Journal of Systems Architecture xxx (2015) xxx–xxx

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Dataflow formalisation of real-time streaming applications on a Composable and Predictable Multi-Processor SOC

Andrew Nelson [a,*], Kees Goossens [a], Benny Akesson [b]

[a] Eindhoven University of Technology, The Netherlands
[b] Czech Technical University in Prague, Czech Republic

ARTICLE INFO

Article history:
Received 12 July 2014
Received in revised form 7 January 2015
Accepted 2 April 2015
Available online xxxx

Keywords:
Real time
Mixed time criticality
Multi-processor
GALS
Dataflow
Performance analysis

ABSTRACT

Embedded systems often contain multiple applications, some of which have real-time requirements and whose performance must be guaranteed. To efficiently execute applications, modern embedded systems contain Globally Asynchronous Locally Synchronous (GALS) processors, network on chip, DRAM and SRAM memories, and system software, e.g. microkernel and communication libraries. In this paper we describe a dataflow formalisation to independently model real-time applications executing on the CompSOC platform, including new models of the entire software stack. We compare the guaranteed application throughput as computed by our tool flow to the throughput measured on an FPGA implementation of the platform, for both synthetic and real H.263 applications. The dataflow formalisation is composable (i.e. independent for each real-time application), conservative, models the impact of GALS on performance, and correctly predicts trends, such as application speed-up when mapping an application to more processors.

1. Introduction

By definition embedded systems are part of a larger device such as a car, mobile phone, or hearing aid. These devices interact with the physical environment that progresses in real time. Embedded systems must therefore respect real-time deadlines to avoid undesirable behaviour (e.g. stuttering audio or video), disallowed behaviour (e.g. violating communication standards), or even unsafe behaviour (e.g. late message in brake by wire).

As integration of computing performance on a single System on Chip (SOC) increases, it is now possible to integrate multiple applications on a single embedded system. For cost reasons this is increasingly common. Running multiple applications on a single (SOC) usually introduces interference of their (timing) behaviours, which complicates the verification of their guaranteed performance. This is especially the case in mixed-time-criticality systems, in which some of the applications do not require those guarantees, and may have unknown or unbounded timing characteristics.

We address the problem of designing embedded systems *executing multiple applications of mixed time-criticality*. We focus on the verification of real-time streaming applications (but in the presence of non-real-time applications). Any solution must include: (1) a *programming model* for real-time applications, (2) a hardware and software *architecture* for mixed-time-criticality systems, (3) a *formal verification* method to guarantee the performance of the real-time applications. Preferably, (4) each embedded system is an instance of a general platform template that can be generated and performance verified *automatically*. (5) Formal verification should be performed independently per real-time application.

In this introduction, we first describe how the existing Composable and Predictable Multi-Processor System-on-Chip (COMPSOC) platform and CompSOC + SDF3 design flow already address Requirements 1, 2, and 4, but 3 only partially. We provide a detailed overview of our novel contributions in Section 1.2. In the rest of the paper, we address Requirements 3 and 5, by defining a *dataflow formalisation to independently model real-time applications executing on a Composable and Predictable CompSOC platform*. This formalisation enables independent performance verification of real-time applications.

1.1. The CompSOC approach

Regarding Requirement 1, the CompSOC platform [1] uses the *dataflow model of computation as the application programming model*, illustrated in Fig. 1(a, left). Cyclo-Static Dataflow (CSDF) [2] offers a good trade off between expressiveness and analysability (Requirement 1). Absence of deadlock, guaranteed minimum throughput, and guaranteed maximum latency can be computed for CSDF programs, even in the presence of cyclic dependencies.

* Corresponding author.

http://dx.doi.org/10.1016/j.sysarc.2015.04.001
1383-7621/© 2015 Elsevier B.V. All rights reserved.

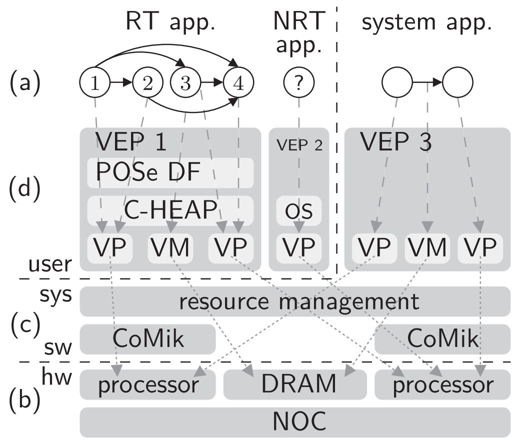


Fig. 1. CompSOC high-level overview.

Regarding Requirement 2, the CompSOC platform consists of hardware resources: processors, distributed shared memories, Direct Memory Access (DMA) modules, and Network on Chip (NOC), see Fig. 1(b) in multiple asynchronous clock domains (i.e. a GALS system). The processors execute system software: microkernel, inter-processor communication library, and dataflow library. To execute applications within guaranteed timing bounds, all hardware and software *components of the platform are individually predictable in their timing behaviour*, i.e. it is possible to compute a Worst-Case Execution Time (WCET).

Resources are shared within and between applications for cost reasons. A resource management library is used to divide physical resources in smaller virtual resources, by allocating resource *budgets*. Every application executes within its dedicated *Virtual Execution Platform (VEP)*, i.e. a set of resource budgets, cf. Fig. 1(d). Only the system application has access to the resource management library and creates and manages VEPs in a safe and controlled manner at run time, e.g. for the real-time and non-real-time applications in Fig. 1(a). A VEP isolates an application from the behaviour of other applications, i.e. *the execution of an application is composable*.

The VEPs are cycle-accurately composable, which means that they do not interfere by even a single cycle. Mixed-time-criticality applications that execute on separate VEPs can therefore be composed on the same physical resources with no interference, satisfying Requirement 2. Composable execution facilitates independent verification of real-time applications when sharing resources with mixed-time-criticality applications. The CompSOC platform's virtualised resources not only have a guaranteed budget quantity but are also guaranteed when they will receive it, enabling real-time applications to share resources with non real-time applications without any timing interference. Real-time applications are verified independently for the given budget allocation of their VEP without the need to further account for interference from concurrent applications on other VEPs, satisfying Requirement 5.

It is shown in [3] that dataflow applications with schedulers that guarantee a minimum service budget within a given periodic interval have an upper bound on the finishing time of task executions. We use dataflow as an application programming model for Requirement 1, and also for Requirement 3 as the mathematical formalism to verify their timing performance. Dataflow is used to capture the application graph as well as the mapping of the application on the platform (i.e. the application's VEP). However, our previous formalisations [4,5] do not include the effects of GALS and system software, and do not consider verification independently per application.

Regarding Requirement 4, platform instances are generated with automated CompSOC tools, directed by the system designer [6]. Applications can be manually mapped to the platform, or alternatively an automated tool such as those described in [7] can be used to map the application to the hardware. Given a dataflow model of the mapped application, the SDF3 tool [8] (or similar dataflow analysis tool) can be used to compute the guaranteed throughput and latency.

1.2. Contributions

The main contribution of this paper is a *dataflow formalisation to independently model concurrent real-time applications executing on a mixed-time-criticality platform*. Regarding Requirement 3, the existing CompSOC hardware and software platform, and the dataflow models of its components are extended in the following important ways. First, we model and quantify the effect of GALS on the application performance. Second, all system software is modelled for the first time. In particular, the intra-application and inter-application scheduling using a microkernel on the processors. The software inter-processor communication using Distributed Shared Memory (DSM) and the dataflow execution library are also modelled. We introduce an algorithm to automatically combine dataflow models of all components into a combined system model. For Requirement 5, we show that the use of Virtual Execution Platform (VEPs) leads to independent verification per application.

In the remainder of this paper, we first introduce the dataflow formalism in Section 2. In Sections 3–7, we introduce the components of the CompSOC platform with their dataflow models. In Section 8, we show how to automatically combine the models of each of the components of earlier sections in a system model of an application mapped to its VEP. In Section 9, we experimentally quantify the tightness of the dataflow models of several applications. We also quantify how the degree of synchronisation of the processors due to GALS affects the tightness. Section 10 discusses related work.

2. Dataflow formalism and modelling

Dataflow formalism, in combination with a worst-case analysis, enables the computation of guaranteed throughput and latency, and proof of absence of deadlock. The dataflow formalism exists in many variants. Homogeneous Synchronous Dataflow (HSDF) is the least expressive of these variants. It is therefore possible to directly represent HSDF models in more expressive dataflow variants such as Cyclo-Static Dataflow (CSDF) [2].

In this work, we demonstrate how an application and CompSOC platform instance are modelled using HSDF. Our technique is directly applicable to dataflow variants, such as CSDF, that can be translated to a timing equivalent HSDF [9,10]. While it is possible to create a timing equivalent Homogeneous Synchronous Dataflow Graph (HSDFG) of a CSDF Graph (CSDFG), the conversion process can lead to a much larger graph than the original CSDFG, however the degree of expansion depends on the CSDFG being translated. The causes and techniques to minimise CSDFG to HSDFG expansion are outside of the scope of this article. Nevertheless, by modelling the CompSOC platform hardware using HSDF models, our models can also be combined with applications modelled using more expressive dataflow variants (A detailed explanation of how this would be achieved is left as future work).

2.1. Dataflow formalism

A HSDFG has *actors* represented by the graph vertices and First In First Out (FIFO) communication channels between actors represented by directed edges. Formally, an HSDFG G is represented using the tuple (V, E, t, d). V is the finite set of annotated vertices. The vertices are dataflow actors and can represent tasks of a dataflow modelled application or other timing delays. E is the finite set of annotated directed edges that connect the vertices. An edge is represented by the tuple $(i, j) \in E$ where $i \in V$ is the actor *producing* tokens on the edge and $j \in V$ is the actor *consuming* tokens from the edge. The execution time of an actor i is given by $t(i)$, with $t : V \to \mathbb{R}^+$. The initial token occupancy of an edge (i, j) is given by $d(i, j)$, with $d : E \to \mathbb{N}$.

