A Task-Level Monitoring Framework for Multi-Processor Platforms

Philipp Ittershagen, Kim Grüttner
OFFIS – Institute for Information Technology
Oldenburg, Germany
{ittershagen,gruettner}@offis.de

Wolfgang Nebel
Carl von Ossietzky Universität
Oldenburg, Germany
wolfgang.nebel@informatik.uni-oldenburg.de

ABSTRACT

In this paper, a monitoring framework for observing properties of tasks running on a multi-processor platform is proposed. We describe the implementation of the framework on a TLM-based virtual platform containing an ARM Cortex A9 multi-core instruction-set simulator and shared memory modules. An application model consisting of periodic tasks and communication channels is used to demonstrate the applicability of the monitoring framework. Based on the application model, we describe a method for deriving a monitor implementation at design time that is able to check the execution order and the platform mapping during run-time. The model is implemented on top of a POSIX-compatible real-time operating system and the monitor is instantiated as a TLM component in the virtual platform. The monitor implementation is then able to check the execution order and the platform mapping of the application against the specification at run-time. Finally, we discuss the monitoring capability and its contribution to a safety concept for fail-safe systems.

CCS Concepts

• **Computer systems organization** → **System on a chip; Redundancy;** *Reliability;*

Keywords

Monitoring; Task execution order; Multi-Processor; Virtual Platform; Transaction Level Model

1. INTRODUCTION

Today's embedded system development is driven by the need to integrate many complex applications on a single system and consequently reduce the cost, size, and power consumption of these integrated systems. This has lead to the development of highly integrated Commercial off-the-shelf (COTS) platforms consisting of high-performance processing elements as well as on-chip programmable logic for implementing dedicated hardware accelerators.

However, ensuring that the functional as well as extra-functional properties of an application mapped to these Multi-Processor Systems-on-a-Chip (MPSoCs) are within the bounds of the specification becomes increasingly difficult due to this performance need. Interfering usage of shared resources on multi-processor platforms, sophisticated software execution features such as out-of-order execution, branch prediction, or pipeline stalls make it impossible to statically check the behaviour with model-based approaches. On the other hand, simulative approaches cannot guarantee that the implemented application behaviour matches the requirements specified by the models due to their limited state space coverage. Furthermore, refining the initial system design towards a target architecture involves many configuration steps and design decision that are hard to trace an validate in an implementation. The ability to check the execution behaviour of the final application against the initial specification during run-time can therefore be a valuable tool in the design of complex, heterogeneous embedded systems. Additionally, a run-time monitoring system can detect execution anomalies caused by faults in fail-safe systems. This is an essential requirement in many safety-critical systems.

Our contribution in this paper is three-fold: First, we introduce a monitoring framework and implement it as a prototype on a virtual COTS MPSoC platform. Next, we present an application model and derive a monitor implementation from its static properties. Finally, we discuss the run-time monitoring framework capabilities applied on an implementation of our application model on the virtual platform. The paper structure follows our contributions. The monitoring framework properties and the virtual COTS MPSoC platform characteristics are described in Section 3. In Section 4, we introduce the application model and derive a monitor implementation from the specified behaviour at design time. Finally, in Section 5, we discuss the monitor implementation and its ability to check the logical task execution order of the application model and its processor mapping. We further discuss restrictions and possible extensions of the monitoring approach towards extra-functional properties such as temporal behaviour and power consumption.

2. RELATED WORK

Execution monitoring with dedicated monitoring systems at run-time is an established academic field and used in many industrial applications. The approaches can be categorized

SCOPES'16 May 23–25, 2016, Sankt Goar, Germany

ISBN 978-1-4503-4320-6/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2906363.2906373

by properties regarding their supported task models, the platforms they are restricted to, and architectural properties such as the partitioning of the monitor implementation on hard- and software components.

Early approaches [6, 8, 10, 15] focussed on the throughput of typical operating system tasks such as scheduling and resource usage coordination. These approaches propose dedicated hardware accelerators to replace critical software parts of the operating system, thereby improving the overall system performance. The concept of *hardware operating systems* with task management and synchronisation primitives implemented in hardware is evaluated in [15] and [8]. This approach can improve the system's real time capability by decreasing the overall response delay. It has also been shown that this approach can increase the performance on multiprocessor platforms [10]. Furthermore, a Portable Operating System Interface (POSIX)-compliant implementation [6] can increase the portability of these approaches. Our approach does not replace the system's scheduling and resource allocation capabilities. Instead, we provide a monitoring interface to the software layer which is capable of checking task properties at run-time by matching them against the initial task model description. This allows us to build our monitoring concept upon existing, robust, and well-established scheduler implementations and additionally gain the verifiable knowledge of a correct property implementation that conforms to the specification.

