.

II. Background

This section first gives an overview of key concepts and then presents the basic internals of the RTSC, a compiler-based tool that performs a structural decomposition of real-time systems. This decomposition results in an IR appropriate for transforming non-functional properties of real-time systems.

A. Overview of the RTSC

The RTSC is a tool that extracts an abstract IR called Atomic Basic Block (ABB) graphs from given implementations of real-time applications and transforms and analyses non-functional system properties. Based on the LLVM [4], the RTSC consists of about 53,000 lines of C++ code, ≈4,300 lines of which make up the scheduler and processing node assigner implementations discussed in this work. The compiler processes the source code of a real-time application and an additional system description. In general the RTSC is not limited to one programming language but accepts any input the LLVM can process. Similar to most compilers, the RTSC is structured in terms of a *Front-End*, a *Middle-End* and a *Back-End*. The *Front-End* extracts independent ABB graphs from the implementation of the source real-time system. The *Middle-End* uses its own high-level WCET analysis implementation in combination with AbsInt's aiT for the low-level analysis, and maps ABB graphs to the – still abstract – target executive. In order to do so, it calculates an offline assignment of ABBs to processing nodes and per-processing-node time-triggered schedules. The RTSC's *Back-End* generates a skeleton that calls the application's event handlers at pre-defined points in time. The application itself is emitted as machine code for the target platform and OS. The resulting executable real-time system has to satisfy two invariants: On the one hand, all structural properties of the source real-time system – like *order of precedence* and *mutual exclusion* – must be observed. On the other hand, the generated real-time system must abide by real-time properties, namely release times and deadlines.

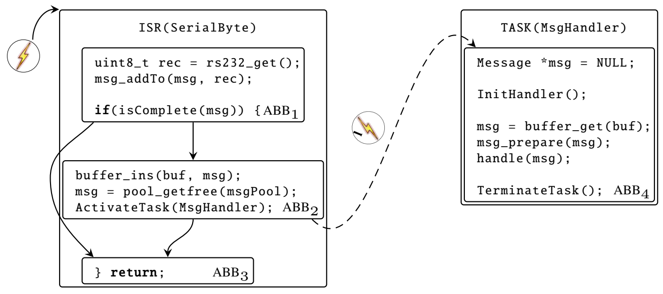
B. Intermediate Representation

The first invariant can be complied with if the real-time system is transformed into an *IR* that encompasses the application's control and data flow as well as all synchronizing interactions with the OS. The RTSC derives such an IR – which we call ABB graphs – from the application's source code. Using synchronizing system calls as boundaries, the application's basic blocks are aggregated and split to form ABBs. Individual ABBs are connected by edges, tracing the basic block graph's flow of control. In the next step, all OS-dependent system calls, like GetResource, ReleaseResource, ActivateTask, . . . , are purged from the ABB graphs and replaced by *directed* (activations) and *undirected* (mutual exclusion) dependency edges. This is important since the target OS's system calls may be different from the source OS's. Only if all edges leading to an ABB have been satisfied does this ABB become ready for execution, ensuring that each ABB can be executed independently of the rest of the system once its dependencies have been fulfilled.

To illustrate the ABB concept, Figure 1 depicts an implementation pattern typical in embedded real-time systems: an event handler consisting of an ISR triggered by a physical event, and a task, activated by the ISR. The ISR fetches a single byte from the serial interface and assembles the message. Whenever a message is complete, the task is activated by a system call. Thus, a directed dependency is created between the ISR and the delayed message handler by means of the underlying OS. This relationship is captured by a global control flow dependency within the ABB graph. Since the *task* contains only a single system call at its end, only ABB_4 is created here. However, the ISR issues a system call that creates a directed dependency to the task. Therefore ABB_2 ends here and in total three ABBs are necessary to model the ISR. Note that the actual system calls – though for better traceability depicted within the ABBs – are not part of the IR – any information included in these calls is represented by dependencies among ABBs.

