Hierarchical Scheduling over Off- and On-chip Deterministic Networks*

Ramon Serna Oliver
TTTech Computertechnik AG
Schoenbrunnerstr. 7, Vienna, Austria
rse@tttech.com

Silviu S. Craciunas
TTTech Computertechnik AG
Schoenbrunnerstr. 7, Vienna, Austria
scr@tttech.com

ABSTRACT

In this paper we present a compositional model for distributed virtualized systems communicating over on-chip/off-chip deterministic networks implementing an end-to-end or partial time-triggered paradigm. We derive system-level constraints for combined task-, virtualization- and network-level static scheduling enabling the end-to-end composition of schedules for systems featuring table-driven (guest) operating systems. In the absence of a time-triggered run-time system, we analyze the composition problem with the aid of hierarchical scheduling methods for abstract resources. Moreover, we identify and discuss possible trade-offs and optimization opportunities that arise when scheduling across multiple (virtualized) software layers in tandem with the deterministic network.

1. INTRODUCTION

The proliferation of embedded hypervisors and system-on-chip technologies open a range of opportunities for the development of complex systems for real-time deployments in various embedded domains, like mixed-criticality systems and the Internet of Things (IoT). Virtualized platforms spanning across multiple levels of on-chip and off-chip components are able to host dynamically relocatable distributed applications with different criticality requirements in isolation of each other. An example of such architecture is the DREAMS harmonized platform [5] currently being exploited within several demonstrators in the EU/FP7 project DREAMS[1].

This paper relies on previous work on end-to-end scheduling of distributed time-triggered systems [3] and develops a descriptive discussion concerning the challenges and implications of extending the concepts to an abstract distributed and virtualized on-chip/off-chip architecture (Section 2), a generalization of the DREAMS harmonized platform, implementing either a full or partial composable end-to-end time-triggered paradigm. We explore in Section 3 the isolation problem as a combination of proper resource allocation and scheduling composability via hierarchical system models. We further present a number of performance trade-offs leading to feasible solutions without excessively sacrificing the schedulability solution space. We round up the paper with a discussion (Section 4) of the trade-offs and optimization opportunities arising from our model followed by a brief survey of related research in Section 5 and concluding remarks in Section 6.

2. SYSTEM AND NETWORK MODEL

We introduce an abstraction of a distributed hierarchical time-triggered system, similar to [13] composed of networked multi-core nodes as depicted in Figure 1. Nodes are essentially systems-on-chip composed of multiple tiles, each consisting of a number of interconnected CPUs. On the tiles, a hypervisor hosts multiple guest operating systems (guest OS) following a table-driven schedule in a fully virtualized platform. Application tasks execute on the guest OS and communicate with other tasks within or outside their tile by exchanging time-triggered messages (tt-message). We differentiate two levels of the network, namely an on-chip network connecting the tiles within one node, and the off-chip network connecting the multiple nodes. Both on-chip and off-chip networks implement a scheduled time-triggered communication paradigm (e.g. [20]) to transfer messages of tasks within the same node (on-chip) or, respectively, across nodes (off-chip). We differentiate two alternative scenarios (see Section 3) with respect to the execution of tasks in the guest OS. On one hand, we consider the time-triggered case (os_{tt}) in which the guest OS and virtualization layers follow a table-driven schedule and, hence, an end-to-end time-triggered paradigm orchestrates the execution of tasks, virtual machines, as well as communicating networks. On the other hand, we elaborate an alternative case (os_{abs}) in which the execution model of the guest OSs is abstracted into a common schedulable resource on the level of hypervisor. In this case, the time-triggered paradigm applies only to the hypervisor and the networks. In our model we assume that both the guest OS and virtualization layers have mechanisms allowing a global time synchronization (e.g. IEEE 802.1AS, IEEE 1588, or [21]).

*The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007- 2013) under grant agreement n° 610640 (DREAMS).

[1]A detailed project description is available at http://www.dreams-project.eu/

.