Similar to the hardware operating system approaches, many contributions provide a hardware monitoring unit capable of creating an on-line schedule of a given task set. The software stack is then modified to execute the tasks according to the scheduling order calculated by the control unit [2, 3, 12, 14]. In [1], the dedicated hardware scheduling unit *CoreManager* is presented. Combined with an instruction-set architecture (ISA) extension [2], it features a hardware-based task management unit with the goal to increase the performance on the targeted MPSoC. Our approach explicitly does not extend the ISA in order to increase the portability on different COTS platforms. The work of [13, 14] proposes a programming model and a task pool unit to drive the execution on the processors in the system. Furthermore, a run-time dependency detection using memory I/O information is performed. The resulting dependency graph is thus built dynamically and used for scheduling the tasks. The approach is targeting the Cell MPSoC platform, which is particularly suitable for image and video processing. Rather than dynamically extracting model properties, our monitoring approach assumes that the properties which should be checked at run-time are already available at design time. This way, we can deliver a traceable property check of the model implementation.

In [3], a special-purpose co-processor is proposed to define and monitor the task execution process in the system. The co-processor features a dedicated ISA to describe the software task model execution semantics that is consequently enforced on the platform. The co-processor's task scheduling implementation is provided to the executing processors through a memory-mapped I/O (MMIO) interface. The authors demonstrate the approach on a bare-metal software stack. Due to our approach targeting COTS platforms containing Field-Programmable Gate Arrays (FPGAs) subsystems, our work does not focus on providing a programmable co-processor. Instead, we use the on-chip programmable logic to implement the hardware module, allowing us to synthesize the monitor implementation based on the given application model properties. Additionally, our implementation builds on top of the existing scheduling implementation instead of introducing its own hardware-based implementation.

Recently, monitoring concepts were proposed which provide additional checks next to validating the functional correctness of the implementation. These monitoring capabilities include extra-functional properties such as timing constraints. A fine-grained timing- and control flow monitoring approach for control flow graphs (CFGs) on a basic-block granularity is presented in [19]. Besides the functional correctness, the monitor also checks the required execution time of basic blocks. Execution time overruns can be detected when comparing the value to a statically analysed Worst-Case Execution Time (WCET). This approaches mainly differs in the granularity and its applicability to run-time environments with dynamic task scheduling, since our monitoring approach is not limited to static task scheduling policies. Another timing-aware monitor implementation has been proposed in [16], where annotations in the source code result in check points for validating the application's timing behaviour. This approach is not easily adaptable to dynamic scheduling scenarios and hence not in the focus of this work. In our approach, the functional and extra-functional correctness is checked by verifying task activities at instrumentation points defined by the designer. This high-level approach allows applications ranging from execution flow monitoring to extra-functional resource consumption checks for power or timing.

Recent approaches emerged featuring opportunistic monitoring concepts. In [12], an opportunistic monitoring mechanism is proposed to mitigate the monitoring overhead of real-time applications. By observing the execution time of a task, any extra monitoring computation can be delegated to execute when the task has enough slack available. However, the approach is restricted to modifiable processor cores, which makes it inapplicable for COTS MPSoC platforms where the targeted processing element is not provided on the FPGA fabric.

The work in [9] proposes a hardware controller for distributed run-time WCET boundary checking in mixed-critical systems. A control-flow graph software model is extended with instrumentation points and a master instance runs on the CPU to control the task execution. The approach however is tightly coupled to the platform due to the use of core-local clock counters and dedicated event/interrupt mechanisms. Additionally, there is no support for dynamic scheduling policies.

To summarise, our approach is targeting highly integrated, heterogeneous MPSoC consisting of high performance general-purpose processing units and programmable logic. In our implementation on a virtual platform, the monitor is accessible through MMIO and uses software instrumentation check points in the task implementation. These instrumentation points are compiled down to simple memory access patterns and thus avoid the need for non-standard compiler extensions or custom instructions. We demonstrate our approach by constructing a monitor semantic for checking the logical execution flow and processor mapping of tasks from an application model. Overall, our approach builds on top of POSIX and is therefore independent of the underlying runtime implementation. It also supports dynamic scheduling

policies, since it does not impose a total, static task execution order. The order of execution is instead checked against the logical dependencies defined in the application model.

3. MULTI-TASKING MONITORING

When designing an embedded system, the active components of the application model are eventually mapped to the resources provided by the platform. Existing application modelling approaches can be divided into two categories.

Dataflow oriented application models denote active components as *actors* which perform operations on their incoming data and produce data for other actors. Control flow dominant application models consist of (hierarchical) event-triggered states. In a partitioning step, these states can be combined into active components that are later executed on the processing resources. Due to these data- and event-driven dependencies, both models define a *logical execution order* on their actors.

In the embedded systems design flow partitioning step, the application model actors and states are partitioned into tasks and mapped to the processing elements of the system. When implementing the tasks on single core platforms, the execution order of the mapped components is defined at design time by providing a static execution sequence which implements the logical execution order of the model. As a consequence, no active component needs to synchronise explicitly when executing the scheduled application model.

