Exploiting Segregation in Bus-Based MPSoCs to Improve Scalability of Model-Checking-Based Performance Analysis for SDFAs

Maher Fakih, Kim Grüttner, Martin Fränzle, Achim Rettberg

Exploiting Segregation in Bus-Based MPSoCs to Improve Scalability of Model-Checking-Based Performance Analysis for SDFAs

Maher Fakih[1], Kim Grüttner[1], Martin Fränzle[2], and Achim Rettberg[2]

[1] OFFIS – Institute for Information Technology, Germany
[2] Carl von Ossietzky Universität, Germany

Abstract. The timing predictability of embedded systems with hard real-time requirements is fundamental for guaranteeing their safe usage. With the emergence of multicore platforms this task becomes even more challenging, because of shared processing, communication and memory resources. Model-checking techniques are capable of verifying the performance properties of applications running on these platforms. Unfortunately, these techniques are not scalable when analyzing systems with large number of tasks and processing units. In this paper, a model-checking based approach that allows to guarantee timing bounds of multiple Synchronous Data Flow Applications (SDFA) running on shared-bus multicore architectures will be extended for a TDMA hypervisor architecture. We will improve the the number of SDFAs being analyzable by our model-checking approach by exploiting the temporal and spatial segregation properties of the TDMA architecture and demonstrate how this method can be applied.

1 Introduction

A look at the current development process of electronic systems in the automotive domain, shows that this development process is still based on the design of distributed single-core ECUs (Electronic Control Units), especially in the hard real-time domain (for safety-critical systems) with a single application running per ECU. Yet, because of the growing computational demand of real-time applications (in automotive, avionics, and multimedia), extending the design process for supporting multicore architectures becomes inevitable. Due to their significantly increased performance and Space Weight and Power (SWaP) reductions, multicores offer an appealing alternative to traditional architectures.

In "traditional" distributed systems, dedicated memory per ECU and predictable field-bus protocols are used. This allows temporal and spatial segregation and timing requirements can be verified using traditional static analysis techniques. In multicore systems, contention on shared resources such as buses and memories makes the static timing analysis of such platforms very hard. To enable the applicability of static analysis techniques in multicore systems, resource virtualization using a static time slot per application has been introduced [1]. Time slots are switched circularly by a resource manager or hypervisor. The hypervisor takes care of the temporal and spatial segregation. Each

application can access all platform resources until its time slot is over. When switching to the next slot, the hypervisor takes care of storing the local state of all platform resources of the terminated slot and restores the local state of the next time slot to be activated. In this paper, we present a composable analysis that is capable to analyze real-time properties of SDFAs on multicore platforms using a shared-bus architecture and dedicated data and instruction memories per processing unit. When using the hypervisor architecture described above, composability can be exploited due to the guaranteed segregation properties.

Our method, allows a model-checking based performance analysis of multiple SDFAs running on a single multicore platform [2]. An SDFA consists of multiple actors that exchange information through tokens and follows the Synchronous Dataflow (SDF) [3] Model of Computation (MoC). SDFAs are represented as Synchronous Dataflow Graphs (SDFGs). Our multicore architecture consists of tiles, each of them has a processor core with its own instruction and data memory. Message passing of tokens is implemented through memory-mapped I/O on a shared memory connected to the tiles via shared bus architecture. The SDF semantics support the clean separation between task computation and communication times which enables the analysis of timing effects though shared memory accesses. The general idea is illustrated with the following example. Four SDFAs, each two of them are mapped onto a dedicated 2-tile platform, as shown on the left side of Fig. 1. Under this mapping all SDFAs meet their required deadlines. On the right side of Fig. 1 all four SDFAs have been integrated on a virtualized 2-tile platform, where every set of applications is statically mapped to a time slot (A, B to slot1 and C, D to slot2). Segregation is implemented through a hypervisor component which implements a TDMA protocol that manages the switching between time slots and guarantees that applications running in different time slots have exclusive access to each tile's processor and have their own private area in the tile's local memories as well as in the shared memory.