C. System Model

The second invariant is satisfied by means of the RTSC's *system model*, which makes all timing-related properties available. By definition real-time systems have to react to external *events*, which can be *periodic* or *non-periodic*. Periodic events are annotated with a *period* while non-periodic events carry a *minimal interarrival time* and a *jitter*. All events can have soft, firm or hard *deadlines*. The entities handling these events are called *tasks* and consist of one or more *subtasks*. Subtasks may be connected by *directed* (activations) and *undirected* dependencies (mutual exclusion).



```
uint8_t rec = rs232_get();
msg_addTo(msg, rec);

if(isComplete(msg)) {
```

```
buffer_ins(buf, msg);
msg = pool_getfree(msgPool);
ActivateTask(MsgHandler);
```

```
} return;
```

```
Message *msg = NULL;

InitHandler();

msg = buffer_get(buf);
msg_prepare(msg);
handle(msg);

TerminateTask();
```

Figure 1. Atomic Basic Blocks tracing CFGs and connecting an ISR to a task by means of a directed dependency.

D. Executive

An important goal of the RTSC is the ability to generate solutions for hard real-time problems, meaning that an unhandled missed deadline may have catastrophic consequences. Therefore the RTSC has to generate real-time systems that are either easy to verify or verified-by-construction, an objective that can only be achieved if the real-time system's *executive* is also verifiable. Since verification becomes easier if the system that is to be verified is straightforward, the executive has to be as simple as possible. Thus we are aiming for a *time-triggered executive*, which in contrast to a clock-driven executive [5] does not allow for any interrupts other than the timer – physical events therefore must be perceived by polling. No facilities for mutual exclusion and directed dependencies are provided and hence these properties of the application are enforced implicitly by the generated schedule. As a consequence the code base of the executive is small and it becomes possible to calculate tight upper bounds for WCETs.

The combination of system model and ABB graphs allows us to describe any real-time system in an abstract way, independent of the underlying real-time architecture. This is necessary in order to perform arbitrary transformations on the real-time system while still guaranteeing all deadlines. Overall the RTSC is a solid foundation for the generation of time-triggered multi-core systems. The ABB graphs feature an IR that is independent of architectural details and easy to extend.

III. Approach

Recent years have seen quite some research into multi-core real-time systems. However, most approaches perform scheduling at runtime [6], [7], making proofs of correctness in the presence of directed and undirected dependencies infeasible. Compiler techniques have been used in real-time systems for program slicing [8], [9] to improve schedulability, and to inject wrappers for porting applications to the logical execution time model. Hierarchical event streams have been extracted from control flow graphs (CFGs) [10], however, this approach is only partially automated. Many approaches require domain-specific languages as input [11], [12], forcing the real-time developer to abandon established development paradigms and thus legacy code. Others generate time-triggered systems from event-triggered ones [13] but operate on the coarse-granular task level or are limited to task sets with harmonic periods [14]. In contrast, the RTSC strives to

1. support directed (activations and events flags) and undirected dependencies (mutual exclusion),
2. apply a structural decomposition of the real-time system into ABBs, instead of the functional one dictated by the application design process, allowing for arbitrary transformations while honouring real-time constraints,
3. employ optimal processing node assignment and scheduling algorithms,
4. adapt these algorithms where necessary to handle the fine-grained load generated by the structural decomposition,
5. transform the ABBs to allow the algorithms to reach meaningful results, and
6. consolidate legacy applications regardless of implementation details.

In general, the RTSC already provides the correct representation of the real-time system and the means for extracting the necessary information from the real-time system implementation. The rest of this section will present the steps necessary to allow the RTSC to fulfil all of the abovementioned requirements in a multi-core setting.

a) Middle-End: The goal of the RTSC is to assign and schedule an arbitrary *static* workload of *interdependent* real-time tasks on the processors of a multi-core system. We assume that missed deadlines have *catastrophic consequences* for the real-time system and its environment. Structural decomposition into ABBs results in more fine-grained jobs that have to be assigned and scheduled. This means that the processing resources required for assignment and scheduling may be much higher than for traditional task-level loads. However, since scheduling and assignment are performed offline, vast resources are available and therefore choosing optimal algorithms makes sense. To extend the RTSC toward multi-core targets, two additional components in the middle-end are required: An algorithm that *assigns* ABBs to processing cores and a *scheduling* algorithm that creates a dispatcher table for each core. Both algorithms must be optimal and handle *precedence constraints* and *mutual exclusion* introduced by interaction between individual ABBs. Since scheduling under precedence constraints is an NP-hard problem, both algorithms should exhibit acceptable behaviour w. r. t. run time as well as memory consumption.