In simple terms, data is communicated along channels, represented by the edges of the graph, in atomic units of *tokens*. The initial placement of a token in a dataflow graph is represented as a black circle on an edge. Edges have an infinite token capacity, meaning that the number of tokens on an edge does not inhibit the production of more tokens on that edge. HSDF actors require a single token on each incoming edge before they are able to *fire*, i.e. the actor has the data required for execution. In a Self-Timed Schedule (STS) actors fire as soon as they are able. When an actor fires, a token is atomically consumed from each incoming edge. Upon completing execution, one token is produced on each outgoing edge. CSDF extends HSDF with the ability to consume or produce zero or more tokens on each edge according to a cyclic schedule. This makes it significantly easier to capture application behaviour.

Each actor has an execution time. The (reciprocal of the) throughput of a HSDFG is computed by finding the critical cycle in the graph. This is derived by calculating the sum of actor execution times divided by the number of initial tokens on that path (i.e. pipeline depth) for all cycles in the graph, and then taking the largest of these values. This Maximum Cycle Mean (MCM) and similar analyses [11] are implemented by dataflow analysis tools such as SDF3 [8].

The execution time of different firings of an actor may vary due to data-dependent behaviour. However, for real-time applications, we want to predict the worst-case (minimum) throughput of the HSDF, that is possible for the actual actor execution times. We achieve this by performing a worst-case analysis. The application's HSDFG is annotated with each actor's WCET, enabling the application's worst-case throughput to be calculated using a so-called Worst-Case Self-Timed Schedule (WCSTS) analysis [12]. The throughput calculated for the WCSTS is guaranteed to be conservative in comparison to the STS of the application's actual execution, i.e. the application's actual throughput can only be better than or equal to the throughput derived from the WCSTS analysis. This is due to the monotonicity of dataflow execution under an STS [12], where a later actor finishing cannot lead to an earlier actor firing, and conversely an earlier actor finishing cannot lead to a later actor firing.

2.2. Modelling CompSOC

We use HSDF to model an application and its mapping, i.e. its VEP, to a CompSOC platform instance, containing multiple shared hardware resources with system software. A dataflow graph is executed on the basis of token availability only: actors fire as soon as they have a token on each input. However, mapping a dataflow application on a real platform introduces resource constraints.

For example, when two actors use the same (single-threaded) resource, such as a processor, then only one executes at any point in time, even when both are enabled in the application graph. The application HSDFG does not capture this behaviour, and must be adapted by introducing new actors and edges, as defined in Section 4. Resource virtualisation and scheduling pose similar problems.

Similarly, the application's communication must also be mapped to platform resources, as illustrated in Fig. 1. A simple edge in the application graph is implemented with a software FIFO communication protocol that uses the processor tile's local memories, DMA, NOC, and Static Random Access Memory (SRAM) or Dynamic Random Access Memory (DRAM) memory tiles. Each resource is modelled by additional actors and edges in the application HSDFG, as explained in Sections 5–7.

Our strategy is therefore to encode all constraints in the HSDFG model such that it models the application, all hardware and software platform components, and the VEP, i.e. allocated resource budgets. In Section 8 we define an algorithm that uses the HSDFG of each component to generate a combined system model for analysis. The final model only depends on the application and its VEP, and thus the performance analysis of the application is independent of other applications.

Our performance analysis computes the minimum guaranteed throughput using the WCSTS. Since the CompSOC platform executes dataflow applications using a STS, this is conservative with respect to any execution of the application on the platform, as discussed above.

3. CompSOC hardware platform overview

CompSOC offers its *predictability* in several stages. First, all resources are predictable, i.e. offer service with a known WCET. Second, all shared resources are scheduled with predictable arbiters. Third, the behaviour of each actor mapped to a resource is characterised by a HSDFG. For compositional performance verification, this subgraph must be independent of other resources. Finally, the subgraphs are then combined in one HSDFG describing the performance of the application in its VEP, which is then verified. Since the VEP is defined by resource budgets for the application only, verification is independent of other applications.

Fig. 2 presents an example CompSOC platform with three processor tiles, one memory tile with DRAM, and a Thin-Film Transistor (TFT) tile. Each tile and the NOC can operate in their own clock domain, with Clock Domain Crossings CDCs used to communicate between domains (DMAs and cmems function as CDCs). In the following sections, we introduce the processor tiles, memory tiles, and NOC, followed by the C-HEAP communication library, and the POSe dataflow library, each with their corresponding dataflow model.

4. Processor tile

As illustrated in Fig. 2, processor tiles contain a Microblaze processor, configured as a single 5-stage in-order pipeline to increase its predictability. Branch prediction is disabled because the branch predictor state carries over from the execution of one actor (or application) to the next, influencing the latter's execution time. This complicates WCET calculations, but worse, invalidates composability, i.e. interference-free execution of multiple applications. The processor has an instruction memory (imem) and a data memory (dmem) that are tightly coupled, i.e. have single-cycle access via an Instruction/Data Local Memory Bus (LMB).

Since real-time applications can share the processor with non-real-time applications that may not have a WCET, the Composable and Predictable Microkernel (CoMik) [13] preemptively divides the processor into Time Division Multiplexed (TDM) time slices (service units) that are allocated to virtual processors. Each virtual processor can perform scheduling [14] and power management [15], receive interrupts and set timed interrupt events, *independently of other virtual processors*. To achieve this, we use a hardware Timer, Interrupt, and Frequency Unit

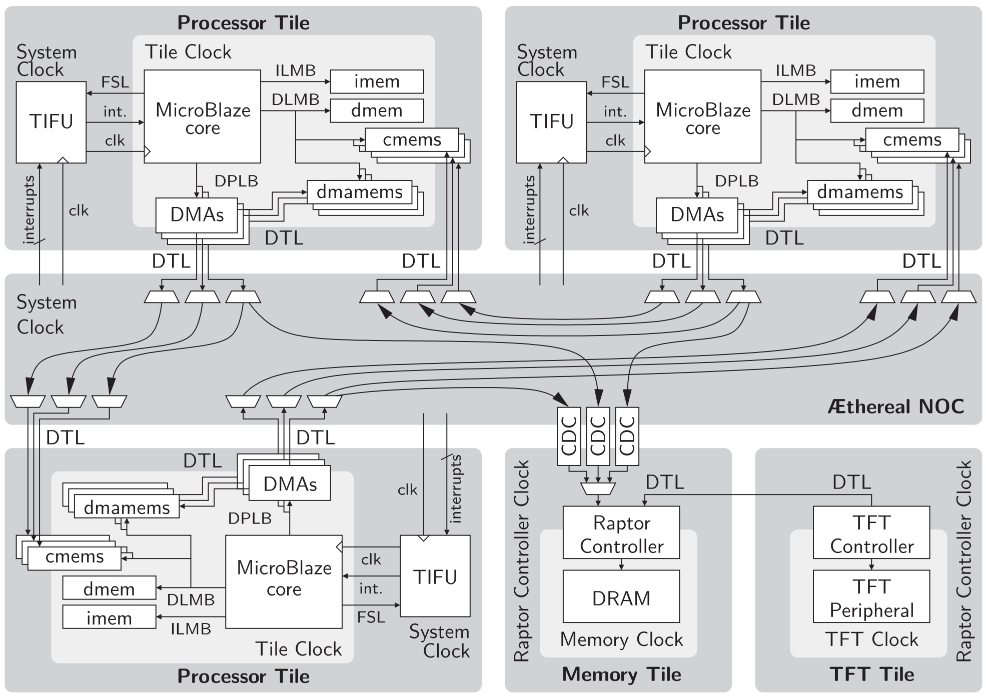


Fig. 2. CompSOC platform overview.

(TIFU) that offers a “HALT until deadline” instruction [16], timers, timed interrupts, and exceptions per virtual processor, as well as for the CoMik microkernel. The TIFU (accessed via a Fast Simplex Link (FSL) connection) also allows run-time scaling of the frequency of the processor and instruction and data memories at run time. The DMAs (accessed via a DATA Processor Local Bus (PLB)) with memories (dmamem) for incoming and/or outgoing data (accessed from DMAs via Device Transaction Level (DTL) connection), and communication memories (cmem) for incoming data (accessed from NOC via DTL connection) are discussed in detail in Section 6.3.

Without virtualisation by CoMik, a single actor in the dataflow application executing on a physical processor is modelled by a single actor in the HSDFG model annotated with its WCET. When an actor fires (details follow in Section 7) it has all its input tokens and space for its output token. Since it only uses the instruction and data memories that are tightly coupled to the processor, execution does not stall on shared resources, I/O, etc. As a result, the computation of the WCET therefore only depends on the actor code and the processor, which is simple for commercial WCET tools [17] to compute. (Coherent) caches are not used in CompSOC because the WCET of an actor would then depend on multiple resources in the system, namely processor, NOC, and memory tile, which is not compositional.

In the application dataflow graph it is possible that the same actor can fire multiple times at the same time instant. This auto-concurrency is not permitted in an implementation, since even if the actor implementation is reentrant, a processor can only execute one instruction at the same time. The real behaviour is easily modelled by introducing an edge from every actor to itself, with an initial token, which forces at most one firing of that actor at any point in time.

4.1. Intra-application processor sharing

In this paper, scheduling of actors within a real-time application is assumed to be cooperative. Hence, only one actor belonging to an application can execute per processor at any given time.

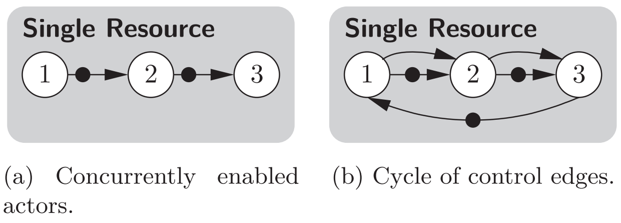


Fig. 3. Single-resource-constraint control edges.