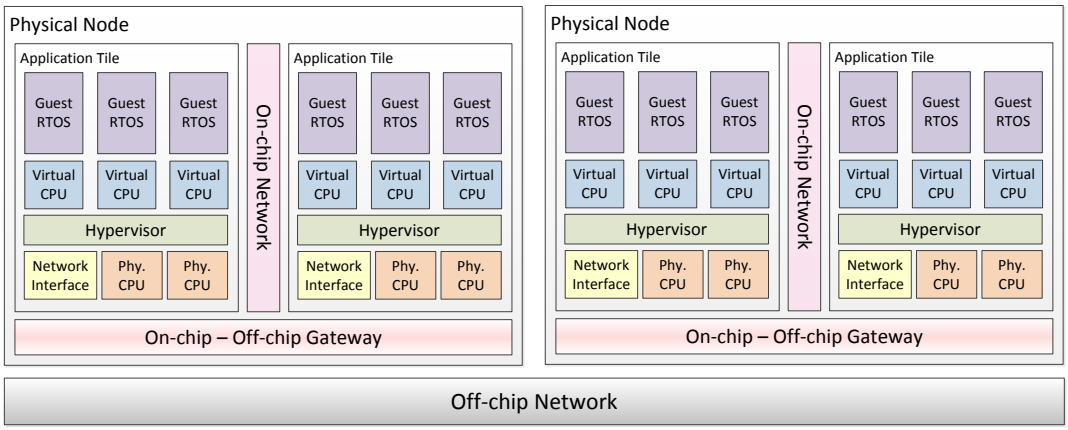

Figure 1: Example of a virtualized on-chip/off-chip architecture.

2.1 Execution Modes

An advantage of virtualization is that, in the event of failures, tasks and guest OSs can be migrated to different tiles or nodes provided that sufficient resources are still available. While a dynamic reconfiguration procedure offers the greatest flexibility towards unexpected failures, it also requires online mechanisms to resolve and orchestrate the migration steps. These mechanisms, however, decrease significantly the deterministic behavior established by the time-triggered paradigm, since it is typically not feasible to maintain system guarantees across unforeseen run-time dynamics. A more deterministic procedure results from an offline planning of reconfiguration modes (e.g. [6]). The advantages arise from the possibility of ensuring the required end-to-end system guarantees for each configuration mode, while clearly, the main disadvantage is the limitation in the number of tolerated failures to those which have been considered offline. In this paper we consider the second case. We claim that predictability and determinism in the domain of mix-criticality systems is of higher value than the adaptability to unforeseen events. Moreover, we motivate our choice in the existence of analysis methods, like [5], compatible with the offline definition of alternative configuration modes.

An execution mode is, therefore, the allocation and configuration for the set of tasks to the respective OS, as well as an assignment for each OS to an explicit tile of a node. Note that the end-to-end communication requirements (i.e. message exchange between application tasks) remains the same regardless of the particular resource where tasks are hosted. Dependent configurations like, e.g. on-chip or on-chip network schedules, may need to change and accommodate to the new arrangement of tasks.

We consider system level timeliness guarantees as the end-to-end application latency of communicating pairs of tasks. To this respect, we follow a communication model, similar to [3] in which tasks either consume or produce at most one message which is respectively received before its release time or transmitted at the end of its execution. We aim at generating the schedule for each time-triggered layer (i.e. off-chip/on-chip network, hypervisor, guest OS) such that end-to-end latency is guaranteed for each execution mode considered. Without loss of generality, we consider timeliness guarantees to be specified as earliest start time, latest execution time (e.g. deadline), and maximum end-to-end latency, where end-to-end refers to the difference between the scheduled release time of the producer task and the latest scheduled execution time of the consumer.

Given the set of execution modes, we elaborate again two alternative scenarios with respect to end-to-end guarantees. On one hand, we consider the loose case in which generation of schedules is decoupled, hence satisfies end-to-end constraints on each mode without imposing any constraints across alternative modes. On the other hand, we consider the strict case in which the same end-to-end timeliness is coupled across the complete set of modes, meaning that the scheduled start time, deadline, and end-to-end latency is strictly the same for each mode. We motivate this scenario in that a mode change can take place without any noticeable change at the application layers with respect to end-to-end timeliness. Note, however, that in order to fully guarantee strict end-to-end timeliness of the application layer we impose the guest OS to be table-driven (case os_{tt}). A more detailed discussion on this topic is presented in section 3.1.1.