Most of the algorithms in recent literature are not optimal [15] or do not support directed dependencies [16], [17] and are therefore unsuitable for our workload. Buttazzo et al. [18] found that their algorithm is optimal but does not scale beyond 20 tasks. The scheduling and assignment algorithms finally chosen for the RTSC implement partitioned EDF scheduling in combination with Branch and Bound (B&B) to ensure optimality. Although in general B&B decreases computational complexity, optimal scheduling algorithms remain challenging in this regard. In Section IV-B we will show how we handled this problem.

Assignment: The RTSC's assignment pass implements an algorithm by Peng et al. [2]. The first solution generated by the algorithm is an empty assignment. This solution is refined successively along the search tree by assigning one more task to each processing node. Inner nodes of the tree are *incomplete* and only leafs are candidates for solutions. The cost function for solutions is the maximum Normalized Task Response Time (NTRT) of all its tasks, which is calculated from the task's completion time c, release time r and absolute deadline d as $\bar{c} := \frac{c-r}{d-r}$. In each solution, the algorithm adjusts the release time of tasks so that all precedence constraints are respected and then arranges the tasks in non-decreasing order of their release times, creating disjoint blocks of jobs. Then the lowest-cost task is removed from its block and the remaining tasks in this block are rearranged in order of their modified release times, respecting all precedence constraints, and the removed block is reinserted into the now freed-up space. These steps are repeated for all tasks. In addition to the NTRT a lower bound for the load caused by unallocated tasks is taken into account.

Scheduling: For scheduling the RTSC uses the algorithm by Abdelzaher et al. [3]. In contrast to the assignment algorithm, the initial solution is already *complete*, containing all tasks that have to be scheduled, and respects their directed and undirected dependencies. If this solution is also *feasible*, the search ends here. Otherwise refined solutions are generated by and by, moving late tasks to earlier points in the schedule. Solutions in the search tree are generated by scheduling all tasks using the EDF with Deadline Inheritance (EDF-DI) algorithm. The cost function is the maximum lateness of all tasks. A lower bound for the cost of refined solutions is calculated by scheduling a simplified version of the load that does not contain any dependencies spanning multiple processing nodes.

b) Back-End: The RTSC's back-end's job is to transform the per-processing-node schedules generated by the middle-end into dispatcher tables for target hardware and OS. Currently, the RTSC supports two multi-core time-triggered back-ends: The first generates tables for time-triggered AUTOSAR, which we then deploy on an Infineon Aurix processor using the ERIKA Enterprise OS. Since AUTOSAR expects a master/slave multi-core environment, the code generated by the RTSC uses one of the cores of the execution platform to kick-start the other cores. From then on, the cores run independently of each other. No injection of OS-specific system calls is necessary for time-triggered execution since any synchronization required is enforced through the dispatcher table. The second back-end generates time-triggered code that runs on POSIX-compatible OSes. Although POSIX does not standardize a time-triggered RTOS API, it is possible to execute code in a clock-driven fashion. The RTSC's POSIX back-end uses POSIX timers to execute dispatcher tables, and the thread-affinity mechanism to bind jobs to processing cores. A semaphore for each core re-synchronizes execution at the start of the hyperperiod.

IV. The Implementation and Its Consequences

In the previous section we presented the algorithms we have chosen for the RTSC. In the following we shift our focus to the modifications necessary and the hurdles we had to overcome to integrate the algorithms. We will present the effects of the modifications and our insights and lessons learned while implementing, using and evaluating the assignment and scheduling algorithms.

Terminology: When we speak of an *optimized algorithm*, we mean the algorithm with our modifications applied while the *naïve algorithm* refers to the version by the original authors. The RTSC *analysing a system* means that the compiler is able to determine if a system is feasible or not. Failure to analyse a system indicates that the RTSC ran out of memory. Where we say that the algorithm was able to *schedule a system*, we mean that a feasible schedule has been found.

A. Conceptual Hurdles

The original *assignment* algorithm by Peng et al. models directed dependencies by moving release times. Explicit dependencies are therefore converted into an implicit form. In the context of the workloads this algorithm is intended for, this may be sufficient. However, in the RTSC all ABBs belonging to the same task inherit that task's deadline, which is valid since all of the task's ABBs must have finished by said deadline and precedence relationships are enforced by directed dependencies. Yet, unlike precedence constraints, deadlines impact a solution's cost and therefore situations arise where ones that should cause different cost are indistinguishable w. r. t. their NTRT, preventing algorithmic termination. Therefore – similar to the approach proposed in [19] –, in addition to shifting release times from front to back, the implementation also performs a transformation on deadlines, shifting these from back to front along the path of the ABB dependencies.

