Run-Time Middleware to Support Real-Time System Scenarios

Kees Goossens, Martijn Koedam, Shubhendu Sinha, Andrew Nelson, Marc Geilen
Eindhoven University of Technology, Netherlands

***Abstract*—Systems on Chip (SOC) are powerful multiprocessor systems capable of running multiple independent applications, often with both real-time and non-real-time requirements. Scenarios exist at two levels: first, combinations of independent applications, and second, different states of a single application. Scenarios are dynamic since applications can be started and stopped independently, and a single application's behaviour can depend on its inputs, on different stages in processing, and so on. In this paper we describe how the CompSOC platform offers system integrators and application writers the capability to implement multiple scenarios.**

I. Introduction

Multiprocessor SOCs (MPSOCs) are increasingly common and being required to perform ever more functionality. This has lead to mixed time-criticality systems, where applications with various degrees of timing requirement severity execute on the same physical hardware resources. Real-time applications require formal verification that their timing constraints are met. This effort is hard due to interference of concurrent applications on shared resources. This is complicated further on mixed time-criticality systems where the interfering application can be a non-real-time application that causes unpredictable interference.

Temporal isolation is a strategy to reduce this complexity, by bounding or eliminating the level of interference that real-time applications experience. The composable and predictable CompSOC platform [1] enforces strict temporal isolation, ensuring that concurrently executing Virtual Execution Platforms (VEPs) cannot interfere with other VEPs by even a single cycle. The actual (cycle-accurate) execution of an application executing on one VEP is therefore isolated from and independent of applications on other VEPs. CompSOC VEPs are formally modelled to enable formal verification of real-time applications executing on them [2].

Applications executing on a VEP can therefore change their behaviour (scenarios) without affecting applications executing on other VEPs. Additionally, VEPs, and the applications running on them, can be started and stopped independently, enabling dynamic combinations of multiple applications.

In this paper, we describe the CompSOC platform's run-time middleware, focussing on our virtual-resource generalisation. Our technique enables multiple heterogeneous resources, VEPs, and applications to be managed in a unified manner, simplifying VEP management and hence scenario configuration. The CompSOC software stack is shown in Figure 1.

II. Related Work

Traditionally, resource abstraction has been achieved with a Hardware Abstraction Layer (HAL) [3], [4]. A HAL enables

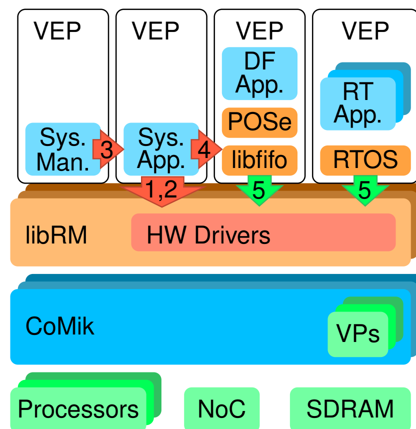


Figure 1. CompSOC platform hardware and software stack.

software portability and reuse. However, the HAL Application Programming Interfaces (APIs) is resource specific. Extensive work has been done on optimizing resource usage via resource managers [5]–[8]. We focus on [5] that presented a generic description of MPSOC run-time management components as the most related to this work. The MPSOC platform in [5] is considered as a single resource and the allocation policies and mechanisms for the platform resources are combined in a single resource manager. Instead our approach is 1) to manage all instances of each type of resource with a single resource manager; 2) to have a uniform API for all different (hierachical) resource types; 3) to manage entire VEPs and 4) applications using the same API.

III. The CompSOC Hardware Resources

SOCs contain multiple types of resources, most of which are run-time configurable enabling them to be managed at run-time. The CompSOC platform [1] is a composable and predictable tile-based architecture consisting of run-time configurable processor tiles and Synchronous Dynamic Random Access Memory (SDRAM) [9] tiles that are connected by a Network on Chip (NOC) [10]. The CompSOC platform implements strict temporally isolated VEPs.

The CompSOC platform can be instantiated to have one or more processor tiles. The CompSOC processor tile consists of a MicroBlaze processor, a Timer, Interrupt, and Frequency Unit (TIFU), local scratchpad Static Random Access Memory (SRAM) memories and Direct Memory Accesss (DMAs), as illustrated in Figure 2. The TIFU is programmed over an Fast Simplex Link (FSL) providing the CoMik microkernel [11] the ability to perform virtualised local tile Dynamic Voltage and Frequency Scaling (DVFS) and interrupt control. The tight coupling of the DVFS and interrupt controller with programmable interrupt timers enables CoMik to switch virtual

978-1-4799-9877-7/15/$31.00 ©2015 IEEE.