One of the challenges in MPSoC featuring preemptive multi-tasking on multiple processing resources is to map the logical execution order defined by the application model to the resources available in the system. The underlying implementation needs to synchronise the active application model components across multiple processing elements. As a consequence, there is a challenge for implementing the logical execution order of the application model, defined by the model semantics, on top of the scheduled execution order of the actors on the platform. In today's presence of high-performance MPSoC platforms, this scenario becomes the common case, since these platforms provide multiple processing elements and are able to execute operating systems that support dynamic scheduling. Thus, to enhance the performance of the system, the execution order is usually not determined statically. Instead, the underlying software stack provides synchronisation primitives to implement the logical execution order on the available processing resources, as defined by the application model. The final execution order of the model thus depends on the dynamic behaviour and correctness of the target implementation.

Based on these observations, we propose a monitoring framework to assess the execution of control- and data-flow driven application models on dynamic multi-tasking MPSoC platforms. The framework provides the infrastructure to check whether application model properties and execution semantics defined in the design flow are correctly implemented on the target implementation at run-time. We will further present an implementation of this framework on top of a virtual MPSoC platform.

The monitoring framework consists of two components. A software-based instrumentation point and a hardware monitoring unit, where the instrumentation points expose the dynamic task execution and completion events to a dedicated monitoring unit. This approach avoids the need of an invasive hardware device to track the dynamic state and thus makes it suitable for a COTS platform. In the monitoring unit, a specific implementation can then define rules based on additional information extracted from the application model, such as elapsed time or logical execution order. The monitor implementation is based on a virtual platform representing the Xilinx Zynq™-7000 series MPSoC.

Virtual platforms provide the advantage of being able to check the complex software stack executing on these MPSoC in an early stage in the design flow, prior to an implementation on a real platform[1]. But, to compensate the functional complexity of the modelled platform simulation, the timing model is only triggered on a coarse-grained level. In this virtual platform, the instruction-accurate timing model of the ARM Instruction-Set Simulator (ISS), although sufficiently accurate for executing the functional behaviour of an operating system and its user-space applications, is incomplete in terms of intra-instruction delays, such as pipeline stalls and branch predictions. Nevertheless, we chose a virtual platform for implementing the monitor framework due to the fast prototyping capabilities and the sufficient model detail in terms of executing the functional behaviour of the hard- and software models executed on the virtual platform. The SystemC-based virtual platform provides a functionally accurate model of a multi-processor platform which is capable of executing an operating system. This allows us to check if the monitor can compare the scheduled execution order on the platform with the logical order and its mapping. The following section describes the platform model characteristics which are of interest to us.

3.1 Platform Setup

The platform model consists of processing elements, an interconnect, and shared memory. Processing elements are able to execute active components, such as actors or tasks, while their shared state is mapped to memory modules. The interconnect connects the processing elements to the memory modules.

Definition 1 (Platform Model)**.** A platform $\mathcal{P} = (\mathbb{P}, \sigma, \mathbb{M}, \mathbb{I})$ is characterised by the set of processing elements $\mathbb{P}$ and their scheduling policy σ, a set of shared memory modules $\mathbb{M}$, and an interconnect $\mathbb{I}$ representing the connections between processing elements and shared memory modules.

Processing Elements. Application model tasks are executed on processing elements. A processing element $P_i \in \mathbb{P}$ is characterized by the tuple (θ_i, μ_i). θ_i refers to the processing element type and μ_i is the size of the local memory in bytes. We assume that local storage is available to map the task code. In particular, this means that instruction fetches do not alter the temporal behaviour of communication across the shared bus.

Shared Memory. Shared memory $M_i \in \mathbb{M}$ is characterized by its size (s_i) in bytes. It will contain the state of mapped communication channels.

Interconnect. An interconnect model $\mathbb{I} = (\rho,)$ is used to represent the routing ρ of the platform interconnect. The interconnect relation $\rho\colon \mathbb{P} \to \mathbb{M}$ describes the structural connection of processing elements $\mathbb{P}$ and memory components $\mathbb{M}$.

[1]Implementation of the proposed monitoring framework in hardware is ongoing work and no subject of this paper.

We use a SystemC-based platform implementation to represent our platform model which contains an ARM Cortex A9 dual-core ISS with symmetric multi-processing (SMP) capability. The interconnect model is implemented with an AXI TLM interconnect and the shared memory modules are implemented as TLM target components. The ISS is further executing an RT-Linux instance on both cores in an SMP configuration, exposing a POSIX API to the user space. Operating systems that support POSIX include RT-Linux, VxWorks [18], and PikeOS [17], or Real-Time Linux (RT-Linux). The platform also contains other components, such as timers, an interrupt handler, a virtual display, and a UART to successfully boot and operate the RT-Linux instance. However, these components are not relevant for our approach are thus not included in the platform model description.

The interconnect model is implemented on top of SystemC-Transaction-Level Modelling (TLM), such that the bus protocol implementation is based on transactions instead of a pin-accurate communication protocol. This modelling approach additionally increases the speedup due to less synchronisation overhead compared to a detailed pin-accurate simulation. SystemC-TLM provides address-based transactions for initiators (bus masters) and targets (bus slaves). The interconnect is modelled on a functional level only. Since it therefore does not contain a timing model, there is no need to arbitrate multiple concurrent requests among the initiators.