In the original *scheduling* algorithm, if a deadline is inherited, only the *currently running task* inherits that deadline. With the original workload this makes sense, since only the running task *needs* the resource. In our case an entire chain of ABBs, connected by directed dependencies, may require a resource. Therefore the deadline is bequeathed to all ABBs in the chain, preventing uncontrolled priority inversion.

B. Performance Hurdles

From a real-time engineer's point of view, it is often necessary to split a system into fine-grained artefacts for scheduling and assignment so it becomes possible to schedule the load at all [20]. The ABB concept provides such a fine-grained subdivision of the system. However, the resulting host of ABBs leads to serious problems: The search space resulting from the fine-granular decomposition is much larger than the task-level problem the algorithms have originally been designed for. Since the search space grows *exponentially* with the number of ABBs it may become so large that finding a feasible assignment and schedule is impossible due to limited computational resources. The target domain of multi-core systems aggravates this problem further. For this reason we introduced a number of optimizations while exercising great care so as not to break (cf. Section V) the algorithm's optimality. The idea here was that in places where the algorithm's behaviour is *unspecified*, it is

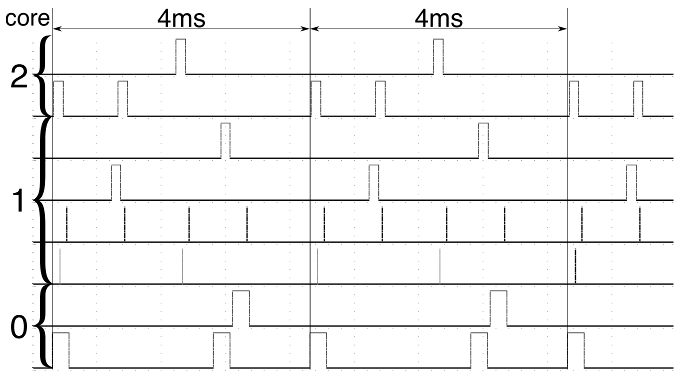


Figure 2. Oscilloscopic trace of the Highstriker experiment: Each row shows one subtask and the hyperperiod is 4 ms. The Highstriker's short but critical tasks are allocated and scheduled on core 1 next to two soft-real-time tasks. The other long-running extra tasks have been scheduled on cores 0 and 2.

adequate to make a deliberate choice that guides the algorithm towards early termination. Therefore, the implementation of the algorithm continues to use the original cost function as long as it can distinguish solutions. However, if two solutions have the same cost, the one that has more ABBs assigned is preferred. This solution is deeper in the tree and thus closer to algorithmic termination.

C. Implementation Hurdles

a) Knowing the Execution Platform: The measures introduced so far enabled the RTSC to employ the two algorithms. However, for deployment on real hardware, additional requirements beyond theoretical models had to be met. The original *scheduling* algorithm adjusts deadlines and release times similar to the assignment algorithm. In the case of the *scheduling algorithm*, however, this approach would have lead to many expensive context switches, negatively affecting schedulability. Therefore, instead of shifting deadlines and release times, an *explicit ready queue* of ABBs is maintained. This measure has the same effect as shifting deadlines and release times but avoids context switches. Additionally, to further reduce detrimental context switches, ties between equivalent ABBs are broken such that a context switch is avoided if possible.

Earlier the *assignment* algorithm's cost function was extended to deal with complexity due to the large number of ABBs. The cost function was extended further to avoid expensive context switches. Furthermore, in the assignment as well as the scheduling algorithm context switches are punished by injecting additional WCET into each ABB that causes a switch.