3. HIERARCHICAL SCHEDULING

Typically the scheduling problem for distributed time-triggered networked systems was decoupled into different layers and handled separately or sequentially. Some approaches schedule the network first and derive out of the resulting schedule constraints for the software layer [4] whereas other methods consider the software layer first and subsequently constrain the schedule of the network layer [9]. Both methods suffer from the problem that sequential scheduling requires a feedback loop in the case of one of the layers not being schedulable as a result of the constraints of the other layer. The root cause for infeasiblity may be very difficult to determine due to the so-called *domino effect* (cf. [2, p. 37]). Some recent papers have considered the combined approach in order to cover the whole solution space [3] but are restricted to a simple model where end-nodes are uniprocessor systems lacking the virtualization layer and off-chip and on-chip architecture discussed in our model.

Our architecture raises two main research questions. First, the virtualization layer together with the multi-core capabilities introduces the problem of creating combined schedules over the hierarchical design. Secondly, the re-

quirements of having multiple modes, where applications can migrate from one node to another, imposes additional constraints on the off-chip and on-chip communication latencies.

3.1 The allocation and scheduling problem

The model presented in this paper requires several levels of scheduling in addition to the decision on which CPU core to schedule which task/virtual machine. As in [3], we assume the sender and receiver tasks communicating over the off-chip/on-chip network have the same rate (period). Multi-rate communication can be resolved, as discussed in e.g. [7], by means of extended logic in the sender/receiver tasks without affecting our approach. Other complex dependencies, like e.g. event chain constraints, can be modeled by a composition of precedence constraints (cf. [3]).

The most important aspect of the scheduling problem is whether the guest OS features table-drive static scheduling (os_{tt}) or some other priority-based mechanism (os_{abs}). In the first case, we generate static schedules for the guest OS layer (Section 3.1.1) whereas in the second case individual guest OS schedulers featuring either dynamic or static priorities cannot be readily integrated into our approach (Section 3.1.2).

3.1.1 Table-driven static scheduling

The model presented in [3], in which preemptable tasks running on uni-processor systems communicate through a deterministic network, offers the basis for our analysis. Our method consists of breaking up the preemptive time-triggered tasks into non-preemptable *pieces* based on the macrotick of the underlying OS and formulate first-order logical constraints defining the point in time where both the task pieces and frames on the network are scheduled fulfilling the end-to-end guarantees. The constraints are then solved by either a Satisfiability Modulo Theories (SMT) or Mixed Integer Programming (MIP) solver, depending on whether optimization objectives are considered or not.

We define the network model as a graph $G(\mathcal{R}, \mathcal{L})$. The set $\mathcal{R}$ defines the set of all resources in the network, i.e., end-nodes and off-chip components (switches). The edges ($\mathcal{L} \subseteq \mathcal{R} \times \mathcal{R}$) are the directional communication links that connect the resources in the given topology. We have two types of schedulable entities, namely tasks and network frames. We model all schedulable entities through the concept of frames similar to [3]. The communication through network links is composed of frames of a certain length that have to be transmitted from one sender to one or multiple receivers. Tasks that are scheduled on CPUs in the end-systems can be split into pieces, where each piece can be viewed as a frame with a certain size that has to be scheduled. Hence, for resources such as off-chip network switches, a frame is the instance of a tt-message scheduled on an outgoing port (physical link). For CPUs, a task is a set of sequential frames scheduled on a CPU (core). We can model both preemptive and non-preemptive tasks by generating either one frame per task (with the length of the frame equal to the WCET of the task) or as many frames as are necessary based on the macrotick and speed of the respective CPU. From this point on we will only concentrate on scheduling frames and refer the reader to [3] for the complete transformation formalism and scheduling constraints for uni-processor end-nodes.

If the guest OSs feature table driven static schedules we can abstract the hierarchical scheduling problem to scheduling n tasks (i.e. tasks from all guest OSs running on the same application tile) on m processors. In this regard, we need to extend the constraints presented in [3] to create schedules for multiple processors, i.e., both the time at which tasks are scheduled as well as the physical CPU where tasks run needs to be determined by the solver engine. In [3] the only relevant scheduling resource was time. For our current model we need to extend this resource by a second dimension that represents the physical CPU where tasks run.

We formulate an additional constraint to the ones presented in [3] that captures the two-dimensional scheduling resource for tasks, allowing pieces of a task to either overlap in time but execute in different CPUs, or run on the same CPU but not overlap in the time domain. A two dimensional resource $R \in \mathcal{R}$ has two scheduling components, namely the time dimension and the CPU dimension. In case of switches, the second dimension has a capacity of 1 while in the case of end-nodes the capacity of the resource $R.C$ defines the number of available CPUs. A task τ_i running on an application tile generates a set of frames $\mathcal{F}_i = \{f_{i,j} \mid j = 1, \ldots, n\}$ where the frames represent the non-preemptable pieces of the task [3]. Usually, the pieces are defined based on the macrotick of the system but they can also be defined depending on optimization objectives (discussed below).

