Virtual Execution Platforms for Mixed-Time-Criticality Applications: The CompSOC Architecture and Design Flow

Kees Goossens,[1] Arnaldo Azevedo,[2] Karthik Chandrasekar,[2] Manil Dev Gomony,[1] Sven Goossens,[1] Martijn Koedam,[1] Yonghui Li,[1] Davit Mirzoyan,[2] Anca Molnos,[2] Ashkan Beyranvand Nejad,[2] Andrew Nelson,[2] Shubhendu Sinha[1]

[1] Eindhoven University of Technology, [2] Delft University of Technology

Abstract—Systems on chip (SOC) contain multiple concurrent applications with different time criticality (firm, soft, non real-time). As a result, they are often developed by different teams or companies, with different models of computation (MOC) such as dataflow, Kahn process networks (KPN), or time-triggered (TT). SOC functionality and (real-time) performance is verified after all applications have been integrated.

In this paper we propose the CompSOC platform and design flows that offers a *virtual execution platform per application, to allow independent design, verification, and execution*. We introduce the composability and predictability concepts, why they help, and how they are implemented in the different resources of the CompSOC architecture. We define a design flow that allows real-time cyclo-static dataflow (CSDF) applications to be automatically mapped, verified, and executed. Mapping and analysis of KPN and TT applications is not automated but they do run composably in their allocated virtual platforms.

Although most of the techniques used here have been published in isolation, this paper is the first comprehensive overview of the CompSOC approach. Moreover, three new case studies illustrate all claimed benefits: 1) An example firm-real-time CSDF H.263 decoder is automatically mapped and verified. 2) Applications with different models of computation (CSDF and TT) run composably. 3) Adaptive soft-real-time applications execute composably and can hence be verified independently by simulation.

I. INTRODUCTION

Systems-on-chip (SOC) complexity grows as more applications are integrated in the same system. Often *applications* are dynamically started and stopped, leading to many *use cases*, i.e. sets of concurrently executing applications. Applications have specific *characteristics*, such as being control- or data-oriented, being more or less time critical (firm FRT, soft SRT, or non real-time NRT), and the extent to which they are adaptive or scalable. The system as a whole may have average or peak power or energy requirements depending on whether it is powered by battery, tethered, or energy-scavenging.

In a SOC applications are usually implemented with multiple communicating tasks, and are executed on an often heterogeneous set of processors and accelerators, memory hierarchy, and advanced on-chip interconnect such as a network on a chip (NOC). Since SOCs are always constrained in terms of area, the processors, interconnect, and memory resources must be shared between applications.

Problem Statement

Integrating a set of applications with different characteristics and requirements on a multi-core SOC is challenging for a number of reasons. First, *resource sharing causes interference* between applications, making their temporal behaviours inter-dependent. In case of an adaptive application the functional behaviour may change too. For example, when decoding H263 video, a B frame instead of an I frame may lead to a shorter computation, fewer memory accesses, lower interconnect throughput, and a different processor task schedule. This affects the performance of other applications (e.g. the audio processing) either positively (e.g. by finishing earlier) or negatively (e.g. in the presence of scheduling anomalies).

Second, SOCs applications are *developed by different groups* in a company, or even by different companies. Often executables are

delivered and integrated, since source code is proprietary and not shared for intellectual property reasons. Individual applications are verified on a system that is different from the final system, since other applications are usually absent. When all applications have been delivered and integrated, the system as a whole is verified. Unfortunately each application may fail at this point, since interference from other applications often leads to previously untested conditions. Use-case verification then becomes a *circular process* that must be repeated if an application is added, removed, or modified [36].

Third, applications are *designed and programmed using different methodologies, models of computation (MOC) or programming models, and design flows*. FRT applications could be programmed and analysed using dataflow, time-triggered, or event-based models of computation. Performance verification would be based on *worst-case* behaviour and formal analysis. NRT applications, on the other hand, could be programmed using threads or tasks using distributed shared memory to communicate. Verification could focus on functional correctness, and be based on *average-case* simulation. SRT applications occupy the middle of the spectrum and could use techniques of both.

The performance verification of *adaptive applications* is yet another case. For example, the quality of a SRT adaptive video decoder is measured with the signal-to-noise ratio (SNR) and deadline misses. The SNR depends on how much work (computation cycles at a certain frequency) and time (before a display deadline) the decoder allocated to decode a compressed video frame. If there is interference from other applications in terms of actual number of computation cycles or scheduling times, then the SNR will change. Hence, while FRT applications and SRT applications are verified on the basis of their worst-case and average-case behaviours, SRT adaptive applications are verified on the basis of their *actual-case* behaviour.

Finally, different application characteristics and MOCs require *different resource-management policies*. For example, different scheduling policies (preemptive vs. non-preemptive, work conserving or not, with jitter constraints and fairness properties, etc.) lead to different worst-case, average-case, and actual-case latency and throughput, but also affect buffer requirements. Regarding power/energy management, FRT applications must use conservative policies, where S/NRT policies can be speculative. It is challenging to map the different scheduling requirements of all applications on a few common schedulers (such as static priority or round robin) that many SOCs offer.

Composable and Predictable Virtual Platforms

The CompSOC platform [1] addresses these problems by offering *a virtual execution platform per application, to allow independent design, verification, and execution*. An automatic *design flow* is available for FRT dataflow applications, and S/NRT dataflow and Kahn process network (KPN) applications. Time-triggered applications are supported but not yet automated. CompSOC relies on two complexity-reducing concepts: *composability* and *predictability*. Composable virtual platforms are completely isolated (partitioned) and cannot affect each other by even a single clock cycle. They

are hence *virtualised in terms of actual execution time*, which enables independent verification of S/NRT applications. Our use of of composability extends [36] to multiple applications. Each virtual platform is also predictable, which means that it can be *virtualised in terms of performance bounds* such as worst-case execution time. This enables independent formal analysis of FRT applications.

In the next section we describe in more detail how composability and predictability solve or mitigate the problems introduced. Following this, we define the platform architecture (Section III), describe how resource sharing is dealt with (Sections IV-V) and elaborate the design flow (Section VI). In Section VII we illustrate our approach with a H.263 video decoder implemented as a FRT dataflow application, a FRT time-triggered application, and as a SRT adaptive dataflow application. In all cases, it runs together with other applications, each in their own virtual platform. After discussing related work in Section VIII we conclude.

