International Journal For Advanced Research In Science & Technology
A peer reviewed international journal www.ijarst.in
ISSN: 2457-0362

OSEK-V: APPLICATION-SPECIFIC RTOS INSTANTIATION IN HARDWARE

[1]*BURRI NARENDER REDDY, M.Tech Assistant Professor, narendarburri@gmail.com*
[2]*P.SHANKAR, M.Tech Assistant Professor, shankardvk48@gmail.com*
Department-ECE
Pallavi Engineering College Hyderabad, Telangana 501505.

ABSTRACT
The employment of a real-time operating system (RTOS) in embedded control systems is often an all-or-nothing decision: While the RTOS-abstractions provide for easier software composition and development, the price in terms of event latencies and memory costs are high. Especially in HW/SW co-design settings, system developers try to avoid the employment of a full-blown RTOS as far as possible. In OSEK-V, we mitigate this trade by a very aggressive tailoring of the concrete RTOS instance into the hardware. Instead of implementing generic OS components as custom hardware devices, we capture the actually possible application–kernel interactions as a finite-state machine and integrate the tailored RTOS semantics directly into the processor pipeline. In our experimental results with an OSEK-based implementation of a quad rotor right controller into the Rocket/RISC-V soft core, we thereby can significantly reduce event latencies, interrupt lock times, and memory footprint at moderate costs in terms of FPGA resources.

KEYWORDS
Application special processor design, hardware-assisted real-time scheduling, OSEK

1 INTRODUCTION

This paper addresses the hardware–operating-system boundary in embedded control systems. Our modern lives are driven by these special-purpose systems [30]: We can already more than a hundred of them in our car [5], dozens of them in our household appliances, and trends like the Internet of Things (IoT) will further increase their role for everyday life. Embedded control systems typically have to full dedicated, pre-denied task in a cyber-physical context, often under the consideration of strong safety and timing requirements. As they are employed in goods of mass production (such as cars), the per-unit

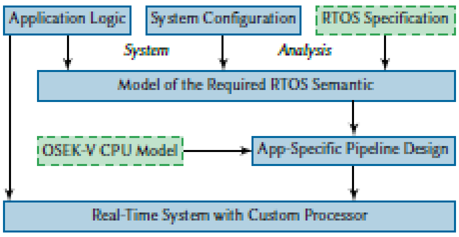


Figure 1: The OSEK-V Approach with Application-Specific (blue) and Generic (green, dashed) Fragments

cost pressure is high. Hence – if at all – a compile-time tailor able real-time operating system (RTOS) is employed as system software, but in many cases developers try to avoid the costs of even a small RTOS kernel. Compared to bare-metal software or even discrete hardware, solutions using an RTOS are typically less analyzable/ predictable and induce much higher event latencies and memory costs. On the other hand, the abstractions by the RTOS (e.g., prioritized threads, alarms, resources) significantly ease the development of more complex and composable control applications. However, even in cases of HW/SW codesign, we often see an all-or-nothing approach: Engineers either avoid employment of RTOS abstractions (which complicates software development) or instantiate a complete RTOS as a (costly) standard software component. In this paper, we resolve the all-or-nothing gap in HW/SW co-design settings by combining the best of both worlds: The idea is to keep the RTOS interface for easy and composable application development, but aggressively tailor its actual implementation to the very specific usage pattern of the concrete application directly into the hardware.

The idea to push the operating system (or parts thereof) into (custom) hardware to improve on event latencies is a long-established of research (e.g., [6, 3, 21, 24, 15, 11]). In contrast to such previous work, we perform a much tighter tailoring of the OS and hardware based on our whole-system approach: Instead of instantiating dedicated components (such as the scheduler) as an additional hardware device besides the CPU, we integrate the RTOS semantics directly into the CPU pipeline. E_ectively, the concrete RTOS interaction model (actually used syscalls and their call-site context) becomes an e_cient and application-tailored extension of the processor's instruction set and register _les. This direct processor integration avoids the costs of a full-blown RTOS, but exposes properties that are hard to achieve software-only on modern architectures: Perfectly

International Journal For Advanced Research In Science & Technology
A peer reviewed international journal www.ijarst.in
ISSN: 2457-0362
predictable timing of all RTOS interactions (which take just a few processor cycles), no kernel-induced cache evictions, drastically reduced interrupt lock times. From the security point of view, the strict tailoring of the RTOS reduces its "misuse capabilities", the instantiation in hardware e_ectively eliminates the possibility to inject code into the kernel domain. In combination with memory protection mechanisms (not addressed in this paper), erfect isolation without executing kernel code would be possible. In previous work, Dietrich et al. [9] have presented a static cross-kernel and whole-system analysis, as well as an applicationspeci _c _nite-state machine (FSM)-based kernel implementation [8]. With this paper, we improve the e_ciency of the system analysis, integrate the FSM-based representation into an actual processor pipeline, and employ further tailoring of system components that become possible at the hardware level. In particular, we claim the following contributions: We present a method to catch the semantics of a concrete RTO instance as a FSM to provide for an e_cient hardware implementation. By application-speci_c instantiation of a standard RTOS interface into the processor pipeline we achieve low syscall and interrupt latencies. The whole tailoring process is fully automated; hardware and software variants are generated on demand. Our open-sourced implementation OSEK-V covers the automotive OSEK/AUTOSAR RTOS standards [23, 2] and integrates their application-tailored semantics into the Rocket RISC-V core [17, 34]. The rest of the paper is organized as follows: Section 2 describes the system model and gives an overview on our approach: Starting with the application, a system state machine is derived in Section 3 and integrated into the CPU pipeline in Section 4. We evaluate our approach in Section 5 on the example of a real-world _ight-control application, discuss the results in Section 6, and give an overview on further related work in Section 7 before concluding in Section 8.

2 SYSTEM MODEL AND IDEA

We assume a static real-time control system with _xed-priority scheduling: One application is combined with a statically con_gured RTOS (all threads and interrupt handlers are known/derivable at compile time) and delivered as a system image. We furthermore assume a static application structure (no dynamic code loading, no invocation of syscalls via nontrivial function pointers). Note that these requirements, while appearing harsh from the viewpoint of general-purpose computing, are common practice and basically ful_lled inherently by the majority of real-time control systems: They are mandated by the dominant safety (e.g., MISRA-C [10], ISO 26262 [13]) and RTOS industry standards anyway. For instance, ARINC 653 partitions [1] (avionics), µITRON [28] and OSEK/AUTOSAR [23, 2] (automotive), but also the POSIX.4 real-time extensions (with SCHED_FIFO) all prescribe _xed-priority scheduling of a well-known set of tasks.