Under a STS, actors start firing as soon as they have sufficient tokens to do so, but multiple actors on the same processor may concurrently have sufficient tokens to fire. Fig. 3(a) illustrates a single resource on which all three actors are (incorrectly) able to fire concurrently. The shared-resource constraint is therefore modelled using additional (so-called) control edges in the HSDFG. Fig. 3(b) shows the updated graph that ensures that actors fire one at a time, in a Static-Order Schedule (SOS), starting with the actor that has the initial token on its incoming control edge (actor 1 in the example).

4.2. CoMik for virtual processors

When an actor executes on a virtual processor it receives only part of the processor capacity. CoMik [18,13] uses TDM arbitration to divide the processor into fixed duration TDM slots with each slot further divided into a CoMik slot followed by a Virtual Processor (VP) slot. VP’s are allocated dedicated slots in the TDM table. There is no upper bound on the number of slots that COMik’s TDM table can have as it is maintained in software, but the number of TDM slots is statically dimensioned at design time and set during platform initialisation. The number of slots in the TDM table provides the upper bound on the number of concurrent VPs on a processor and hence the number of concurrent applications also.

CoMik’s TDM arbitration can be conservatively modelled as a latency-rate server [19] that can be represented as a HSDFG of two actors [20], see Fig. 4. The R actor represents the conservatively sustainable rate of the TDM table while the L actor represents the latency before the rate R is conservative. Rate $R = S/T$, where S is the number of cycles of service received in a single table length divided by the number of cycles for a single table length T. Latency $L = \bar{r} - R^{-1}$, with $\bar{r} = T - S + 1$ is the TDM table's Worst-Case Response Time (WCRT). An actor a that has a WCET t on a physical processor, its WCET on a virtual processor is modelled by a latency-rate graph with unchanged latency L and updated $R_a^{-1} = t \times R^{-1}$.

For example, in the CompSOC system of Section 9.1, a CoMik slot has 4,096 cycles and a virtual processor slot 65,536 cycles. A virtual processor receiving five out of ten TDM slots has a sustainable rate R of 8 virtual cycles for every 17 cycles of the physical processor (1). This rate is conservative after a latency of $L = 368,639$ cycles (2). If actor a's WCET is 5000 cycles, then its virtualised WCET is $R_a^{-1} = 10,625$ cycles (3).

$$R = \frac{5 \times 65,536}{10 \times (65,536 + 4096)} = \frac{8}{17} \tag{1}$$

$$L = 10 \times (65,536 + 4096) - 5 \times 65,536 + 1 - \frac{17}{8} = 368,638.875 \tag{2}$$

$$R_a^{-1} = 5000 \times \frac{17}{8} = 10,625 \tag{3}$$

4.2.1. HSDF application virtualisation

We now combine the above techniques, in a novel manner, to map multiple actors of a single application on a single virtual processor. CoMik's TDM table length and virtual processor slot allocation are configured (possibly differently) per core. The CompSOC platform is also a GALS system, and as such, the clock on each processing tile, and therefore CoMik's TDM tables cannot be assumed to be synchronised. All these, coupled with dynamic variations in actor execution time, mean that data can arrive and enable an actor to fire at any moment within CoMik's TDM table.

Fig. 5 illustrates how the HSDF modelled latency-rate server is incorporated into the SOS, to express that each actor could experience the WCRT of the virtual processor's allocation in CoMik's TDM table. Each application actor v is split into a latency and rate actor, as defined in Section 4.2. The incoming edges of the actors are connected to their respective latency actors that represent the worst-case duration before the task is processed at the sustainable rate of the virtual processor, once the task is enabled to fire. The outgoing edges from the task actor v are connected to the their respective rate actors. The self edges of the rate actors, are optional here, as the SOS control edges already prevent their auto-concurrency.

4.3. GALS multi-core CoMik TDM alignment

When the model that we present in Section 4.2.1 is annotated with the calculated latency and rate values derived from Section 4.2, it enables the derivation of conservative timing bounds assuming inter-core communication always experiences the WCRT of CoMik's TDM table on the receiving core, i.e the communicated

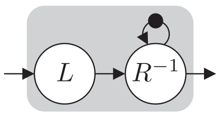


Fig. 4. HSDFG representation of a latency-rate server, used to model virtual-processor, NOC, and DRAM service.

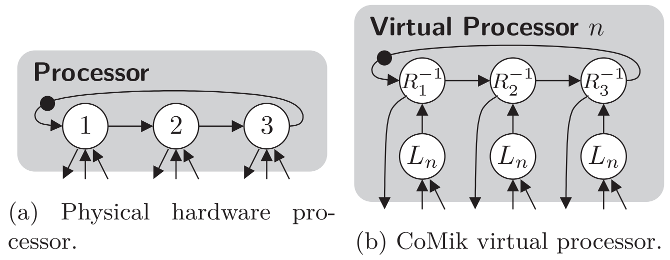


Fig. 5. HSDFGs of a three-actor SOS on a physical and CoMik virtual processor.

data always has a Worst-Case Arrival (WCA) time. While this can be the case, it is an over pessimistic assumption in many instances. This pessimism can be reduced for GALS systems in which the TDM tables are symmetrically dimensioned and the alignment (phase difference) of the TDM tables is temporally bounded. These bounds can be achieved using GALS clock tree techniques, such as those presented in [21].

In a multi-core system, CoMik TDM tables are symmetrically dimensioned when for each core (1) the CoMik slot has the same duration (2) the VP slot has the same duration, (3) the TDM tables have the same number of slots, and (4) the application is allocated the same slots in each table. If the TDM tables are also Fully Aligned (FA), then the latency of the TDM table's latency-rate timing abstraction only needs to be taken into account once for the rate to be conservative for the rest of the execution. The FA case is the same as having a bounded alignment of zero. If the bounded alignment is greater than zero, then the TDM table's latency minus the alignment bound is taken into account once, and the VP latency actors, L_n in Fig. 5(b), are annotated with a duration equal to the alignment bound. If the alignment bound equals the TDM table's latency then the analysis is therefore the same as for the WCA assumption.

We provide an experimental analysis of the conservativeness and accuracy of modifying the annotated timing of the VP's latency component of its latency-rate server abstraction in Section 9. Our modification improves the tightness of the prediction and is not required by our technique to produce a conservative timing analysis. We leave a formal proof of the conservativeness of our timing modification as future work.

5. Memory tiles

Memory tiles support both SRAM [4], and DRAM memories [22–24]. As illustrated in Fig. 2, each requestor for a memory has a connection to the memory tile. Since transactions of a requestor sent to memory tiles may be of any size and may arrive at any rate, so-called atomisers chop transactions into smaller finite-size transactions called atoms. These atoms are arbitrated using round robin, TDM, or Credit Controlled Static Priority (CCSP) [25] arbitration. Each atom is then executed non-preemptively with a guaranteed WCET by the memory [26]. SRAM tiles offer single-cycle read and write access, and DRAM tiles offer fixed-size read and write access, implemented with DRAM patterns (of about 30 cycles for a 64 B access to the DDR3 memory on our FPGA). (If TDM is not used, then composability requires a delay block that delays each response to its WCRT [27,28]. Since this does not affect predictability, it will be ignored in the remainder.) Responses for a single original transaction are coalesced on return.

When a dataflow token is stored in a memory tile, then it will be copied there by a tile DMA (see next section). As a result, all transactions arriving at a memory tile have a known size and the effect of atomisers can be taken into account in the memory tiles dataflow model. Here, we abstract the guaranteed timing behaviour of each requestor at a memory tile by a latency-rate server modelled as an HSDFG, with latency and rate values as computed in [28].

6. Communication with NOC and C-HEAP

In this section, we describe the NOC hardware, how to model finite-capacity channels, and the C-HEAP software communication protocol.

6.1. Network on Chip

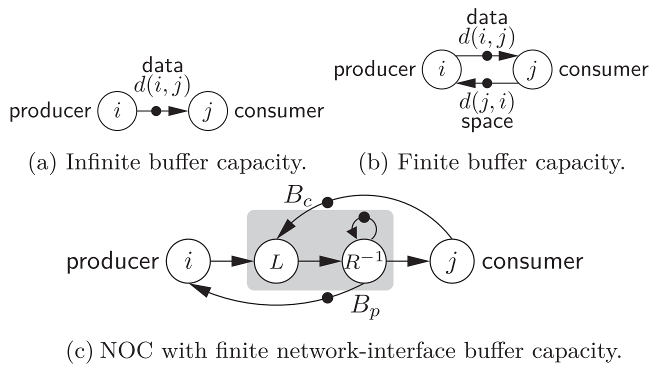
The Æthereal Network on Chip (NOC) [29,30] connects the tiles. It provides virtual point-to-point connections that have predictable upper bounds on throughput and latency. The connections time share the hardware along their path using TDM arbitration, configured to avoid all contention. Transaction requests and responses use independent connections. Since as for memories, incoming transactions may be infinitely long or stall halfway, the NOC network interfaces preempt incoming data into service units (flits) of three words. Each connection in the NOC can be modelled as a HSDFG at various levels of abstraction [5]. Below, we use a latency-rate model for a NOC connection, with latency and rate computed as defined in [5].

6.2. Finite FIFO capacity

Application actors communicate using FIFO channels (Fig. 6(a)) with infinite capacity. However, after mapping on resources, i.e. in a Virtual Execution Platform (VEP), their capacity is finite. In implementation, tokens in memory have a static finite size and the FIFO buffer is an array with capacity for a finite number of tokens, as described in Section 6.3. A FIFO buffer with finite capacity is modelled as a HSDFG with two opposing edges. One edge represents data transfer between the two actors, and the other the transfer of free space in the buffer, as shown in Fig. 6(b). Thus, the effect on throughput of a producer stalling on a slower consumer, or vice versa, i.e. back pressure or flow control, is modelled correctly. Fig. 6(c) illustrates two buffers of B_p and B_c tokens between a producer (e.g. processor tile), NOC modelled with a latency-rate HSDFG, and a consumer (e.g. another processor tile), respectively. Note that individual components (e.g. NOC, memory tile) are modelled with a latency-rate abstraction that does not incorporate back pressure – this is handled by the combined HSDFG instead.