II. COMPSOC CONCEPTS

In essence a virtual execution platform is a set of *resource budgets*; intuitively, a percentage of the resource capacity, enforced by a budget scheduler. Resources include processors, NOC, distributed on-chip SRAMs, and off-chip DRAM memory. In other words, each physical resource is divided in smaller virtual resources that are handed out to different applications. Resource budgets are computed at design time (see Section VI), and programmed at run time. Conceptually, composability is ensured on each resource by using *preemptive time-division multiplexing (TDM) between applications*, avoiding all interference between applications. Within a virtual platform, each application can further budget and schedule each virtual resource using whatever scheduling policy is appropriate for the application and resource. CompSOC therefore employs *two-level scheduling* on (selected) resources: *composable and predictable between applications, and application-specific within an application*. However, several different techniques are used to implement these concepts in a concrete platform, as discussed in Sections III-IV. First we discuss how our concepts address the problems raised in the introduction.

How Composability and Predictability Address The Problems

Composability, i.e. virtualisation in terms of actual execution time, has several important consequences. First, the behaviour of an application depends only on its own virtual platform. From the application writer's perspective, sharing with and interference from other applications have disappeared. Verification no longer depends on others, and has become a non-circular process. An application is designed and verified in isolation, and no reverification is required after integration in a larger system.

Second, each application may have its own performance verification methods. Since the *actual-case* behaviour is independent of other applications that may or may not be running at the same time, so is the *average-case* performance of an (adaptive) S/NRT application. Clearly, the *worst-case* behaviour of FRT applications is independent too. As a result, the verification of an application, of whatever criticality, depends only on its own virtual platform.

Third, application-specific scheduling policies are supported through CompSOC's two-level scheduling. Each virtual platform is composable and predictable, and within it, each application defines its own scheduler. Specifically, on a processor, the first-level TDM scheduler multiplexes applications, and a second-level application-specific scheduler multiplexes actors/processes/tasks of an application in a way that fits its characteristics and MOC. For example, a FRT application can use cooperative non-preemptive priority-based
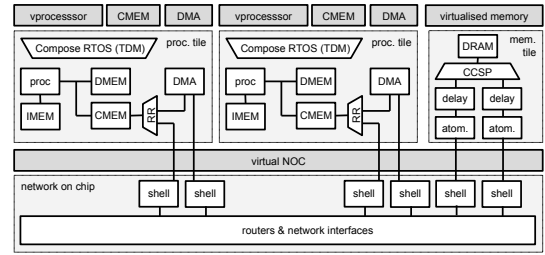

Fig. 1.   Example instance of the CompSOC architecture template.

scheduling between dataflow actors, a SRT application could use preemptive earliest deadline first (EDF) between tasks, and a NRT application preemptive round robin between KPN processes. Similarly, each virtual processor has its own virtual battery, managed by each application's power/energy manager.

Each virtual platform, i.e. a set of resource budgets, is predictable, which means it can be characterised by performance bounds on throughput and latency. For FRT applications, these bounds may be combined with the bounds given by its own predictable intra-application scheduler. In our FRT dataflow design flow, described in Section VI, an actor budget (intuitively, a percentage of the resource capacity) is the product of the virtual-platform budget and the actor budget within the virtual platform.

We conclude that composability and predictability are complementary concepts that both solve important parts of the verification problem for mixed time-criticality systems, and provide a complete solution when combined.

III. NON-SHARED PLATFORM ARCHITECTURE

It is essential that the platform architecture is designed with composability and predictability in mind, as it is not possible to graft these qualities on later. We first explain our user-resource model, and the consequences of composability and predictability on it. We then describe the platform components, and how they are combined into a larger system, and used at run time. A CompSOC platform consists of *resources* that are used by *users*, according to an allocated budget. Specifically, a resource is one of (cf. Figure 1):

- A processor with its local instruction memory (IMEM) and data memory (DMEM), used by computation tasks.
- A direct memory access unit (DMA), used by a DMA task.
- A communication memory (CMEM), used by storage tasks.
- A network on a chip (NOC), used by communication tasks.
- An on-chip SRAM memory, used by storage tasks.
- An off-chip DRAM memory, used by storage tasks.

As described in more detail in Section VI, for mapping and performance analysis purposes, users represent the resource usage of a "real" application tasks and channels. Application tasks are split in computation, DMA, and storage tasks, and channels are split in communication and storage tasks. A processor tile contains a single processor resource including its IMEM and DMEM, and one or more DMAs with their CMEMs. A S/DRAM tile contains the S/DRAM memory. To share the resource between multiple users each tile also contains additional hardware/software, e.g. buffering, multiplexing and scheduling. A platform instance contains any number of processor, SRAM, and DRAM tiles, and a single NOC connecting them all. For physical scalability, CompSOC uses GALS (globally asynchronous locally synchronous), i.e. each tile can operate on its own (scalable) clock frequency, and the NOC has its own single clock. Since tiles operate asynchronously, there is no shared global notion of time, and each resource scheduler operates asynchronously.

For simplicity, we first show how an application executes on a platform, where none of the resources are shared. Each computation task of the application uses a single processor, and its instructions and data must fit in the IMEM and DMEM, respectively, and be loaded before the task executes. The processor has no caches, and the application tasks cannot use interrupts (although its scheduler can). A task communicates with other tasks using distributed shared memory. To communicate with another task on the same tile it uses local DMEM, for another tile it uses the CMEM on the remote tile. A third alternative is to use a remote shared memory, either SRAM or DRAM. A single global address space is used for all memories, except that remote IMEMs and DMEMs are not visible to a processor. The CMEM and DMA are used to access a memory outside of the tile. To write to a remote memory, a task writes the data in its CMEM and then uses a communication library to instruct the DMA to copy the data from the local CMEM to the remote memory. Reading is similar, and copies remote data into the local CMEM. A task and its DMA(s) operate in parallel. The read and write requests of the DMA are transported by a NOC to the remote memories, where they are executed, after which requested responses return over the NOC.