2.1 Our Approach In a Nutshell

Figure 1 visualizes our concept of usage-based tailoring of the RTOS down into the hardware: First, we perform an analysis of the speci_c application and its described system conguration to extract all possible interaction between application and RTOS as a _nite-state machine. This system state machine (SSM) mimics the semantics of the RTOS for the concrete application; it receives input signals (synchronous system calls and interrupts), adapts its internal state, and exposes the currently running thread as an output signal. Second, we integrate this SSM into an applications CPU design: the application triggers the SSM with newlyintroduced instructions and the pipeline reacts by dispatching to another hardware thread. As the result, we get an automatically tailored computing system for the concrete real-time application. Of course, in the general case such a RTOS-FSM would be intractable due to state explosion: The internal kernel state of an event-triggered RTOS encompasses the ready list, thread contexts, the running thread, and so on. Every syscall is a potential point of rescheduling at which, depending on the chosen scheduling strategy and the dynamic state of the ready list, some other thread may be continued that, in turn, may trigger further syscalls. Still, our approach is tractable due to two facts: _rst, we rely on a system model with an inherently bounded number of possible system states. Second, we supply the system analysis with application-speci_c information to reduce indeterminism as far as possible at compile time: We exploit static knowledge about the RTOS con_guration and its semantics in combination with a whole-system analysis across all control _ows of the application gure out how the RTOS is actually used. Thereby, we can reduce he number of possible states drastically, as the outcome of many scheduling decisions can be derived (or at least constrained) ahead of time [9]. This, in turn, provides for an e_cient implementation in hardware, where we integrate the RTOS and its tailored hardware components (e.g., thread contexts or timers) directly into the processor pipeline. Without loss of generality, we describe our approach in the following on the example of the system model mandated by OSEK. Our actual implementation named OSEK-V covers OSEK/AUTOSAR systems [23, 2] up to conformance class ECC1.

2.2 Overview of OSEK-OS

The OSEK standard de_nes a widely used class of _xed-priority RTOSs and has been the dominant industry standard for automotive applications for the last two decades. Without loss of generality, we based our approach on the RTOS interface mandated by the OSEK-OS standard [23]. In the following, we brie_y introduce the abstractions provided by its API. Basically, OSEK o_ers two main control-_owabstractions: interruptservice routines (ISRs) and tasks (traditionally called threads). ISRs are activated asynchronously by the hardware an have limited access to system services, while threads possess a statically assigned priority and are activated

IJARST
International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
www.ijarst.in
ISSN: 2457-0362

synchronously by software. Threads are allowed to use all system services and are executed according to a _xed-priority preemptive scheduling policy. On each new activation, threads start from the very beginning until their (self-) termination. Critical sections can be synchronized either by a coarse-grained global interrupt lock, or more _ne-grained resource objects. Based on a stack-based priority-ceiling protocol [4], OSEK resources ensure mutual exclusion while preventing deadlocks and unbounded priority inversion. Furthermore, a thread can wait for an event to be set and remains in the waiting state until another control _ow signals the arrival of the event.

app.c

```
TASK(T) {
  kickoff();
  do_computation();
  TerminateTask();
}
```

generated.c

```
TASK(idle) {
  while(1) {
    idle();
  }
}

ISR(alarm_tick) {
  isr();
  increase_counter(C);
  if (check_alarm(C, A)) {
    ActivateTask(T);
  }
  iret();
}
```

app.oil

```
TASK T {
   PRIORITY = 10;
   SCHEDULE = FULL;
};

COUNTER C {
  MAXALLOWEDVALUE = 1000;
};
ALARM A {
  COUNTER = C;
  ACTION = ACTIVATETASK {
    TASK = T;
  };
  AUTOSTART = TRUE {
   ALARMTIME = 35;
   CYCLETIME = 70;
 };
};
```

Figure 2: Example OSEK System. The system conguration (**app.oil**) describes a single thread T and its periodic activation every 70 ticks, which automatically begins at system boot. The application code (**app.c**) includes the implementation of T and the generator materialized the system description into **generated.c**

Recurring periodic as well as aperiodic thread activations or events can be triggered with the help of statically declared alarms. Every alarm is connected to a counter, which typically is driven through a hardware timer. Alarms can be started with a phase/period automatically at system startup, or dynamically at run time. For a speci_c application, the developer declares all system objects and their parameters in a domain-speci_c con_guration _le. Typically, a system generator derives the concrete RTOS instance statically at compile time and links application and OS library into a single system image. In Figure 2, an example OSEK system with one task and one periodic alarm is shown. The application code (app.c) contains the task T, which executes a computation and terminates itself afterwards. The system con_guration (app.oil) denotes that task T has a static priority of 10, is fully preemptable (SCHEDULE = FULL). Furthermore, a counter C is declared and connected to the alarm A, which expires every 70 ticks after an initial phase shift of 35 ticks. On expiration, the alarm A activates task T. During compile-time, the system generator produces a system harness (generated.c): An idle task runs at the lowest priority; the alarm_tick ISR handler manages counters and alarms, when the timer interrupt occurs. In order to explicitly anchor system behavior, we added arti_cial syscalls (idle, isr, iret, kickoff) to the code. In this work, we focus on the OSEK extended conformance class 1 (ECC1), which includes waiting states and resources, but excludes multiple tasks per priority and multiple activations per task. Subsequently, we consider the described RTOS primitives as a markup language for expressing the real-time system (RTS)'s behavior, and use the terms threads (for OSEK tasks) and ISRs to distinguish between the control-_ows types.

SYSTEM STATE MACHINE

Since our approach is application-speci_c, we start with a system analysis on one speci_c RTS to extract an interaction model of application, external environment and the RTOS. The system state machine (SSM) captures the desired kernel behavior (i.e., rescheduling sequence) in the presence of the analyzed application and the environmental model. Dietrich et al. [9] described an applicationspeci _c state-transition graph (STG) that enumerates and connects all possible system states. In this work, we improve the e_ciency of the STG calculation and subsequently derive the applicationspeci _c SSM from it. Within an event-triggered RTS, the RTOS receives signals from two sides: the control application issues synchronous syscalls and external components deliver asynchronous hardware interrupts. In both cases, the RTOS is activated, manipulates its internal state, and materializes the scheduling result through dispatching. The internal

International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
www.ijarst.in
ISSN: 2457-0362
IJARST

syscall issue ordering is restrained by the application logic; the possible sequences of external events is shaped by the surrounding environment.

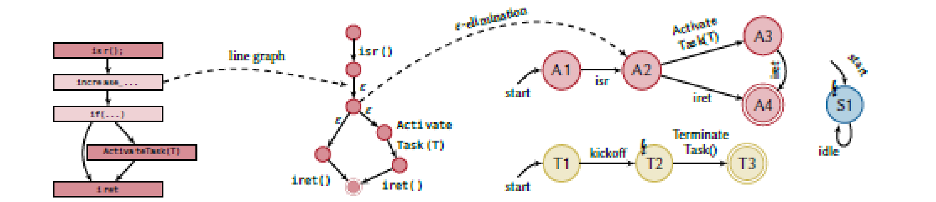
3.1 Application State Machines

For the system analysis, we express the internal application-structure as a set of application-speci_c _nite state machines; every thread and every ISR becomes an application state machine (ASM). These state machines function as signal generators towards the operatingsystem model, which, in turn, orchestrates the execution of several ASMs. In previous work, Dietrich et al. [9] used a (condensed) CFG to express the syscall ordering. In contrast, the ASM representation is more dense and we achieve shorter analysis times due to a reduced number of states.