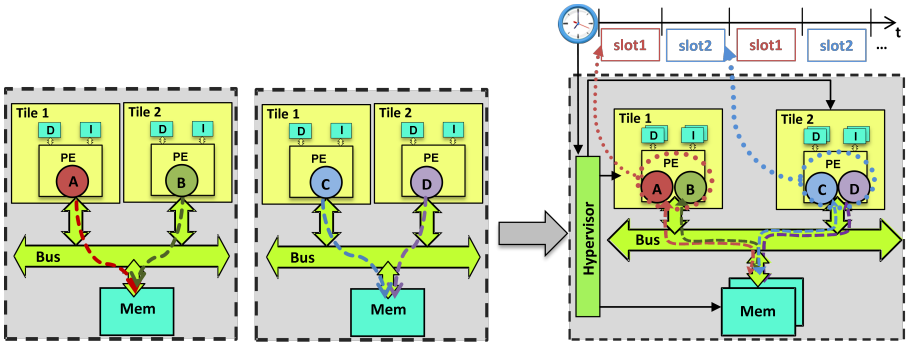

Fig. 1. Integrating four SDFAs on a 2-tile virtualized platform

In this paper, we improve the number of actors being analyzable by our model-checking approach [2] on a fixed number of tiles. This improvement benefits significantly from a composable analysis based on the temporal and spatial segregation properties of virtualized multicore platforms as described above. In Section 2, we discuss the related work addressing the performance analysis

of synchronous dataflow graphs (SDFGs) on multicores. We extend our system model description for virtualized platforms in Section 3. In Section 4 we describe our compositional performance analysis for virtualized systems and evaluate its improvements with regard to scalability of our model-checking approach. The paper closes with a conclusion and gives an outlook on future work.

2 Related Work

2.1 Model-checking

Lv et al. [4] presented an approach based on model-checking (UPPAAL) combined with abstract cache interpretation to estimate WCET of non-sharing code programs on a shared-bus multicore platform. Gustavsson et al. [5] moved further and tried to extend the former work [4] concentrating on modeling code sharing programs and enhancing the hardware architecture with additional data cache but without the consideration of bus contentions. In their work, they considered general tasks modeled at assembly level and analyzed these when mapped to an architecture where every core has its private L1 cache and all cores share an L2 cache without sharing a bus. Yet, the instruction level granularity of the modeled tasks lead to scalability problems even with a platform of four cores, on which four (very simple) tasks run and communicate through a shared buffer. Despite the advantage of the former two approaches being applicable to any code generated/written for any domain, the fine granularity of the code-level or instruction-level impedes the scalability of the model-checking technique. In [2], we intended to limit the application to an SDF MoC and limit the hardware architecture by removing caches, in order to reason about the scalability of a model-checking-based method for the performance analysis of SDFGs. We showed in [2] that our model-checking approach scales up to 36 actors mapped to 4-tiles and up to 96 actors on a 2-tiles platforms. In this paper, we intend to improve the number of actors being analyzable by our method on a fixed number of tiles (up to 4 tiles). In [6] an approach which combines model-checking with real-time analysis was presented to extend the scalability of worst-case response time analysis in multi-cores. Tasks are composed of superblocks where resource access phases can be identified. In this paper, we concentrate on SDF based applications with their specific properties and constraints. It is possible to use the abstraction techniques from [6] to analyze SDF applications. Dong et al. [7] presented a timed automata-based approach to verify the impact of execution platform and application mapping on the schedulability (meeting hard real-time requirements). The granularity of the application is considered at the task level. With tasks and processors having their own timed automata, the approach scales up to 103 tasks mapped to 3 cores. Yet, the communication model is missing in this approach.

2.2 Performance Analysis of SDFGs

Bhattacharyya et al. [3] proposed to analyze performance of a *single* SDFG mapped to a multi-processor system by decomposing it into a homogeneous