IV. SHARING IN THE COMPSOC PLATFORM

For composability and predictability, each user is statically bound to one resource. Multiple users can be bound to a single resource, except for DMAs that are not shared. A user cannot use multiple resources, and resources cannot use each other; we discuss this further in Section VI. As a result, we can discuss the sharing of each resource in isolation. Composable sharing requires that the service an application receives is independent of others, and predictable sharing requires a bound on the minimum service an application receives. For each resource, predictable and/or composable scheduling require at least some of the following ingredients: IV-A) bounded scheduling interval, IV-B) fixed scheduling interval, IV-C) a neutral resource state between scheduling intervals, and IV-D/IV-E) an appropriate scheduler. The concepts behind these ingredients are detailed in [2]. Next, we discuss each of these in turn.

A. Bounded Scheduling Interval

Service is given out in *service units*, such as real-time operating system (RTOS) slots or NOC flits, that take a *bounded scheduling interval* to execute. (For simplicity we ignore pipelining here; see [2] for details.) A bounded scheduling interval is ensured either when each user of the resource has a *bounded execution time*, or each resource can be *preempted in a bounded time*. In the former case, cooperative (non-preeemptive) scheduling may be used, and the scheduling interval is the worst-case execution time of any user. However, note that in a mixed-criticality system the worst-case execution time for some users may not exist. A NRT task on a processor may never finish (perhaps intentionally), or memory transactions may be infinitely long (as allowed by, e.g., DTL and AHB protocols). This effectively locks up the resource for a single user, which breaks predictability and composability.

In the case of preemption, the scheduling interval is independent of the (worst-case) execution time of users, which is advantageous. However, it may not be possible to preempt a user in a bounded time. Many processors, such as Xilinx Microblaze and simpler ARM processors do not serve interrupts when there is an outstanding read or write transaction to a non-local memory. In the context of CompSOC, IMEM, DMEM, and CMEMs are local (on a local memory bus without handshake / flow control), and serve read/write transactions in a single cycle. Other remote CMEM or S/DRAM memories on the NOC are accessed via a handshaked PLB bus, and this may take an arbitrarily long (possibly infinite) time, depending on the scheduling on the NOC and remote memory. Hence CompSOC uses a DMA with a CMEM to communicate to remote memories, with the processor polling DMA for completion. This results in a bounded scheduling interval because the DMA works in parallel with the processor, and polling to the DMA is local and interruptible.

B. Fixed Scheduling Interval

Regarding the duration of a scheduling interval, most often it is actually *not* fixed. On a processor, although the interrupt timer may be cycle accurate, the interrupt service latency (in seconds) depends on the instruction that is being executed, and on the frequency the processor runs at. Similarly, reading or writing to a DRAM take a different number of cycles. There are two ways of dealing with this. Either all scheduling intervals are *made the same length* by padding with idle cycles or by disabling the clock of the resource to make each scheduling interval to be as long as the worst-case scheduling interval. In CompSOC, NOC time slots are padded, whereas the processor RTOS time slots are made longer with clock gating. Alternatively, the scheduling interval is *left variable*, and the interference arising from the variable scheduling interval and from variations in timing due to scheduling are removed by delaying to the worst-case interference. This approach is taken for the DRAM. Similar approaches are used in [3], [4], [36]. We introduce the details when discussing the individual resources.

C. Neutral State

Often resources have internal state that persists between the execution of successive service units. This state often influences the (timing of) execution. When scheduling a resource it is advantageous to return it to a *neutral state* between two service units. Composability requires that the execution of a user on a resource is independent of others. In particular, when a user starts or resumes execution the state of the resource must be independent of others. Often this is not the case, such as for processor pipeline, branch predictor, and caches, the state of which depends on the instructions of the application(s) that were executed before. The timing behaviour thus depends on other applications. Similarly, for DRAM memories, the open/closed status of pages at the end of a scheduling interval strongly affect the execution time of the following scheduling interval. Since all CompSOC resources are shared composably, we therefore *reset or restore the resource state between scheduling intervals*. In particular, for processors, branch prediction is turned off, and the pipeline timing and operating frequency are restored to the state when the user was previously interrupted and swapped out, and no caches are used. The DRAM uses close-page policy to enforce a neutral state.

D. Predictable Scheduling

A predictable scheduler must provide a bound on the service that is delivered to each user of a resource. For this, the scheduling interval must be bounded but not necessarily be fixed. Similarly, a neutral state is not required, as long as the interference of previous users of the resource can be bounded. However, since all resources are shared both composably and predictably, CompSOC returns every resource to a neutral state between scheduling intervals (except the DMA and CMEM, see below). In terms of scheduling policy, for predictability any *budget scheduler* will do. A budget scheduler guarantees a user a minimum number of service units in a given time frame. Essentially, any scheduler that is free of starvation suffices. In CompSOC we use latency-rate arbiters, such as as TDM, credit-controller static priority

(CCSP) [6], round-robin (RR), and static-order (SO) scheduling, depending on the resource.

E. Composable Scheduling

Finally, the resource scheduler determines in what order the bounded scheduling intervals are given to applications. For composability, this must be done such that the behaviour of an application is not affected by other applications. Scheduling cannot, therefore, take information that depends on (other) applications, such as availability of data or task readiness, into account. The obvious solution, which we use on the processor and NOC, is to use time-division multiplexing (TDM), which is static and not work-conserving. For the same reasons, TTA [36], MERASA [7], and [8] use TDM.

However, TDM inversely couples latency and rate, and couples the scheduling interval and frame size to the latency. A user that requires low-latency service must be overallocated in terms of bandwidth. Overallocation is not acceptable for scarce resources, since a composable, i.e. non-work-conserving, scheduler cannot give unused capacity to other applications that may want it. For this reason, a second technique is also used. *Any predictable scheduler can be made composable* by delaying the result of the execution of a user on a resource until its worst-case interference [9]. In particular, in the case of DRAM, the response of a read transaction is delayed as if it had experienced its worst-case interference from other users whether they are present or not. Since the worst-case interference of others is constant, the nett result is a composable behaviour.

