Towards Performance Analysis of SDFGs Mapped to Shared-Bus Architectures Using Model-Checking

Maher Fakih*, Kim Grüttner*, Martin Fränzle† and Achim Rettberg†
*OFFIS Institute for Information Technology, Germany
†Carl von Ossietzky University, 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 became very challenging. In this paper, a model-checking based approach will be described which allows us to guarantee timing bounds of multiple Synchronous Data Flow Graphs (SDFG) running on shared-bus multicore architectures. Our approach utilizes Timed Automata (TA) as a common semantic model to represent software components (SDF actors) and hardware components of the multicore platform. These TA are explored using the UPPAAL model-checker for providing the timing guarantees. Our approach shows a significant precision improvement compared with the worst-case bounds estimated based on maximal delay for every bus access. Furthermore, scalability is examined to demonstrate analysis feasibility for small parallel systems.**

I. Introduction

Multicores are becoming standard in the area of embedded systems. Due to their significantly increased performance and decreased energy consumption, they offer an appealing alternative to traditional architectures. This fact, together with the growing computational demand of real-time applications (in automotive, avionics, and multimedia), stresses the need for methods to prove the timing predictability of software applications running on such architectures for guaranteeing their safe usage in a real-time domain. Yet the timing prediction of multicore platforms with hard real-time requirements is very challenging. The high contention on shared resources such as busses and caches makes the static timing analysis of such platforms very hard. To cope with this challenge, predictable by construction platforms with predictable arbitration protocols have been suggested. In general, predictability is coupled with performance degradation.

There are mainly two performance analysis approaches for embedded applications: simulative and formal methods. Because exhaustive simulation is never complete and might not cover all interesting corner cases, a formal approach is needed to estimate safe upper bounds on the application execution time. In this approach, a mathematical (static) analysis is performed on a formal representation of both the software and the hardware. This analysis takes into consideration all possible inputs and combinations of the running applications with all different hardware states of the proposed platform. Since multicore architectures are composed of concurrent components and their synchronization depends on timing constraints, formal models like timed automata and model-checkers like UPPAAL [1] are very suitable to capture and verify the behavior of these systems. Model-checking techniques are capable of verifying the performance properties of a system with vigor, in contrast to simulative approaches. Furthermore, for unmet timing properties counter examples are provided. Unfortunately, these techniques are not scalable when analyzing full featured multicore systems (with cache, preemption and shared bus) on which general applications are run. In this paper, we limit the application being analyzed to the Synchronous Dataflow Flow (SDF) [2] Model of Computation (MoC). SDF semantics support the clean separation between task computation and communication time which makes it easier to predict timing effects of global memory accesses. Furthermore, we consider a simple multicore architecture where each core has its own instruction and data memory and is called a "tile". Tiles are connected through an "unpredictable" shared bus and message passing between them is realized through memory-mapped I/O on a shared memory. We claim the following contributions:

1) We utilize model-checking to find the timing bounds of multiple (hard real-time) SDF-based applications mapped to a multicore platform considering variable access delays due to the contention on the bus.
2) We evaluate our approach and show that it allows a better scalability than the approach in [3]. At the same time, it gives more precise guarantees on hard real-time SDFGs than a pessimistic analysis method [4].

This paper is structured as follows: In Section II we discuss the related work mainly addressing the performance analysis of synchronous dataflow graphs (SDFGs) on multicores. Next, we describe the considered system model. Section IV describes our proposed performance analysis method. Afterwards, the method is evaluated in terms of scalability and in comparison to a pessimistic analysis method. Finally, the last Section concludes the paper and gives an outlook on future work.

II. Related Work

A. Model-checking

Lv et al. [5] 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. [3] moved further and tried to extend the former work [5] concentrating on modeling code sharing programs and enhancing the hardware architecture with additional data cache but

978-3-9815370-0-0/DATE13/

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 this work 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. 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 ressource 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.

B. Performance Analysis of SDFGs

Bhattacharyya et al. [2] 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. [4] 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 as we will show in Section V-B.