SDFG (HSDFG). This could result in an exponential number of actors in the HSDFG compared to the SDFG. This in turn may lead to performance problems for the analysis methods. Ghamarian [8] presented novel methods to calculate performance metrics for single SDF applications which avoid translating SDFGs to HSDFGs. Nevertheless, resource sharing and other architecture properties were not considered. Moone [9] analyzed the mapping of SDFGs on a multiprocessor platform with limited resource sharing. The interconnect makes use of a network-on-chip that supports network connections with guaranteed communication services allowing them to easily derive conservatively estimated bounds on the performance metrics of SDFGs. Kumar [10] presented a probabilistic technique to estimate the performance of SDFGs sharing resources on a multi-processor system. Although this analysis was made taking into account the blocking time due to resource sharing, the estimation approach was aimed at analyzing soft real-time systems rather than those of hard real-time requirements. The work presented in [11] introduces an approach based on state-space exploration to verify the hard real-time performance of applications modeled with SDFGs that are mapped to a platform with shared resources. In contrast to this paper, it does however not consider a shared communication resource. Schabbir et al. [12] presented a design flow to generate multiprocessor platforms for multiple SDFGs. The performance analysis for hard real-time tasks is based on calculating the worst-case waiting time on resources as the sum of all tasks' execution times which can access this resource. This is a safe but obviously a very pessimistic approach. In [13] the composability of SDFGs applications on MPSoC platforms was analyzed. The resource manager proposed in their work, relies on run-time monitoring and intervenes when desired to achieve fairness between the SDFGs running. In difference to their work, we utilize a TDMA-based hypervisor which allows exclusive resource access for SDFGs assigned to time slots. Furthermore, in every time slot a model-checking-based method is utilized for the timing validation of multiple hard real-time SDFGs on a multi-core platform, considering the contention on a shared communication medium with flexible arbitration protocols such as First Come First Serve (FCFS).

3 System Model Definition

Definitions and terms of the system model are based on the X-Chart based synthesis process defined in [14]. We decided to use a formal notation (inspired from [3, 15]) to describe in an unambiguous way, the main modeling primitives and decisions of the synthesis process. This synthesis process takes as first input a set of behavior models, each implemented in the SDF MoC. The second input comprises resource constraints on the target architecture. The output is a model of performance (MoP) that serves as input for our performance analysis.

3.1 Model of Computation (MoC)

An SDF graph (SDFG) is a directed graph which typically consists of nodes (called actors) modeling functions/computations and arcs modeling the data

flow. In SDFGs a static number of data samples (tokens) are consumed/produced each time an actor executes (fires). An actor can be a consumer, a producer or a transporter actor. We describe the formal semantics of SDFGs as follows:

Definition 1. *(Port) A Port is a tuple* $P = (Dir, Rate)$ *where* $Dir \in \{I, O\}$ *defines whether* P *is an input or an output port, and the function* $Rate : P \rightarrow \mathbb{N}_{>0}$ *assigns a rate to each port. This rate specifies the number of tokens consumed/produced by every port when the corresponding actor fires.*

Definition 2. *(Actor) An actor is a tuple* $A = (\mathcal{P}, F)$ *consisting of a finite set* $\mathcal{P}$ *of ports* P*, and* F *a label, representing the functionality of the actor.*

Definition 3. *(SDFG) An SDFG is a triple* $SDFG = (\mathcal{A}, \mathcal{D}, T_s)$ *consisting of a finite set* $\mathcal{A}$ *of actors* A*, a finite set* $\mathcal{D}$ *of dependency edges* D*, and a token size attribute* T_s *(in bits). An edge* D *is represented as a triple* $D = (Src, Dst, Del)$ *where the source (*Src*) of a dependency edge is an output port of some actor, the destination (*Dst*) is an input port of some actor, and* $Del \in \mathbb{N}_0$ *is the number of initial tokens (also called delay) of an edge. All ports of all actors are connected to exactly one edge, and all edges are connected to ports of some actor.*

Definition 4. *(Repetition vector) A repetition vector of an SDFG is defined as the vector specifying the number of times every actor in the SDFG has to be executed such that the initial state of the graph is obtained. Formally, a repetition vector of an SDFG is a function* $\gamma : A \rightarrow \mathbb{N}_0$ *so that for every edge* $(p, q) \in \mathcal{D}$ *from* $a \in \mathcal{A}$ *to* $b \in \mathcal{A}$*,* $Rate(p) \times \gamma(a) = Rate(q) \times \gamma(b)$*. A repetition vector* γ *is called non-trivial if and only if for all* $a \in \mathcal{A} : \gamma(a) > 0$*. In this paper, we use the term repetition vector to express the smallest non-trivial repetition vector.*

3.2 Model of Architecture (MoA)

Fig. 1 depicts our proposed platform architecture template. Each tile is made up of a processing element (PE) which has a configurable bus connection. A shared bus is used to connect the tiles to shared memory blocks. In addition, every PE has two local disjoint memories for instructions (IM)and data (DM). Furthermore, we assume that all tiles have the same bit-width as the bus and the architecture is fully synchronous using a single clock. This architecture enables the actors of SDFGs to communicate via buffers implemented in the shared memory. Only explicit communication (message passing) between actors will be visible on the interconnect and the shared memory. We assume constant access time for any memory block in the shared memory (as in [4]). Furthermore, we assume that bus block transfers are supported.