For each control _ow, we calculate the ASM from the local CFG.1 First, we partition the code into basic blocks that are not maximal, but do contain either a single syscall or only computation code; the later cannot in_uence the system state synchronously. For this separation of in_uences, we demand the application structure (CFG and syscall locations) to be known at system-generation time. Figure 3a shows the basic-block partition for the alarm_isr handler from Figure 2 (syscalls in dark red).

In order to generate a state machine that produces a signal for every executed basic block, we calculate the line graph from the CFG. Each CFG edge becomes a vertex, while each basic block becomes an edge labeled with the block's contents. However, since the OS state can only be in_uenced through syscalls, we replace all computation by "-transitions. It is noteworthy that the transition labels correspond to syscall sites and not syscall types. Figure 3b shows the line graph for the alarm_tick ISR.

We remove the "-transitions by applying standard "-elimination to each ASM. Furthermore, we mark thread states that are reachable through a "-transition as interruptible by an ISR (E) . For each ASM state, the set of outgoing edges names those syscalls possible at one point in the application. Figure 3c depicts the three ASMs for the running example: when the alarm handler is in state A2, ActivateTask and the iret syscall site can be executed next and sent to the SSM.



(a) CFG for alarm_tick ISR (b) Initial Line Graph (c) ASMs: A1-4: alarm_tick ISR; T1-3: thread T; S1: idle thread

Figure 3: Application State Machines Construction for Example from Figure 2.

International Journal For Advanced Research In Science & Technology
A peer reviewed international journal www.ijarst.in
ISSN: 2457-0362

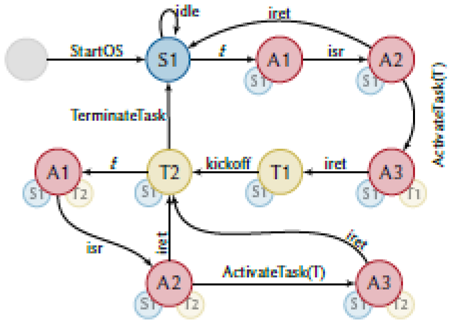


Figure 4: System State Machine. Every vertex is an abstract system state; transitions are triggered by syscalls. The currently running thread overlays the preempted threads. S: idle thread, A: alarm, T: thread.

Currently, we use a very simplistic model to include other analysis results to restrict the external-event model. When a thread (or a group of threads) has an implicit deadline, the triggering event is blocked until the event processing is _nished [9]. Nevertheless, other logic of actions on application level could be derived and used to restrict the event model. For example, it could be de_ned, that a "send bu_er empty" interrupt could only occur after the associated SendMessage() function had been invoked.
The application model and the external-event model are connected to an abstract operating-system model. The OS model is instantiated with the system con_guration and adheres the OSEK semantics [23]. We use the system-state enumeration (SSE) [9] method to combine all three parts and to explicitly enumerate all possible system states in the STG.
The STG is a directed graph of abstract system states (AbSSs), which capture a possible system state at one point in time and hold the information that can in_uence scheduling decisions. Particularly, each AbSS includes a vector of ASM states that indicate the current preemption point of each control _ow. In every state, exactly one _ow is marked as the currently running thread. For details on the STG construction we refer you to Dietrich et al. [9].
Since every transition label in an ASM corresponds to an kernel activation and the SSE only combines several ASMs according to the system semantic, the STG can directly be used a SSM. In Figure 4, the STG/SSM for the running example is given: The system starts

from the idle loop. On an interrupt (E), the alarm handler can activate thread T or directly return to idle. If activated, thread T is dispatched, executes the computation, and terminates itself. Although the alarm can expire again during the computation, the can be activated only once.
3.2 System State Machine Minimization
The resulting SSM already exposes the correct behavior, but the number of states and transition edges is not minimal yet. However, as state-machine minimization is a well covered and long standing topic [22, 12], we will only investigate on the SSM speci_cs.
For the SSM minimization, we meld all states that expose the same observable behavior into a single state. In our case, the observable behavior is the sequence of possible re-scheduling events.
For example, if two states always occur in the same sequence but dispatching happens only after the second one, the _rst state can be merged into the last one. One instance of such a state pair are

IJARST
International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
www.ijarst.in
ISSN: 2457-0362

the A1 and A2 states in our running example (Figure 4). From a system perspective, all A1 states, which are activated by the hardware event (E), are followed by the same re-scheduling sequence as the A2 states Therefore, the A1 state is subsumed by its respective A2 state. We identify such equivalent states by using the Moore algorithm [22] and merge them into one state. If a transition label occurs only at self-loops after the merging, we can safely remove the signal completely from the system.

3.3 Static Alarms

One bene_t of our application-speci_c hardware tailoring is the possibility for optimized components that match the static system con_guration. Besides the SSM, we also detect static alarms within the application, which are very common in embedded control systems: A static alarm starts automatically at boot time, is never recon_gured, and triggers at a constant rate. We check for these properties by static analysis of con_guration and application code. All non-static alarms are dynamic alarms and can, depending on the con_guration, be driven by a timer IRQ with a lower base rate, reducing the IRQ load.

The alarm in the running example (Figure 2) is static: It starts automatically at boot (AUTOSTART = TRUE), has a phase of 35 ticks and a period of 70 ticks, and is never recon_gured.

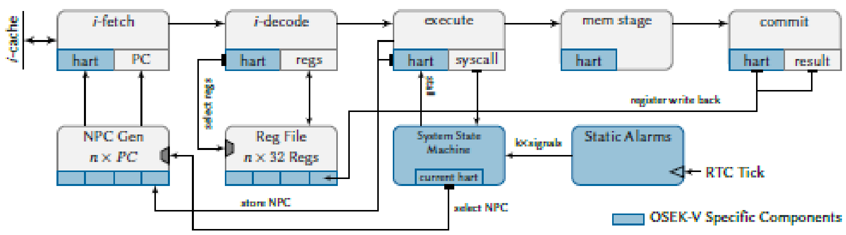


Figure 5: OSEK-V Pipeline

4 DERIVING THE OSEK-V PROCESSOR

In the system analysis, we gathered information to tailor a parametrizable processor pipeline towards the application requirements. We map each OS thread to a hardware thread (harts) and introduce specialized instructions, which interact with the SSM component that controls hart scheduling. Furthermore, a static-alarm component generates periodic signals and activates the SSM asynchronously.