Another difficulty are the timers, which have a finite resolution and hence can trigger alarms only at *discrete points in time*, and therefore release tasks only at these points. This fact, however, is abstracted away in the design of the original scheduling algorithm, which assumes that tasks can be released at arbitrary points in time. Unless a discrete time base is enforced, the algorithm might slide into solutions that are schedulable on a continuous time basis but cannot run on actual hardware. Therefore, we introduced an *explicit scheduling grid* into our model, having ABBs only scheduled at points in time where a hardware tick occurs.

b) Migration: One central problem on the way to a multi-core-capable RTSC is that inevitably jobs have to be cut apart so they can migrate between cores, since otherwise it may be impossible to find a feasible schedule [20]. However, one requirement for the RTSC is that it must be possible to execute the resulting system on a minimalist, time-triggered run-time system that does not provide the facilities necessary for migration. Therefore, it is the RTSC's job to conduct all steps necessary for *migration at compile time*. The RTSC breaks the tasks and subtasks of the real-time system up into ABBs and migration can only happen at ABB boundaries. Therefore – from an external perspective – tasks and jobs migrate, while – from an internal point of view – ABBs always run to completion on the core they were started on. From a technical perspective, whenever migration becomes necessary, a *shadow stack* for the resulting job fragments is created, converting formerly local job state into global state shared by the job fragments. For the second part of the job fragment a function wrapper is generated which is passed a reference to the shadow stack so the shared state becomes accessible.

D. Experimental Setup

a) Case Study – Consolidation of Legacy Systems: The Highstriker real-time demonstrator consists of a vertical acrylic glass tube ringed by electromagnets. Inside the tube is a freely moving ferromagnetic core that can be manipulated by the electromagnets. Above each electromagnet there is a light barrier that detects the core shortly before it enters the coil's area of influence. The Highstriker's aim is to pass the core from electromagnet to electromagnet without ever dropping it to the ground. To do so the electromagnets have to be activated by an Infineon Aurix processor in time to catch the core but not so early that the core is accelerated towards the ground. The Aurix is a five-core microcontroller with a CPU frequency of 200 MHz commonly used in automotive applications. Four cores run in pair-wise lock step.

The relative deadline of the event that signals that a light barrier has been passed is 500 µs. For the RTSC the event-handler job is sampled with a period of 1 ms and a second job managing the Highstriker's state machine is sampled with 1 ms too. Both job's deadlines are hard. To show that it is possible to consolidate legacy software with the help of the RTSC we added five additional periodic tasks connected by directed dependencies, standing in for long-running soft-real-time jobs. Figure 2 shows the result of the experiment. All of the highstriker's deadlines as well as the soft ones of the additional tasks were met.

b) Statistical Evaluation: We decided to use synthetically generated random input systems that can be analysed automatically to evaluate the RTSC on a large scale and to examine the influence of various parameters on the performance of the compiler. Therefore, we implemented a generator tool that creates random real-time systems, consisting of sources and system descriptions for an OSEK OS executive. The generation process can be parametrised by the number of tasks, directed dependencies and the utilization of the system, just to name a few.

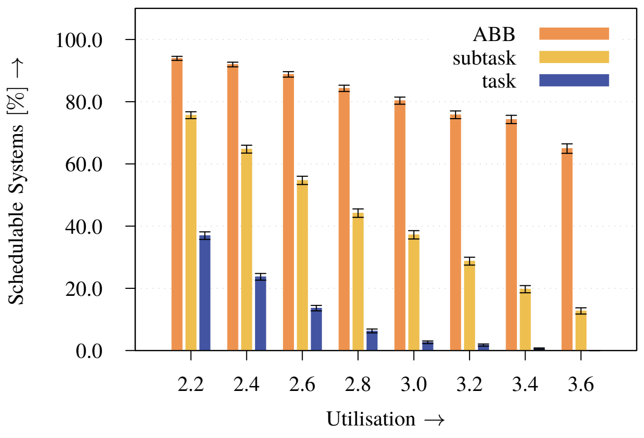


Figure 3. Prevalent assignment and scheduling on *task* or *subtask-level* compared to scheduling at the granularity of ABBs for the same set of 12,812 random systems with up to 68 ABBs.

The solution space of both assigning and scheduling grows exponentially with the number of ABBs, while the number of tasks and subtasks is largely irrelevant. As exploring the impact of varying amounts of ABBs was more interesting to us, we decided to fix the number of cores to four. Nevertheless due to the large system sets we wanted to explore with resources available to every developer, we limited the available memory per RTSC process to 7 GB for each run.