The hypervisor component implements global time slots on the platform using a TDMA protocol. Each time slot represents a subset (or all) of the platform resources to be used exclusively by the subset of SDFGs statically assigned to this slot. For this reason a shadow memories per slot for each local IM, DM and shared memory location is available to guarantee spatial segregation between different slots. The hypervisor has the role of switching cyclically between the

slots, storing and restoring the local and global state through management of the different shadow memories. We describe our virtualized architecture model as follows:

Definition 5. *(Tile) A tile is a triple* $T = (PE, i, M_p)$ *with processing element* $PE = (PE_{type}, f)$ *where* PE_{type} *is the type of the processor and* f *is its clock frequency,* $i \in \mathbb{N}_{>0}$ *is the number of manageable slots and* $M_p = ((m_{I_0} \dots m_{I_{i-1}}), (m_{D_0} \dots m_{D_{i-1}}))$ *where* $m_I, m_D \in \mathbb{N}_{>0}$ *are the instruction and data memory sizes (in bits) respectively, and the index* i *represents the slot number of this memory. Total size of instruction and data memory is* $i * m_X$ *for* $X \in D, I$.

Definition 6. *(Execution Platform) An execution platform* $EP = (H, \mathcal{T}, B, M_S)$ *consists of a hypervisor component* $H = (Sl, h)$ *where* Sl *is the number of slots the hypervisor can handle, and* h *is the delay of switching from one slot to another, a finite set* $\mathcal{T}$ *of tiles* $T_{H.Sl}$*, a shared bus* $B = (b_b, AP)$*, where* b_b *is the bandwidth in bits/cycle and* AP *is the arbitration protocol (FCFS, TDMA, Round-Robin), and* $M_S = (m_{s_0} \dots m_{s_{H.Sl-1}})$ *a shared memory where all slots have their own dedicated shared memory of the same size* m_s *in bits. Total size of the shared memory is* $H.Sl * m_s$.

Definition 7. *(Virtual Execution Platform) Let* $\mathbf{Sl} := EP.H.Sl$ *be the number of slots managed by the hypervisor. Then a virtual execution platform* $VEP = (l, EP.\mathcal{T}, EP.B, EP.M_{S(0)})_0 \times \dots \times (l, EP.\mathcal{T}, EP.B, EP.M_{S(\mathbf{Sl}-1)})_{\mathbf{Sl}-1}$ *consists of the duration of each slot* l*, the tiles* $\mathcal{T}$*, the shared bus* B *and one shared memory partition* $M_{S(j)}, 0 \leq j \leq EP.H.Sl - 1$ *of* EP*. We define* $VEP(i) = (l, \mathcal{T}, B, M_S)_i$ *as the configuration of EP at the* i*-th time slot.*

3.3 System Synthesis

The system synthesis includes the definition of the Virtual Execution Platform, and the partitioning of SDFGs for different VEPs and the binding and scheduling of the SDFGs on the resources of their VEP. The mapping of the partitioned SDFGs on our VEP model is defined as follows:

Definition 8. *(Mapping) Let* $\mathbf{Sl} := EP.H.Sl$ *be the number of slots managed by the hypervisor. Then for every slot* $0 \leq i \leq \mathbf{Sl} - 1$*, let* $\mathcal{A}_i$ *be the set of actors and* $\mathcal{D}_i$ *the set of all edges of all SDFGs assigned to this slot. Then a mapping for each slot can be defined as a tuple* $M_i = (\alpha_i, \beta_i)$ *with*

1. *the function* $\alpha_i : \mathcal{A}_i \rightarrow VEP(i).\mathcal{T}$ *mapping every actor to a tile (multiple actors can be assigned to one tile)*
2. *the function* $\beta_i : \mathcal{D}_i \rightarrow \bigcup_{VEP(i).\mathcal{T}} M_{p(i)} \cup VEP(i).M_S$ *mapping every edge of the SDFG either to the slot's private memory of the tile or to the slot's shared memory partition.*

An edge mapped to a private or to the shared memory represents a consumer-producer FIFO buffer in an actual implementation [3]. The following three definitions allow us to express the scheduling behavior of multiple SDFGs mapped to the tiles of $VEP(i)$:

Definition 9. *(Static-order schedule) For an SDFG with repetition vector γ, a static-order schedule SO is an ordered list of the actors (to be executed on some tile), where every actor a is included in this list $\gamma(a)$ times.*