F. Mixed-Time-Criticality Multi-MOC Applications

Given that each resource can be shared, it is useful to see how different applications, with different time criticality (F/S/NRT) and different models of computation (MOC), are mapped on it and run concurrently. In fact, the CompSOC platform is independent of the MOC of the applications that run on it [10]: a single platform execution model supports multiple models of computation. Processors execute generic tasks, and DMAs and NOC transport generic data.

To run a (FRT) *cyclo-static dataflow (CSDF) application* on CompSOC, its actors are automatically converted to tasks by adding a wrapper that receives tokens before firing and sends them after firing, using the C-HEAP FIFO communication library [11] that uses the DMAs. The user can chose a given (F/S/NRT) scheduler that implements actor firing rules (e.g. preeemptive round robin, non-preemptive static order), or supply his/her own. Similarly, a conservative FRT power management policy is provided [12]. As described in Section VI, CSDF applications are automatically mapped on the platform, and are FRT, (given, of course, appropriate scheduler and power manager, and worst-case execution times for actors).

The processes of (SRT) *Kahn process network (KPN) applications*, are normal CompOSe RTOS [13] tasks that use a blocking read/write API on FIFO channels, also implemented with [11]. A mapping of processes and tasks on the platform is automatically generated, along with default schedulers, but without real-time guarantees. The user can use a built-in speculative SRT power manager, or supply another.

Since dataflow and KPN applications work on the basis on availability of data/space in their channels, the entire CompSOC platform uses flow control / backpressure. All resources (processors, NOC, DMA, memories) use handshakes to transfer data, avoiding channel overflow or underflow. Equally important this enables work conserving behaviour within an application, unlike time-triggered architectures [36]: if data arrives early (when there is slack in the virtual platform), the application can work ahead. Conversely if data arrives later than expected, the consumer waits until it arrives.

Hence, dataflow actors and KPN processes tend to be scheduled on the basis of data availability (using firing rules and information regarding blocking on channels) rather than on the basis of time. Tasks of *time-triggered applications*, on the other hand, use a (often static periodic) schedule in time, or use a schedule based on task deadlines. (Non)-preemptive time-triggered task scheduling within an application is supported. Mapping is like for KPN applications.

V. Sharing Per Resource Type

To recapitulate, predictable sharing requires IV-A) a bounded scheduling interval, and IV-D) a predictable scheduler. Composable sharing is additionally either achieved with a IV-B) fixed scheduling interval, and IV-C) neutral state, with a IV-E) TDM scheduler, or by IV-E) converting predictable scheduler to be composable. In the following (see Table I), we describe for each resource what its users are, what its service unit is, and how its scheduling interval is (made) bounded, whether its scheduling interval is fixed, and how its neutral state is enforced. The composable and predictable interapplication scheduling and (optionally predictable) intra-application scheduling policies are also discussed. As we shall see, different resource characteristics (service unit size, size of state, abundant or not) lead to the use of different combinations of techniques.

A. Processor

A processor serves computation tasks. The CompOSe RTOS [13] runs on Microblaze and ARM processors and multiplexes multiple tasks on the same hardware using preemptive scheduling since tasks may have an unbounded or infinite execution time. Preemption is implemented using an interrupt generated by a timer that runs at the maximum frequency of the tile. Although the interrupt is generated at fixed points in time, the interrupt service routine (ISR) takes a variable number of cycles (e.g. depending on the currently executing instruction). Moreover, the time (rather than cycles) to serve the interrupt varies too, since each task can run at its own frequency. On receiving an interrupt, the ISR first sets the frequency to the maximum, saves the stack, records the ISR service delay due to the processor pipeline emptying, and jumps to the RTOS. Scheduling and housekeeping of the RTOS take a variable amount of time too. Therefore, to enforce a fixed scheduling interval, the processor gates its clock and instructs its clock generator to re-enable the clock at a fixed time at the end of the scheduling interval. In this way, a complete RTOS slot comprising task execution, interrupt handling, and RTOS scheduling, each of which may take a variable amount of time, is served with a constant scheduling interval. In Table I this is indicated by the worst-case scheduling interval (WCSI) delay. The next task slot length is reduced by the cycles that it took to empty the pipeline when the interrupt arrived, otherwise the task would receive more cycles than budgeted. The neutral state at the start of a scheduling interval is enforced by restoring the stack, and by enabling the clock at the task's frequency. Since the frequency may be changed several times in each RTOS slot, it must be quite fast (in the order of tens of clock cycles). Technology such as [14], [15] offers this capability for ASIC implementation. The CompSOC FPGA prototype models frequency scaling by clock subsampling [12].

A fixed scheduling interval coupled with a TDM scheduler results in composable scheduling. Since a processor is an expensive resource, its utilisation is important. As we saw before, composable scheduling cannot be work-conserving, and any slack is therefore lost. However, by offering two levels of scheduling, first between applications and then between tasks within the application, only slack between applications is lost. In fact, during unused slots the processor can be

resource	user	service unit	bounded SI through	fixed SI size (by)	SI neutral state	L1 inter-app. sched. (comp. & pred.)	L2 intra-app. sched.
processor	proc. task	OS slot	interrupt	WCSI-delay	pipel. delay & restore freq.	TDM	any
DMA	DMA task	transaction	comm. lib.	no	no state	—	—
CMEM	storage task	transaction	DMA	no	—	—	RR
NOC	comm. task	flit	shell	padding	no state	TDM	—
SRAM	storage task	word	atomiser	yes	no state	TDM	—
SRAM	storage task	word	atomiser	yes	no state	any pred. & WCI-delay	—
DRAM	storage task	pattern	atomiser	no	closed pages	any pred. & WCI-delay	—

switched off, reducing energy/power. Within a single application any application-specific (non)-work-conserving scheduler can be used. The cost of using two levels of scheduling implemented in software is amortised over the relatively large scheduling interval (at least 10,000 cycles at maximum tile frequency). Similarly, each application can have its own power manager that computes and sets its frequency, as part of the scheduler or its tasks. Both intra-application scheduling and power management are user-supplied code and run in user time, i.e. a misbehaving scheduler or power manager cannot affect the RTOS or other applications.