To the best of our knowledge, our proposed methodology is novel in two regards w.r.t. above approaches: 1) it utilizes model-checking for the timing validation of multiple hard real-time SDFGs on a multicore platform and 2) it considers the contention on a shared communication medium with flexible arbitration protocols such as First Come First Serve (FCFS), providing more flexible analysis than above approaches, which analyzed SDFGs on interconnects with predictable protocols (such as Time Division Multiple Access (TDMA)).

III. System Model Definition

All the definitions and terms of the system model are based on the X-Chart based synthesis process defined and described in [12]. We decided to use a formal notation (inspired from [2, 13]) to describe in a precise and 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.

A. 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. Fig. 1 depicts the SDFG of a JPEG encoder (cf. [4]). While the producer (*get_MB*) and consumer (*VLC*) actors respectively either produce or consume tokens, a transporter actor (*CC*, *DCT*) does both. The JPEG encoder SDFG consists of four actors: a macroblock sampling (get_MB) which parses an input BMP file and sends 3 macro-blocks (each 16x16 pixels) to a color conversion (CC) actor. The CC actor can fire if 128 pixels are available on its input edge and sends 64 pixels to the discrete cosine transform (DCT) actor which in turn sends with each firing 64 pixels to the variable length coding (VLC) 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

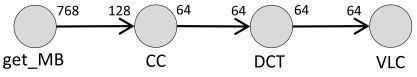

Figure 1. SDFG of a JPEG encoder

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.

B. Model of Architecture (MoA)

Fig. 2 depicts our proposed architecture template. The tile is made up of a processing element (PE) which has a configurable bus connection. In addition, every PE has two local disjoint memories: an instruction (IM) and a data memory (DM). An interconnect such as a bus is used to connect the tiles to shared memory blocks in order to allow communication using shared memory. This enables SDFGs mapped to different tiles to communicate via buffers mapped to this 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 [5]). Furthermore, we assume the architecture to be synchronous so that no possible delays due to different clock cycles of architecture components are considered.

Definition 5: (Tile) A tile is a tuple $T = (PE, M_p)$ with processing element $PE = (PE_{type}, f)$ where PE_{type} is the type of the processor and f is its clock frequency, and $M_p = (m_i, m_d)$ where $m_i, m_d \in \mathbb{N}_{>0}$ are the instruction and data memory sizes (in bits) respectively.

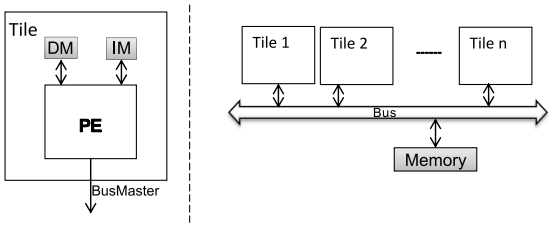

Figure 2. Proposed Platform

Definition 6: (Execution Platform) An execution platform $EP = (\mathcal{T}, B, \mathcal{M}_\mathcal{S})$ consists of a finite set $\mathcal{T}$ of tiles, a shared bus $B = (b_b, AP)$ with b_b being the bandwidth in bits/cycle, AP is the arbitration protocol (FCFS, TDMA, Round-Robin) and $\mathcal{M}_\mathcal{S}$ a finite set of shared memories, each of them having specific size m_s in bits.

C. System Synthesis

The system synthesis includes the processes of binding and scheduling the behavioral model on the defined architecture. Mapping the SDFG on our MoA is defined as follows:

Definition 7: (Mapping) If $\mathcal{A}$ is the set of actors of all SDFGs, $\mathcal{D}$ the set of all edges, $\mathcal{T}$ the set of tiles of the platform configuration, $\mathcal{M}_\mathcal{S}$ the set of all shared memories, $\mathcal{M}_\mathcal{P}$ the set of all private memories, then a mapping can be defined as a tuple $M = (\alpha, \beta)$ with

1) the function $\alpha : \mathcal{A} \rightarrow \mathcal{T}$ mapping every actor to a tile (multiple actors can be assigned to one tile)
2) the function $\beta : \mathcal{D} \rightarrow \mathcal{M}_\mathcal{P} \cup \mathcal{M}_\mathcal{S}$ mapping every edge of the SDFG either to a private memory of the tile or to a shared memory.