4.1 State Assignment and Logic Minimization

For instantiating the SSM in hardware, we have to provide an e_cient implementation of the state-transition function: Besides the current SSM state, it consumes one system event and returns a new system state together with the (next) hart.

$$SSM : \langle \text{system event} \rangle \times \langle \text{state} \rangle \mapsto \langle \text{state} \rangle \times \langle \text{hart} \rangle$$

The system analysis produces a SSM with symbolic signals (e.g.“TerminateTask”, “ActivateTask(T)”) and states. For a hardware implementation, we have to choose bit vectors for these symbolic values (e.g., hActivateTask¹T ºi = 1012). This choice, known as the state-assignment problem, largely in_uences the minimal required complexity of the hardware implementation. Luckily, several methods have been proposed to solve this problem for di_erent hardware designs [33, 7, 32].

We use the NOVA program [33] to solve our state-assignment problem. NOVA targets optimal encoding for two-level logic implementations and chooses bit vectors for the system calls and the SSM states accordingly, when given a "thread id" encoding, which we choose arbitrarily. Since NOVA internally uses a logic minimizer, we get a minimized truth table for the transition function as a result. With this truth table, and the static-alarm information, we proceed to instantiate the OSEK-V processor.

4.2 The OSEK-V Processor

We built OSEK-V on top of the Rocket core [17], a 64 bit, 6-stage, in-order pipeline. While this soft core is not primarily targeted for embedded systems, a compatible stripped-down 32 bit variant is currently developed. The Rocket implements the RISC-V interface [34], an ISA designed to support computer-architecture research. The Rocket resembles a hardware product family and exposes a multitude of con_guration switches to adapt the implementation towards application requirements.

We have integrated OSEK-V into the Rocket chip generator, which is able to generate a cycle accurate C++ simulator at the

register-transfer level. With our adaptions, it instantiates all components according to the results of the system analysis and wires them up into the pipeline (see Figure 5).

In order to provide fast control-_ow switching, we have extended the pipeline to support hardware threading. The processor is enabled to track di_erent execution _ows (harts) and their contexts simultaneously: Each pipeline stage has a tag to hold information about the currently executed hart; register _le and program-counter generator (NPC Gen) are extended to hold the execution context for multiple hardware threads (harts). The issuing of instructions from di_erent harts is controlled by the SSM.

In our current implementation, every OSEK task and the idle thread is mapped as separate harts, while ISRs still execute in the context of the current hart. This is a trade-o_ between ISR activation times and hardware resource consumption, but could be softened by using one dedicated hart to execute all ISRs.

4.3 Special Instructions and Static Alarms

Furthermore, the OSEK-V pipeline provides two new instructions to interact with the SSM: ssm.ld and ssm.tx. The ssm.tx instruction is used in the boot code to set the instruction pointers for all harts. The ssm.tx instruction communicates its immediate operand as a system event (see Section 4.1) to the SSM, which, in turn, invokes the state-transition function on it.

When an ssm.tx instruction enters the pipeline and reaches the execution stage, the preceding stages are stalled until memory and commit stage have emptied. This stall ensures that all exceptions preceding the ssm.tx instruction remain precise. The execute stage sends the system event to the SSM. While the SSM applies the transition function and updates the "current hart" signal, the pipeline is stalled. If a re-schedule happens, the branch-mispredict logic is reused to _ush the pipeline and to issue an instruction fetch for the new hart's program counter.

Besides the ssm.tx instruction, the static-alarm component also issues system events. Internally, it derives clock signals with di_erent phases and periods from the real-time–clock tick and communicates with the SSM. If one or more alarms expire, the static-alarm component pauses the pipeline and waits for the current instructions to _nish. Afterwards, multiple system events are transmitted atomically to ensure that alarms can trigger simultaneously. The transmission is initiated with one hisr ¹ºi, multiple alarm actions (e.g., hActivateTask¹…ºi) build the body, and the hiret ¹ºi triggers a possible rescheduling.

Static-alarm events are handled like other external events; they are only accepted when interrupts are currently not locked in the processor. Therefore, all regions with locked interrupts still have a run-to-completion semantic.

The OSEK-V functionality is incorporated into the Rocket chip generator; con_guration parameters are encoded as JSON and directly read and interpreted by the hardware design. The parameters include the minimized truth table of the SSM logic, the number of harts, and the static-alarm setup. While the con_guration states only the intentional behavior, the generator decides how implement these requirements. For example, the static-alarm component uses di_erent strategies to derive clock signals depending on the phase and period of the alarm activation.

4.4 System Generation and Startup

Tailored hardware also requires the system software to be tailored. By pushing the OS logic in hardware, only little functionality is left to the software part of the kernel. At boot, the kernel con_gures the program counters with the ssm.ld instruction. The stack pointer is initialized by the thread itself as one of the _rst instructions. When a terminated thread is reactivated, it starts executing at the stack–setup code. The remaining dynamic alarms, as well as regular interrupts are handled by the software kernel. Within the application, syscall sites are replaced with ssm.tx instructions carrying system-event identi_ers. The syscall sites have to be enclosed by a pair of interrupt disable/enable commands, to ensure symmetry between the re-schedule points in ISRs and threads.

We made the process of tailoring the RTOS and the hardware fully automated. The system analysis and transformation is implemented in the dOSEK framework and requires no manual intervention. The Rocket generator reads in the processor con_guration and generates the OSEK-V instance; either as cycle-accurate emulator or as Verilog code.

5 EXPERIMENTAL RESULTS

For our evaluation scenario, we employ the I4Copter [31], a safetycritical embedded control system (quadrotor helicopter) developed in cooperation with Siemens Corporate Technology.

We analyzed the task setup of the I4Copter control application (Figure 6): Threads are activated both periodically and sporadically by three alarms and one ISR. Inter-thread synchronization is realized with OSEK resources and a watchdog thread observes the remote control communication. In total, the scenario consists of eleven threads, three periodic events (alarms), one sporadic interrupt, and one resource. One alarm, which controls the watchdog thread and runs with a low activation rate, is recon_gured at run time, and, therefore, is a dynamic alarm; the two others are static.

We replaced the application logic with checkpoint markers, since we are interested in the interaction between application and kernel. The substitution does not in_uence the analysis, but only exchanges the contents of computation blocks. In total, the system includes 52 system-call sites.

During the SSM construction (Section 3), we used application knowledge about implicit deadlines to restrict the external-event model. For example, the “Sampling”, “Signal Processing”, and “Flight Control” tasks always _nish execution before the 3-milliseconds alarm triggers again. As described Dietrich et al. [9], this incorporation of available information eases the system analysis.