B. DMA and CMEM

A DMA serves DMA tasks. When a computation task wishes to communicate with a remote memory it uses a DMA communication library to instruct the DMA to copy (a finite amount of) data to/from the corresponding local CMEM. A computation task and its DMA task(s) run in parallel, and do not affect each other because they use distinct ports on the CMEM. When needed, the computation task polls for DMA completion, which ensures that a computation task can always be interrupted in a short time, as described previously. The DMA service unit is a single DMA transaction, which results in a variable, but bounded, number of read/write transactions to the NOC, and then a remote memory. When a remote DMA accesses a local CMEM to read or write (synchronisation) data, it uses the same port on the CMEM as the local DMA. Since only tasks from the same application communicate, this sharing needs to be at most predictable and not composable. For this reason, a round-robin scheduler is used here. The DMA has no state after completion, and the round-robin scheduler does not need to have a neutral state.

C. Network on Chip

The Æthereal NOC [16] serves communication tasks. A communication task (usually called a connection) is a virtual wire over which the NOC transports requests from master (DMA) to slave (memory), and optional responses from slave to master. Requests are received from the master using a DTL handshaked protocol. In theory, a transaction may be infinitely long, or may not arrive in a finite amount of time (since each word of the transaction is separately handshaked). A "shell" hardware block serialises the command and data groups of the request to a stream of words. The NOC executes a global TDM schedule of flit service units, with a scheduling interval of 3 cycles. Whenever a network interface port can inject a flit in the NOC, it consumes 3 words from the shell, or else pads with dummy words. This guarantees a fixed service unit. When flits arrive in the receiving network interface over their allocated connection, dummy words are removed, and the result deserialised to a DTL request. Responses are handled similarly.

From a scheduling perspective the NOC behaves as a single deeply-pipelined resource, even though it is made up of many routers and network interfaces. Logically it executes a global static TDM schedule. Flits are injected such that they never wait in the NOC to

minimise buffering since this is the main contributor to the area cost. Introducing a second level of intra-application arbitration would require an additional level of buffering (per-application virtual circuits), which is prohibitively expensive. Additionally, two-level scheduling at wire-speed is hard, and in any case, NOC bandwidth is essentially relatively cheap. For these reasons, all NOC communication tasks (connections), whether they belong to the same application or not are scheduled composably (and predictably).

D. SRAM

SRAM memory is the simplest resource, and serves transactions that are generated by storage tasks (or requestors). Transactions are chopped in single-word transactions; e.g. a 8-word write is converted in 8 single-word writes. The service unit is a single-word transaction, and the scheduling interval one cycle. Regarding atomicity of transactions, note that this chopping may interleave execution of multi-word transactions. While perhaps unexpected for software engineers, on-chip communication protocols, such as DTL and AXI, explicitly allow this, as they only require byte-level atomicity. Higher-level software synchronisation, such as C-HEAP, must be used to avoid unexpected results. In our current CompSOC platform, a single-level TDM arbiter is used most frequently, since bandwidth overallocation tends not to be a problem. However, all the techniques of the DRAM can also be used for the SRAM.

E. DRAM

A DRAM memory serves transactions of storage tasks. CompSOC uses the Predator real-time DRAM memory controller [17]. Like for the SRAM, incoming transactions are chopped into fixed-size transactions. However, DRAMs are only efficient with large access granularity, of at least 64 or 128 bytes. We refer to [18] for more information on how to select an appropriate access granularity, and corresponding DRAM configuration. The time to serve a read or write transaction on a DRAM varies greatly depending on the current state of the DRAM, i.e. whether the requested page is open or not, and whether the previous transaction was also a read or write. A simple worst-case approach would yield a maximum guaranteeable bandwidth of about 20%, which is unacceptably low: at least 80% utilisation is required. For this reason, Predator uses service units based on patterns, which are precomputed sequences of memory commands corresponding to a read, write, or read/write or write/read switch. These patterns leave the DRAM in a neutral state (all pages are closed), and a tight scheduling interval can be computed. Like the NOC, the DRAM is pipelined, and the scheduling interval is less than the service time.

TDM is not an acceptable scheduler for the DRAM for two reasons. First, the service units do not have the same length, since that would reduce utilisation of each service unit too much. Using TDM with variable scheduling intervals is not composable. Second, TDM couples latency and rate, which means that low-latency requestors require overallocation, which when unused cannot be given to other

applications through work-conserving scheduling. We therefore use the credit-controller static priority (CCSP) [6] scheduler, which does not have these defects. However, as CCSP is only predictable, we convert it in a composable arbiter by delaying each response until the time it would have had with worst-case interference [9]. In Table I this is indicated with WCI-delay. In essence, by simulating that each request always encounters worst-case interference, its behaviour is independent of other requestors whether they are present or not. As a result, without overallocation the DRAM is efficiently used. For S/NRT applications, the actual/average-case DRAM performance has degraded to the worst-case performance. But there are no drawbacks for FRT applications, since the worst-case performance is unaffected.

VI. Mixed-Criticality Design flow

Programming a multi-processor SOC is hard at the best of times, but additionally guaranteeing real-time performance is even more challenging. For this reason, given a FRT cyclo-static dataflow (CSDF) [19] application with worst-case execution time and memory usage per actor, the CompSOC design flow automatically computes the mapping of users to resources and the scheduling budgets of each user on each resource, such that required end-to-end, i.e. first to last actor, throughput and latency are guaranteed. All required C wrappers for actors, drivers that configure the resources and their schedulers at run time, as well as boot code to load the application(s) at run time are automatically generated. As mentioned before, KPN, time-triggered, and applications without a specific MOC must be mapped and compiled by the user, but are loaded automatically at run time. In the remainder of this section we focus on the FRT design flow, since it is the hardest, and is a superset of what is required for other applications. Lacking space for a full formal description, we aim to convey the intuition, and refer to [1], [20], [21] for technical details.

At the basis of the FRT design flow are the notions of user, resource, mapping, and budget. Each user (i.e. computation, DMA, communication, or storage task) is bound to a single resource (i.e. processor, DMA, NOC, or CMEM/SRAM/DRAM, respectively). Multiple users can be bound to a single resource, except for DMAs that are not shared. A user cannot use multiple resources, and resources cannot use each other. The essence of these restrictions is that the performance of a user (e.g. computation or storage task) depends only on the one resource (a processor or memory, respectively) it is bound to. This is essential for *compositional performance analysis*, as we discuss below. Each user has a budget on the resource it is bound to; a budget is a percentage of the resource. A budget is expressed as a minimum number of cycles of service during a given finite time duration. Equivalently it may be stated in terms of service units and scheduling intervals.