A frame $f_{i,j} \in \mathcal{F}_i$ when scheduled on a resource R generates a frame instance denoted by $f^R_{i,j}$. A frame instance is defined by the tuple $\langle f^R_{i,j}.\phi, f^R_{i,j}.\pi, f^R_{i,j}.L, f^R_{i,j}.T \rangle$, where $f^R_{i,j}.\phi$ is the offset in the time dimension, $f^R_{i,j}.\pi$ is the offset in the CPU dimension, $f^R_{i,j}.L$ is the frame length (piece size), and $f^R_{i,j}.T$ is the period of the frame in the time dimension.

The set of frames generated by all n tasks running on an application tile $\mathcal{F}^R = \{f^R_{i,j} \mid i = 1, \ldots, n; j = 1, \ldots, m_i\}$ scheduled on a resource R is subject to the following constraints: $\forall f^R_{i,j} \in \mathcal{F}^R : (f^R_{i,j}.\phi \geq 0)$ and $\forall f^R_{i,j} \in \mathcal{F}^R : (f^R_{i,j}.\pi \geq 0) \wedge (f^R_{i,j}.\pi \leq R.C)$, where $R.C$ is the number of available CPUs on resource R. Additionally, we have to ensure that, regardless of the CPU the pieces of an individual task run on, they are in the correct order, i.e. $\forall f^R_{i,j} \in \mathcal{F}^R_i, j = 1, \ldots, m_i - 1 : f^R_{i,j}.\phi + f^R_{i,j}.L \leq f^R_{i,j+1}.\phi$.

We now refine the link constraint presented in [3] to capture a multi-core/multi-processor architecture. The resulting constraint for any resource $R \in \mathcal{R}$ is

$$
\begin{aligned}
&\forall f^R_{i,j} \in \mathcal{F}^R_i, f^R_{k,l} \in \mathcal{F}^R_k, i \neq k, \\
&\forall \alpha \in \left[0, \frac{HP^{k,l}_{i,j}}{f^R_{i,j}.T} - 1\right], \forall \beta \in \left[0, \frac{HP^{k,l}_{i,j}}{f^R_{k,l}.T} - 1\right] : \\
&\Big((f^R_{i,j}.\phi + \alpha \times f^R_{i,j}.T \geq f^R_{k,l}.\phi + \beta \times f^R_{k,l}.T + f^R_{k,l}.L) \vee \\
&(f^R_{k,l}.\phi + \beta \times f^R_{k,l}.T \geq f^R_{i,j}.\phi + \alpha \times f^R_{i,j}.T + f^R_{i,j}.L) \Big) \vee \\
&\Big(f^R_{i,j}.\pi \neq f^R_{k,l}.\pi \Big),
\end{aligned}
$$

where $HP^{k,l}_{i,j} \stackrel{def}{=} lcm(f^R_{i,j}.T, f^R_{k,l}.T)$.

Other constraints, such as precedence and latency requirements, can be defined complementary. The end-to-end latency of the communication between a sender and receiver task can be thus defined by the system requirements and imposed as a constraint on the generated schedule (cf. [3]).

The length of a frame generated by a task is usually based on the macrotick of the respective system. In this case

the length is equal to 1 since the generated schedule is defined on a time-line with macrotick granularity. However, it also possible to model non-preemptable tasks by setting the length of the frame to the WCET of the task. Moreover, we can also define a design problem allowing a arbitrary split of tasks into frames between the two corner-cases defined above. Here, there is a fundamental trade-off between solution space and runtime of the solver. A larger length of the frames, and hence a smaller number of frames that need to be scheduled, may result in a faster runtime of the solver at the expense of solution space. Furthermore, there is also an optimization problem that can be defined with respect to the switching overhead. For every frame that needs to be scheduled, a context switch overhead may be introduced, hence lowering the available CPU bandwidth for the execution of tasks and increasing the overhead of the underlying guest OS and virtualization layer. The optimization problem consists in finding the optimal split of tasks into frames that minimizes the overall context switch overhead.