6.3. C-HEAP FIFO communication

Finite-capacity FIFOs are used for streaming peer-to-peer communication, e.g. for requests or responses of a transaction. When an actor sends a token to another actor of the same application,



(a) Infinite buffer capacity. (b) Finite buffer capacity.

(c) NOC with finite network-interface buffer capacity.

Fig. 6. HSDFGs for FIFO and NOC.

multiple transactions are required. The C-HEAP [31] library implements this higher-level protocol for multiple processors with distributed shared memories, cf. Fig. 2. Actors on the same processor tile communicate using the local data memory (dmem). When producer and consumer actors reside on different tiles, e.g. actors 1 and 3 in Fig. 1, the producer writes the token in a dmamem and instructs the DMA to write the data over the NOC in a cmem of the consumer tile. The processor tile's local memories (dmem, cmem, dmamem) are fast since they are tightly coupled to the processor, but also relatively small. Larger on-chip SRAM or off-chip DRAM remote shared memories can also be used, in which case the producer DMA writes data into the remote memory, and the consumer DMA reads the token from the remote memory.

C-HEAP safely synchronises access to shared data in the distributed shared memories of the CompSOC platform. Resources are never locked with semaphores or otherwise, since this invalidates composability: the execution time of one actor would depend on the behaviour of another actor, rather than only on its allocated budget. Instead each C-HEAP channel uses a Write Counter (WC) and a Read Counter (RC) in a shared memory that indicate the position to which data has been written and read, respectively, in a circular FIFO. Actors poll the WC and RC to compute the availability of data or space in the FIFO before either writing data or reading data. After this, they either increment the WC (producer only) or RC (consumer only). Since memory accesses to a single word are atomic, no locking is therefore required. For best performance, only posted writes over NOC are used, with a local copy of the counters. Fig. 7(a) and (b) illustrate this for communication without and with shared remote memory, respectively, which are the most common channel mappings.

Fig. 7(a) is modelled by the novel HSDFG model shown in Fig. 7(c). The actors in the HSDFG are marked with the transaction enumeration from Fig. 7(a) to show which transaction timings they represent, with some actors representing the timing of multiple transactions.

The C-HEAP FIFO is initially empty with its buffer capacity represented by B initial tokens in Fig. 7(c). The producing task checks for space in the buffer by comparing RC and local WC. The producing actor can subsequently fire if there is space in the FIFO and also enough space in its local output buffer of capacity B_p. Once the producing task has completed, DMA transactions 1 & 2 write the data and updated WC into the consuming tile's cmem. The timing of these transactions is represented by two dataflow actors. The first actor models the time taken by the DMA to write the data followed by the WC onto the NOC. The second actor models the time taken for the last word of transactions 1 & 2 to be transported by the NOC and written into the consumer cmem.

The consuming actor is enabled to fire whenever it observes the presence of the data in the buffer by comparing its local RC and WC. After its firing has completed, it updates the local RC and performs transaction 3 to release the space. The timing of transaction 3 is modelled using two dataflow actors, which is similar to what was described for the producer. The producing task is now able to observe this space by comparing its RC and local WC.

Fig. 7(b) and (d) illustrate the same process when a SRAM or DRAM memory tile is used. The HSDFGs capture that producer, consumer, their DMAs and NOC operate in a parallel and pipelined manner.

7. POSe dataflow execution library

The hardware and system software components described up to this point are sufficient to write applications consisting of communicating tasks running on multiple shared processors, using distributed shared memories, including C-HEAP applications [31],

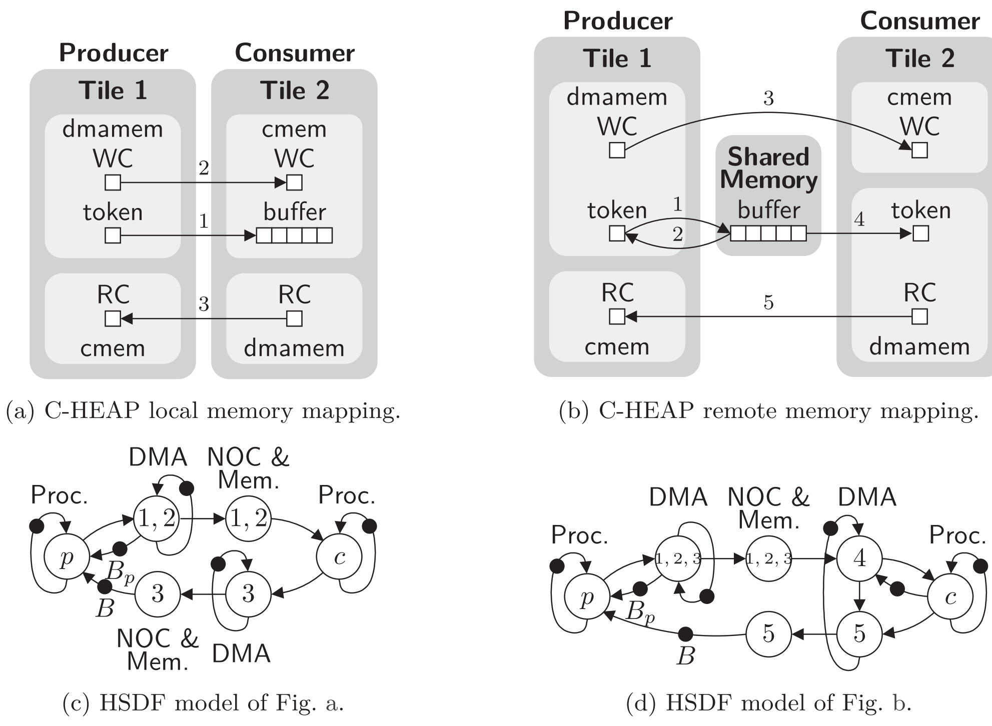

Producer
Consumer
Tile 1
Tile 2
dmamem
WC
token
cmem
WC
buffer
RC
cmem
RC
dmamem
Shared Memory
(a) C-HEAP local memory mapping.
(b) C-HEAP remote memory mapping.
DMA
NOC & Mem.
Proc.
(c) HSDF model of Fig. a.
(d) HSDF model of Fig. b.

Fig. 7. Inter-tile C-HEAP FIFO communication using local memories only (a, c), and with local and remote memories (b, d).

Kahn process networks [32], dataflow applications, or time-triggered applications [14].

The POSe library simplifies writing dataflow modelled applications (HSDF/Synchronous Dataflow (SDF)/CSDF). The behaviour of an actor is defined using a single function written in the C language. Input/output tokens are function inputs by reference. The function must be pure, i.e. is not stateful and has no side effects, such as using static or volatile variables, or pointers to shared state. The application programmer defines a C function for each dataflow actor, and defines the firing rules and mapping of the actors. As indicated by our novel dataflow model in Fig. 8, POSe wraps each C function with standard functionality required for every dataflow actor: (1) checking the actor's firing rule; (2) reading the input tokens into local memory; (3) firing the actor, i.e. executing the C function on the input tokens, producing output tokens; (4) writing the output tokens into appropriate memories.

In Step 1, for each input and output channel of the actor, C-HEAP is used to check that input tokens are available, and space is available for output tokens. Input tokens may reside in dmem (local producer), cmem or remote shared memory (remote producer). In Step 2, they are copied to local dmamem by the DMA, before they can be accessed by the C function. The C function fires in Step 3, producing any output tokens either in its dmem or dmamem. In Step 4, if required, the DMA copies the tokens to a remote consumer tile's cmem or shared remote memory. Recall that the use of DMAs enforces composability by ensuring that the WCET of the (wrapped) C function only depends on the processor and its local memories.

Each actor in the application graph is expanded to the HSDFG of Fig. 8, which captures the cost of inter-actor communication in a physical tile. Sharing the processor with other actors (of same or different applications) must be modelled additionally, using techniques presented in Section 4. The next section explains how this is done.

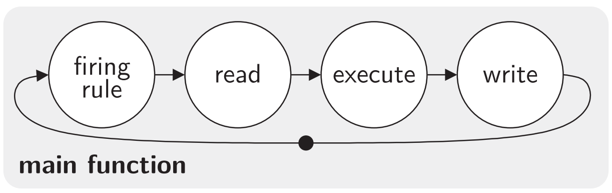


Fig. 8. Actors and POSe dataflow library.

8. Modelling applications mapped on a platform

In the previous sections, we described the components required to execute dataflow applications, as well as their individual models. We now combine all pieces to *translate a dataflow application mapped on a platform to a dataflow graph that models both the timing of the application and the timing of the hardware and software components of the CompSOC platform*. Each application has its own HSDFG that can be analysed independently.

Next, we explain our novel automated method to translate the original application dataflow graph by modifying and adding dataflow actors and edges, to take into account dependencies and delays due to (8.1) mapping, (8.2) C-HEAP communication (incl. DMAs, NOC, and memory tiles), (8.3) POSe dataflow library, and (8.4) CoMik processor virtualisation using TDM.

8.1. Incorporating application mapping

The starting point of our technique is a mapped dataflow application, produced by SDF3 or equivalent tools. Each actor of a dataflow application is mapped on a processor, assuming a deadlock-free SOS per processor. An example mapping of the dataflow application of Fig. 9(a) is shown in Fig. 9(b). Actors 1 and 3 are mapped to the processor on tile 1, and actors 2 and 4 to the processor on tile 2. Actors are scheduled with per-tile SOS which is captured with the additional control edges, cf. Section 4.1.

8.2. Incorporating C-HEAP communication

The application mapping of Fig. 9(b) has two FIFOs between actors on the same processor. These use C-HEAP, but require no additional hardware resources. Also, since they contain only one token, the reverse edges modelling their capacity may be omitted. The two inter-tile C-HEAP FIFO communication channels do require hardware resources: one or two DMAs, the NOC, and possibly a memory tile (cf. Section 6.3). The capacity of each FIFO is modelled using a reverse edge with the number of initial tokens equal to its capacity. The HSDFG of the FIFO depends on the memories on which the read and Write Counter and data are mapped, as discussed in Section 6.3. Fig. 9(c) illustrates the HSDFG for the example application.