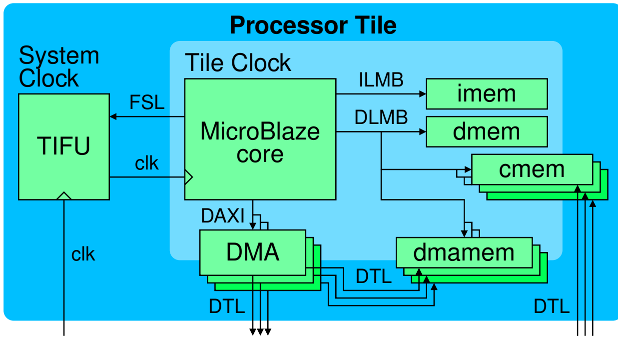


Figure 2. CompSOC processor tile.

processor contexts, including application frequency, following a Time Division Multiplexed (TDM) schedule.

The processor tile also contains local scratchpad memories for instructions (imem) and data (dmem) with single-cycle access via local memory busses (I/DLMB). These memories are virtualised by allocating dedicated regions of memory to the virtual processors. Inter-tile communication is carried out using DMAs. Each DMA has a dedicated dual-ported single-cycle access memory (dmamem) shared with the local processor. The DMAs are programmed by the processor on the Data AXI (DAXI) bus to read/write data from/to the dmamem, from/to distributed shared memories via the NOC. For direct tile-to-tile communication, each processor tile has dual-ported communication memories (cmems) that are connected to both the DLMB and the NOC. To achieve composability, DMAs, dmamems and cmems are dedicated virtual platform resources, i.e. not shared between virtual platforms.

The scratchpad memories are relatively small (up to a few hundred KB). To store more data, the CompSOC platform supports shared SRAM and SDRAM memory tiles attached to the NOC. They are composably shared using either a composable arbitration scheme, such as TDM, or a predictable arbitration scheme, such as Round Robin (RR) or Credit Controlled Static Priority (CCSP), with delay blocks [12]. The SDRAM memory controller back-end, and the memory arbiters and delay blocks are run-time configurable [9].

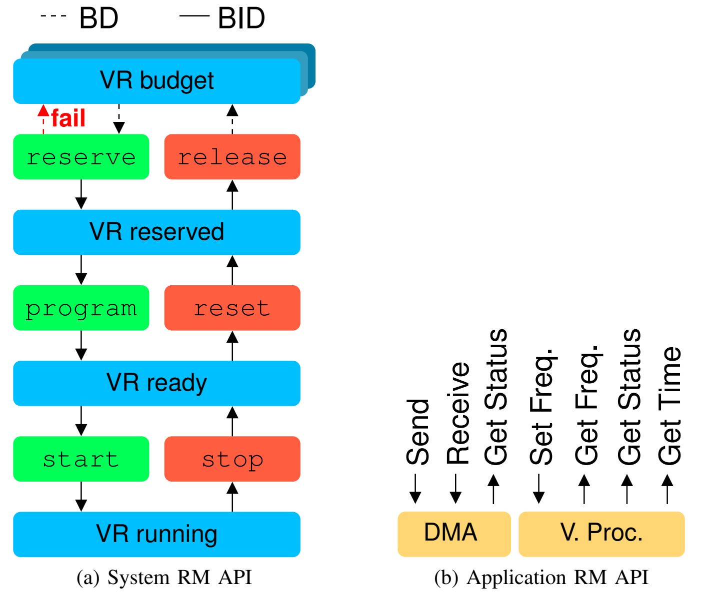
The CompSOC platform uses the predictable and composable dAElite [10] NOC. The NOC offers point-to-point connections with latency and bandwidth guarantees [13]. Connections (“virtual wires”) are implemented with run-time programmable TDM arbitration of physical NOC wires.

IV. Virtual-Resource Management

The CompSOC platform has various virtualisable resources where the dimensioning of the arbitration is run-time configurable. A processor can be divided in multiple smaller virtual processors, the NOC wires in multiple virtual wires (connections), and SRAM and SDRAM memories can be divided spatially (memory regions) as well temporally (access bandwidth). Although the specific nature of the configuration may differ between resources, CompSOC platform provides a generalised configuration interface, as illustrated in Figure 3a.