Captured in our constraint is the ability of a task to migrate from one CPU to another by allowing each piece of the task to be freely scheduled on any CPU. The overhead involved in migration can be substantial thus raising another optimization/design problem that aims at minimizing the involved overhead (discussed in Section 4).

3.1.2 *Priority-based scheduling*

We now consider the case in which the guest OS running on top of the virtualization layer feature either earliest deadline first (EDF) or rate monotonic (RM) scheduling. In this case, the combined scheduling approach cannot include the guest OS layer, since there is either dynamic priorities or static priorities that define the timing behavior of tasks at runtime and not a precomputed offline schedule. One approach that can be used to handle such systems within our combined scheduling method is the compositional hierarchical scheduling framework [19] in which real-time timing requirements of individual periodic tasks scheduled in one guest OS can be combined (abstracted) into one single periodic resource requirement on the level of the virtualization layer. In this approach, a scheduling component model, defined as $C(W, R, A)$, where W is the task set, R is the abstracted resource model, and A is the scheduling algorithm, is guaranteed to be schedulable in isolation to the other scheduling models [18]. Given the set of tasks $W = \{T_i(e_i, p_i)\}$, where e_i is the WCET and p_i the period of task T_i, a method is given in [18, 19] to extract a periodic resource $\Gamma(\Theta, \Pi)$ for both EDF and RM which ensures that, if the virtualization layer supplies Θ time units every Π time units for workload W, each task T_i will execute its workload of e_i within the given period p_i. From this abstraction, a periodic virtual real-time task with execution time Θ and period Π can be defined that is then scheduled offline using our combined approach on the virtualization layer.

The resulting scheduling problem is similar to the one described above (Section 3.1.1), namely computing a schedule for n virtual tasks that result from the guest OS abstractions on m processors with one very important difference. Above, the end-to-end latency was an input constraint from the user on individual sender-receiver pairs. Here, we cannot control the deadlines and release times of individual sender and receiver tasks running in the guest OSs, hence the end-to-end latency cannot be specified as a user constraint but is rather

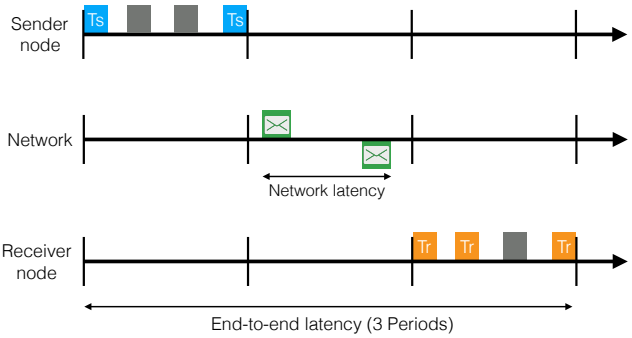

Figure 2: End-to-end latency example.

computed as the worst-case resulting from the period of the communication.

Since we only can guarantee that a sender/receiver task will start and finish execution within its period, but not at which point exactly within the period, the moment in time when a message is available for transmission and reception on the end-system side is at the end of the current and beginning of the next period instances, respectively. We illustrate this issue in Figure 2 via a simplified example in which a sender task $T_s(2, 10)$ running on a guest OS abstracted into a resource $\Gamma_s(4, 10)$ is communicating via the network to a receiver task $T_r(3, 10)$ running on a guest OS abstracted into a resource $\Gamma_r(4, 10)$. All times are given in ms.

Illustrated in the figure is a possible schedule of both sender and receiver tasks, where the grey areas define the other tasks that run within the abstracted resources Γ_s and Γ_r. The uncertainty regarding the moment of execution of sender and receiver tasks within the guest OS results in a worst-case end-to-end latency of 3 periods. For the network, the end-to-end latency between the sending of the message from the sender node and the reception of the message on the receiving node is defined to be one period. In the example the resulting worst-case end-to-end latency is $30ms$.

As discussed in [3] individual tasks may have precedence constraints due to shared resources or system design requirements. As opposed to the table-driven approach within the guest OS, having priority-driven scheduling in a hierarchy with dependencies both within a guest OS and between tasks of different guest OSs, requires more advanced hierarchical abstraction mechanisms. Such mechanism have been studied in [10], where hierarchies of tasks with dependencies are considered within the RTW (Real-Time Workshop) and LET (Logical Execution Time) semantics, exposing a trade-off between end-to-end latency and composability.