An edge mapped to a private or to a shared memory represents a consumer-producer FIFO buffer in an actual implementation [2]. The following three definitions allow us to express the scheduling behavior of multiple SDFGs mapped to the tiles:

Definition 8: (*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 9: (Scheduling Function) Let $\mathcal{SO}$ be the set of all SO schedules for all SDFGs considered in the system. A scheduling function is a function $S : \mathcal{T} \rightarrow \mathbf{so}$, which assigns to every tile $t \in \mathcal{T}$ a subset $\mathbf{so} \subseteq \mathcal{SO}$.

Definition 10: (Scheduler) A scheduler is a triple $S = (\mathbf{so}, F, HS)$ where $\mathbf{so} \subseteq \mathcal{SO}$ 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 is performed at run-time depending on the buffer state [2]. 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 V is the *period*, defined in this paper as the time needed for one static order schedule of an SDFG to be completed. Fig. 3 shows two SDFGs for a *JPEG encoder* and a *Sobel filter* (left) and a 4-tiles platform (right). One possible synthesis would be realized by:

1) Mapping actors with the same color to corresponding colored tile for e.g. *DCT* and *GY* are mapped to *Tile-3*.

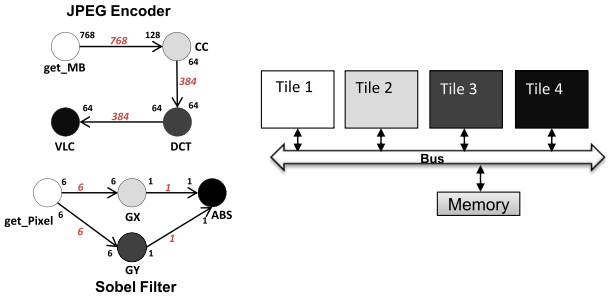

Figure 3. Mapping of JPEG encoder and Sobel Filter on a 4-tiles platform

2) Mapping each edge of both SDFGs to a FIFO buffer in the shared memory, which can be accessed through the bus (the buffer sizes are annotated on the middle of every edge with an italic/red number).
3) Calculating a static order schedule for *JPEG encoder*: $(get\_MB)(CC)^6(DCT)^6(VLC)^6$ and one for the *Sobel filter*: $(get\_Pixel)(GX)(GY)(ABS)$, where the exponent indicates how often the actor is executed in the schedule.
4) Choosing *static-order* strategy for the hierarchical scheduling with priority(*JPEG*)>priority(*Sobel*).

D. 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 mapped SDFGs to the 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) [14] abstraction for the communication. This means that the application layer issues read/write transactions on the bus, abstracting away from the communication protocol (see CAAM model [14]). 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 $\mathcal{A}$ is the set of actors, $\mathcal{S}$ the set of schedulers, $\mathcal{D}$ the set of edges, $\mathcal{C}$ the set of communication drivers, B the bus, $\mathcal{M}_\mathcal{S}$ the set of shared memories, and $\mathcal{M}_\mathcal{P}$ the set of private memories, when considering the performance of the synthesized model, the following delay functions are defined:

- $\Delta_A : \mathcal{A} \times \mathcal{T} \rightarrow \mathbb{N}_{>0} \times \mathbb{N}_{>0}$ 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.
- $\Delta_S : \mathcal{S} \times \mathcal{T} \rightarrow \mathbb{N}_{>0} \times \mathbb{N}_{>0}$, $\Delta_C : \mathcal{C} \times \mathcal{T} \rightarrow \mathbb{N}_{>0} \times \mathbb{N}_{>0}$ assigns in analogy to Δ_A 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.
- $\Delta_D : \mathcal{D} \times \mathcal{M}_\mathcal{P} \cup \mathcal{M}_\mathcal{S} \rightarrow \mathbb{N}_{>0}$ assigns to each communicating edge $d \in D$ 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, 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.