V. Budgets and Virtual Resources

Virtual-resource requirements are expressed as a Budget Descriptor (BD) that captures the requirement as a specific configuration of the resource, e.g. arbiter settings such as



(a) System RM API (b) Application RM API

Figure 3. States (blue) and transition API calls (green & red). VR (yellow)

CCSP priorities for the memory controller, TDM slots for NOC, memory controller, or microkernel, range of memory locations for any of the memories, or energy or power budget of a processor [14]. Given a BD the *reserve* API call makes a resource reservation, locking out other requesters from reserving overlapping resource configurations. A reservation request may therefore fail due to the required resources having been locked earlier by others. Of course, other reservation attempts with different BDs may be made. A successful budget reservation returns a Budget Identifier (BID) that refers to the record of the reservation, and the virtual resource is in the reserved state. A successful reservation that is no longer needed can be cancelled using the *release* API call.

The virtual resource is put in the ready state using the *program* API call with the BID as the argument. This instantiates the logical reservation by programming the budget on the physical resource using the resource driver. The virtual resource enters the running state after the *start* API call, which indicates that the virtual resource is functionally operational, e.g. a NOC connection accepts data. The performance of a resource is guaranteed according to its BD until it is stopped.

To release a resource reservation of a running virtual resource, it must first return to the quiescent ready state, by calling the *stop* API call. This ensures that all operational activity that is normally performed in the running state has ceased, e.g. all data has finished crossing a NOC connection, a DMA has no data in flight, and a memory controller pipeline is empty. The resource reservation can be de-programmed with the *reset* API call that takes a ready virtual resource to the reserved state. The resource reservation can be changed to a new BD with the *rereserve* API call, or be released.

A. Hierarchical Budgets and Hierarchical Virtual Resources

Each kind of resource (processor, DMA, NOC, imem, dmem, cmem, dmamem, and distributed memory) has its own driver to program it and a resource manager to administrate the virtual resources mapped on it. All resources of the same type are managed by a single resource manager, except processor tiles that each have their own, for performance reasons.

A processor tile contains multiple resources, namely processor, DMAs, imems, dmems, cmems, and dmamems, and the resource management API allows *hierarchical resource reservations*. A processor tile BD contains BDs for the subresources, and will hierarchically reserve, program, and start the subresources. If reservation fails for any resource, the hierarchical resource as whole fails. Stopping, resetting, and releasing work similarly. In fact, since a VEP is a set of virtual resources, e.g. multiple processor tiles, NOC connections, SDRAMs and SRAMs, it too is implemented as a hierarchical BD. Thus the same API shown in Figure 3a is used to manage 1) individual resources, 2) hierarchical resources such as processor tiles, as well as 3) entire VEPs. As shown in Figure 1, 1 and 2 are used by the system application, 3 by system manager to the system application.

B. Application Virtual-Resource Management API

Once a virtual resource has been created and an application is running on it (explain in the next section), the application can use the virtual resource, e.g. compute, communicate, and store/retrieve data. However, through the virtual-resource application API the application can change the state of the virtual resource, such as gettting and setting processor frequency and timers, or send and receive data using a DMA, or configure SDRAM memory controller patterns. The user API is specific to a virtual resource type, see e.g. Figure 3b.

Only the system application can create and modify virtual resources, i.e. VEPs, using the system resource management APIs shown as red arrows in Figure 1. Applications live within a VEP and can modify its state (the green arrows in Figure 1), but not its size, to ensure strong isolation of applications.

C. Application Management API

Once a VEP has been created, an application must be started in it. As discussed in more detail in the next section, this follows a process similar to creating a virtual resource namely: *reserving*, i.e. (optional) checking if the code and data of the application fits in the reserved VEP. Then *programming*, i.e. loading code and data of the Real-Time Operating System (RTOS) and application, followed by booting / configuring the RTOS or application. Then *starting*, i.e. executing the application. Since these steps are the same as those of a single (hierarchical) virtual resource, the system application uses the resource management API (arrow labelled 4 in Figure 1) to manage applications to start and stop in their VEP. In fact, work by Kramer [15] on dynamic change management for distributed applications inspired our virtual-resource and application management API, especially the notion of quiescent state (our virtual-resource ready state).