3.2 Mode changes

In the context of this paper execution modes are defined as a collection of configuration artifacts for the schedulable time-triggered resources (e.g. hypervisor, guest-OS, or on-chip/off-chip network). Eventually, a re-configuration of resources may be necessary in order to maintain system guarantees upon changes in the distributed deployment. These runtime changes may be triggered as a response to system failures or due to changes in the application demand for given resources. Regardless of the triggering event, we define the action of switching from one system-wide configuration (execution mode) to a new one as a `mode change` [16].

We restrict the runtime adaptability to changing events to a set of pre-computed execution modes. We also assume an offline established relation between each such mode and one or several alternative triggering events. Once an event occurs, a system wide decision to switch to a new mode is made. Note that a discussion concerning the mechanism to detect, decide, and orchestrate mode changes during runtime is not in the scope of this paper.

In any case, we consider the action of switching to a new mode as a critical operation with a non-zero cost in terms of complexity and performance degradation [6], [17], [14]. In general terms, loading a new configuration may require the reconfigured resource to restart (e.g. off-chip switch, guest-OS) or imply a certain loss of application data (e.g. routing tables, application cache). For certain resources, typically those with lower utilization, it is possible to leverage the reconfiguration efforts by defining super-sets of configuration modes. In particular, if the resource is managed according to a table-driven schedule, the construction of the super-set reduces to the union of multiple tables.

For example, consider execution modes m_a and m_b for alternative tasks sets in a given guest-OS, namely $T_a = \{\tau_i^a, \tau_j^a\}$ and $T_b = \{\tau_i^b, \tau_k^b\}$. Instead of reconfiguring the system at run-time based on the respective table-driven schedules tt_a and tt_b we define a merging step of the task-set, prior to the scheduling step, to determine a combined task-set $T_{a\cup b}$, where $T_{a\cup b} = T_a \cup T_b = \{\tau_i, \tau_j, \tau_k\}$. This essentially results in one larger task-set that can be scheduled following the same approach described so far. Therefore, configuring the resulting schedule (i.e. $tt_{a\cup b}$) is sufficient to fulfill both execution modes, m_a and m_b. In this case, the activation of the already scheduled tasks for each mode is leveraged to the application logic as it does not require any additional re-configuration mechanism at run-time.

One such resource is the off-chip network, where several time-triggered communication schedules can be merged into a super-schedule. Switching to a new mode which is part of the current super-set results in zero cost, since the frame transmission events required in the new mode are already present in the current configuration. Only when the union of schedules becomes unfeasible due to conflicting constraints or over-utilization of the resource it becomes necessary to switch to an alternative super-set. It is relevant, therefore, to build the super-sets based on the union of those modes which are prone to correlate in order to minimize the need to reconfigure a new super-set. In other resources, even those that may not follow a time-triggered paradigm (i.e. an abstract OS), a union of modes can still be achieved based on the analysis of resource requirements detailed in Section 3.1.2.

4. TRADE-OFFS AND OPTIMIZATION

We have identified several fundamental trade-offs and optimization problems arising from the presented model which we discuss here in more detail.

From the scheduling perspective, the different layers open up several interesting trade-offs in terms of solution space and runtime of the solver engine, which is important for NP-complete problems, as well as optimization problems regarding several overhead parameters. As presented in Section 3 one important factor influencing the schedulability of the system is the macrotick length. A larger macrotick will reduce runtime of the solver since the search space for frame offsets is defined by the granularity of the time-line at the expense of also reducing the solution space. Moreover, a smaller macrotick, while increasing system responsiveness, will also reduce the available CPU bandwidth for executing tasks since, for some systems, the constant overhead of the timer interrupt (which happens at macrotick granularity) in addition to the context switch overhead will be more relevant if the macrotick is smaller. In Figure 3 we illustrate the dependency between the available CPU bandwidth for task execution and the macrotick size for a custom RTOS developed at TTTech [4] running on a TMS570 platform.