5.1 Performance

As a _rst evaluation, we ran the benchmark scenario for di_erent degrees of tailoring and measured the required clock cycles in the cycle-accurate simulator for di_erent system operations. We ran the benchmark for three hyper periods and give the results in Figure 7. The _rst two variants do not touch the underlying processor, and are only put in place to give a context for the OSEK-V results: The Baseline variant in Figure 7 is the standard dOSEK implementation, where all alarms are dynamic. The Specialized variant uses the system-analysis results to replace the syscall sites with specialized code fragments [9]. Specialized syscalls may omit operations (e.g., _nd the highest-priority runnable thread), if the result can be deduced statically. In the OSEK-V–SSM variant, the pipeline is enriched by a tailored SSM component, while all alarms are managed dynamically. Finally, the SSM+Alarms variant includes the SSM, as well as a static-alarm component that manages two of three alarms. In the cycle-accurate trace, we identify operations on the kernel state that execute atomically. These operations are synchronous syscalls, the timer ISR that manages the dynamic alarms, and the transmission of static-alarm signals. We count the clock cycles required for these operations, while separating cycles that actually execute in the pipeline (a, b), from additional cache-stall cycles (c, d). This separation allows us to discriminate the actual computational cost of OSEK-V from the processor-speci_c cache hierarchy. For the whole benchmark, the e_ective clock cycles, where the processor is not in idle, decrease by specializing syscalls, and even more by using specialized hardware components (a). The reduction stems from the shorter atomically-executed kernel activations, which are synchronized with an interrupt lock. Shorter interruptlock intervals are of special interest for real-time systems; the responsibility of the system increases, when the interrupt latency goes down. Without considering cache stalls, the average length of interrupt locks for all operations decrease by about 80 percent from 195 cycles to 41 cycles (SSM+Alarms). This decrease is mainly driven by the timer ISR. Nevertheless, even without static alarms the average operation takes only 138 cycles (SSM). We distinguish instances of synchronous syscalls, as well as ISR activations, into two classes: events that do not cause a rescheduling, and the ones that actual dispatch to another control _ow. Furthermore, we consider the event with the longest processing time as a relevant information, since we reason about real-time capabilities. Therefore, we give not only average run times, but also maximal run times for each operation (upper bar). First, we consider synchronous syscalls issued by the application code. When no re-schedule occurs, the worst-case times for OSEK-V without static alarms decreases (□79.29%) in a similar range as the version with specialized syscalls (□75.71 %). On average, the tailored hardware is about 50 percent faster than the specialized software. This di_erence is caused by the fact, that the scheduler and dispatcher are often already eradicated through specialization for these non-dispatching syscalls. The advantage for OSEK-V grows for syscall operations with dispatching: The OSEK-V hardware has

International Journal For Advanced Research
In Science & Technology
A peer reviewed international journal
www.ijarst.in
ISSN: 2457-0362
IJARST

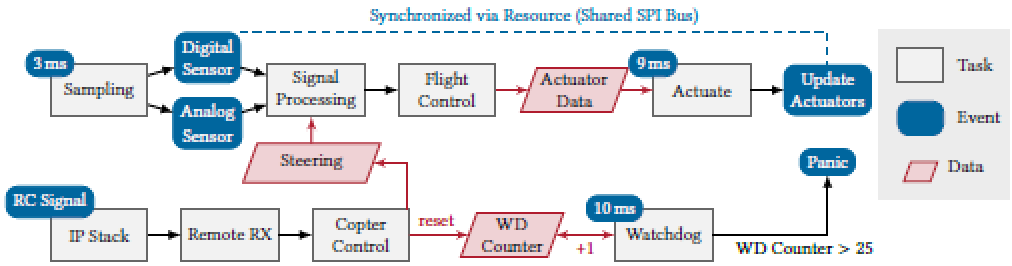


Figure 6: The _ight-control application of the I4Copter quadrotor helicopter.

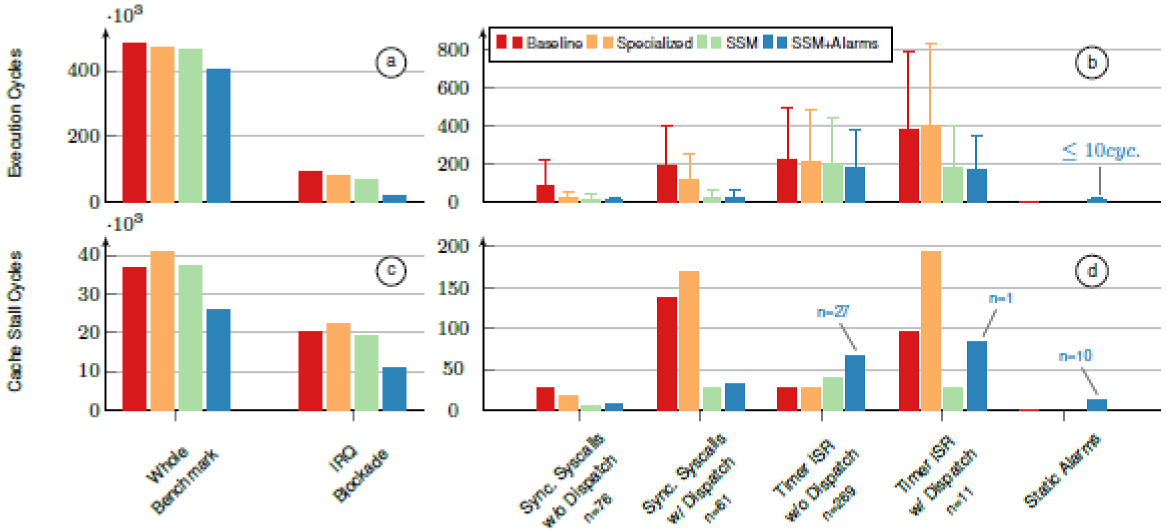


Figure 7: Results for I4Copter in the Cycle-Accurate Simulator generated by the Rocket Toolchain.

at least a 75 percent bene_t over the baseline, while the average bene_t is even over 90 percent.

In purely software-based implementations, every alarm is managed dynamically through a timer ISR. We measured the executed cycles for the whole timer ISR as a single system operation, since the e_ect of the alarm activation manifests atomically at the iret. Again, we distinguished between operations with and without dispatch of another thread. The syscall specialization has only minor in_uence on the cycle counts, regardless of actual re-scheduling. When a timer interrupt does not cause a rescheduling, the SSM variant shows only a minor worst-case improvement (□12.77 %). However, in case of a dispatch, the operation executes about twice as fast (□47.16 %) in the worst case and causes signi_cant lesser cache stalls on average (□72 stalls).