The cyclo-static dataflow (CSDF) paradigm is used both as a programming model and as a model of computation for performance analysis. To disambiguate, we talk about application actors and application channels versus actors and channels in the CSDF model. As shown in Figure 2(a), a CSDF application is written as a set of application actors that communicate using application channels. Each application actor is implemented by a computation task (bound to a processor), and as many DMA tasks (bound to DMAs), communication tasks (bound to the NOC) and storage tasks (bound to CMEMs) as it has outgoing application channels. In other words, the implementation of an application channel involves DMA, NOC, and CMEM, which must be modelled as three separate resources to comply with the mapping restrictions outlined above. (The distributed SRAM and DRAM memories are not used in the automated binding, but can be analysed when bound manually.)
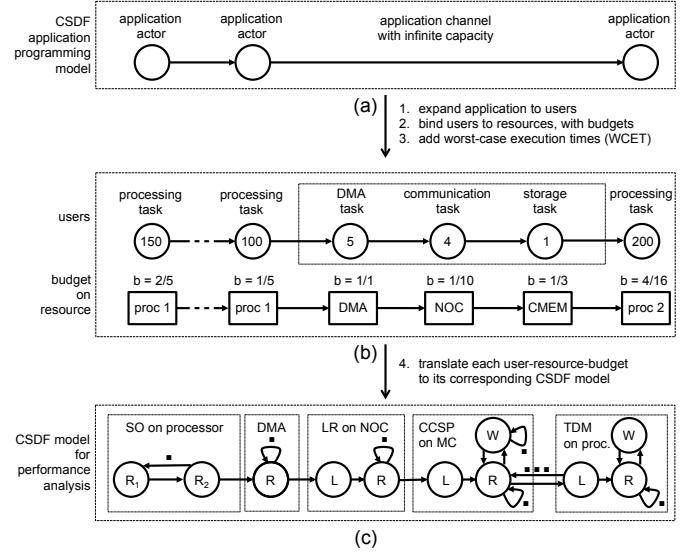

Fig. 2. (a) Application written in a dataflow programming model; (b) its user-resource representation; and (c) its dataflow performance analysis model.

A CSDF application graph is converted into its corresponding CSDF model for performance analysis in four steps. In Step 1 the application is expanded into its users, as explained above. In Step 2 users are bound to resources, i.e. receive resource budgets. At this point the application has its virtual platform, as shown in Figure 2(b). Steps 1 and 2 are performed for all applications, i.e. including those not written in the CSDF MOC. Later steps are for CSDF applications only. Step 3 adds the worst-case execution time (WCET) of each user on its non-shared resource, i.e. when it has a budget of 100%. Our flow computes the WCET of all users, except computation tasks, which it assumes as given. Step 4 translates each user with its budget on a resource to its corresponding CSDF model. Shared resources contain a scheduler, which is also part of the model. Different resources and schedulers result in different CSDF models.

For example, in Figure 2(c), the first two computation tasks are bound to the same processor 1, which uses a static-order (SO) scheduler. The rates R_i capture the effect of the budgets (2/5 and 1/5) on the WCETs of the tasks. (Intuitively, $R_1 = 150/(2/5)$ and $R_2 = 100/(1/5)$.) Note that the third task, which is bound to processor 2 with TDM has a different model [22]. The DMA task is bound to the DMA that is not shared. The NOC model uses a latency-rate (LR) abstraction of the communication performance [1]. CMEM is scheduled with CCSP, which has another model [6]. The self-edge on an actor ensures that the actor cannot restart before it finished.

The resulting CSDF graph with WCETs is used to compute end-to-end throughput and latency of the mapped application using standard dataflow maximum-cycle-mean analysis or similar. Note that this performance analysis is compositional in several senses. First, the WCET of a user bound to a resource is independent of other users and resources (since the resource is not yet shared, users can only use one resource, and resources cannot use each other). This is very advantageous in computing e.g. WCET of application actors on processors: DMA, NOC, and memories do not need to be taken into account. Second, sharing is modelled with budgets that only depend on the user and the resource. An actor that receives e.g. a 3/10 budget does not care if the remaining 7/10 of the resource are used (or not) by other users belonging to the same, or indeed different applications. Finally, a virtual platform, i.e. a set of resource budgets for an application, can be defined and given to an application
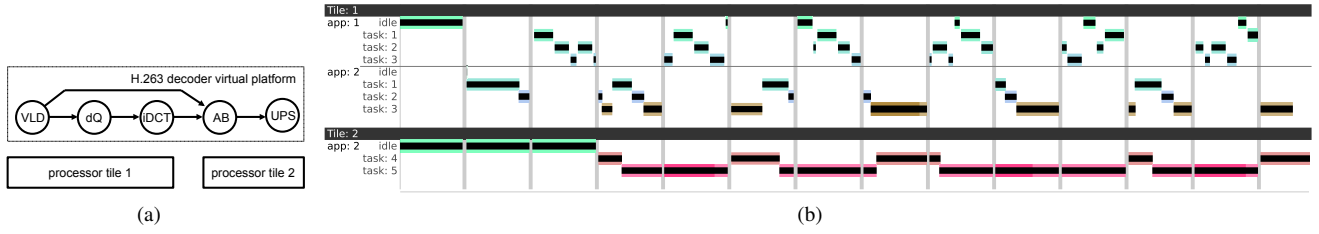
Fig. 3. (a) Binding of a FRT H.263 decoder, and (b) schedule traces of the synthetic application (App 1) and the H.263 decoder (App 2).

designer. Each resource budget can be subdivided when multiple users of the application share the resource. However, nowhere is there any dependency on other applications that may or may not co-exist on the same physical platform.

The FRT CSDF design flow is based [23]. It analyses the trade-off between the storage space assigned to the channels (i.e. size of CMEMs) and the throughput of the graph, binds users to resources, and generate static-order schedules for processors. All application C code, and system configurations, drivers and boot code are generated.