Definition 10. *(Scheduling Function) Let $\mathcal{SO}_i$ be the set of all SO schedules for all SDFGs considered in slot i. A scheduling function for slot i is a function $S_i : VEP(i).\mathcal{T} \rightarrow \mathbf{so}_i$, which assigns to every tile $t \in VEP(i).\mathcal{T}$ a subset $\mathbf{so}_i \subseteq \mathcal{SO}_i$.*

Definition 11. *(Scheduler) A scheduler for a slot i is a triple $S_i = (\mathbf{so}_i, F, HS)$ where $\mathbf{so}_i \subseteq \mathcal{SO}_i$ is the set of different SDFGs schedules assigned to one tile, F represents the functionality (code) of the scheduler and HS is the hierarchical scheduling, defining the order (priority) of execution of independent lists of different SDFGs assigned to one tile according to an arbitration strategy (Static-Order, Round-Robin, TDMA).*

We assume that all SDFGs running in the system are known at design time. Furthermore, while the actors execution order is fixed, the consumer-producer synchronization in each tile is performed at run-time depending on the buffer state [3]. A producer actor writes data into a FIFO buffer and blocks if that buffer is full, while a consumer actor blocks when the buffer is empty. An important performance metric of SDFG that will be evaluated in Section 5 is the *period*, defined in this paper as the time needed for one static order schedule of an SDFG to be completed.

3.4 Model of Performance (MoP)

In order to be able to verify that the performance of the SDFG stays within given bounds, we must keep track of all possible timing delays of all SDFGs per slot, with regard to the physical resources of the underlying multicore platform. To achieve this, a MoP is extracted from the synthesis process which includes only the SW/HW components where the timing delay is critical. From the hardware abstraction point of view, we consider a Transaction Level Model (TLM) [16] abstraction for the communication. This means that the application layer issues read/write transactions on the bus, abstracting from the communication protocol (see CAAM model [16]). After synthesis, the following system components can be annotated with execution times/delays: the *scheduler* that implements the static order schedule within an SDFG and the hierarchical scheduling across different SDFGs, the *actors*, the *tiles*, the *bus* and the *shared memories*. A new component (*communication driver)* is introduced into our system, which is responsible of implementing the communication between actors mapped to a tile with other components such as the private memory and the shared memory. In addition, when an actor blocks on a buffer, this driver implements a polling mechanism. If $\mathcal{A}_i$ is the set of actors, $\mathcal{S}_i$ the set of schedulers, $\mathcal{D}_i$ the set of edges, $\mathcal{C}_i$ the set of communication drivers, $VEP(i).B$ the bus, $VEP(i).M_S$ the set of shared memories, and $\bigcup_{VEP(i).\mathcal{T}} M_{p(i)}$ the set of private memories per slot, when considering the performance of the synthesized model, the following delay functions per slot i are defined:

- $\Delta_{A_i} : \mathcal{A}_i \times VEP(i).\mathcal{T} \rightarrow \mathbb{N}_{>0} \times \mathbb{N}_{>0}$ which provides an execution time interval $[BCET, WCET]$ for each actor representing the cycles needed to execute the actor behavior on the corresponding tile. This delay can be estimated using a static analyzer tool.
- $\Delta_{S_i} : \mathcal{S}_i \times VEP(i).\mathcal{T} \rightarrow \mathbb{N}_{>0} \times \mathbb{N}_{>0}$, $\Delta_{C_i} : \mathcal{C}_i \times VEP(i).\mathcal{T} \rightarrow \mathbb{N}_{>0} \times \mathbb{N}_{>0}$ assigns in analogy to Δ_{A_i} to every scheduler and communication driver a delay interval, which can be estimated using a static analyzer tool depending on the code of both components and the platform properties.
- $\Delta_{D_i} : \mathcal{D}_i \times \bigcup_{VEP(i).\mathcal{T}} M_{p(i)} \cup VEP(i).M_S \rightarrow \mathbb{N}_{>0}$ assigns to each communicating edge $d \in D_i$ mapped to a communication primitive a delay which depends on the number and size of the tokens being transported on the edge and the bandwidth of the corresponding communication medium. We assume that the delay on the edge mapped to a private memory is included in the interval calculated by the static analyzer tool for the actors. Likewise, the shared memory access delay is included in the delay of the bus needed to serve a message passing communication.