3.2 Monitor Implementation

Any task start and stop activities are exposed via events triggered by instrumentation points of the framework. We propose a C++ macro for implementing these software instrumentation points, defined as `SCOPED_CHECK`. With the help of the C++ object lifetime semantics, the macro instantiates a wrapper class to perform an operation when the scope is entered and exited, which we call a *scoped instrumentation*. The entry (exit) of the scoped instrumentation triggers a function call to the monitor's `on_entry` (`on_exit`) implementation.

To integrate the monitor in the virtual platform, we implement a TLM adaptor to refine the monitor method calls to TLM transactions. The adaptor consists of a SystemC module and a TLM target socket which connects it to the platform interconnect. Figure 1 gives an overview of the monitor embedded in the virtual platform. The monitor framework provides a driver to refine function calls to TLM-based MMIO access operations. For both entry and exit functions of the scoped instrumentation, a unique task identifier of the active task instance is encoded in the address operator of the generated bus transaction and the transaction value contains the current processor identifier.

Encoding the task identifier into the address operand makes it possible to use both the transaction address and its value for communicating both function arguments without the need for software-based logical operations in the generated annotation or multiple transactions.

With the POSIX `mmap` call, we are further able to make arbitrary physical address regions accessible for a process. The software driver which converts the function calls to the monitor to address-based transactions uses `mmap` to access the address space of the monitor adaptor. Listing 1 shows the generated ARM target instructions of the scoped instrumentation. As can be seen by the output, only one write

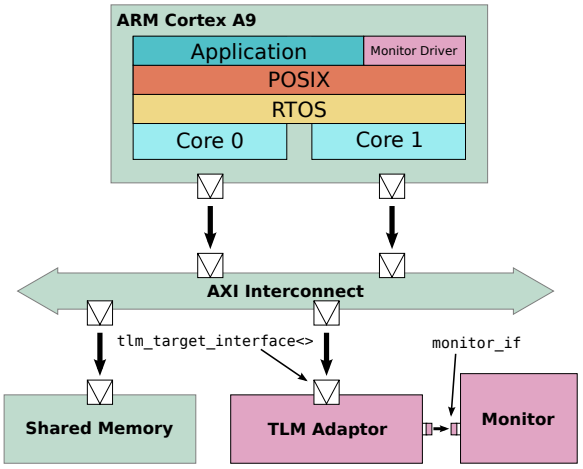

Figure 1: Monitor implemented in the virtual platform.

```
1  ldr   r5, [r4]
2  bl    sched_getcpu
3  ldr   r3, [r4, #4]
4  str   r0, [r3, r5, lsl #2] ; on_entry
5
6  ; inside SCOPED_CHECK
7
8  ldr   r5, [r4]
9  bl    sched_getcpu
10 ldr   r3, [r4, #4]
11 str   r0, [r3, r5, lsl #2] ; on_exit
```

Listing 1: ARM target code instructions implementing the scoped instrumentation entry and exit function calls through MMIO.

transaction needs to be generated for communicating both values. Even in the case of a dynamic task identifier saved in a register (`r5`), the compiler is able to use the register content as an address offset. Since the monitor target base address returned by `mmap` cannot be determined at design time, it is saved on the stack during application setup. The final address for the transaction therefore needs to be calculated by adding the task identifier to the register containing the mapped base address.

4. APPLICATION MODEL

This section defines a structural representation of an example application and its mapping to the platform resources introduced in Section 3.1. The application model captures the behaviour of the system through the notion of communicating actors and a mapping defines which application model entities are mapped to which platform resources.

The following application model represents a simplified version of the mixed-criticality task model presented in [7]. Without loss of generality, this paper targets simple dataflow oriented applications through the notion of communicating tasks that form a directed acyclic graph. Based on Synchronous Data Flow (SDF) [11], the system's behaviour is therefore partitioned into actors we denote as *tasks*. Interaction between tasks is exposed via communication channels we call *Shared Objects*. At design time, tasks are *bound* to Shared Objects through the use of matching *ports*, which

provide the Shared Object interface to the tasks. Thus, the application model provides a structural representation of communicating tasks.

Definition 2 (Application Model)**.** The *application model* $\mathbb{A}_i$ is defined as a tuple $(\mathcal{T}_i, \mathcal{C}_i, \mathcal{E}_i)$ consisting of

- a set of tasks $\tau_i \in \mathcal{T}$,
- a set of Shared Objects $c \in \mathcal{C}$, and
- a relation $\mathcal{E} : \mathcal{T} \rightarrow \mathcal{T}$ denoting the (directed) acyclic communication edges between tasks.

Tasks model sequentially executed functional behaviour. Similar to SDF actors, they depend on the result of other tasks and can only execute when the necessary input data is available. After computation, a task communicates the result through its output channels to other tasks in the system. The following definition serves as a programming model for tasks.

Definition 3 (Task)**.** Let $\mathcal{I}$ be the set of all Shared Object interfaces in the application model $\mathbb{A}$. A *task* $\tau_i \in \mathcal{T}$ is characterized by a tuple (C_i, D_i, T_i, π_i), consisting of a platform-dependent worst-case computation time C_i, a relative deadline D_i, a period T_i, and a set of ports $\pi_k \subseteq \mathcal{I}$, representing the connection to the associated Shared Object interfaces.