Algorithm 1 incorporates the C-HEAP models from Section 6.3 into the mapped-application HSDFG. For example, applying it to HSDFG of Fig. 9(b) results in Fig. 9(c). The algorithm iterates over all of the C-HEAP edge pairs $\{(p,c),(c,p)\} \subseteq E$ that represent the forward data path and the reverse space path of a single C-HEAP FIFO from producer p to consumer c, replacing them with the appropriate C-HEAP HSDFG, which is Fig. 7(c) in this case. getCHEAPHSDFG: $E \times E \rightarrow G$ returns the appropriate HSDFG representation of the C-HEAP FIFO for the C-HEAP edge pair.

Algorithm 1. Incorporate Inter-Tile C-HEAP

```
Require: input HSDFG $G$ (e.g. Fig. 9(b))
  for all C-HEAP edge pairs $\{(p,c),(c,p)\} \subseteq E$ do
    $E \leftarrow E \setminus \{(p,c),(c,p)\}$
    $G \leftarrow G \cup \text{getCHEAPHSDFG}(\{(p,c),(c,p)\})$
  end for
  for all processors $P$ do
    $V_p \leftarrow \text{getActors}(G,P)$
    for all DMAs $D$ local to $P$ do
      $V_d \leftarrow \text{getActors}(G,D)$
      $G \leftarrow \text{createSOS}(G,V_d)$
      $G \leftarrow \text{orderSOSactors}(G,V_p,V_d)$
    end for
  end for
  return HSDFG $G$ (e.g. Fig. 9(c))
```

C-HEAP FIFOs can share a single DMA, requiring a cycle of control edges to ensure sequential firing of all DMA actors. To add these edges, Algorithm 1 iterates over all the DMAs of all the processors, finding all the actors that belong to each DMA, before forming a single SOS per DMA and ordering the actors relative to the SOS of the task actors on the local processor. The function getActors: $G \times H \rightarrow V$ takes a graph G and a hardware resource H and returns the set of actors V from G that represent the timing of H. It is used to find the set of actors V_p that represent the processor P and the set of actors V_d that represent the DMA D. createSOS: $G \times V \rightarrow G$ takes an HSDFG G and a subset of actors V and returns a graph G with a cycle of control edges to the actors V in G, forming a SOS. orderSOSactors: $G \times V \times V \rightarrow G$ takes the graph G and two subsets of actors V, ordering the SOS of the second set of actors relative to the first. The algorithm uses this to order the SOS of the DMA actors V_d relative to the ordering of the task actors on the local processor V_p.

The result of applying Algorithm 1 to the HSDFG of Fig. 9(b) is shown in Fig. 9(c). The two inter-tile C-HEAP edge pairs are replaced with the C-HEAP HSDFG for C-HEAP communication using local scratchpad memories, cf. Section 6.3. Both C-HEAP FIFOs use the same DMA on each processing tile. Following Algorithm 1, the actors modelling DMA transactions are added to a SOS per DMA and ordered relative to the task execution order of the local processor.

8.3. Incorporating POSe dataflow library

The actors representing the processing actors of Fig. 9(c) are the same as the original actors in the application graph in Fig. 9(a). Section 7 explained how POSe implements the dataflow execution model including firing rule checking, token copying, and actor firing (C function execution). Algorithm 2 describes how the POSe dataflow library is incorporated into the combined application and CompSOC platform HSDFG.

Algorithm 2. Incorporate POSe dataflow library

```
Require input HSDFG $G$ (e.g. Fig. 9(c))
  for all processors $P$ do
    $V_p \leftarrow \text{getActors}(G,P)$
    for all actors $v \in V_p$ do
      $G \leftarrow \text{substitute}(G, v, \text{POSeTaskHSDFG}(v))$
    end for
  end for
  return HSDFG $G$ (e.g. Fig. 9(d))
```

The algorithm iterates over all actors in the HSDFG that model processor actors, substituting each of them with the POSe HSDFG. The function substitute: $G \times V \times G \rightarrow G$ takes a graph G with an actor V to be substituted with a graph G and returns the resulting graph G. The incoming edges of the substituted actor are transferred to the first actor in the actor order of the replacement graph, and the outgoing edges to the last actor. POSeTaskHSDFG: $V \rightarrow G$ takes an actor and returns the POSe HSDFG for that actor.

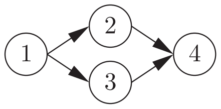
The result of applying Algorithm 2 to the graph of Fig. 9(c) is shown in Fig. 9(d). Each actor n is expanded to actors firing rule s_n, execute e_n and write w_n. No read r_n actors are present because in this example no remote shared memory (SRAM or DRAM) is used.

8.4. Incorporating CoMik microkernel

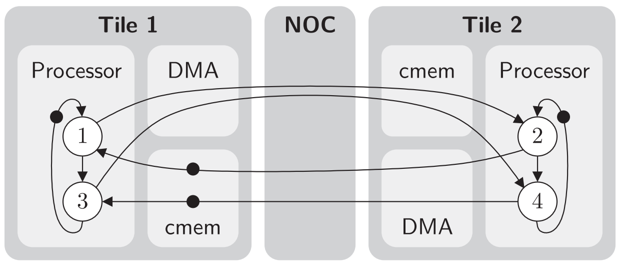
CoMik virtualises a single processor into multiple virtual processors using TDM arbitration. Section 4.2 defined how to incorporate CoMik's latency-rate server abstraction into the application HSDFG. Algorithm 3 implements this, translating e.g. Fig. 9(d) into Fig. 9(e).

Algorithm 3. Incorporate CoMik TDM Timing

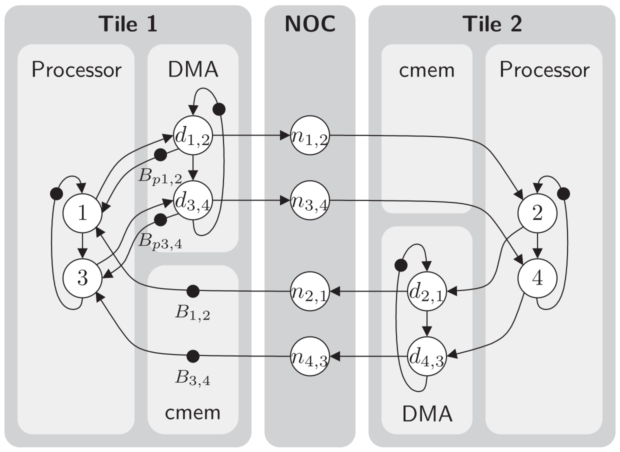
```
Require: input HSDFG $G$ (e.g. Fig. 9(d))
  for all processors $P$ do
    $V_p \leftarrow \text{getActors}(G,P)$
    for all edges $(i,j) \in E$ do
      if $i \notin V_p$ and $j \in V_p$ then
        $G \leftarrow \text{substitute}(G, j, \text{CoMikLRserver}(j))$
        $V_p \leftarrow V_p \setminus j$
      end if
  end for
    $V_p \leftarrow \text{getActors}(G,P)$
    for all actors $v \in V_p$
      $G \leftarrow \text{updateLRtiming}(G, v)$
    end for
  end for
  return HSDFG $G$ (e.g. Fig. 9(e))
```



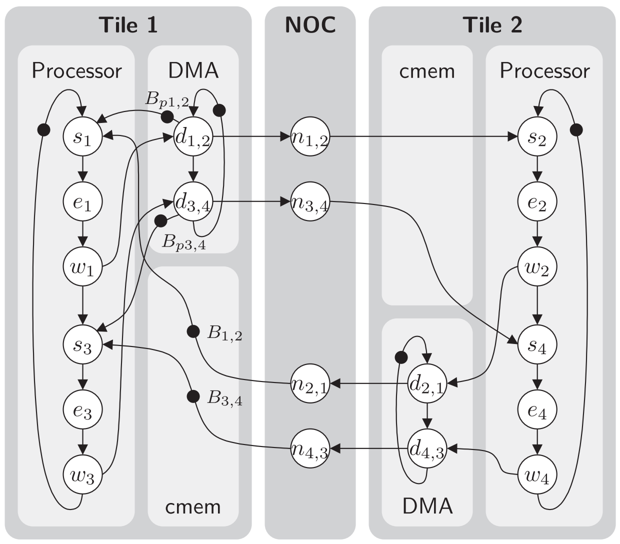
(a) Example HSDFG modelled application.



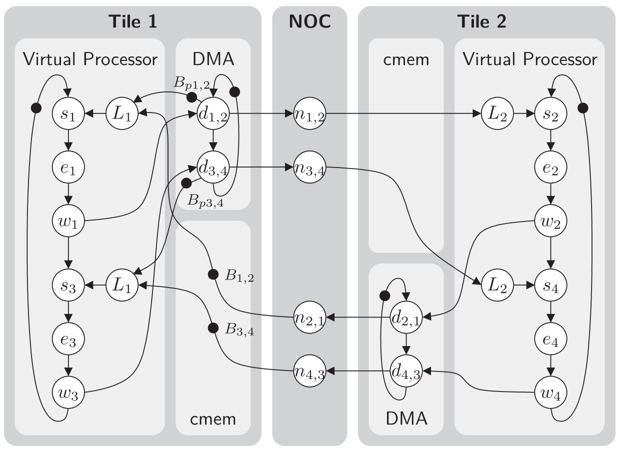
(b) Application and resource constraints mapped onto CompSOC hardware resources.



(c) Incorporating hardware timing in the application's dataflow graph.



(d) Dataflow actors replaced by POSe library HSDFG.



(e) Latency-rate abstraction of CoMik TDM table.

Fig. 9. Combined application and CompSOC platform dataflow modelling.