IV. Performance Analysis of SDFGs

The components of the MoP identified in the last Section can be formalized using the timed automata semantics of UPPAAL[1]. The composition can be described as follows:

System = ExecutionPlatform $\|\ _{i=1}^{q}$ SDFG$_i$
SDFG$_i$ = $\|\ _{j=1}^{r}$ Consumer$_j$ $\|\ _{k=1}^{s}$ Producer$_k$ $\|\ _{l=1}^{t}$ Transporter$_l$
ExecutionPlatform= $\|\ _{m=1}^{u}$ 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 is the number of private FIFO buffers. The edges in the SDFG with the port-to-port bindings, the mapping decisions, and system configuration parameters (e.g. WCET and buffer sizes) were implemented as global variables in UPPAAL. Fig. 4 depicts the interactions between the timed automata of different components of the MoP. The scheduler starts and activates the actors mapped to a tile according to a scheduling mechanism (in this paper *static order* between and among SDFGs). When an actor needs to communicate with another actor it issues a *Read/Write* signal to the communication driver which realizes the communication with the bus and the private FIFO buffers depending on the mapping. The bus arbitrates different requests from different tiles according to a specific arbitration mechanism (in this paper FCFS) and issues a communication to the specific buffer. If the communication is successful then a *FinishSharedFIFO* signal

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

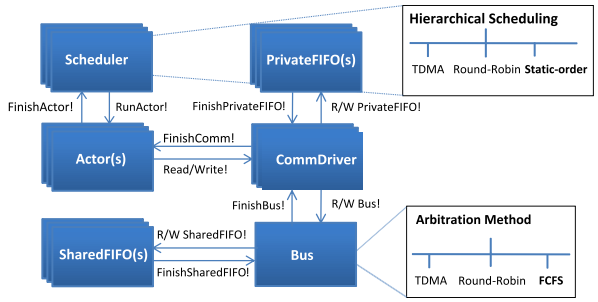

Figure 4. MoP in UPPAAL with all interactions

is returned to the communication driver, which acknowledges this by sending a *FinishComm* signal to the actor. If the target buffer is blocked, it issues a *FinishBlock* signal that the bus propagates to the communication driver which in turn waits for some time before it retries the communication (polling). Fig. 5 shows in detail the timed automaton template of a *transporter* actor. The states of the actor alternate between *Idle, ReadAllPorts, WaitCommRead, Compute, WriteAllPorts, WaitCommWrite* and *Finish*. After receiving a *runActor* signal from the scheduler, the actor reads on all its ports according to their rates, and for every communication issues a *read* signal to the communication driver. When the communication of the last port is finished, an interval of BCET/WCET is delayed (*Compute* state). Then the actor writes to all its ports depending on their rates, and after writing to all ports, the actor sends a *finishActor* signal indicating a single successful execution.

A. Performance Analysis by verification

UPPAAL can verify whether a property holds for a given network of timed automata or not. The verification properties can be formalized in a subset of CTL (Computation Tree Logic). By checking `A [] not deadlock`, we can verify if our system is always deadlock free. We also take use of the model-checker operator sup which searches for the *supremum* of a variable or a clock value in the system. Likewise, we could find the *infimum* by utilizing the inf operator.

To obtain the period of an SDFG, we need to implement an observer automaton which traces the finishing time of the last instance of the sink actor in the static order schedule of that SDFG. We can now utilize the sup/inf operator to search for the maximum/minimum delay between two consecutive finishing instances of the sink actor which coincides with the (worst/best case) period of the graph.

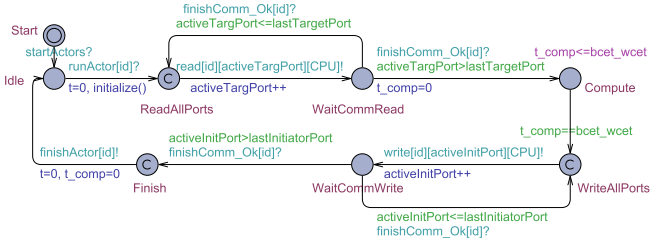

Figure 5. Template of SDF transporter actor

Table I
SCALABILITY RESULTS ANALYSIS TIME IN (S)

Actors Count	2 Tiles	4 Tiles	6 Tiles
8	0.50	34.60	$> 1h$(aborted)
16	1.79	36.70	
32	11.90	277.90	
36	102.50	1038.00	
40	45.79	$> 0.75h$(aborted)	
64	213.24		
96	1050.10		
100	$> 1h$(aborted)		