The usage of a static-alarm component (SSM+Alarms) results in several changes in the system's behavior regarding the alarm handling. On the one hand, the number of timer interrupt requests (IRQs) dropped from 280 to 28, since the base rate for the remaining dynamic watchdog alarm could be lowered. This mainly drove the drop on the interrupt-blockade times for the whole benchmark. Additional to the reduced interrupt rate, the execution times for the ISR dropped for the static alarm variant: Since only one alarm had to be manged instead of three, ISRs without dispatch (□29.43 %), as well as with re-scheduling (□59 %) executed signi_cantly faster. Furthermore, a static alarm activation takes at most 10 cycles and in_uences the SSM directly, without utilizing the processor.

The decrease in cache-stall cycles is proportional to the degree of

IJARST
International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
ISSN: 2457-0362
www.ijarst.in

specialization and goes up to □46.79 percent (c, SSM+Alarms). For both OSEK-V variants, the remaining cache stalls for synchronous syscalls and static-alarm activations stem from the instruction fetch for the application code (d). Only the dynamic-alarm handling can lead to cache evictions that stem from executed kernel instructions.

5.2 FPGA Synthesis Cost for the OSEK-V Core

Besides the run-time and latency bene_ts of the OSEK-V approach, we also evaluated the actual cost of having specialized hardware components next to the pipeline. We start this evaluation with an

		Example	*I4Copter*
Generation Time	Seconds	0.06	73.68
Initial SSM	States	9	4834
	Transitions	13	7479
Minimized SSM	States	6	701
	Transitions	9	1246
Transition Function	Clauses	4	781

Table 1: System Analysis Results for the Benchmarks

	Rocket	Example	*I4Copter*	
	(Baseline)	(Figure 3)	SSM	SSM+AL
LUT	29 460	29 216	32 041	32 341
Mem-LUT	1033	1160	2016	2016
Flip-Flops	14 208	14 117	14 129	14 196
F_{max} [Mhz]	26.37	26.57	26.7	25.68

Table 2: Synthesis Results for the tailored OSEK-V Core

overview about the results of the system analysis for the I4Copter benchmark and the running example from Figure 2. These numbers lay the ground to understand the actual implementation costs that arise when we synthesize the di_erent OSEK-V cores.

In Table 1, we show the results for the system analysis, which is executed within the Python-implemented dOSEK framework on a single Intel Core i7-2600 core. The running example from Figure 2 is a small system: its analysis is fast, the initial SSM is already small, and the SSM minimization does not cut away much redundancy.

The resulting minimized logic block, which implements the statetransition function, consists only of four AND clauses (four AND gates with the outputs combined in one OR gate).

For the I4Copter benchmark, the system analysis takes more than one minute, where the run time is mainly driven by the stateassignment phase (96.95 %). Nevertheless, the size of initial SSM still grows exponentially with the size of the system (#IRQs, #Tasks) and reveals a large state machine. Compared with the previous work [8], the size of the initial STG could be cut down signi_cantly (□75.91 %) by the usage of ASMs instead of the control-_ow graph. Still, the state-machine minimization can remove 85.5 percent of the states. The resulting state-transition function takes a 15 bit input vector (state: 10 bits, system event: 5 bits) and produces a 14 bit output signal (hart id: 4 bits).

We used the Xilinx Vivado 2015.2 toolchain to synthesize the di_erent OSEK-V cores for the Zynq-7020 FPGA chip, which is integrated into the ZedBoard platform. The Rocket's pipeline was constrained to run with at least 25 Mhz, while the FPGA features a Fmax of 100 Mhz for a single logic unit.

As expected, the Figure 2 example resulted (see Table 2) only in a small increase in FPGA resource usage (+127 memory LUTs),

International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
ISSN: 2457-0362
www.ijarst.in
IJARST

when compared with the baseline Rocket core. This increase mainly stems from the doubled register _le, since the synthesis tool uses distributed RAM cells to implement the second register _le for the additional hart (idle thread, thread T).
The I4Copter benchmark results in a quite larger core. Without static-alarms, 9 percent more look-up tables (LUTs) are required; these LUTs are mainly used for the SSM component (76.09 %). The

bytes	Baseline	SSM	SSM+Al.
Text Segment (kernel)	14 368	8669	8393
Data Segment (w/o stacks)	1908	410	354

Table 3: Required Flash and RAM Space for the I4Copter
983 additionally required memory LUTs were used mostly for the register _le (96.24 %) to hold the additional hart contexts. These increased FPGA-resource requirements are directly connected to the decreased memory consumption within the system image (see Table 3); the SSM avoids most kernel code and the expanded register _les avoid RAM consumption for the thread contexts.
When we add the static-alarm component (SSM+Alarms), the FPGA resource consumption increases only negligible compared to the variant without this additional hardware component. The static memory consumption for the system image changed not signi_cantly. For all variants, the Xilinx synthesis tool took at least 10 minutes and was always able to ful_ll the 25 MHz timing constraint for the pipeline.

6 DISCUSSION
Compared to other HW/SW codesign approaches, we focus on a single application instead of a small class of applications to unveil emergent system properties. Our narrowed focus exposes unique properties, but we will also discuss the consequential limits.

6.1 Specialization vs. Standardization
We target real-time control systems based on customizable hardware designs, where either a FPGA is employed or a custom chip (ASIC) is intended. This appears to be in stark contrast with the current industry trend of the domain to reduce HW/SW development costs by consolidating custom designs into high-volume (and, thus, cheaper) Commercial o_-the-shelf (COTS) platforms.
We are convinced, however, that the increasing degree of automation on all levels of the customization process will partly reverse this trend – on the longer term an "ASIC on demand" industry will drastically reduce development and per-unit costs of custom hardware. This is already happening, as Patterson and Nikolic outline in a recent EETimes blog post [25]. OSEK-V goes well with this vision as we stay completely compatible on the software side: The application is developed against a standard RTOS interface – but the automatically derived optimized implementation can optionally be pushed into the hardware.

6.2 Application Domain and Scalability
Our approach is applicable, when the in_exibility of static tailoring, culminated in application-speci_c chips, is tolerable. An OSEK-V chip manifests the internal solution structure in silico; an employed ASIC cannot be updated but can only be replaced. For a FPGA system the situation is di_erent, there the OSEK-V processor would become part of the deployed system update. Nevertheless, an update of the OSEK-V core is only required if the application structure (system con_guration and ASMs) changes; other updates can be

deployed as usual. Also, a partial push-down in hardware is possible with a hierarchical scheduling scheme: high-priority threads can be directly mapped to harts, while low-priority might be combined