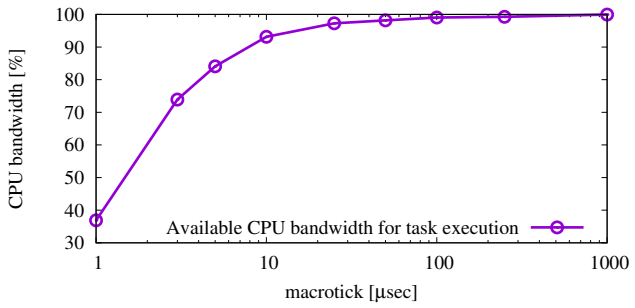

Figure 3: Available CPU bandwidth for task execution in function of the macrotick length on a TMS570 platform. Measurements were obtained from our previous experiments conducted for [4].

The length of the macrotick also influences the task splitting into non-preemptable task pieces (frames). A finer macrotick granularity reduces the wasted CPU bandwidth when the WCET of the task is not a multiple of the macrotick since, in such a case, there is some over-provisioning, i.e, the number of generated frames n_i of task τ_i with computation time $WCET_i$ running on a system with macrotick mT is computed as $n_i = \lceil \frac{WCET_i}{mT} \rceil$.

An arbitrary split of a task into pieces spanning multiple macroticks also exposes a trade-off between solution space and runtime of the solver and offers some space for optimization. Lowering the number of frames that need to be scheduled may result in a faster runtime of the solver and may also result in fewer context switches at the expense of solution space. An interesting multi-objective optimization problem in this context is to find the optimal number of pieces (frames) for each task such that overall context switch overhead and the wasted CPU bandwidth from over-provisioning are minimized. The problem can also be formulated as finding the maximum macrotick size for which the system is still schedulable.

The second dimension that yields optimization opportunities is the multi-core design. The ability of a task to migrate from one CPU to another by allowing each piece of the task to be freely scheduled on any CPU may result in substantial migration cost. Here there is also a fundamental trade-off between overhead cost of migration and schedulability. If a task is not allowed to migrate by design, we can impose a constraint, $\forall f_{i,j}^R \in \mathcal{F}_i^R, j = 1, \ldots, m_i - 1 : f_{i,j}^R.\pi = f_{i,j+1}^R.\pi$ that forbids migration. However, while reducing the migration cost to 0, this also reduces the feasible solution space. As with the context switch overhead, an interesting optimization problem is to let tasks migrate in order to increase schedulability but define an optimization objective that minimizes the resulting migration overhead.

For systems in which the guest OS layer features priority-driven scheduling we also identify an optimization problem. The main issue with extracting the single periodic resource requirement $\Gamma(\Theta, \Pi)$ out of multiple tasks is finding the optimal period Π such that the resource capacity of the virtualization layer is not wasted, i.e., minimize the abstraction overhead [19]. Additionally, the virtual task resulting from the abstracted resource must be schedulable using the macrotick of the underlying virtualization layer. In other words, the optimization problem here is to find x and y, with $\Theta = x \times mT$ and $\Pi = y \times mT$ and where mT is the macrotick of the virtualization layer, such that the abstraction overhead, defined in [19] as $\mathcal{O}_{\mathcal{P}} = \frac{U_{\mathcal{P}} - U_W}{U_W}$, where $U_W = \sum_{T_i} e_i/p_i$ and $U_{\mathcal{P}} = \Theta/\Pi$, is minimized.

5. RELATED WORK

Hierarchical scheduling models have been extensively studied within the scope of real-time scheduling with the most prominent being the work on the hierarchical scheduling framework presented in [18, 19] and extended in [10].

Combined task and network-level scheduling featuring preemptive or non-preemptive tasks running on uniprocessor end-nodes that are connected through a switched deterministic network has been studied in [3] and [22] using SMT solvers and MIP multi-objective optimization, respectively. Approaches dealing with tasks communicating through a bus are [1] and [15] which discusses the combination of fixed-priority task and TTP bus message scheduling. A SAT-based approach is used in [11] to schedule the application and network layers where tasks that communicate though a bus are scheduled using a fixed-priority scheme.

Within the context of virtualized platforms, the work in [12] discusses a scheduling method based on Mixed Integer Linear Programming (MILP) that features both the allocation as well as the scheduling of tasks on CPUs together with scheduling and routing communication on networks-on-a-chip. The work in [8], which addresses a similar problem to ours, presents a schedulability analysis in which the scheduling model on one virtualization layer within a two-level hierarchical system is abstracted into a set of time windows.