Now, for each slot i we can abstractly represent every tile by the actors mapped to it, the scheduler, a communication driver, each with their delay as defined before, and its private memory. Each of the private memories in the tiles and the shared memories can be abstracted in a set of (private/shared) FIFO buffers with corresponding sizes depending on the rate of the edges mapped to them and the schedule (each edge is mapped to exactly one FIFO buffer). Note that although no delays are explicitly modeled on the private and shared buffers, these buffers are still considered in the MoP because of their effect on the synchronization which in turn affects the performance.

4 Compositional Performance Analysis Method

4.1 Model-checking based Performance Analysis within a Slot

The following method is used to analyze the performance of all SDFGs mapped to a single slot accessing a subset of the multicore's compute, memory and shared bus resources. The components of the MoP identified in the last Section can be formalized using the timed automata semantics of UPPAAL[3]. The composition can be described as follows:

System = VirtualExecutionPlatform $\|_{i=1}^{q}$ SDFG$_i$
SDFG$_i$ = $\|_{j=1}^{r}$ Consumer$_j$ $\|_{k=1}^{s}$ Producer$_k$ $\|_{l=1}^{t}$ Transporter$_l$
VirtualExecutionPlatform= $\|_{m=1}^{u}$ Tile$_m$ || Bus || $\|_{o=1}^{v}$SharedFIFO$_o$
Tile$_i$= Scheduler$_i$ || CommunicationDriver$_i$ || $\|_{p=1}^{w}$ PrivateFIFO$_p$

where || means parallel composition of timed automata in UPPAAL, q is the number of SDFGs, r, s, t represent the number of actors (distinguished according to their type), u is the number of tiles, v is the number of shared FIFO, and w

[3] UPPAAL 4.1.11 (rev. 5085), has been used in the experiments

is the number of private FIFO buffers. In [2], we described the implementation and the interactions of timed-automata of different components of the MoP. In addition we illustrated, how performance metrics such as the Worst Case Period (WCP) can obtained with the help of UPPAAL model-checker. The evaluation in [2] showed that this method suffers from scalability limitations. E.g. up to 36 actors on a 4-tile platform and up to 96 actors on a 2-tile platform could be analyzed in a reasonable amount of time.

4.2 Performance Analysis across the Slots

With the proposed extension to our system model definition in Section 3, our approach in [2] can be used to obtain the WCP of multiple SDFGs per slot with the help of model-checking. In this subsection we describe how this WCP changes when considering other slots where different SDFGs are mapped. As described above, the hypervisor implements a temporal and spatial segregation between SDFGs of different slots. I.e. all SDFGs of a slot have exclusive access to the resources and no contention with other SDFGs from other slots can appear. Yet every SDFG in one slot can still have contention with other SDFGs mapped to the same slot. This contention and its effect can be analyzed using the model-checking method presented in the last Section 4.1 and in [2].

For the construction of the slots in the VEP, the length $VEP(i).l$ of every slot i is set to be equal to the maximum WCP of all SDFGs mapped to this slot. Since the hypervisor has the role to dispatch/suspend SDFGs in every slot, an execution platform dependent slot switching delay overhead $\mathbf{h} := EP.H.h$ is induced at the beginning of each slot. Assuming that SDFGs running in one slot are independent from those running in other slots, the following formula can be used to determine the WCP_{compos} of every SDFG after the composition:

$$WCP_{compos} = \sum_{i=0}^{\mathbf{Sl}} WCP_{max}(i) + (\mathbf{Sl} \times \mathbf{h}), \tag{1}$$

where $WCP_{max}(i)$ is the maximal WCP among the SDFGs running in slot i and $\mathbf{Sl} := EP.H.Sl$ is the total number of slots.

5 Evaluation

5.1 Performance Analysis

Suppose we have four SDFGs, each two of them are mapped onto a 2-tile platform (see Fig. 1). Now, we have the task to integrate both platforms on one multicore platform, such that they still meet their timing requirements. This is indeed, a typical use-case in many domains nowadays (automotive, avionics). The goal of this experiment, is to demonstrate how our proposed method can be applied to above use-case, and to show that in case the contention on the bus is high, partitioning induces only minor performance penalties.