VI. System Scenarios

System scenarios incorporate both the combinations of independent VEP that can be instantiated concurrently and also the different states of the software that executes on the VEP. User software is mapped onto VEPs enabling cycle-accurately composable execution of concurrent software that is mapped onto other VEPs. The software within the VEP can have multiple execution scenarios, e.g. due to data dependent behaviour, and dynamically switch between them. In this section, we describe in detail how the CompSOC platform supports run-time configuration of system scenarios.

A. CompSOC Middleware

Software support for run-time scenario configuration is enabled by the scenario middleware. The CompSOC platform's software stack is illustrated in Figure 1. The *system manager* is an application that runs in its own small VEP on a single tile. It decides what applications run when and in which VEPs, and instructs the *system application* to effectuate these decisions. The system application runs in its own VEP consisting of a small virtual processor on each processor tile, and uses DMAs and NOC connections to synchronise its distributed operation. The system applications use the resource-management library (libRM) on each processor tile to configure virtual resources. libRM implements the virtual-resource management APIs described in the previous section. As described there, the interactions between system manager and application manager, and application manager and libRM, and application manager and applications all use the same API. Individual virtual resources, hierarchical virtual resources, entire VEPs, as well as applications are all managed uniformly, proving that our resource management is versatile and generic.

B. Dynamic Bundle Instantiation

An application always runs in a VEP, the BD that specifies the VEP is therefore always packaged together with the application's executables (code, data, and so on, in the form of ELF files). This combination is called a *bundle*. When the system manager detects that a new application is available (after its bundle has been uploaded at run time to SDRAM, for example), it decides whether it should run or not. In the next section we describe this decision process; we first, describe the bundle instantiation process.

The system manager sends (a pointer to) the bundle to the application manager which hierarchically reserves, programs, and starts the VEP, the virtual processor tiles, local tile memories, DMAs, NOC connections, SDRAM and SRAMs, etc. All virtual resources need to complete each before the next phase can start. If any phase fails, the application manager releases the entire VEP and informs the system manager. After successful creation of the VEP, virtual processors can compute using their local memories, can store data in local and remote virtualised memories, and communicate with each other using their DMAs.

Next, the application must be started. To reduce the load on the application manager, this process is performed by a boot loader that executes in the application VEP, i.e. using the application's virtual resources. It contains a number of steps.
1) *reserve* Optional step to check that the application fits in the given VEP, as explained in the next section.
2) *program* For each task on each processor, load its code and data (ELF) in the processor tile's imem and dmem, by copying it from (e.g.) SDRAM. Booting a bespoke scheduler (e.g. static order or preemptive) or RTOS, e.g. CompSOC's POSe [2] or a standard μC OS-III RTOS (`www.micrium.com`). If the application uses multiple processor or memory tiles then it also configures the application communication channels, e.g. dataflow or Kahn Process Network (KPN) First In First Outs (FIFOs) [16], using libFIFO.
3) *start* executing.
VEPs are created and run composability and predictably, i.e. cause no interference to other applications running their own

VEPs, as described in [17]. Multiple applications can boot and/or execute at the same time.

C. VEPs Scenarios

Given a set of bundles, i.e. applications with the specification (BD) of their VEPs, the system manager must decide which applications to run, i.e. which bundles to load. Clearly, if different bundles ask for the same virtual resources, they cannot execute concurrently, and the system manager must decide one way or another which application has priority. It is possible for an application to have multiple bundles, with different VEPs, e.g. a slow small-VEP implementation, and a fast large-VEP implementation. The system manager can try the different bundles in some order.

If the set of applications and their combinations is known at design time, then defining the VEP that each application receives is a (complex) optimisation problem. Conceptually, each VEP must receive (e.g.) a set of TDM slots (on all of its subresources) that do not conflict with any other VEP that must be able to run concurrently. The dAElite NOC design flow acomputes the NOC connections this way, optimising over all application combinations [13], but the rest of the CompSOC platform flow does not. For now we leave this complex job to the system designer / integrator, but this static allocation of virtual resources to applications is also common practice.

If the set of applications is not known at design time, then the system manager can check if a new bundle's VEP does not conflict with an already running application's VEP, and instantiate it if it doesn't. Alternatively, if the running application is deemed less important, it can be told to stop, using the application management API. Its VEP can then be reclaimed and the new bundle loaded. In all of the above situations, already running applications run composably, i.e. are either stopped or are not affected by new arrivals.