V. EVALUATION

A. Scalability

In order to test the scalability of our method, we took the JPEG encoder SDFG (see Fig. 1) and used its parameters to instantiate the timed automata templates. We then varied the number of JPEG SDFGs in the system and the number of tiles. For every variation, we measured the analysis time (on a Quad-core running at 2.4 GHz with 4 GB of RAM) consumed, checking that the system does not deadlock. The results achieved are shown in Tab. I and indicate a better scalability than in [3]. Our approach scales up to 36 actors mapped to 4-tiles and up to 96 actors on a 2-tiles platform. The verification run was aborted by the tool with 6 tiles and 8 actors after 1 hour of analysis, when the memory was exhausted.

B. Accuracy

Our goal is to make a comparison between the output of our analysis method with that of a pessimistic method considered in [4]. In their work, the authors calculate the worst-case waiting times for non-preemptive systems with FCFS strategy by assuming that all other competing actors mapped to this resource come to run before the waiting actor. In our case, this means that for every tile (*tile A*) access to the bus, it should be assumed that the actor with the maximal communicating time on every other tile runs to completion before tile *A* gets access to the bus. The authors admit pessimistic results for large number of applications. We will show in the following how pessimistic these estimations can be. In order to do that, we will be using the system already described in Fig. 3, consisting of two real-life SDFGs mapped to a 4-tiles platform and configured with the parameters listed in Tab. II. t_{wcet} (in cycles) is the WCET given by static analyzer for every actor (values were adopted from [4]). t_{com} (in cycles) is the communication time needed by every actor firing to transport a number of tokens each of size 32 bits over a bus with a bandwidth of 32 bits/cycle. First we configured the timed automata templates to evaluate different mappings and schedules of the considered SDFGs (see Tab. III). All edges in all mappings were mapped to the shared memory in order to achieve a high contention on the bus. To obtain the

Table II
ACTORS EXECUTION TIMES IN CYCLES

	getMB	CC	DCT	VLC	getPixel	GX	GY	ABS
$\mathbf{t}_{wcet}$	13220	4446	20950	5420	320	77	77	123
$\mathbf{t}_{com}$	768	192	128	64	12	7	7	2

Table III
STATIC ORDER SCHEDULES EXPERIMENTED

Sched.	Tile-1	Tile-2	Tile-3	Tile-4
S1	(getMB) (CC)^6 (getPixel) (GX)	(DCT)^6 (VLC)^6 (GY) (ABS)	-	-
S2	(getMB) (CC)^6 (DCT)^6 (VLC)^6	(getPixel) (GX) (GY) (ABS)	-	-
S3	(getMB) (getPixel)	(CC)^6 (GX)	(DCT)^6 (GY)	(VLC)^6 (ABS)
S4	(getPixel) (getMB)	(GX) (CC)^6	(GY) (DCT)^6	(ABS) (VLC)^6

worst case period duration (WCP) for an SDFG, the schedule of the SDFG and the *Worst Case Response Time (WCRT)* of every actor are needed. In Section IV-A we pointed out how the WCP can be computed using our model-checking-based approach. For the pessimistic method, we define the WCRT for every actor as follows:

$$t_{wcrt} = t_{wcet} + t_{com} + (MNC \times t_{wait}), \quad (1)$$

where *MNC* is the *Maximum Number of Communication* attempts that an actor (by one activation) can launch on the bus in a given period. The waiting time t_{wait} of actor *a* mapped to a tile *m* on every communication attempt is defined as:

$$t_{wait} = \sum_{i=0}^{n} AWCT_i - AWCT_m, \quad (2)$$

where *n* is the number of tiles in the system, AWCT$_i$ is the *Actor with the Worst Communication Time* (t$_{com}$) among the actors mapped to tile *i*.