6. CONCLUSION

Driven by the proliferation of embedded hypervisors and system-on-chip technologies in real-time systems, we presented methods for the generation of combined time-triggered schedules for virtualized multi-core/multi-processor systems connected through deterministic off-chip/on-chip networks with two alternative types of guest operating systems. Moreover, we discussed several fundamental trade-off and optimization challenges that open up interesting directions for future work.

References

[1] Abdelzaher, T. F., and Shin, K. G. Combined task and message scheduling in distributed real-time systems. *IEEE Trans. Parallel Distrib. Syst. 10*, 11 (1999), 1179–1191.

[2] Buttazzo, G. C. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag, 2004.

[3] Craciunas, S. S., and Serna Oliver, R. Combined task- and network-level scheduling for distributed time-triggered systems. *Real-Time Systems 52*, 2 (2016), 161–200.

[4] Craciunas, S. S., Serna Oliver, R., and Ecker, V. Optimal static scheduling of real-time tasks on distributed time-triggered networked systems. In *Proc. ETFA* (2014), IEEE Computer Society.

[5] Durrieu, G., Fohler, G., Gala, G., Girbal, S., Gracia Pérez, D., Noulard, E., Pagetti, C., and Pérez, S. DREAMS about reconfiguration and adaptation in avionics. In *Proc. ERTS* (2016).

[6] Fohler, G. Changing operational modes in the context of pre run-time scheduling. *IEICE Transactions on Information and Systems Special Issue on Responsive Computer Systems* (November 1993).

[7] Forget, J., Grolleau, E., Pagetti, C., and Richard, P. Dynamic priority scheduling of periodic tasks with extended precedences. In *Proc. ETFA* (2011), IEEE Computer Society.

[8] Guasque, A., Balbastre, P., Brocal, V., and Crespo, A. Schedulability analysis of hierarchical systems with arbitrary scheduling in the global level. In *Proc. CESCIT* (2015).

[9] Hanzalek, Z., Burget, P., and Šucha, P. Profinet IO IRT message scheduling. In *Proc. ECRTS* (2009), IEEE Computer Society.

[10] Matic, S., and Henzinger, T. A. Trading end-to-end latency for composability. In *Proc. RTSS* (2005), IEEE Computer Society.

[11] Metzner, A., Franzle, M., Herde, C., and Stierand, I. Scheduling distributed real-time systems by satisfiability checking. In *Proc. RTCSA* (2005), IEEE Computer Society.

[12] Mursheed, A., Obermaisser, R., Ahmadian, H., and Khalifeh, A. Scheduling and allocation of time-triggered and event-triggered services for multi-core processors with networks-on-a-chip. In *Proc. INDIN* (2015).

[13] Obermaisser, R., et al. Architectural style of DREAMS. *Distributed Real-time Architecture for Mixed Criticality Systems (DREAMS) D1.2.1* (July 2014).

[14] Pedro, P., and Burns, A. Schedulability analysis for mode changes in flexible real-time systems. In *Proc. ECRTS* (1998), IEEE Computer Society.

[15] Pop, P., Eles, P., and Peng, Z. Schedulability-driven communication synthesis for time triggered embedded systems. *Real-Time Syst. 26*, 3 (2004), 297–325.

[16] Real, J., and Crespo, A. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems 26*, 2 (2004), 161–197.

[17] Sha, L., Rajkumar, R., Lehoczky, J., and Ramamritham, K. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems 1* (1988), 243–264.

[18] Shin, I., and Lee, I. Periodic resource model for compositional real-time guarantees. In *Proc. RTSS* (2003), IEEE Computer Society.

[19] Shin, I., and Lee, I. Compositional real-time scheduling framework with periodic model. *ACM Trans. Embed. Comput. Syst. 7*, 3 (2008), 30:1–30:39.

[20] Steiner, W., Bauer, G., Hall, B., and Paulitsch, M. TTEthernet: Time-Triggered Ethernet. In *Time-Triggered Communication*, R. Obermaisser, Ed. CRC Press, Aug 2011.

[21] Steiner, W., and Dutertre, B. Automated formal verification of the TTEthernet synchronization quality. In *NASA Formal Methods*, vol. 6617 of *Lecture Notes in Computer Science*. Springer, 2011.

[22] Zhang, L., Goswami, D., Schneider, R., and Chakraborty, S. Task- and network-level schedule co-synthesis of Ethernet-based time-triggered systems. In *Proc. ASP-DAC* (2014), IEEE Computer Society.