Tab. 1 shows the parameters of the four artificial SDFGs, we constructed to examine the claim above. The actors' worst-case execution times were generated

randomly (uniformly distributed) within a range of [5..500] cycles, and a timing requirement (WCP_{req}) was imposed on every SDFG. We have set the ports' rates deliberately high, in order to impose contention on the bus. High rates lead to longer communication time of the active actor, and this in turn leads to longer waiting time of other actors trying to access the bus. In addition, all edges of all SDFGs in all mappings were mapped to the shared memory in order to achieve a high contention on the bus. The bus has a bandwidth of 32 bits/cycle, a FCFS arbitration protocol and all tokens are of size 32 bits. Moreover, all SDFGs were scheduled according to a static order schedule.

First, we configured the timed automata templates to evaluate the mapping of the considered SDFGs, each pair on a 2-tile platform (see Fig. 1 left). The Worst-case Period (WCP_{isol}) values for every SDFG were calculated using the model-checking method as described in Section 4.1. Next, we integrated the four SDFGs and mapped them on a 2-tile platform but without the hypervisor component. Again, we utilize the model-checking based analysis to find the new WCP (WCP_{nocomp}) of every SDFG (see Tab. 1 (Exp. 2 tiles)). After that, we take use of the hypervisor extension, configuring two time slots. SDFGs A, B are assigned to $slot1$, and C, D are assigned to $slot2$ (see Fig. 1 right). The length of every slot is equivalent to the maximum WCP_{isol} (WCP_{max}) among the SDFGs assigned to this slot (slot1: 59895, slot2: 85001). The new WCPs (WCP_{compos}) are calculated according to Formula (1) assuming a hypervisor delay h of 100 cycles. The results depicted in Tab. 1 (Exp. 2 tiles), show that all SDFGs still respect their requirements, with a minor performance degradation of average 12.5% in the case of temporal and spatial segregation through the hypervisor.

5.2 Scalability

The model-checking method presented in [2] does not scale beyond 36 actors mapped to a 4-tile platform. In order to demonstrate the scalability improvement of our proposed extension, we consider the same set of artificial SDFGs presented above which have in total 36 actors, and another set of SDFGs (E, F, G and H) also having 36 actors (see Tab. 1). Every set was mapped on a 4-tile platform (without hypervisor) and both were first analyzed in isolation with the help of the model-checking method. After obtaining the WCP_{isol} of the single SDFGs

Table 1. Experiments Setup and Results, WCP in cycles

	SDFGs Parameters			**Exp. 2 tiles**				**Exp. 4 tiles**	
	Actors	Chan	Ports' Rate	WCP_{req}	WCP_{isol}	WCP_{nocomp}	WCP_{compos}	WCP_{isol}	WCP_{compos}
A	10	9	[1200,2400]	160000	54529	140863		135400	
B	10	9	[200,600]	160000	59895	117439	**145096**	171000	
C	10	9	[220,440]	160000	85001	141734		135400	
D	6	5	[100,200]	160000	44236	119466		69600	
E	10	9	[500,2000]					107850	**279050**
F	10	9	[300,600]					64500	
G	10	9	[700,1400]					66950	
H	6	5	[150,300]					37300	

in isolation (see Tab. 1: Exp. 4 tiles), we now map the 8 SDFGs onto a 4-tile platform with a hypervisor with two slots and a slot switching delay h of 100 cycles. SDFGs A, B, C, D were assigned to $slot1$ with the length 171000 cycles and E, F, G, H to $slot2$ having a length of 107850. Afterwise, we calculated the new WCP_{compos} of the single SDFGs according to (1) (see Tab. 1: Exp. 4 tiles). The results show that our composable analysis doubles the number of actors, which can be analyzed compared to [2] for this example, at the cost of performance degradation.

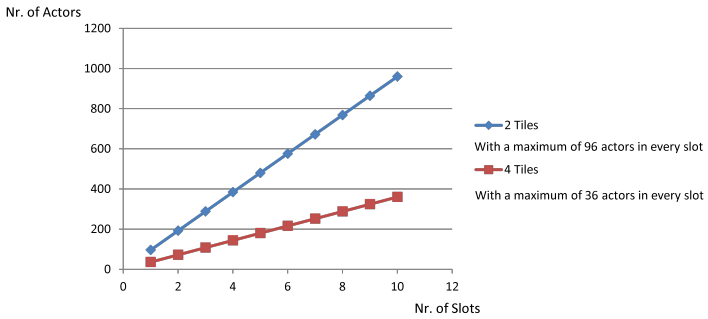

Fig. 2. Scalability Results

Clearly, we can now increase the number of SDFGs that can be analyzed by increasing the number of slots managed by the hypervisor. Fig. 2 shows that by 10 slots we could analyze up to 960 actors on a 2-tile platform and 360 actors on a 4-tile platform. Nevertheless, the designer should be acquainted with the fact that by increasing the number of slots the performance overhead of the single SDFG would be increased (for Exp. 4 tiles an average of 255%).