in one hart and managed in software. This would provide low latencies in critical situations and preserve _exibility otherwise.
The main scalability challenge is the state explosion of the STG. In theory, a system could have an exponentially higher amount of states compared to the number of threads and interrupts. Even when feasible, this burden would precipitate in long analysis and construction times. Nevertheless, we could show that our prototypical implementation handles real-world scenarios faster than the resulting hardware description could be synthesized.
The hardware scalability is determined by two cost factors: the register _le and the SSM logic. Since the register _le has to hold n thread contexts, we must allocate the storage capacity. However, it scales linearly with the number of hardware threads and it could be placed in the FPGA's block RAM. The SSM logic scales with the application complexity and the external indeterminism the system has to face. Small systems that come with a large knowledge about the surrounding environment will bene_t the most from the instantiation of the RTOS semantic in hardware.
Besides the classic real-time control systems used in industry, we see the emerging IoT _eld as a possible application domain. When small control systems become ubiquitous, the trade-o_ between specialization and _exibility will be renegotiated. IoT systems are sold in large numbers, strive a high price pressure, and are tightly coupled to the task and the life span of the employed device. We believe that application-speci_c highly tailored chips are a good _t with these changing design factors.
6.3 Restrictions on Semantic and Application
Besides scalability issues, the restriction we put on (1) the RTOS semantics and (2) the application structure is a threat to the general applicability of our semantic extraction. In essence, the STG includes the inherent determinism that is available at compile time due to the RTOS semantics and its utilization by the application; even in the presence of external interrupts. While this works reasonably well for _xed-priority scheduling, the usefulness is limited on systems that o_er signi_cantly less determinism, such as an RTOS with an earliest deadline _rst (EDF) scheduler or any other scheduler that performs online acceptance tests. On the application side, all interactions with the RTOS have to be detectable at compile time. This forbids any sort of dynamic code loading, the invocation of syscalls via function pointers, and syscall arguments that are not computable at compile time.
Nevertheless, for many domains these restrictions impose little impact in practice – they are prescribed and demanded by the relevant industry and real-time safety standards anyway: EDF scheduling, for instance, is barely used in embedded control systems; the relevant industry standards (such as OSEK/AUTOSAR [23, 2], ARINC 653 [1], µITRON [28], but also POSIX.4) all employ _xedpriority scheduling; the usage of function pointers and any sort of dynamic code modi_cations is discouraged by the relevant coding and safety standards (e.g., [10, 13]). In summary, most of our requirements have to be ful_lled anyway by embedded control systems that needs to pass certi_cation.

6.4 Predictable RTOS Implementation
Real-time developers use worst-case execution time (WCET) analysis

to give upper bounds to the execution budget a job requires.
Tight bounds will simplify the provisioning required to obtain a timing-predictable system. We can foster our predictive power by improving the analysis itself, or by making the underlying platform more predictable in the _rst place. Like the T-CREST/Patmos project [27], OSEK-V provides a more predictable processor platform for real-time applications.
With OSEK-V, the in_uence of the RTOS on the timing behavior is minimized. An SSM activation requires only a few cycles, which are dominated by the instruction-fetch delay for a re-scheduled hart. The SSM execution itself does not evict any cache lines; only the required pipeline _ush in_uences the application. Furthermore, the static-alarm component o_oads periodic system orchestration and, thereby, reduces the interrupt load.
Though timeliness is an important aspect of predictability, security becomes more and more of an issue, especially when control systems are connected to a (public) network. An OSEK-V core inherits only the required RTOS semantic, cuts down on the trusted code base, and pushes the controlling component into the more trustable domain of hardware implementations. When combined with tailored memory protection, where a hart switch implies a protection-domains switch, perfect isolation could be achieved without executing a single kernel instruction. However, that is a topic of further research. RELATEDWORK
Interpreting the OS interface as an extension to the actual processor interface that de_nes a hierarchical machine [29] is an established view in the systems community. Therefore, it is nearby to resolve the partial interpretation of syscalls by moving the OS (or parts thereof) into the (custom) hardware to improve on di_erent nonfunctional properties.
HybridThreads [3] accomplishes low run-time overhead and fast interrupt handling by placing OS component besides the actual processor. Scheduling decisions are dispatched in software through an ISR. Sloth [11] achieves similar advantages for OSEK on standard hardware by delegating all scheduling and dispatching to the interrupt hardware. The FlexPRET processor [35], which also exposes a RISC-V instruction set architecture, achieves a predictable execution of mixed-criticality systems through _ne-grained hardware multithreading, while inter-thread dependencies and synchronization are not considered. The ReconOS project [18], in contrast, provides a uni_ed OS interface, resembling POSIX, for threads and hardware components; coordination and synchronization is still done in software. Mooney and Blough [21] instantiate OS components in hardware to provide an application-speci_c platform; the developer can manually select pre-built components, which are orthogonal to the core services. In contrast to all these approaches, OSEK-V performs an in-depth tailoring of the RTOS: We catch the RTOS semantic from the viewpoint of a speci_c application instead of reproducing a (generic) software implementation in hardware. Furthermore, we directly integrate components into the processor pipeline to achieve _ne-grained and application-speci_c tailoring.

In essence, OSEK-V derives its tailored RTOS semantic by a complete specialization of each syscall at each call site. This somewhat resembles the path-speci_c syscall optimization known from Synthesis [26, 19] or partial specialization as provided by the Tempo [20] framework. Both of these, however, specialize at run time, which (a) requires expensive run-time support and (b) facilitates probabilistic optimizations that can be reverted when necessary. In

IJARST
International Journal For Advanced Research
In Science & Technology
A peer reviewed international journal
ISSN: 2457-0362
www.ijarst.in
contrast, OSEK-V is tailored at compile-time, so all specializations have to be sound and complete in the sense that the resulting RTOS instance can be represented as an FSM.
The usage of FSMs as a whole-system model has also been proposed for deeply embedded sensor nodes to enhance simplicity and energy e_ciency: SenOS [14] is a software event dispatcher and executor for multiple manually-encoded state machines. Kothari et al. [16] derive compact state machines (< 16 states) from TinyOS programs by symbolic execution to foster understanding of existing applications.
8 CONCLUSION
With OSEK-V, we explore the HW/SW design space for eventtriggered _xed-priority real-time systems at the hardware–OS boundary. Starting from a single application and the standardized OSEKOS API, we extract the actual used RTOS behavior as a _nite-state machine. This system state machine is triggered by syscalls and interrupts and controls the thread dispatching. The OSEK-V core maps each RTOS thread to a hardware thread and is accompanied by application-speci_c hardware components that implement the extracted RTOS semantic. Thereby, we unveil desirable nonfunctional properties, like low event latencies (□79 % average IRQ lock times), interference-reduced RTOS execution (□47 % cache stalls in the kernel), and fast thread re-scheduling (□81 % cycles for dispatching syscalls). These improvements come at moderate FPGA cost of 10 percent more LUTs and 86 distributed memory cells per mapped RTOS thread.