E. Lessons Learned

a) Usefulness of the IR: In order to assess the usefulness of the ABB IR an experiment was conducted to compare assignment and scheduling for the usual *task*, *subtask* and the ABB granularity. 12,812 Systems with up to 68 ABBs were scheduled for a four-core CPU with each of these granularities. As can be seen in Figure 3, our approach improves schedulability, especially in case of higher system utilization. This effect can be explained with the fact that fine granular decomposition into ABBs provides many more migration opportunities and degrees of freedom to the algorithms.

b) From Theory to Reality: Our experience with the RTSC shows that the selected algorithms are fragile if their *often implicit* assumptions are not met. These algorithms are designed with much smaller problem sizes in mind, namely assignment and scheduling on the task level of real-time systems designed 20 years ago. Due to the inherently NP-hard nature of the scheduling problem, the solution space explodes if the algorithm and its input are not tuned to the assumptions made by the underlying theoretical model. Here, tool support is mandatory since it is impossible for an engineer to explore the entire solution space by hand.

Fortunately, the selected algorithms allowed for optimization on the implementation level as randomness is used to simplify some decisions in performance-critical paths. For a B&B algorithm, this randomness is not an inherent property of its working principle, as would have been the case with a heuristic algorithm. We exploited this underspecification by adding more

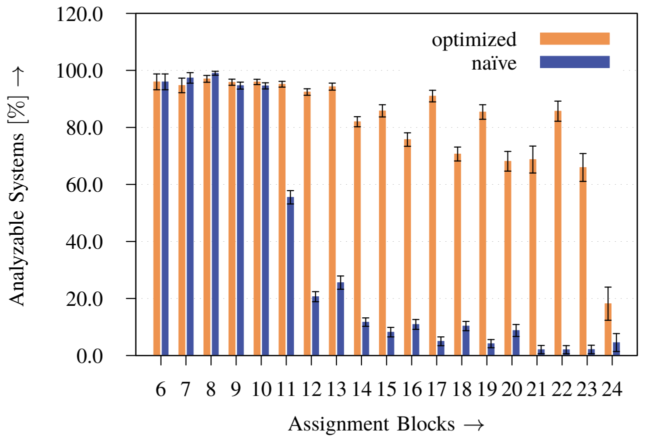


Figure 4. Effect of optimizations: The same random system set of 4,814 systems was analysed two times for a four core CPU, at first with the original algorithm and then with our optimizations.

distinct criteria to force the assignment algorithm to terminate as quickly as possible. For the experiment depicted in Figure 4, the same random system set was analysed by the RTSC once with and once without the optimizations. The generated systems consisted of 2 to 3 tasks including 1 to 4 subtasks summing up to a maximum of 12 interacting jobs connected with up to 6 additional directed dependencies. These parameters lead to systems consisting of 6 to 24 ABBs that had to be allocated. For both experiments we made a maximum of 7 GB of RAM available to each RTSC process. The range of analysable systems could be extended massively. Besides that, one optimized RTSC run took only 4.460 s and consumed 424 MB on average compared to 44.455 s and 4.844 GB without optimizations.

c) Relate Intention and Effect: The primary purpose of the RTSC is to provide the engineer with a tool in which a real-time system can be expressed conveniently. Instead of teaching engineers yet another method for the design of real-time systems, as would be the case with model-based approaches, it should be possible to stay with the textbooks. To enable transformations, the systems engineer's intention for the real-time system has to be translated into models that first the tool can deal with (ABBs and their dependencies), second the employed algorithms can handle, and finally the target platform is compatible with (release times).

One example of such a translation is the way we moved deadlines from back to front in the assignment algorithm: The original algorithm by Peng et al. cannot deal with the explicit model employed in the RTSC, which expresses directed dependencies by edges connecting ABBs. These relationships were initially not respected in the assignment algorithm's scheduler and had to be translated into shifted deadlines in order to be represented in the cost function. This example demonstrates the overall need for transformations between varying models, to make the different algorithms usable and interoperable. Consequently, the abstract and implementation-independent concept of ABBs is the enabler to achieve this very transformation, making it the ideal vehicle to implement and evaluate different algorithms.

With this enhanced version of the RTSC we now have a tool that handles tasks efficiently that up to now engineers painstakingly had to do by hand or couldn't do at all.

V. Discussion and Threats to Validity

We are aware of the fact that threats to validity arise from various sources. In this section, we discuss our approach and face the experimental results with the most common issues from appropriateness to generalisability.