6 Conclusion

In this paper, we have presented a composable extension to our model-checking-based performance analysis method for the validation of hard real-time SDFGs mapped to a virtualized shared-bus multicore platform. Exploiting the temporal and spatial segregation properties of the hypervisor, significantly improves scalability depending on the number of slots (by ten slots) up to 360 actors mapped to 4-tile and up to 960 actors on a 2-tile platforms. Future work will address relaxing the MoC towards dynamic data flow graphs, and relaxing architecture constraints towards interrupts, cross-bar switches, and dedicated FIFO channels.

Acknowledgement

This paper has been partially supported by the MotorBrain ENIAC project under the grant (13N11480) of the German Federal Ministry of Research and Education (BMBF).

Bibliography

[1] Aeronautical Radio, I.: Arinc 653: Avionics application software standard interface. Technical report, ARINC, 2551 Riva Road Annapolis, MD 21401, U.S.A (2003)

[2] Fakih, M., Grüttner, K., Fränzle, M., Rettberg, A.: Towards performance analysis of SDFGs mapped to shared–bus architectures using model–checking. In: Proceedings of the Conference on Design, Automation and Test in Europe. DATE '13, Leuven, Belgium, European Design and Automation Association (March 2013)

[3] Sriram, S., Bhattacharyya, S.S.: Embedded Multiprocessors: Scheduling and Synchronization. 1 edn. CRC Press (March 2000)

[4] Lv, M., Yi, W., Guan, N., Yu, G.: Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software. In: 2010 31st IEEE Real-Time Systems Symposium. (2010) 339–349

[5] Gustavsson, A., Ermedahl, A., Lisper, B., Pettersson, P.: Towards WCET Analysis of Multicore Architectures Using UPPAAL. 10th (2011) 101–112

[6] Giannopoulou, G., Lampka, K., Stoimenov, N., Thiele, L.: Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems. In: Proc. International Conference on Embedded Software (EMSOFT), Tampere, Finland, ACM (Oct 2012) 63–72

[7] Dong-il, C., Hyung, C., Jan, M.: System-Level Verification of Multi-Core Embedded Systems Using Timed-Automata. (July 2008) 9302–9307

[8] Ghamarian, A.: Timing Analysis of Synchronous Data Flow Graphs. PhD thesis, Eindhoven University of Technology (2008)

[9] Moonen, A.: Predictable Embedded Multiprocessor Architecture for Streaming Applications. PhD thesis, Eindhoven University of Technology (2009)

[10] Kumar, A.: Analysis, Design and Management of Multimedia Multiprocessor Systems. PhD thesis, Ph. D. thesis, Eindhoven University of Technology (2009)

[11] Yang, Y., Geilen, M., Basten, T., Stuijk, S., Corporaal, H.: Automated bottleneck-driven design-space exploration of media processing systems. In: Proceedings of the Conference on Design, Automation and Test in Europe. DATE '10, Leuven, Belgium, European Design and Automation Association (2010) 1041–1046

[12] Shabbir, A., Kumar, A., Stuijk, S., Mesman, B., Corporaal, H.: CA-MPSoC: An Automated Design Flow for Predictable Multi-processor Architectures for Multiple Applications. Journal of Systems Architecture **56**(7) (2010) 265–277

[13] Kumar, A., Mesman, B., Theelen, B., Corporaal, H., Ha, Y.: Analyzing composability of applications on MPSoC platforms. J. Syst. Archit. **54**(3-4) (March 2008)

[14] Gerstlauer, A., Haubelt, C., Pimentel, A., Stefanov, T., Gajski, D., Teich, J.: Electronic System-Level Synthesis Methodologies. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **28**(10) (October 2009) 1517 –1530

[15] Stuijk, S.: Predictable Mapping of Streaming Applications on Multiprocessors. Volume 68. University Microfilms International, P. O. Box 1764, Ann Arbor, MI, 48106, USA (2007)

[16] Cai, L., Gajski, D.: Transaction Level Modeling: an Overview. In: First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2003. (October 2003) 19–24