# Design and Implementation of an Operating System for Composable Processor Sharing

Andreas Hansson[a], Marcus Ekerhult[b], Anca Molnos[c], Aleksandar Milutinovic[a], Andrew Nelson[c], Jude Ambrose[c], Kees Goossens[d]

[a] *University of Twente, Enschede, The Netherlands*
[b] *Lund Institute of Technology, Lund, Sweden*
[c] *Delft University of Technology, Delft, The Netherlands*
[d] *Eindhoven University of Technology, Eindhoven, The Netherlands*

---

## Abstract

Multi-Processor Systems on Chip (MPSoC) run multiple independent applications, often developed by different parties. The applications share the hardware resources, e.g. processors, memories and interconnect. The sharing typically causes interference between the applications, which severely complicates system integration and verification. Even if the applications are verified in isolation, the system designer must verify the combined behaviour, leading to an explosion in design complexity. Composable MPSoCs have no interference between applications, thus allowing independent design and verification. For an MPSoC to be composable, all the hardware resources must offer composability. A particularly challenging resource is the processors, often purchased as off-the-shelf intellectual property.

In this work we present the design and implementation of CompOSe, a light-weight (only 1500 lines of code) composable operating system for MPSoCs. CompOSe uses fixed-size time slices, coupled with a composable scheduler, to enable composable processor sharing. Using instances of ARM7, ARM11 and the Xilinx MicroBlaze we experimentally demonstrate the ability to provide temporal composability, even in the presence of dynamic application behaviour and multiple use cases. We do so using a diverse set of processor architectures, without requiring any hardware modifications. We also show how CompOSe allows slack to be distributed within and between applications through a novel two-level scheduler and slack-distribution system.

*Keywords:* Composable, Operating System, Multi-Processor, System on Chip

## 1. Introduction

Embedded systems are seeing an increasing number of *applications* integrated on a single chip [1, 2, 3, 4, 5]. A large part of the applications are from the signal-processing domain [1, 4], which is also the focus of this work. Applications in this domain, e.g. a modem, filter or decoder, typically consist of *tasks* that communicate in a streaming fashion by performing actions on input data and producing output data. Three such applications are illustrated in Figure 1. The applications are realised by hardware and software Intellectual Property (IP), e.g. processing elements and application code, and are often *developed independently*, both by in-house design teams and by Independent Software Vendors (ISV).

With growing application heterogeneity, the application requirements are becoming more multifaceted [6], with non-functional aspects like timeliness and security growing in importance. For example, many signal-processing applications have firm or soft real-time requirements, either to satisfy standards (e.g. certification) such as WiMAX and WLAN, or to deliver a certain level of user-perceived quality, e.g. in a video or audio decoder. The real-time requirement could relate to a target on deadline miss-rate, or strict periodicity like a audio or video ADC or DAC. The requirements and verification methodology thus depend on each application. Moreover, the applications are started and stopped at run time, creating a large number of *use-cases*, making the system-level constraints increasingly complex.

During the design process, the tasks of the applications are mapped to processing elements, typically heterogeneous, as illustrated in Figure 2. The tasks communicate through the interconnect and possibly also use it to access memory for private data/instructions. The multi-processor platform shown in Figure 2(b) allows for distributing the tasks (within and between applications)

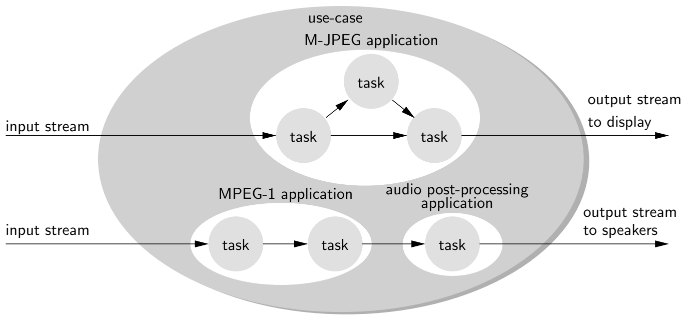
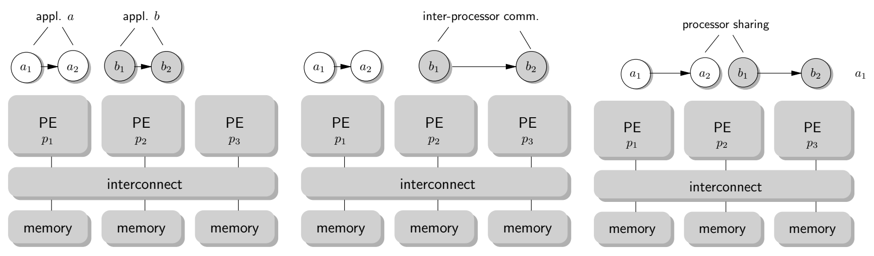


Figure 1: Application model.

across processors to deliver sufficient performance or reduce power consumption.