In this paper we introduced several modifications to the original algorithms as well as to the RTSC. Here, we argue that these modifications do not endanger the correctness of the algorithms or change real-time systems in a way that would violate real-time properties. The assignment algorithm's cost function was changed slightly to make it easier to find a feasible assignment quickly and to discourage context switches. However, these modifications were limited to areas where the original algorithm makes an arbitrary decision anyway. Since the former did not harm the algorithm's optimality, even if that arbitrary decision is always unfavourable, we conclude that a guided decision would not do harm either. The measurements showed that due to this change the RTSC was able to schedule $\approx 50\,\%$ more real-time systems on average with the limited resources available. The same reasoning applies to the optimization of the cost function avoiding context switches.

The RTSC's assignment algorithm's scheduler moves deadlines from back to front along chains of dependent ABBs; deadlines are only tightened, never relaxed. This modification of the workload does not violate real-time properties. Here, the RTSC's explicit model (directed dependencies) is converted to the implicit model employed by the algorithm (individual deadlines). Bequeathing deadlines to chains of ABBs is justified too. It enforces correct behaviour w. r. t. synchronization – otherwise jobs would yield an inherited priority before being done with a resource, leading to uncontrolled priority inversion.

The fact that deadlines and release times are not shifted in the scheduling algorithm is correct, too. Release times and deadlines still end up where they would have without the modification while avoiding the harm done by encouraging costly context switches.

One serious limitation of the RTSC is not due to the algorithms employed or the fact that the workload is modified in some way. Rather, the problem of accurately estimating WCETs on current CotS multi-core systems has not been solved, not even in theory [21]. The impact of shared memory buses and caches on actual execution times may be tremendous, and using the worst-case scenario would yield overly pessimistic results. As a consequence, workloads that are schedulable in practice would appear unschedulable to the RTSC. Therefore our current solution is limited to applications that fit into core-local caches or scratch-pad memory. This limitation is partially alleviated by the fact that most current multi-core processors intended for real-time applications, like Infineon's Aurix, rely almost completely on core-local memory. Special-purpose hardware like the one proposed in [21] may allow for tighter WCET bounds but currently exists only as a research prototype.

Furthermore, the modified versions of the algorithms still have to deal with NP-hard problems. Therefore the number of ABBs in a system that can be mapped successfully to multi-core hardware is limited.

To make it easier for other researchers to reproduce our results, we made the input systems used for creating the plots in this paper, the source code of the processing node assigner and that of the scheduling algorithm available.

Project details, data and sources:
`http://www4.cs.fau.de/Research/RTSC`

VI. Conclusion and Outlook

Undeniably, time-triggered systems have their advantages in terms of verifiability. Still, such systems are poorly adopted in the industry. We believe that, beyond pure theory in terms of algorithms for assignment and scheduling, proper tool support is a key aspect for time-triggered multi-core systems to be widely used. One step towards this goal is the RTSC, a compiler-based tool that is able to effectively and efficiently generate time-triggered multi-core systems from given real-time applications.

By way of example of two existing algorithms for assignment and scheduling, we illustrated the challenges that arise in the transition from theory to practice: a naïve implementation quickly becomes infeasible even for small applications and systems. Moreover, implementation aspects such as explicit consideration of runtime costs like context switches can have a strong influence on the feasibility of given real-time systems.

We were able to solve those issues by combining tool-based analysis with a tailored implementation of the algorithms relying an architecture-independent IR by ABB graphs. These contain the system's structural and temporal properties as intended by the developer such that it can be transformed automatically to fit the algorithm's needs, creating the correct temporal effect in the time-triggered variant. We showed the feasibility of consolidation of legacy applications by generating code for the Aurix processor that executes the Highstriker real-time demonstrator running in parallel with additional load.

With the RTSC, developers now have an efficient tool at hand, which allows for a tight feedback loop within the development process – moving the automatic generation of time-triggered multi-core real-time systems a fraction closer to reality.

We continue our way to improve the general applicability of the RTSC and envision the following future improvements:

Locality: To improve the schedulability and WCET estimates, we intend to implement an integrative approach: assignment and scheduling details available within the RTSC can be leveraged by the aiT's WCET analysis. This way, beneficial effects of locality of ABB chains can be taken into account.

Heterogeneity: Asymmetric multi-core processors are becoming more and more common in embedded applications. We intend to extend the RTSC to support these settings, subsequently advancing into the realm of distributed systems.