In the task model execution behaviour, all input channels are read first, then a local computation is performed, and finally the result is written to the output channels.

Communication between tasks in the system is explicitly modelled using Shared Objects. To further separate computation from communication, tasks contain ports which are bound to Shared Object interfaces and provide access to the interface implementation without structurally depending on the Shared Object. Thus, Shared Objects can be replaced by re-binding the task ports without modifying their internal behaviour. Ports are structurally bound to the Shared Object's interface at design time. The structural aspects of Shared Objects are defined as follows. Instead of a specific implementation, the following Shared Object definition provides a programming model for different implementations. In this paper, we derive a Shared Object *instance* from the definition which is used as a channel in the application model [5].

Definition 4 (Shared Object)**.** A *Shared Object* c_i is a tuple $(\Sigma_i, M_i, \mathcal{I}_i)$, containing

- the *internal state* Σ_i,
- a set of *methods* or *services* $M_i \subseteq \Sigma_i \times \Sigma_i$, representing operations on the object's internal state,
- a set of *interfaces* $\mathcal{I}_i \subseteq \mathcal{P}(M_i)$ the object provides, and

Each interface can be accessed by at most one port by binding it to the Shared Object. The port type has to match the Shared Object interface to which it is bound.

Our Shared Object instance is configured as a first-in first-out (FIFO) channel with two methods $put, get \in M_f$ and a FIFO size of $N = 1$, and its current token size representing the Shared Object state. Each of these methods inserts or removes one token at a time. The Shared Object provides two interfaces put_{if}, get_{if} which tasks can bind their ports to. Requests to get_{if} are blocked when they cannot be handled due to an insufficient amount of tokens in the FIFO. Similarly, calls to put_{if} block if the FIFO contains a token. In both cases, the corresponding method needs to be called to unblock access to the FIFO.

4.1 Mapping and Execution Behaviour

Tasks and Shared Objects are mapped to the platform model for execution according to the mapping relation defined below. Tasks can be mapped to processing elements and Shared Objects to shared memory modules. When multiple tasks are mapped to the same processing resource, a scheduling policy σ partitions the resource for all mapped tasks. Communication channels described by Shared Objects in the application model are mapped to shared memory regions of the platform. Ports are used by tasks to access the Shared Object interface implementation, they act as communication drivers and are executed in the task context on the corresponding processing element.

Definition 5 (Application Mapping)**.** The mapping $M = (M_t, M_c)$ of the application model to the platform components is defined by the functions $M_t : \mathcal{T} \rightarrow \mathbb{P}$ and $M_c : \mathcal{C} \rightarrow \mathbb{M}$ which map the tasks to processing elements and the communication channels to memory modules.

A task may be blocked by accessing the channel due to insufficient tokens for reading or not enough space for writing its tokens. Upon invocation, each task consumes and produces exactly one token on each of its channels. The final task execution order on the processing element is determined by the availability of the input data from all task predecessors.

4.2 Implementation

The application model is implemented on top of POSIX. To implement the periodic task execution behaviour, we use POSIX threads and timers. The task computation takes place in its own thread of control which is periodically executed on each specified period. POSIX additionally provides thread barriers to synchronise the initial execution of the task model. Listing 2 shows the task `execute` function that implements the read-compute-write cycle within a POSIX thread. Before executing the period, it waits on a `pthread_barrier_t` object inside of the `wait_for_init()` method for all POSIX threads to be initialised.

The task then iterates over all incoming ports mapped to the Shared Object interfaces to read the incoming tokens. This operation may possibly block the task due to insufficient tokens available in the FIFOs. When the task has read all incoming tokens, it performs its computation, guarded by the scoped instrumentation. After writing the result to the outgoing ports, the task waits via `wait_for_next_period()` which blocks via POSIX timers until the next period starts. To map tasks to a specific processing element, POSIX offers setting the thread's core affinity, causing the scheduling policy to restrict the execution to a subset of processing elements. The underlying POSIX implementation provides two real-time scheduling policies `SCHED_FIFO` and `SCHED_RR`, representing FIFO- (static) and round robin-based (dynamic) scheduling. In our mapping function, we do not use dynamic task migration and thus restrict each task execution statically to one specific processing element. We also configure the task scheduling policy to `SCHED_RR` to utilise dynamic scheduling.

Listing 3 shows an implementation of the FIFO Shared Objects on top of POSIX primitives. We use POSIX semaphores as synchronisation objects to emulate the FIFO behaviour of the Shared Object interface: access to the `get` (`put`) interface is guarded by a semaphore to block the calling task in the case of an empty (full) FIFO. To map Shared

```
void task::execute() {
  wait_for_init();

  while (true) {
    // read all input tokens
    for (port<fifo_get_if>&& in : ins_)
      /* initial_state = */ in->get();

    SCOPED_CHECK() {
      compute();
    }

    // produce output tokens
    for (port<fifo_put_if>&& out : outs_)
      out->put(/* computed_state */);

    wait_for_next_period();
  }
}
```

Listing 2: Task setup and period execution behaviour.