To reduce cost, hardware resources are shared by the applications within and between use-cases, as illustrated by one of the processing elements and the interconnect in Figure 2(c). However, sharing of resources causes problems with verification of functional and non-functional requirements of the application behaviours, as the behaviour of one application depends on all other applications in a circular (monolithic) fashion. This pushes the responsibility of application verification to the system integrator, making it one of the major challenges in SoC design.

To tackle the growing design and verification complexity, a divide-and-conquer approach, i.e. *composability*, is required. Composability takes the verification responsibility away from the system integrator and leaves it with the applications designers and developers by providing a *virtualised platform per application* [7, 8]. Thus, a composable platform ensures that the behaviour of one application is independent of all other applications. Traditionally, composability placed strict limitations on the applications [9, 2], unsuitable for e.g. the consumer-electronics domain. Recently, it has been shown in [10, 7, 11] how the on-chip interconnect and memories can be shared in a composable (and predictable) fashion. However, without a composable operating system it is only possible to run one application per processor (and still achieve system-level



(a) One application per PE (b) Distributed applications (c) Shared PEs

Figure 2: Different mappings for two applications onto three Processing Elements (PE).

composability). Although this is reasonable for a very simple RISC or VLIW processor, recent processors from e.g. ARM, such as the ARM11 are much too powerful to have one such processor per application.

Run-time scheduling (also known as on-line scheduling [12]) is offered by many (probably hundreds) of different operating systems and hypervisors, and some even offer bounds on the temporal behaviour. However, they all lack composability, due to e.g. priority-based scheduling and cache pollution. Moreover, real-time operating systems are typically focused on one processor and do not address the interfacing between tasks distributed across multiple processors and the *communication and synchronisation* between processors and memories. To achieve composable MPSoC also the processor I/O must be considered.

In this work, our main contribution is the design and implementation of a composable operating system for MPSoCs. To share processors in a composable fashion we ensure that tasks execute *without any interference*, i.e. that the time and processor state when an application is scheduled are independent of other applications. This requires:

1. pre-emption-based (enforced) sharing so that tasks are not required to be well behaved or well characterised.
2. a context switch mechanism that runs in constant time, even in the presence of instructions that take several cycles to complete (most notably I/O).
3. composable inter-application scheduling and cache management.

We show how to achieve the three aforementioned requirements in a lean (less than 1500 lines of code) operating system called CompOSe. It uses a novel concept based on scheduling of fixed-size *service units*, implemented by means of pre-emptive scheduling (item 1 and 2) and uses a *budget-enforcing scheduler* (item 3). CompOSe also offers a *two-level scheduler* to enable different task schedulers per application and a *slack manager* to maximally benefit from any unused capacity. The functionality of CompOSe, and the ability to deliver *temporal composability at clock cycle resolution*, is demonstrated using gate-level simulation (on a multi-processor system) and actual hardware (on a single-processor system). We show experiments using ARM7, ARM11 and MicroBlaze processors to demonstrate the concepts behind CompOSe on a diverse set of processor architectures, both with and without caches.

The remainder of the paper is structured as follows. We start by introducing related work in Section 2. Next, the the application software and hardware platform is described in Section 3, including an introduction to composability in MPSoCs. As the major contribution of this paper, a detailed description of the proposed processor tile and composable operating system is given in Sections 4 and 5. Experimental results, using different processor architectures, abstraction levels and system instances, are presented in Section 6, followed by conclusions in Section 7.

## 2. Related work

Many scheduling algorithms have been proposed and commercially used in embedded operating systems. In Symbian [13], for example, the (preemptive) scheduler uses priority levels and Round Robin inside each level. The priority-based arbitration inherently couples the applications, making it non-composable. Other common scheduling algorithms like Rate-Monotonic Scheduling (RMS) [14] and Earliest Deadline First (EDF) [15] make a large number of assumptions on the tasks, e.g. that there are no task dependencies, and also assume a (correct) characterisation with respect to deadlines and execution times.