VII. CASE STUDIES

This section contains three case studies illustrating: 1) automatically mapping a FRT CSDF application on a platform instance, using formal performance analysis; 2) running applications with different MOCs and intra-application schedulers at the same time; 3) running multiple SRT adaptive applications at the same time, and using simulation-based performance verification. The CompSOC platform is prototyped on a Xilinx ML605 FPGA board.

In the first case study, a FRT H.263 decoder is implemented as a CSDF graph, and worst-case execution times for all application actors were obtained manually. After this, our design flow automatically finds a mapping on the given two-tile platform instance that satisfies resource (e.g. memory) and timing constraints. Non-preemptive static-order schedulers are used for intra-application task scheduling on the Microblaze processors. The CompSOC instance with the generated C code (application actors with CSDF wrappers) and C drivers and boot code were mapped on FPGA using standard synthesis tools. Experiments on the FPGA board confirmed that our platform provides a composable and predictable behaviour when running the H.263 decoder.

The second case study uses the same decoder, but with a different manual mapping, shown in Figure 3(a). A second synthetic application with three tasks also runs on tile 1, and uses preemptive time-triggered static-priority scheduling. Figure 3(b) is a trace measured on the FPGA, and illustrates CompSOC's two-level scheduling on processor tiles. Each composable RTOS slot is delineated by gray lines. In particular, we observe 1) the preemptive TDM schedule between virtual platforms, and 2) the preemptive fixed-priority schedule within the synthetic application's virtual platform (App 1), and 3) the non-preemptive static-order schedule within the H.263 application's virtual platform (App 2). Although the exact clock cycles are not shown for legibility, each RTOS slot is exactly the same duration.

The final case study uses an adaptive SRT CSDF dataflow implementation of H.263. At run time it trades the image quality (peak signal to noise ratio) for lower energy usage, depending on the available energy in the application's *virtual battery*, i.e. energy budget on each processor. In fact, *two H263 decoders run in different virtual platforms*. Both decoders use the same manual mapping as before. The four curves illustrate that four runs of one the decoders are identical. Situation 1, run 1 & 2 are two successive runs of one decoder, when it runs alone on the platform. In situation 2, run 1 & 2 are two successive runs, when the other decoder runs too. All

runs are identical in terms of energy (as shown) and cycles (not shown). The screen capture of Figure 4 shows the output of the system. Each decoder sends it output to shared memory, where it is sampled (composably) by a picture-in-picture application running on a third processor tile. The energy in the virtual battery of each application, and the energy per frame on each processor (C1/C2) per application are also shown at run time. The "%CPU" graph displays the frequency at which each frame is decoded. Decoder 1 decodes and upsamples to the large window, while decoder 2 decodes and downsamples to the smaller picture-in-picture (PIP) window, which results in a lower energy usage on tile 2. This case study shows that multiple adaptive SRT applications can run composably in the same CompSOC platform, and can hence be verified independently, using simulation or other means.

VIII. RELATED WORK

Specific related work was cited when we discussed particular techniques. Therefore, here we focus on general related work. Wilhelm et al. [24] give an overview on methods to predict the temporal behaviour and the implications of various hardware architecture details on predictability. Moreover, several approaches [7], [25], [26] propose to discard the modern architectural features that aim to improve average performance, but are, however, unpredictable. From this body of work we learn how to achieve predictable hardware.

Conventional real-time systems rely on priority-based scheduling, on top of predictable hardware, and are temporally verified using formal analysis [27]. RTOSs that follow this approach include. μC/OS-II [28], RTEMS (www.rtems.com), eCOS [29], QNX Neutrino (www.qnx.com), ERIKA (www.erika.tuxfamily.org), and FreeRTOS (www.freertos.org). Since priority-based scheduling is used across applications, these approaches are not composable, and often not suitable for a mix of F/S/NRT applications.

In the RTOS domain, the techniques that aim at temporal isolation, but not yet composability, are resource reservation, partitioning, and two-level scheduling. Resource reservation simplifies temporal verification by isolating applications, and ensuring independent temporal bounds. Few examples of such approaches are timing isolation [30]–[34], and real-time virtual resources [35]. Moreira [21] uses the same CSDF approach for real-time performance analysis as advocated here, but for FRT applications only. All these techniques provide the isolation of temporal bounds, but not of actual-case temporal behaviour, which is required for composability.

A resource is partitioned when each user has allocated a (fixed) fraction of the resource (in time and/or space). In two-level scheduling the processor time is first split between applications, and second within an application between its tasks. The RTOSs that implement partitioning are TTA [36] and VxWorks (www.windriver.com). Two-level scheduling is implemented in OKL4 [37] and hypervisors, in general. Only PikeOS (www.sysgo.com), INTEGRITY (www.ghs.com), and LynxOS-178 (www.lynuxworks.com) implement both. PikeOs and INTEGRITY have multi-core versions.
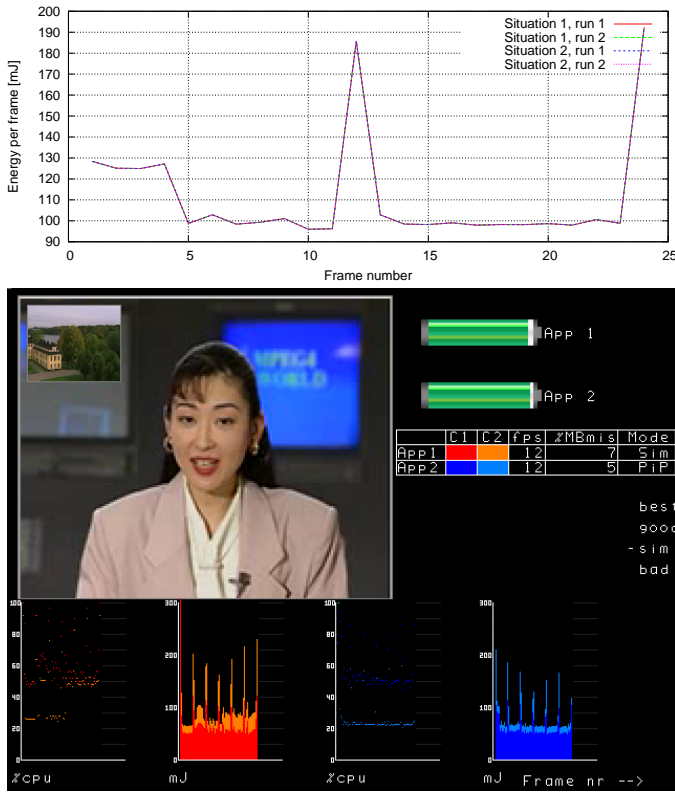
Fig. 4.   (top) Comparison of energy usage of a H263 decoder in its virtual platform, and (bottom) screenshot of PIP with two adaptive H263 decoders.