Objects to specific memory regions of the platform, they can be allocated on a dedicated memory region provided by the operating system through the POSIX `mmap` system call.

4.3 Monitor Construction

In this section, we demonstrate the applicability of the proposed monitoring infrastructure. First, we use properties of the presented application model and its mapping to construct a state transition system that checks whether the logical execution order as well as the task mapping on the platform match the specification. We use the task dependency order $\mathcal{E}$ to construct a transition system that accepts all correct logical execution orders of the task graph. The platform mapping is checked by verifying that the task is executing on the processor specified by M_t. An overview of the application model properties and the monitor construction is given in Figure 2.

The monitor's internal state consists of a list of predecessors for each task, which is derived from the application model at design time. It then checks if a task is eligible to run by consulting the predecessor list and checking if all prior tasks have already been executed. Finally, the monitor updates its internal state after a task has been executed. The following definition describes the internal state of the monitor and the process of determining the predecessor list based on the application model.

Definition 6 (Monitor)**.** Let $(\mathcal{T}_i, \mathcal{C}_i, \mathcal{E}_i)$ be the application model $\mathbb{A}_i$ and $M_t \in M$ the mapping of tasks to processing elements. A *monitor* is a tuple $(\varepsilon, pre, \gamma, \delta)$, consisting of

- an execution count $\varepsilon : \mathcal{T}_i \to \mathbb{N}$ for each task, defining the internal monitor state,
- a predecessor set for each task $pre : \mathcal{T}_i \to 2^{\mathcal{T}_i}$, defined as $pre(t) = \{t_1, \ldots, t_n\}$, which can be constructed from the task graph $\mathcal{E}_i$,
- a function $\gamma\colon \mathcal{T}_i \times \mathbb{P} \to \{0, 1\}$ to monitor task run-time properties, and
- a transition function $\delta\colon \mathcal{T}_i \times \varepsilon \to \varepsilon$ for performing a monitor state transition by incrementing the execution count of the given task.

The monitor's initial state is constructed by defining the immediate task predecessor set *pre*. Given an application model $\mathbb{A}_i = (\mathcal{T}_i, \mathcal{C}_i, \mathcal{E}_i)$, the predecessor set $P(\tau_i)$ for each task is defined as the set of immediate task predecessors:

$$\forall\, \tau_i \in \mathcal{T} : pre(\tau_i) = \{\tau_k \mid (\tau_k, \tau_i) \in \mathcal{E}_i\}$$

The execution count $\varepsilon(\tau_i)$ is set to 0 for all tasks $\tau_i \in \mathcal{T}$.

To perform the application-level execution order and platform mapping checks, the monitor provides a function that determines whether the calling task is eligible to run, given the current task dependency graph state and the executing processing element. A second function is used to update the monitor's internal state, reflecting the completed task execution. The monitor check function $\gamma\colon \mathcal{T}_i \times \mathbb{P} \to \{0, 1\}$ is defined as:

$$\begin{aligned}\gamma(\tau_k, P) = (\forall \tau_j \in pre(\tau_k) : \varepsilon(\tau_j) - \varepsilon(\tau_k) = 1)\\ \wedge\, (\tau_k, P) \in M_t\end{aligned}$$

After a task has finished its computation, a state transition $(\tau_k, \varepsilon, \varepsilon') \in \delta$ is executed, where $\varepsilon'(\tau_k) = \varepsilon(\tau_k) + 1$, which updates the monitor's internal state by incrementing the execution count $\varepsilon(\tau_k)$ of the given task τ_k. This state change reflects the completed execution of the task τ_k.

To summarise, prior to a task computation, a check is performed to validate that its dependencies have been provided by all predecessors by checking that the logically preceding tasks have been executed.

After the task execution, the internal monitor state needs to be updated. The functions γ and δ are therefore invoked before and after a task computation has been performed. The calls to these functions are implemented using the generic

```
struct shared_object
: public fifo_get_if, fifo_put_if
{
  shared_object()
  : size_{1}, tokens_.resize( size_ ) {
    sem_init( &num_free_, 0, size_ );
    sem_init( &count_, 0, 0 );
  }

private:

  // implement get and put interfaces
  virtual int get() {
    sem_wait( &count_ );
    int tok = tokens_.front();
    tokens_.pop_front();
    sem_post( &num_free_ );
    return tok;
  }

  virtual void put(int tok) {
    sem_wait( &num_free_ );
    tokens_.push_back( tok );
    sem_post( &count_ );
  }

  sem_t num_free_;
  sem_t count_;
  int size;
  std::vector< int > tokens_;
};
```

Listing 3: Shared Object FIFO implementation with two semaphores guarding the access to the interface provided to the tasks.