REFERENCES
[1] AEEC. Avionics Application Software Standard Interface (ARINC Speci_cation 653-1). ARINC Inc, 2003.
[2] AUTOSAR. Speci_cation of Operating System (Version 5.1.0). Tech. rep. Automotive Open System Architecture GbR, 2013.
[3] Jason Agron,Wesley Peck, Erik Anderson, David Andrews, Ed Komp, Ron Sass, Fabrice Baijot, and Jim Stevens. “Run-Time Services for Hybrid CPU/FPGA Systems on Chip”. In: RTSS ’06. 2006. doi: 10.1109/RTSS.2006.45.
[4] H. Almatary, N.C. Audsley, and A. Burns. “Reducing the Implementation Overheads of IPCP and DFP”. In: RTSS ’15. 2015. doi: 10.1109/RTSS.2015.35.
[5] Manfred Broy. “Challenges in Automotive Software Engineering”. In: ICSE ’06. 2006. doi: 10.1145/1134285.1134292.
[6] Wayne P. Burleson, Jason Ko, Douglas Niehaus, Krithi Ramamritham, John A. Stankovic, Gary Wallace, and Charles C. Weems. “The Spring Scheduling Coprocessor: A Scheduling Accelerator”. In: IEEE Transactions on Very Large Scale Integration (VLSI) Systems 7.1 (1999). doi: 10.1109/92.748199.
[7] S. Devadas, Hi-Keung Ma, A.R. Newton, and A. Sangiovanni-Vincentelli. “MUSTANG: state assignment of _nite state machines targeting multilevel logic implementations”. In: Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 7.12 (1988). doi: 10.1109/43.16807.
[8] Christian Dietrich, Martin Ho_mann, and Daniel Lohmann. “Back to the Roots: Implementing the RTOS as a Specialized State Machine”. In: OSPERT ’15. 2015.
[9] Christian Dietrich, Martin Ho_mann, and Daniel Lohmann. “Cross-Kernel Control-Flow-Graph Analysis for Event-Driven Real-Time Systems”. In: LCTES ’15. 2015. doi: 10.1145/2670529.2754963.
[10] Guidelines for the Use of the C Language in Critical Systems (MISRA-C). 2004.
[11] Wanja Hofer, Daniel Lohmann, Fabian Scheler, andWolfgang Schröder-Preikschat. “Sloth: Threads as Interrupts”. In: RTSS ’09. (Dec. 1–4, 2009). 2009. doi: 10.1109/RTSS.2009.18.
[12] John Hopcroft. An n log n algorithm for minimizing states in a _nite automaton. Tech. rep. Computer Science Department, University of California, 1971.
[13] ISO 26262-4. ISO 26262-4:2011: Road vehicles – Functional safety – Part 4: Product

development at the system level. 2011.
[14] Tae-Hyung Kim and Seongsoo Hong. “State Machine Based Operating System Architecture for Wireless Sensor Networks”. In: Parallel and Distributed Computing: Applications and Technologies. Vol. 3320. LNCS. 2005. doi: 10.1007/978-3-540-30501-9_158.
[15] Paul Kohout, Brinda Ganesh, and Bruce Jacob. “Hardware Support for Real-Time Operating Systems”. In: CODES+ISSS ’03. 2003. doi: 10.1145/944645.944656.
[16] Nupur Kothari, Todd Millstein, and Ramesh Govindan. “Deriving State Machines from TinyOS Programs Using Symbolic Execution”. In: IPSN ’08. 2008. doi: 10.1109/IPSN.2008.62.
[17] Yunsup Lee, A. Waterman, R. Avizienis, H. Cook, Chen Sun, V. Stojanovic, and K. Asanovic. “A 45nm 1.3GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators”. In: European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014 - 40th. 2014. doi: 10.1109/ESSCIRC.2014.6942056.
[18] Enno Lübbers and Marco Platzner. “ReconOS: Multithreaded Programming for Recon_gurable Computers”. In: ACM Trans. Embed. Comp. Syst. 9.1 (2009). doi: 10.1145/1596532.1596540.
[19] Henry Massalin and Calton Pu. “Threads and Input/Output in the Synthesis Kernel”. In: SOSP ’89. 1989. doi: 10.1145/74850.74869.
[20] Dylan McNamee et al. “Specialization Tools and Techniques for Systematic Optimization of System Software”. In: ACM Trans. Comp. Syst. 19.2 (2001). doi: 10.1145/377769.377778.
[21] Vincent J. Mooney and Douglas M. Blough. “A Hardware-Software Real-Time Operating System Framework for SoCs”. In: IEEE Journal on Design and Test of Computers 19.6 (2002). doi: 10.1109/MDT.2002.1047743.
[22] Edward F. Moore. “Gedanken-experiments on sequential machines”. In: Automata studies. Annals of mathematics studies, no. 34. 1956.
[23] OSEK/VDX Group. Operating System Speci_cation 2.2.3. Tech. rep. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2014-09-29. OSEK/VDX Group, 2005.
[24] Arnaldo SR Oliveira, Luís Almeida, and António B Ferrari. “The ARPA-MT embedded SMT processor and its RTOS hardware accelerator”. In: Industrial Electronics 58.3 (2011). doi: 10.1109/TIE.2009.2028359.
[25] David Patterson and Borivoje Nikolic. Agile Design for Hardware. EE|Times blog post. 2015. url: http://www.eetimes.com/author.asp?section_id=36&doc_id=1327291.
[26] Calton Pu, Henry Massalin, and John Ioannidis. “The Synthesis Kernel”. In: Computing Systems 1.1 (1988).
[27] Martin Schoeberl et al. “T-CREST: Time-predictable multi-core architecture for embedded systems”. In: Journal of Systems Architecture 61.9 (2015). doi: 10.1016/j.sysarc.2015.04.002.
[28] Hiroaki Takada and Ken Sakamura. “μITRON for Small-Scale Embedded Systems”. In: IEEE Micro 15.6 (1995). doi: 10.1109/40.476258.
[29] Andrew S. Tanenbaum. Structured Computer Organization. Fifth. 2006.
[30] David Tennenhouse. “Proactive Computing”. In: CACM (2000).
[31] Peter Ulbrich, Rüdiger Kapitza, Christian Harkort, Reiner Schmid, and Wolfgang Schröder-Preikschat. “I4Copter: An Adaptable and Modular Quadrotor Platform”. In: SAC ’11. 2011.
[32] D. Varma and E.A. Trachtenberg. “A fast algorithm for the optimal state assignment of large _nite state machines”. In: ICCAD ’88. 1988. doi: 10.1109/ICCAD.1988.122483.
[33] T. Villa and A. Sangiovanni-Vincentelli. “NOVA: State Assignment of Finite State Machines for Optimal Two-level Logic Implementations”. In: 26th ACM/IEEE Design Automation Conference. DAC ’89. 1989. doi: 10.1145/74382.74437.
[34] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0. Tech. rep. UCB/EECS-2014-54. EECS Department, University of California, Berkeley, 2014.

[35] Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. "FlexPRET: A processor platform for mixed-criticality systems". In: RTAS '14. 2014. doi: 10.1109/RTAS.2014.6925994.