A final alternative is to resize the VEPs of running and new applications such that some appropriate combination of applications can execute. In this case, already running applications have to adapt to a new, perhaps smaller, VEP. While applications already know what VEP they run in, by virtue of a read-only copy of their VEP's BD (also known as a device tree), we have not yet investigated how they should internally reconfigure, should their VEP change. Note that dynamic reconfiguration of a VEP is likely to be infrequent. The interference that an application experiences due to other applications is therefore limited in terms of frequency and duration, and communicated up front by the system application through the application management API. The application should therefore be able to cleanly handle this process according to the states and transitions of Figure 3a.

D. Application Scenarios

Applications can be dynamic either because they have to react to their VEP being resized by the application manager, or by virtue of their own data-dependent behaviour. CompSOC allows applications to be written as multiple "unstructured" C tasks communicating through shared memory, (cyclo-static) dataflow actor, KPN processes, and as time-triggered jobs. Although none of these explicitly support the notion of scenario, program phases that are resource-intense versus those that are not, can often be identified. For example, the H264 video decoder [18] clearly distinguishes compute-intensive I frames from easier B and P frames, as well as images that are harder to compress than other. It uses this information in its energy-quality trade off. The system manager can use this information to decide if the application's VEP can be reduced in size or needs to be expanded. An API from the application to the application manager, and models of computation that have an explicit notion of scenario, such as Finite State Machine Scenario Aware Dataflow (FSM-SADF) is another possibility that should be investigated.

VII. Conclusion

We have presented an overview of the CompSOC run-time middleware and how it enables system scenarios in the sense of dynamically changing the set of concurrent applications, and dynamic application behaviour. Our resource management framework uniformly handles (hierarchical) virtual resources, entire virtual execution platforms (VEP), and user applications. The system manager application dynamically manages VEPss at run time using the API.

Acknowledgement Partially funded by FP7 288248 Flextiles, CA505 BENEFIC, CA703 OpenES, ARTEMIS 621429 EMC2, 621353 DEWI.

References

[1] K. Goossens et al., "Virtual Execution Platforms for Mixed-time-criticality Systems: The CompSOC Architecture and Design Flow," *SIGBED Rev.*, vol. 10, no. 3, Oct. 2013.

[2] A. Nelson et al., "Dataflow Formalisation of Real-Time Streaming Applications on a Composable and Predictable Multi-Processor SOC," *JSA, to appear*, 2015.

[3] K. Popovici et al., "Hardware Abstraction Layer," in *Hardware-dependent Software*. Springer 2009.

[4] S. Yoo et al., "Introduction to Hardware Abstraction Layers for SoC," in *Embedded Software for SoC*, A. Jerraya, et al., Eds. Springer 2003.

[5] V. Nollet et al., "A Safari Through the MPSoC Run-Time Management Jungle," *J. of Signal Proc. Sys.*, vol. 60, no. 2, 2010.

[6] O. Moreira et al., "Online Resource Management in a Multiprocessor with a Network-on-chip," in *SAC*, 2007.

[7] J. Teich et al., "Invasive Computing: An Overview," in *Multiprocessor System-on-Chip*, M. Hbner and J. Becker, Eds. Springer 2011.

[8] F. Hannig et al."Resource-aware Programming and Simulation of MPSoC Architectures Through Extension of X10," in *SCOPES*, 2011.

[9] S. Goossens et al., "A Reconfigurable Real-time SDRAM Controller for Mixed Time-criticality Systems," in *Proc. CODES+ISSS*, 2013.

[10] R. Stefan et al. "dAelite: A TDM NoC Supporting QoS, Multicast, and Fast Connection Set-Up," *IEEE. Trans. on Comp.*, vol. 63, no. 3, 2014.

[11] A. Nelson et al., "CoMik: A predictable and cycle-accurately composable real-time microkernel," in *DATE*, 2014.

[12] B. Akesson et al., "Architectures and modeling of predictable memory controllers for improved system integration," in *DATE*, 2011.

[13] A. Hansson and K. Goossens, "Trade-offs in the configuration of a network on chip for multiple use-cases," in *NOCS*, 2007.

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

[15] J. Kramer et al., "The evolving philosophers problem: Dynamic change management," *IEEE Trans. on Softw. Eng.*, vol. 16, no. 11, 1990.

[16] 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," *TODAES*, vol. 7, no. 3, 2002.

[17] S. Sinha et al., "Composable and Predictable Dynamic Loading for Time-Critical Partitioned Systems," in *DSD*, 2014.

[18] A. Nelson et al., "Power versus quality trade-offs for adaptive real-time applications," in *ESTIMEDIA*, 2012.