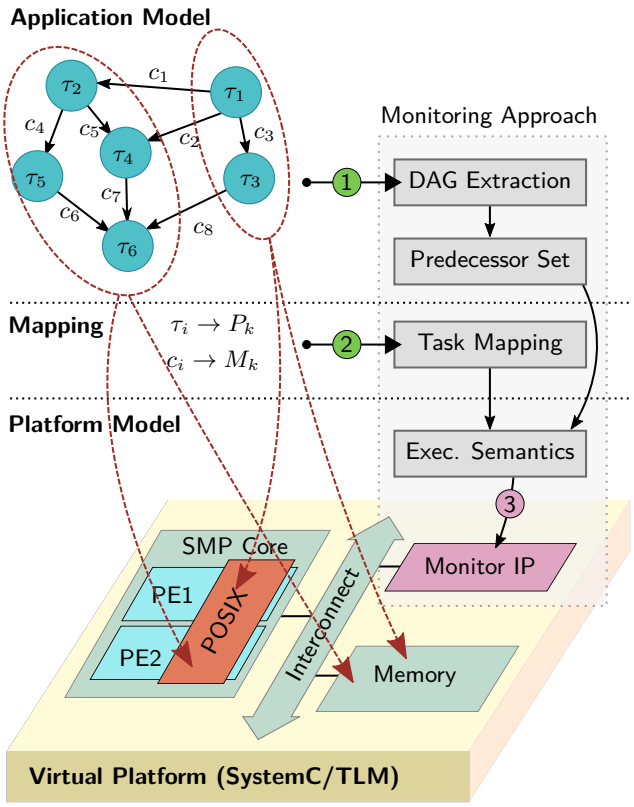

Figure 2: Overview of the monitoring generation approach and its instantiation in a virtual platform.

`on_start` and `on_end` methods of the proposed monitoring framework. Thus, they are triggered when entering and exiting the `SCOPED_CHECK` macro.

5. MONITOR APPLICATIONS AND DISCUSSION

To test the correct behaviour of the monitor for different task graphs, we have built a setup that executes and monitors synthetic task graphs generated by the Task Graphs for Free (TGFF) [4] library. A configuration file is emitted which contains the task graph description and a random mapping to the processing elements and memory modules. The file is read during system configuration in order to set up the monitor state, as described in Definition 6. The POSIX task implementation is configured in a similar manner with tasks as well as Shared Objects being instantiated, connected, and mapped according to the specification. Typical errors which were manually injected after the setup, mimicking a misconfiguration of a manual task graph instantiation or its mapping were successfully detected by the run-time monitoring mechanism. This section reviews and discusses the implications of the proposed model as well as possible extensions of the monitoring framework.

The spatial complexity of the monitor can be directly derived from its state and depends on the input application model. For each task, the monitor keeps an execution count ε and the processing element identifier. Further, a predecessor set is constructed for each task. In a worst-case scenario where a task model is given with N tasks, the first task contains at maximum $N-1$ successors. The second task may point to to $N-2$ successors and so on. The sum of all edges is thus $(N-1)+(N-2)+\ldots+(N-N)=\frac{(N-1)N}{2}$ which denotes the maximum size of all predecessor sets. With N tasks in the application model, the complexity in terms of space is thus upper-bounded by $O(N^2)$.

In our SystemC model, the execution of the check function γ and the state transition δ also depends on the number of tasks defined in the application model. It has a time complexity of $O(\log N)$ due to the execution count mapping lookup ε being implemented as a binary search tree. If implemented in hardware, the lookup can be unrolled at design time.

We have demonstrated the approach on a dataflow oriented application model with a fixed rate of one token per channel. Although this is a very limited model, it allows us to show the possibility to derive a monitor description and implement it within our framework. With different token rates on the channels, the monitor complexity will naturally increase due to the inclusion of the channel token rates in the state space. However, the complexity such a monitor state implementation is orthogonal to the framework capabilities.

We chose to model a static task mapping to processing resources, although RT-Linux and POSIX in general support dynamic task migration. In most scenarios, pinning tasks to processing elements can be useful, e.g. to mitigate bus contention by exploiting cache locality which in turn may help in reaching certain performance goals. To support task migration, the processing element identifier needs to be replaced by a bit mask that allows the underlying POSIX implementation to choose between a set of processing elements for scheduling. The monitor can also be constructed to keep track of the defined processing element set.

The monitor implementation represents an abstract view on the application model. It currently only validates a correct task execution sequence based on the dependency graph without performing a check of the functional behaviour or other task properties. However, it is possible to extend the monitor to check other extra-functional, task-level properties. One aspect is to trace the execution time of tasks. In case of a static task scheduling policy, this can be directly applied with the scoped instrumentation and a hardware timer. To support preemptive task scheduling, an additional instrumentation point has to be added to the system scheduler such that the monitor is able to track task context switches. With this approach, task WCETs as well as their response times can be validated in a multi-tasking context. Due to the lack of a detailed timing model on the virtual platform, this aspect has not been included in the presented approach. If the approach is implemented on a real platform, with the capabilities for measuring power consumption, the approach even allows for checking power usage constraints annotated to the tasks.

Finally, with a backward communication path which can be implemented as an interrupt, the underlying operating system is able to trigger a fail-safe state change and can initiate counter measures, such as dynamic power management or the switch to a degraded system mode. In particular, this is useful in mixed-criticality scenarios [7], where execution time budget overruns are acceptable for non-critical parts of the system.