The MNC highly depends on the number of ports of the actor and on the polling parameters (when blocking on shared buffer). To achieve a fair comparison, we extracted for every configuration in Tab. III the MNC of every actor with the help of the model-checker and used it to calculate the WCRT of every actor according to (1). This guarantees that both methods work with the same MNC for every actor. For every configuration, we calculated the WCP once using our model-checking-based approach (MC WCP) and once with the help of the pessimistic method (Pess. WCP). Except for S2 configuration of the JPEG encoder, where the WCP estimated by our method gave only an improvement of 0.1%, all other results in Tab. IV indicate significant accuracy improvements over the pessimistic method. The minimal improvement in S2 is due to the fact that the waiting time (t_{wait}) of JPEG actors mapped to *Tile-1* by every bus access was minimal (12 cycles, compared to the Sobel filter actors in S2 where t_{wait} was 768 cycles). Another factor was that the MNC of all actors in S2 was the smallest among all other configurations (ranging from 1 to 2 communication attempts on the bus, where as in S1 the MNC of the actors ranged from 2 to 56 access attempts).

Table IV
WORST CASE PERIOD (WCP) RESULTS IN CYCLES

Sched.	JPEG encoder			Sobel Filter		
	MC. WCP	Pess. WCP	% Impr	MC. WCP	Pess. WCP	% Impr
S1	178450	367112	106%	178671	370339	107%
S2	201292	201606	0.1%	1425	6793	377%
S3	151381	591450	291%	151512	471987	211%
S4	151448	607050	300%	150444	535491	256%

VI. CONCLUSION

In this paper, we have demonstrated the applicability of our model-checking-based performance analysis method to validation of hard real-time SDFGs mapped to a shared-bus multicore platform. In terms of scalability, the proposed method scales up to 36 actors mapped to 4-tiles and up to 96 actors on a 2-tiles platforms. In addition, our method shows a significant reduction in the worst-case response time prediction, compared to a pessimistic analysis method known from literature. Future work will address improving our approach w.r.t. scalability by using a discrete-time model checker (UPPAAL uses a dense time model), 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).

REFERENCES

[1] J. Bengtsson and W. Yi, "Timed Automata: Semantics, Algorithms and Tools," in *In Lecture Notes on Concurrency and Petri Nets. LNCS 3098*. Springer-Verlag, 2004, pp. 87–124.

[2] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, 1st ed. CRC Press, Mar. 2000.

[3] A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson, "Towards WCET Analysis of Multicore Architectures Using UPPAAL," *10th*, pp. 101–112, 2011.

[4] A. Shabbir, A. Kumar, S. Stuijk, B. Mesman, and H. Corporaal, "CA-MPSoC: An Automated Design Flow for Predictable Multi-processor Architectures for Multiple Applications," *Journal of Systems Architecture*, vol. 56, no. 7, pp. 265–277, 2010.

[5] M. Lv, W. Yi, N. Guan, and G. Yu, "Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software," in *2010 31st IEEE Real-Time Systems Symposium*, 2010, pp. 339–349.

[6] G. Giannopoulou, K. Lampka, N. Stoimenov, and L. Thiele, "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, pp. 63–72.

[7] C. Dong-il, C. Hyung, and M. Jan, "System-Level Verification of Multi-Core Embedded Systems Using Timed-Automata," C. Myung, Ed., Jul. 2008, pp. 9302–9307.

[8] A. Ghamarian, "Timing Analysis of Synchronous Data Flow Graphs," Ph.D. dissertation, Eindhoven University of Technology, 2008.

[9] A. Moonen, "Predictable Embedded Multiprocessor Architecture for Streaming Applications," Ph.D. dissertation, Eindhoven University of Technology, 2009.

[10] A. Kumar, "Analysis, Design and Management of Multimedia Multiprocessor Systems," Ph.D. dissertation, Ph. D. thesis, Eindhoven University of Technology, 2009.

[11] Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal, "Automated bottleneck-driven design-space exploration of media processing systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '10. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2010, pp. 1041–1046.

[12] A. Gerstlauer, C. Haubelt, A. Pimentel, T. Stefanov, D. Gajski, and J. Teich, "Electronic System-Level Synthesis Methodologies," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1517 –1530, Oct. 2009.

[13] S. Stuijk, *Predictable Mapping of Streaming Applications on Multiprocessors*. University Microfilms International, P. O. Box 1764, Ann Arbor, MI, 48106, USA, 2007, vol. 68, no. 04.

[14] L. Cai and D. Gajski, "Transaction Level Modeling: an Overview," in *First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2003*, Oct. 2003, pp. 19–24.