None of these RTOSs offer cycle-accurate partitioning, or two-level scheduling with preemptive intra-application scheduling.

To the best of our knowledge, CompSOC is the only platform that implements application independence down to the cycle level, power management per application, mixed-criticality and multi-MOC applications, with an automatic design flow for FRT CSDF applications.

IX. CONCLUSIONS

This overview paper presented the CompSOC platform architecture and its accompanying real-time design flow. By offering a virtual execution platform to each application, it is possible to simultaneously execute applications that have different requirements in terms of timing (firm/soft/non real-time), different models of computation (dataflow, Kahn process networks, time triggered, and C), and different implementations (e.g. task scheduler, power management). Since each application experiences no interference from other applications in its virtual platform, it can be designed, verified, and executed in isolation. This *composability* reduces the cost and complexity of integrating multiple applications in a single platform. Additionally, since each virtual platform is *predictable*, any mix of firm/soft/non-real-time applications can be supported. The CompSOC design flow automatically maps and verifies cyclo-static dataflow applications on the platform.

REFERENCES

[1] A. Hansson et al., "CoMPSoC: A template for composable and predictable multi-processor system on chips," *ACM TODAES*, vol. 14, 2009.

[2] B. Akesson et al. "Composability and predictability for independent application development, verification, and execution," in *Multiprocessor System-on-Chip*, M. Hübner and J. Becker, Eds., Springer, 2010, ch. 2.

[3] S. Matic et al., "Trading end-to-end latency for composability," *RTSS*, 2005.

[4] J. Lee et al., "Meterg: Measurement-based end-to-end performance estimation technique in qos-capable multiprocessors," *RTSS*, 2006.

[5] D. Bui et al., "Temporal isolation on multiprocessing architectures," *DAC*, 2011.

[6] F. Siyoum et al. "Resource-efficient real-time scheduling using credit-controlled static-priority arbitration," in *RTCSA*, 2011.

[7] J. Wolf et al., "RTOS support for parallel execution of hard real-time applications on the Merasa multi-core processor," in *ISORC*, 2010.

[8] A. Schranzhofer et al. "Worst-case response time analysis of resource access models in multi-core systems," in *DAC*, 2010.

[9] B. Akesson et al., "Composable resource sharing based on latency-rate servers," in *DSD*, 2009.

[10] A. Nejad et al., "A unified execution model for data-driven applications on a composable MPSoC," in *DSD*, 2011.

[11] 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 TODAES*, vol 7(3), 2002.

[12] A. Nelson et al., "Composable power management with energy and power budgets per application," in *SAMOS*, 2011.

[13] A. Hansson et al., "Design and implementation of an operating system for composable processor sharing," *MICPRO*, vol. 35(2), 2011.

[14] M. Meijer et al., "On-chip digital power supply control for system-on-chip applications," in *ISLPED*, 2005.

[15] P. Vivet et al., "On line power optimization of data flow multi-core architecture based on vdd-hopping for local DVFS," in *Integrated Circuit and System Design*. Springer, 2011.

[16] K. Goossens et al., "The Aethereal network on chip after ten years: Goals, evolution, lessons, and future," in *DAC*, 2010.

[17] B. Akesson et al., "Predator: A predictable SDRAM memory controller," in *CODES+ISSS*, 2007.

[18] B. Akesson et al., "Automatic generation of efficient predictable memory patterns," in *RTCSA*, 2011.

[19] G. Bilsen et al., "Cyclo-static data flow," in *ICASSP*, 1995.

[20] S. Stuijk et al., "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," in *DAC*, 2007.

[21] O. Moreira, "Temporal analysis of hard real-time radios on a multi-processor," Ph.D. thesis, Eindhoven univ. of technology, 2012.

[22] A. Lele et al., "A New Data Flow Analysis Model For TDM Alok Lele," in *EMSOFT*, 2012.

[23] S. Stuijk et al., "A predictable multiprocessor design flow for streaming applications with dynamic behaviour," in *DSD*, 2010.

[24] R. Wilhelm, et al., "The worst-case execution-time problem – overview of methods and survey of tools," *ACM TECS*, vol. 7, 2008.

[25] R. Wilhelm et al. "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *TCAD*, vol. 28, 2009.

[26] B. Lickly, et al., "Predictable programming on a precision timed architecture," in *CASES*, 2008.

[27] L. Sha et al., "Real time scheduling theory: A historical perspective," *Real-Time Sys.*, vol. 28, 2004.

[28] J. Labrosse, *Microc/OS-II*, 2nd ed.   R & D Books, 1998.

[29] A. Massa, *Embedded Software Development with eCos*.   Prentice Hall Professional Technical Reference, 2002.

[30] C. Mercer et al., "Processor capacity reserves for multimedia operating systems," in *ICMCS*, 1994.

[31] I. Shin et al., "Periodic resource model for compositional real-time guarantees," in *RTSS*, 2003.

[32] R. Bril, "Towards pragmatic solutions for two-level hierarchical scheduling: A basic approach for independent applications," Technische Universiteit Eindhoven, CS-report 07/19, 2007.

[33] F. Nemati, et al., "Independently-developed real-time systems on multi-cores with shared resources," in *ECRTS*, 2011.

[34] G. Buttazzo et al., "Partitioning real-time applications over multicore reservations," *IEEE Trans. Industrial Informatics*, vol. 7(2), 2011.

[35] A. Mok et al., "Real-time virtual resource: A timely abstraction for embedded systems," in *EMSOFT*, 2002.

[36] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*.   Springer, 2011.

[37] G. Heiser et al., "The OKL4 Microvisor: Convergence point of microkernels and hypervisors," in *Asia-Pacific Workshop on Sys.*, 2010.