6. CONCLUSION

We have presented a monitoring framework implemented on a COTS MPSoC multi-tasking platform running RT-Linux. The monitor implementation is based on a virtual platform representing the Xilinx Zynq™-7000 series MPSoC and defines a scoped instrumentation for software tasks. To demonstrate the monitoring framework usage, we have defined an application model containing tasks, Shared Objects, and a platform mapping description. The application model was implemented on top of a POSIX operating system layer. In a second step, a monitor implementation based on the application model was derived statically at design time. We were then successfully able to validate the execution order of the tasks derived from its dependency graph as well as their processor mapping at run-time.

The framework presented in this paper enables synchronising an abstract, redundant implementation of the specified application model that is executed along with the implemented application model on top of POSIX primitives. In the future, we would like to extend this approach towards a target implementation on a physical platform, where we are able to validate extra-functional properties such as task-level temporal behaviour or power consumption. Additionally, we want to dynamically react on the execution anomalies caused by faults to further explore the usage of COTS MPSoC platforms in safety-critical systems.

Acknowledgements

This work has been supported by the *EMC2* collaborative project (ARTEMIS, grant agreement 01IS14002R), funded by the German BMBF.

References

[1] O. Arnold and G. Fettweis. Adaptive runtime management of heterogenous MPSoCs: Analysis, acceleration and silicon prototype. In *International Symposium on System-on-Chip (SoC'2014)*, pages 1–4. IEEE, Oct. 2014.

[2] O. Arnold, B. Noethen, and G. Fettweis. Instruction Set Architecture Extensions for a Dynamic Task Scheduling Unit. In *2012 IEEE Computer Society Annual Symposium on VLSI*, pages 249–254, Amherst, MA, USA, Aug. 2012.

[3] J. Castrillon, D. Zhang, T. Kempf, B. Vanthournout, R. Leupers, and G. Ascheid. Task management in MPSoCs: an ASIP approach. In *Proceedings of the 2009 International Conference on Computer-Aided Design (ICCAD'09)*, page 587. ACM Press, 2009.

[4] R. Dick, D. Rhodes, and W. Wolf. TGFF: task graphs for free. In *Proceedings of the Sixth International Workshop on Hardware/Software Codesign (CODES/CASHE'98)*, pages 97–101, Mar. 1998.

[5] K. Grüttner, H. Kleen, F. Oppenheimer, A. Rettberg, and W. Nebel. Towards a Synthesis Semantics for SystemC Channels. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS'10, pages 163–172, Scottsdale, Arizona, USA, 2010. ACM.

[6] R. Hankins, G. Chinya, J. Collins, P. Wang, R. Rakvic, Hong Wang, and J. Shen. Multiple Instruction Stream Processor. In *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA'06)*, pages 114–127. IEEE, 2006.

[7] P. Ittershagen, K. Grüttner, and W. Nebel. Mixed-Criticality System Modelling with Dynamic Execution Mode Switching. In *Proceedings of the Forum on Specification and Design Languages (FDL'2015)*, Barcelona, Spain, 2015.

[8] P. Kohout, B. Ganesh, and B. Jacob. Hardware support for real-time operating systems. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, page 45. ACM Press, 2003.

[9] A. Kritikakou, C. Rochange, M. Faugère, C. Pagetti, M. Roy, S. Girbal, and D. G. Pérez. Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems (RTNS'14)*, pages 139–148. ACM Press, 2014.

[10] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th annual international symposium on Computer architecture*, volume 34, pages 162–173, 2007.

[11] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[12] D. Lo, M. Ismail, Tao Chen, and G. E. Suh. Slack-aware opportunistic monitoring for real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS'2014)*, pages 203–214. IEEE, Apr. 2014.

[13] C. Meenderinck and B. Juurlink. A Case for Hardware Task Management Support for the StarSS Programming Model. In *13th Euromicro Conference on Digital System Design (DSD'2010)*, pages 347–354. IEEE, Sept. 2010.

[14] C. Meenderinck and B. Juurlink. Nexus: Hardware Support for Task-Based Programming. In *14th Euromicro Conference on Digital System Design (DSD'2011)*, pages 442–445. IEEE, Aug. 2011.

[15] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai. Hardware implementation of a real-time operating system. In *Proceedings of TRON'95*, pages 34–42. IEEE Comput. Soc. Press, 1995.

[16] M. Paolieri and R. Mariani. Towards functional-safe timing-dependable real-time architectures. In *2011 IEEE 17th International On-Line Testing Symposium*, pages 31–36, Athens, Greece, July 2011.

[17] SYSGO. *PikeOS Hypervisor Product Datasheet*, 2015.

[18] Windriver. *VxWorks 6.2: Kernel Programmer's Guide*, 2005.

[19] J. Wolf and T. Ungerer. An Optimized Timing and Control Flow Checker for Hard Real-Time Systems. In *Proceedings of 2013 26th International Conference on Architecture of Computing Systems (ARCS'2013)*, Feb. 2013.