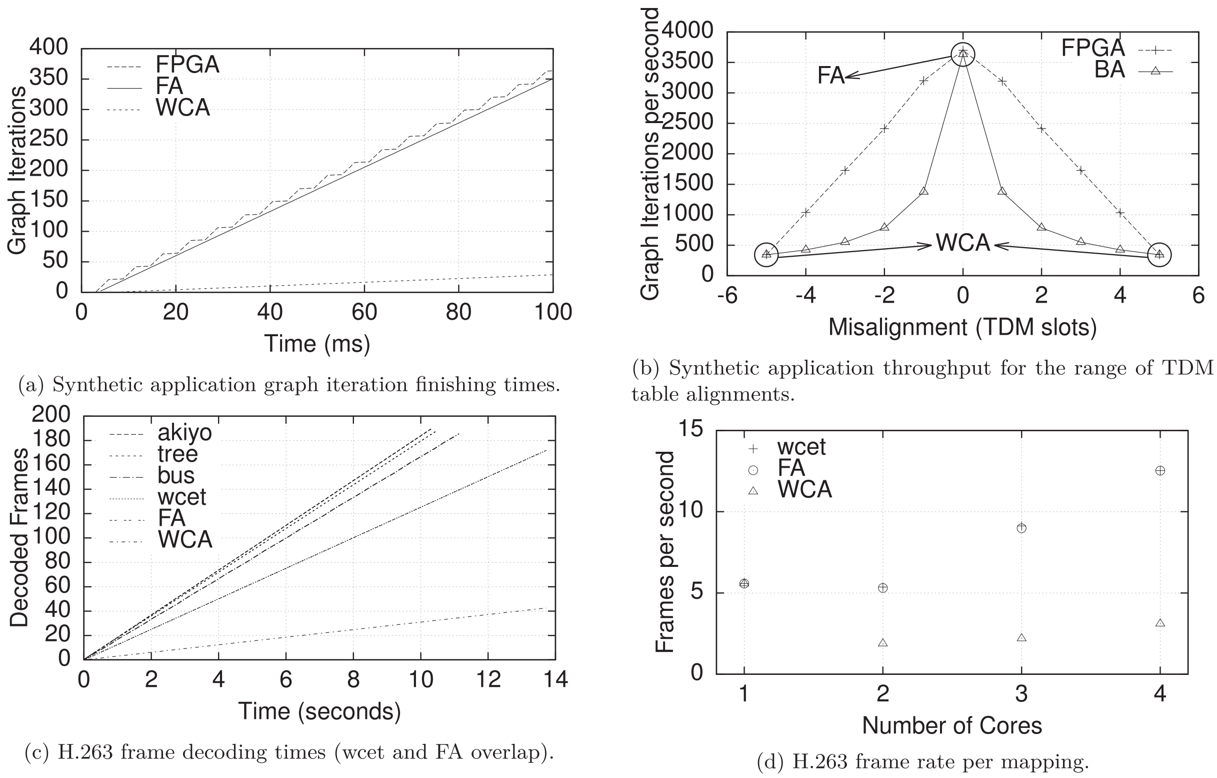
The algorithm iterates over all the edges in the graph (i, j) for every processor P. For the set of actors V_p belonging to P, if the producing actor i is not in V_p and the consuming actor j is in V_p then the edge (i, j) is an incoming inter-tile communication edge. The function CoMikLRserver: $V \rightarrow G$ takes an actor and returns the latency-rate server HSDFG for that actor with the CoMik TDM latency as described in Section 4.2. The algorithm then substitutes the latency-rate graph for j in graph G. The actor j is then removed from the set of actors V_p as the substitution only needs to occur once per consuming actor. The function updateLRtiming: $G \times V \rightarrow G$ takes the HSDFG G and an actor v and updates its annotated timing $t(v)$ to correspond with its execution time on the virtual processor R_v^{-1}, as described in Section 4.2. The latency actors of CoMik's latency-rate server abstraction, are appended to the incoming inter-tile communication edges.

Finally, Algorithms 1–3 are executed sequentially to create a HSDFG of an application executing in a VEP. This algorithm is executed independently for each dataflow application running on the CompSOC platform.

9. Experiments

Having described in the previous sections how dataflow modelled real-time streaming applications that are mapped onto a CompSOC platform are modelled as an HSDFG for timing analysis, we proceed to demonstrate the accuracy of our modelling technique. To do this, we execute applications on a Field Programmable Gate Array (FPGA) prototyped four core CompSOC platform and compare their actual timings with those predicted by timing analysis of the application's associated HSDFG. For this purpose, we use both the example application from Fig. 9(a) (which we will henceforth refer to as the synthetic application), and an H.263 decoder application with which we decode various video streams. In this section, we will show:

- our technique applied to both a synthetic and H.263 decoder application,
- the tightness of our technique using applications with tasks that constantly execute with WCETs,
- that our technique conservatively bounds the timings of GALS systems, ranging from 100% synchronous FA clock and symmetric TDM table, to systems where the alignment of TDM tables are unknown and all inter-core communications are conservatively assumed to arrive with WCA,
- that the tightness of the bounds predicted by our technique depends on the amount of knowledge of the system, i.e. the alignment of the TDM tables,



(a) Synthetic application graph iteration finishing times.

(b) Synthetic application throughput for the range of TDM table alignments.

(c) H.263 frame decoding times (wcet and FA overlap).

(d) H.263 frame rate per mapping.

Fig. 10. Results comparing measurements for the synthetic and H.263 decoder applications, with their models.

- that our technique can correctly predict trends, e.g. whether an improvement in throughput is expected when moving between mappings.

The CompSOC platform that we use for our experiments has four homogeneous processor tiles executing at 120 MHz, with local instruction, data, communication and DMA memories. Each tile has multiple DMAs, but for the purposes of composability, each DMA is only used by a single application on each tile. To simplify exposition, in our experiments, each application is allocated a single DMA per tile.

9.1. Synthetic application

We start our experimentation by comparing the actual graph iteration finishing times of the synthetic application, as measured on the FPGA prototype of the CompSOC platform, with the predicted latency and throughput of its HSDF model from Fig. 9(e). The synthetic application is structured following Fig. 9(a) with each actor representing a task and each edge a C-HEAP FIFO between the tasks. For the purposes of ascertaining the accuracy of the model for worst-case analysis, each task has a constant execution time and therefore always executes at its worst-case. Each CoMik instance is configured to have a TDM table length of ten slots with five slots allocated to the synthetic application's VP. Each TDM slot comprises of a CoMik slot and a VP slot, with durations of 4,096 and 65,536 cycles, respectively. The finishing times of the application's graph iterations, as measured from the FPGA instance of the CompSOC platform, are presented in Fig. 10(a). The synthetic application can be seen to make progress (complete graph iterations) whenever its VP is scheduled for five out of the ten iterations of the TDM table, creating the impression of "steps". The CoMik overhead between its five allocated slots also prevents application progress, but is so small that it is unnoticeable in the graph.

The predicted timings of the application's HSDF model are also presented in Fig. 10(a), and the predicted latency and rate can be seen to conservatively bound the synthetic application's actual finishing times. The effect on timing performance of the VPs' TDM allocation is captured by a latency-rate abstraction, as described in Section 4.2. The rate of execution of each VPs is calculated as 8 virtual cycles per 17 cycles of the physical processor. The timings of actors $s_{\{1,2,3,4\}}, e_{\{1,2,3,4\}}, w_{\{1,2,3,4\}}$, from Fig. 9(e), are modified to reflect the virtual rate of execution.

The VP's rate of execution is sustainable after a latency of 368639 cycles, as derived by (2) in Section 4.2. In Section 4.3, we described how knowledge about the alignment and dimensioning of the TDM table is used to reduce pessimism in the model. In the most pessimistic case, it is assumed that data communicated between cores always arrives at the WCA time. In this case, the latencies L_1 and L_2 from Fig. 9(e) are annotated with the latency of the CoMik TDM table for that tile. Alternatively, if the TDM tables are symmetrically dimensioned and FA (100% synchronous), the latency of the VP's latency-rate abstraction is conservatively taken into account as a single offset before the rate of the VP is sustainable, as described in Section 4.3, with zero delay being assigned to L_1 and L_2. The results presented in Fig. 10(a) for the FPGA implemented CompSOC platform has cycle-accurate FA CoMik TDM tables on each core. The HSDF model's prediction achieves a conservative bound for both the WCA and FA cases. The FA method achieves a tighter bound (predicting throughput to within 1.65% of the actual case) than the WCA method (predicting throughput to within 91.68%). In some GALS systems, it may not be possible to bound the TDM table alignment, or it may be desirable to use a non-symmetric CoMik TDM configuration, and for these instances, the WCA method still gives a guaranteed conservative timing bound.

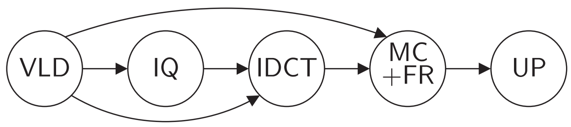


Fig. 11. HSDFG of the H.263 decoder application.

If it is possible to bound the alignment of the TDM tables, it is possible to get a less pessimistic conservative timing bound. Fig. 10(b) presents the synthetic application's measured throughput, from the CompSOC FPGA implementation, for the complete range of TDM table alignments, which is achieved by misaligning the application's TDM allocations on each core. It also presents the throughput predicted by the HSDF model when annotated with the TDM table's alignment bound. The bounded alignment (BA) annotation method uses a single VP latency offset minus the alignment bound, and annotates latencies L_1 and L_2 with the alignment bound. If the misalignment is zero, then the BA annotation matches the FA case, while if the misalignment is equal to the VP latency, then the BA annotation matches the WCA case. From Fig. 10(b) it can be seen that the predicted throughput is quite tight (to within 1.86%) when there is no misalignment, but becomes less tight as the TDM tables are more misaligned. This happens because the model assumes that every iteration of the graph is affected by the misalignment, but since multiple graph iterations can finish within a single slot, as can be seen in Fig. 10(a), while the application still has some slots that are scheduled concurrently, this is not the case. As the tables become more unaligned, more iterations of the graph are affected by the misalignment and the accuracy of the model increases again. From this, it can be seen that the WCA annotation can be tightly accurate (to within 0.4% in Fig. 10(b)) while conservatively bounding all other alignments.