Mixing Real-Time Paradigms: The time-triggered paradigm has serious limitations when handling high-frequency sporadic events. In a time-triggered real-time system these have to be polled as per the *sampling theorem*, even if the minimum interarrival time is only a corner case. We intend to extend the RTSC to generate hybrid real-time systems, handling high-frequency sporadic events in an event-triggered fashion and periodic task in a time-triggered way.

References

[1] F. Scheler and W. Schröder-Preikschat, "The RTSC: Leveraging the migration from event-triggered to time-triggered systems," in *13th IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '10)*. Washington, DC, USA: IEEE, May 2010, pp. 34–41.

[2] D.-T. Peng, K. G. Shin, and T. F. Abdelzaher, "Assignment and scheduling communicating periodic tasks in distributed real-time systems," *IEEE TOSE*, vol. 23, no. 12, pp. 745–758, 1997.

[3] 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.

[4] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Washington, DC, USA: IEEE, Mar. 2004.

[5] J. W. S. Liu, *Real-Time Systems*. Englewood Cliffs, NJ, USA: Prentice Hall, 2000.

[6] B. B. Brandenburg and J. H. Anderson, "On the implementation of global real-time schedulers," in *30th IEEE Int. Symp. on Real-Time Sys. (RTSS '09)*. IEEE, 2009, pp. 214–224.

[7] F. Cerqueira, M. Vanga, and B. B. Brandenburg, "Scaling global scheduling with message passing," in *20th IEEE Int. Symp. on Real-Time and Embedded Technology and Applications (RTAS '14)*. IEEE, 2014, pp. 263–274.

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

[9] P. Gopinath and R. Gupta, "Applying compiler techniques to scheduling in real-time systems," in *11th IEEE Int. Symp. on Real-Time Sys. (RTSS '90)*. Washington, DC, USA: IEEE, Dec. 1990, pp. 247–256.

[10] K. Albers, F. Bodmann, and F. Slomka, "Hierarchical event streams and event dependency graphs: A new computational model for embedded real-time systems," in *18th Eurom. Conf. on Real-Time Sys. (ECRTS '06)*. Washington, DC, USA: IEEE, 2006, pp. 97–106.

[11] T. A. Henzinger, C. M. Kirsch *et al.*, "Distributed, modular HTL," in *30th IEEE Int. Symp. on Real-Time Sys. (RTSS '09)*. Washington, DC, USA: IEEE, Dec. 2009.

[12] I. Gray and N. C. Audsley, "Exposing non-standard architectures to embedded software using compile-time virtualisation," in *2009 Int. Conf. on Compilers, Architectures, and Synthesis for Embedded Systems (CASES '09)*. New York, NY, USA: ACM, 2009, pp. 147–156.

[13] P. Eles, A. Doboli *et al.*, "Scheduling with bus access optimization for distributed embedded systems," *IEEE VLSI*, vol. 8, no. 5, pp. 472–491, Oct. 2000.

[14] M. Freier and J.-J. Chen, "Time-triggered communication scheduling analysis for real-time multicore systems," in *10th Int. Symp. on Industrial Embedded Systems (SIES '15)*. Washington, DC, USA: IEEE, June 2015, pp. 1–9.

[15] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comp. Surv.*, vol. 43, no. 4, Oct. 2011.

[16] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers," in *31st IEEE Int. Symp. on Real-Time Sys. (RTSS '10)*. Washington, DC, USA: IEEE, Dec. 2010, pp. 14–24.

[17] E. Massa, G. Lima *et al.*, "Optimal and adaptive multiprocessor real-time scheduling: The quasi-partitioning approach," in *26th Eurom. Conf. on Real-Time Sys. (ECRTS '14)*. IEEE, 2014, pp. 291–300.

[18] G. Buttazzo, E. Bini, and Y. Wu, "Partitioning real-time applications over multicore reservations," *IEEE TII*, vol. 7, no. 2, pp. 302–315, 2011.

[19] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *Real-Time Systems Journal*, vol. 2, no. 3, pp. 181–194, 1990.

[20] G. Levin, S. Funk *et al.*, "Dp-fair: A simple model for understanding optimal multiprocessor scheduling," in *22rd Eurom. Conf. on Real-Time Sys. (ECRTS '10)*. Washington, DC, USA: IEEE, 2010, pp. 3–13.

[21] M. Schoeberl, S. Abbaspour *et al.*, "T-crest: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449 – 471, 2015.