9.2. H.263 decoder

We proceed to apply our technique to an H.263 decoder that has data dependent task execution times. The HSDFG of the H.263 decoder is illustrated in Fig. 11. The decoder consists of five tasks; Variable Length Decoding (VLD), Inverse Quantisation (IQ), Inverse Discrete Cosine Transform (IDCT), Motion Compensation (MC) and Frame Reconstruction (FR) (combined into a single task), and Up Scaling (UP). Fig. 10(c) presents the frame finishing times from multiple decoding runs and the predictions from the FA and WCA modelling cases. Three different videos (akiyo, bus and tree [33]) are used as input for the H.263 decoder. To demonstrate the accuracy of our technique for worst-case analysis, the wcet run executes the H.263's tasks with the constant worst-case task timings measured from the runs of the three videos.

The H.263 decoder is mapped onto all four cores of the FPGA prototyped CompSOC platform, with FA symmetric CoMik TDM tables. Each table has three slots, of which two are allocated to the H.263 decoder's VPs. Each TDM slot is configured to have a CoMik slot duration of 4096 cycles and a VP slot duration of 196,608 cycles. From Fig. 10(c), it can be seen that the FA and WCA annotated HSDF model predictions conservatively bound the three video runs and the wcet run, although it is hard to see from Fig. 10(c) that the FA annotated model conservatively bounds the wcet runs timing because the FA prediction is so tight that the wcet and FA lines appear to overlap at this scale. As the three videos (akiyo, tree and bus) generally have better than worst-case task execution times, their achieved throughput is higher than for the wcet execution and therefore also the HSDF model predictions. This is a limitation of worst-case analysis in general and is not specific to our technique. What is important, is that our technique conservatively bounds the application's worst-case execution, and it achieves this.

For our final experiment, we compare the accuracy of the H.263 decoder's HSDF model throughput prediction with the throughput achieved by the H.263 decoder executing with constant task WCET on the FPGA instance of the CompSOC platform (wcet), for the H.263 decoder mapped onto one to four cores. Apart from the application mapping, the platform is configured the same as for the previous H.263 decoder experiment. From Fig. 10(d), it can be seen that the FA annotated model that matches the platform's alignment achieves a tight conservative bound for the four mappings. The WCA also achieves a conservative bound, which is less tight but also bounds the application's throughput if the TDM table's alignment was unknown. While the WCA bound is conservative for all table alignments, Fig. 10(d) shows that for a platform with a bounded and relatively small variation in TDM, the WCA bound becomes less accurate as the number of cores increases. This is due to the pessimistic assumption that every inter-core communication arrives at the worst-case time in the TDM table, and therefore incurs a WCRT. Both the FA and WCA HSDF annotations correctly show that the two core mapping of the H.263 decoder does not offer an advantage for application throughput over the single core mapping. The FA predicted throughput also correctly shows that the three and four core mappings perform better for application throughput than the single core mapping, whereas the WCA predicts throughputs worse than the single core mapping for the three and four core mappings. Both FA and WCA correctly predicts an improvement in graph throughput moving from a two core to three core mapping, and from a three core to four core mapping.

From our experimentation, we can conclude that our technique conservatively and accurately models application worst-case timings when mapped onto multiple cores of the CompSOC platform. The accuracy of our technique is better if the alignment of symmetrically dimensioned CoMik TDM tables on the cores can be bounded. Regardless of the TDM table's alignment or dimensioning, our technique is still able to produce conservative timing predictions for application graph latency and throughput.

10. Related work

Our work touches on many aspects of design and performance verification of real-time systems and their components, and there is hence a larger body of related work than can be exhaustively discussed. To structure the discussion, this section addresses related work in two parts. The first part focuses on complete systems and system-level approaches to address the performance verification problem, while the second considers design and analysis of individual platform components.

10.1. Systems

Time-Triggered Architectures [34] (TTA) and Precision-Timed Systems (PRET) [35] are two well-known approaches to design time-predictable and composable real-time systems. TTA [34] is a well-established approach to building safety-critical embedded systems with real-time applications, popular in the avionics and automotive domains. Timing isolation is provided by only allowing interactions between components at pre-computed points in time. In terms of scheduling, it means that TTA uses non-work-conserving scheduling both within and between applications, while our processor scheduling within an application is work-conserving to improve efficiency. The PRET programming model [36] and platform [37] focus on timing repeatability, which is similar to our notion of composability, to address the verification problem of real-time systems. However, unlike our approach, they achieve this property by privatising (parts of) their hardware resources, negatively impacting scalability. We discuss this issue further in the context of their platform components in the following section.

Earlier work has considered resource budgeting/reservation and virtual platforms, similar to our VPs, to simplify verification by providing performance isolation for applications. A vision for a multi-core resource management framework based on virtual private machines was proposed in [38], although it does not concretely explain how to satisfy real-time requirements. In contrast, the ACTORS project [39] focused on adaptive resource management for virtual platforms based on resource reservations. However, the only resource that is explicitly managed is the CPU time and interference in other shared resources, such as interconnect or memory, is seen as load variations that are dealt with by adapting the CPU time allocation and application service levels. In contrast, we support static resource reservations in all shared resources and have an automated tool flow to find configurations that satisfy requirements. Other approaches based on timing isolation briefly worth mentioning are [40–43], and real-time virtual resources [44].

Many recent European research projects, such as (par) MERASA [45,46], T-CREST [47] (this issue) and PROARTIS [48] have worked on time-predictable architectures and developed both hardware components and their corresponding timing analyses. In fact, the T-CREST MPSOC uses modified versions of our NOC and memory controller. However, all these projects focus on determining the WCET of application tasks on different platforms of varying complexity using static analysis [45–47], measurement-based approached [45,46], or probabilistic analysis [48]. The obtained WCETs enable system-level analysis using traditional real-time techniques, which are well-suited for many real-time applications. In contrast, we model all real-time applications and hardware components as data-flow actors, which fits very well with real-time streaming applications that may have cyclic dependencies, and also accurately captures hardware constraints, such as finite buffers.

This article extends our earlier work [4,5,1] by including processor sharing, DRAM, atomisers, and the complete software stack in the platform and performance analysis. Closest to our work are [49,50] that also use dataflow for real-time performance analysis. These works can verify multiple real-time applications, but do not consider the presence of non-real-time applications.

10.2. Platform components

A definition of time-predictability and a survey of recent advances for building time-predictable embedded systems, considering both hardware and software components, is provided in [51]. A commonly used approach [52,53,36] for designing time-predictable hardware components is to discard the architectural features that only improve average performance.

Similarly to our processor tile, PRET's predictable processor [16,54] has timing instructions like HALT with deadline [55]. An important difference with our processor is that PRET uses a privatised hardware thread per application, making it less scalable than our use of TDM implemented with a software microkernel. Many existing microkernels offer partitioning, e.g. with the ARINC standard [56], VxWorks, OKL4 [57], PikeOS, INTEGRITY, and LynxOS-178. However, unlike the CoMik microkernel, neither of these provide cycle-accurate isolation between partitions. These microkernels hence suffice for verification of real-time applications, but they cannot offer composable verification to non-real-time applications.

Several real-time memory controllers have been proposed. Similarly to our controller, most of them use a close-page policy to reduce the worst-case response time of each memory request [58,59]. The PRET memory controller [60] combines this with memory bank privatisation per application to achieve repeatable timing. Bank privatisation is also used in [61], which combines it with an open-page policy and applies static analysis of memory addresses to guarantee locality among the memory requests of an application to reduce its WCET. Although these two works clearly demonstrate benefits with bank privatisation, the number of applications is limited by the number of banks, causing the approach to scale poorly to complex systems with many applications. Furthermore, with the exception of [60], the provided analyses focus on bounding execution times of single outstanding requests and do not efficiently deal with memory traffic generated by DMAs or hardware accelerators that are common in systems with streaming applications.

TDM-based NOCs communication have also been proposed in [62,63]. For the T-CREST platform [47] (this issue), our NOC has been improved [64] by removing end-to-end flow control at the cost of not offering streaming connections. It is also asynchronous, which improves scalability in combination with GALS. Alternative NOCs based on more expensive virtual-circuit buffering have also been proposed.

11. Conclusions

In this paper, we presented a *dataflow formalisation to independently model concurrent real-time streaming applications executing on a mixed-time-criticality platform*. We focussed on the composable and predictable CompSOC platform that contains multiple processors, NOC, and SRAM/DRAM Distributed Shared Memory (DSMs), connected in a GALS manner, executing system software, in particular a microkernel and dataflow communication libraries. Real-time streaming applications running on top of the system software are written following any dataflow paradigm that is translatable to a timing equivalent HSDF, whereas other applications may use any programming model. We presented all of the hardware and software components together with their dataflow models, and the algorithms that combine them into a single model of a real-time streaming application executing on the platform. The models of the software stack and the algorithms are novel. Each application can be verified independently of other (non) real-time applications. Finally, we verified that the performance of real-time applications as computed automatically by our tool flow is conservative and accurate compared to the performance as measured on an FPGA implementation of the hardware & software platform for synthetic and real H.263 applications.

Acknowledgements

This work was partially funded by projects EU FP7 288008 T-CREST and 288248 Flextiles, CA505 BENEFIC, CA703 OpenES, ARTEMIS-2013-1 621429 EMC2, 621353 DEWI, and the European Social Fund CZ.1.07/2.3.00/30.0034.

References

[1] K. Goossens et al., Virtual execution platforms for mixed-time-criticality systems: the CompSOC architecture and design flow, in: ACM Special Interest Group on Embedded Systems (SIGBED) Review, 2013.

[2] G. Bilsen, Cyclo-static dataflow, in: IEEE Transactions on Signal Processing, 1996.
[3] M.H. Wiggers et al., Monotonicity and run-time scheduling, in: Proc. ACM Int'l Conference on Embedded Software (EMSOFT), 2009.
[4] A. Hansson et al., CoMPSoC: a template for composable and predictable multi-processor system on chips, in: ACM Transactions on Design Automation of Electronic Systems, 2009.
[5] A. Hansson et al., Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis, IET Comput. Digital Tech. (2009).
[6] S. Goossens et al., The CompSOC design flow for virtual execution platforms, in: Proc. FPGA World, 2013.
[7] A. Kumar et al., Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA, ACM Trans. Des. Autom. Electron. Syst. (2008).
[8] S. Stuijk et al., SDF^3: SDF for free, in: Proc. Int'l Conference on Application of Concurrency to System Design (ACSD), 2006.
[9] T.M. Parks et al., A comparison of synchronous and cycle-static dataflow, in: 1995 Conference Record of the Twenty-Ninth Asilomar Conference on Signals, Systems and Computers, 1995.
[10] S. Sriram et al., Embedded Multiprocessors: Scheduling and Synchronization, CRC Press, 2012.
[11] F. Baccelli et al., Synchronization and Linearity, Wiley, New York, 1992.
[12] O.M. Moreira et al., Self-timed scheduling analysis for real-time applications, EURASIP J. Adv. Signal Process. (2007).
[13] A. Nelson et al., CoMik: a predictable and cycle-accurately composable real-time microkernel, in: Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014.
[14] A. Beyranvand Nejad et al., A software-based technique enabling composable hierarchical preemptive scheduling for time-triggered applications, in: Proc. Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013.
[15] A. Nelson et al., Composable power management with energy and power budgets per application, in: Proc. Int'l Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS), 2011.
[16] N.J.H. Ip et al., A processor extension for cycle-accurate real-time software, in: Proc. Int'l Conference on Embedded and Ubiquitous Computing (EUC), 2006.
[17] R. Wilhelm, The worst-case execution-time problem – overview of methods and survey of tools, ACM Trans. Embed. Comput. Syst. 7 (3) (2008) 36:1–36:53.
[18] A. Hansson et al., Design and implementation of an operating system for composable processor sharing, Elsevier J. Microprocess. Microsyst. (MICPRO) (2011).
[19] D. Stiliadis et al., Latency-rate servers: a general model for analysis of traffic scheduling algorithms, IEEE/ACM Trans. Network. (ToN) (1998).
[20] M.H. Wiggers et al., Modelling run-time arbitration by latency-rate servers in dataflow graphs, in: Proceedings of the 10th International Workshop on Software & Compilers for Embedded Systems, 2007.
[21] D. Dolev et al., Hex: scaling honeycombs is easier than scaling clock trees, in: Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures, 2013.
[22] S. Goossens et al., A reconfigurable real-time SDRAM controller for mixed time-criticality systems, in: Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013.
[23] S. Goossens et al., Conservative open-page policy for mixed time-criticality memory controllers, in: Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE), 2013.
[24] Y. Li et al., Dynamic command scheduling for real-time memory controllers, in: Proc. Euromicro Conference on Real-Time Systems (ECRTS), 2014.
[25] B. Akesson et al., Real-time scheduling using credit-controlled static-priority arbitration, in: Proc. Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2008.
[26] B. Akesson et al., Architectures and modeling of predictable memory controllers for improved system integration, in: Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE), 2011.
[27] B. Akesson et al., Composable resource sharing based on latency-rate servers, in: Proc. Euromicro Symposium on Digital System Design (DSD), 2009.
[28] B. Akesson et al., Composability and predictability for independent application development, verification, and execution, in: Multiprocessor System-on-Chip – Hardware Design and Tool Integration, Springer, 2010.
[29] K. Goossens et al., The real network on chip after ten years: Goals, evolution, lessons, and future, in: Design Automation Conference (DAC), 2010 47th ACM/IEEE, 2010.
[30] R. Stefan et al., dAElite: a TDM NoC supporting QoS, multicast, and fast connection set-up, IEEE Trans. Comput. 7 (3) (2012) 233–270.
[31] A. Nieuwland et al., C-HEAP: a heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems, ACM Trans. Des. Automat. Embedded Syst. (2002).
[32] A.B. Nejad et al., A unified execution model for multiple computation models of streaming applications on a composable MPSoC, Elsevier J. Syst. Archit. (JSA) (2013).
[33] Benchmark Videos, <https://media.xiph.org/video/derf/>, 2014.
[34] H. Kopetz, Real-Time Systems: Design Principles for Distributed Embedded Applications, Springer, 2011.
[35] S.A. Edwards et al., The case for the precision timed (PRET) machine, in: Proceedings of the 44th annual Design Automation Conference, 2007.
[36] B. Lickly et al., Predictable programming on a precision timed architecture, in: Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, 2008.
[37] I. Liu et al., A PRET microarchitecture implementation with repeatable timing and competitive performance, in: Proc. Int'l Conference on Computer Design (ICCD), 2012.
[38] K.J. Nesbit et al., Multicore resource management, Proc Microarchitect. (MICRO) 28 (3) (2008) 6–16.
[39] E. Bini et al., Resource management on multicore systems: the ACTORS approach, Proc. Microarchitect. (MICRO) (2011).
[40] I. Shin et al., Periodic resource model for compositional real-time guarantees, in: Proceedings of RTSS, 2003.
[41] R. Bril, Towards pragmatic solutions for two-level hierarchical scheduling: a basic approach for independent applications, Technical Report, Technische Universiteit Eindhoven, The Netherlands, 2007.
[42] G.C. Buttazzo et al., Partitioning real-time applications over multicore reservations, IEEE Trans. Ind. Inf. (2011).
[43] F. Nemati et al., Independently-developed real-time systems on multi-cores with shared resources, in: Proc. Euromicro Conference on Real-Time Systems (ECRTS), 2011.
[44] A.K. Mok et al., Real-time virtual resource: a timely abstraction for embedded systems, in: Proceedings of the Second International Conference on Embedded Software, 2002.
[45] T. Ungerer et al., MERASA: multi-core execution of hard real-time applications supporting analysability, IEEE Micro 30 (5) (2010) 66–75.
[46] T. Ungerer et al., parMERASA – multi-core execution of parallelised hard real-time applications supporting analysability, in: Proc. Euromicro Symposium on Digital System Design (DSD), 2013.
[47] M. Schoeberl et al., T-CREST: time-predictable multi-core architecture for embedded systems, Elsevier J. Syst. Architect. (JSA) (2015) in this issue.
[48] F.J. Cazorla et al., Proartis: probabilistically analyzable real-time systems, ACM Trans. Embed. Comput. Syst. 12 (2s) (2013) 94:1–94:26.
[49] O. Moreira et al., Scheduling Real-Time Streaming Applications onto an Embedded Multiprocessor, Springer, 2014.
[50] A. Shabbir et al., CA-MPSoC: an automated design flow for predictable multi-processor architectures for multiple applications, Elsevier J. Syst. Architect. (JSA) (2010).
[51] P. Axer et al., Building timing predictable embedded systems, ACM Trans. Embedded Comput. Syst. (TECS) 13 (4) (2014) 82:1–82:37.
[52] R. Wilhelm et al., Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems, IEEE J. Comput. Aided Des. (2009).
[53] J. Wolf et al., RTOS support for parallel execution of hard real-time applications on the MERASA multi-core processor, in: Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, 2010.
[54] M. Schoeberl et al., Fun with a deadline instruction, Technical Report UCB/EECS-2009-149, 2009.
[55] K. Goossens et al., Composable dynamic voltage and frequency scaling and power management for dataflow applications, in: Proc. Euromicro Symposium on Digital System Design (DSD), 2010.
[56] ARINC Specification 653, Avionics Application Software Standard Interface, 1997.
[57] G. Heiser et al., The OKL4 Microvisor: convergence point of microkernels and hypervisors, in: Proceedings of the First Asia-Pacific Workshop on Systems, 2010.
[58] M. Paolieri et al., Timing effects of DDR memory systems in hard real-time multicore architectures: issues and solutions, ACM Trans. Embedded Comput. Syst. (TECS) 12 (1s) (2013) 64:1–64:26.
[59] H. Shah et al., Bounding WCET of applications using SDRAM with priority based budget scheduling in MPSoCs, in: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012, 2012.
[60] J. Reineke et al., PRET DRAM controller: bank privatization for predictability and temporal isolation, in: Proceedings of the seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2011.
[61] Z.P. Wu et al., Worst case analysis of DRAM latency in multi-requestor systems, in: Real-Time Systems Symposium (RTSS), 2013 IEEE 34th, 2013.
[62] M. Millberg et al., Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip, in: Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE), 2004.
[63] C. Paukovits et al., Concepts of switching in the time-triggered network-on-chip, in: Proc. Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2008.
[64] E. Kasapaki et al., Router designs for an asynchronous time-division-multiplexed network-on-chip, in: Proc. Euromicro Symposium on Digital System Design (DSD), 2013.

Andrew Nelson received his M.Sc. degree in Embedded Systems at Eindhoven University of Technology, Netherlands in 2009. After this, he moved to Delft University of Technology, Netherlands and received his Ph.D. there in 2014. He is currently employed as a Postdoctoral Researcher at Eindhoven University of Technology. His research interests include real-time (including mixed time-criticality) low-power multi-core embedded-systems and the timing analyses thereof.

Kees Goossens received his Ph.D. in Computer Science from the University of Edinburgh in 1993 on hardware verification using embeddings of formal semantics of hardware description languages in proof systems. He worked for Philips/NXP Research from 1995 to 2010 on networks on chip for consumer electronics, where real-time performance, predictability, and costs are major constraints. He was part-time full professor at Delft university from 2007 to 2010, and is now full professor at the Eindhoven university of technology, where his research focusses on composable (virtualised), predictable (real-time), low-power embedded systems. He published 3 books, 150+ papers, and 16 patents.

Benny Akesson received his M.Sc. degree at Lund Institute of Technology, Sweden in 2005 and a Ph.D. from Eindhoven University of Technology, the Netherlands in 2010. Since then, he has been employed as a Postdoctoral Researcher at Eindhoven University of Technology, CISTER-ISEP Research Unit, and Czech Technical University in Prague. His research interests include design and analysis of multi-core real-time systems with shared resources.