In the domain of signal-processing applications this information is not always available and tasks often have data-dependent input and output behaviour, causing significant variation in their execution times and execution rates. Moreover, even in the presence of a correct characterisation there are significant variations in the schedule caused by the other applications making these approaches non-composable.

Hypervisors, on the other hand, are used to virtualise processors, including memory accesses, file systems, interrupts, I/O etc. They are typically designed to run several independent operating systems and applications without placing any restrictions on the latter. However, commercially available hypervisors focus on the functional behaviour and offer limited support for real-time applications. VirtualLogix VLX [16] and Open Kernel L4 [17], for example, use priority-based arbitration and can thereby give real-time bounds to one of the virtualised operating systems. In those approaches, the temporal behaviour depends on all higher priority operating systems (and thus applications). Although these hypervisors provide many important aspects of application isolation, there is no commercial hypervisor that offers temporal composability, and the real-time analysis of any general application in isolation is rendered invalid by resource sharing. Unlike any commercial hypervisor, our goal is to ensure that applications sharing a processor *do not affect each other even on the clock-cycle level*. Our work is, however, less general and does not virtualise interrupts and consequently does not allow the applications to use such functionality.

The operating system introduced in [18] aims to enable real-time guarantees without restricting the applications running on the processor. A budget scheduler guarantees every task a *minimum* amount of time in a maximum interval. This is to be compared with the *fixed* amount of time offered by CompOSe. For well-behaved and well-characterised applications the minimum time enables bounds on the temporal behaviour, e.g. throughput, latency and periodicity, by means of dataflow analysis [19]. With dataflow models, the bounds are sufficient to provide independent application analysis, but assumes that the model is conservative and that the implementation of the tasks is correct and bug free. In general, the provision of a minimum budget is not sufficient to ensure that the applications do not affect each other as the time intervals at which the application is scheduled depends on the other applications.

In addition to the challenges involved in sharing a single processor between multiple tasks, an operating system for an MPSoC must also address communication and synchronisation. Neither of the aforementioned operating systems and hypervisors account for blocking I/O operations (e.g. a read to an off-chip SDRAM), and either assume single-cycle memory access latencies or completely ignores the impact on the execution time of the applications and the scheduler.

This work extends [20], and unlike [18] our emphasis is on composability rather than predictability. We do not require known (and correct) worst case execution times, and also have weak requirements on the task semantics in terms of input and output behaviour. Much like the aforementioned hypervisors the aim is to separate applications logically and thereby enable a divide-and-conquer design methodology. In [20] code and data of tasks are assumed to fit in the local tile memory, whereas this work shows how it is possible to also incorporate caches in the processor tile architecture. Compared to [20] this work gives more implementation details and shows how the concepts of CompOSe can be applied to a diverse set of processor architectures.

## 3. Background

In this section we elaborate on the application software and hardware platform targeted for our composable operating system. We start in Section 3.1 with a description of the existing (composable) hardware platform and continue in Section 3.2 by detailing the assumptions on the application software. We end the background description with an outline of the overall design flow in Section 3.4.

### *3.1. Hardware platform*

The platform used in this work is an extensions of the CoMPSoC architecture introduced in [7]. CoMPSoC uses the Æthereal Network on Chip (NoC) [21], which offers composability and predictability for every logical connection between pairs of memory-mapped initiator and target ports, e.g. the six ports shown in Figure 2. The composable and predictable services also extend to the shared memories (target ports) [11], thereby isolating all the communication between the IPs (ports) in the system. However, the CoMPSoC platform, as described in [7] does not address sharing of the processor tiles, including both the processor itself and its initiator port(s). Both are essential in providing a complete composable platform, and the latter involves the architecture of both the processor tile and the NoC. Next we look at these two issues in more detail.

In contrast to the VLIW processors used in [7], we use the ARM7TDMI (hereafter ARM7), ARM1176JZF-S (hereafter ARM11) and the Xilinx MicroBlaze, further discussed in Section 4. These processors all support pre-emption through (precise) interrupts and thus allow us to enforce context switches, something that is central to the functionality of CompOSe. While the ARM7 and MicroBlaze have no caches (in our implementation), the ARM11 architecture uses a read-only instruction cache and write-back data cache, with software control for invalidation and flushing. The NoC does not provide any hardware cache coherency due to its inherent scalability issues and performance implications. As a consequence, cache control is critical for CompOSe and one of the challenges addressed in this work.

Composable sharing of a processor is not restricted to the processor core (pipeline, register file, etc), but also the I/O interface. For all commercial processors we are aware of, the initiator interface is single threaded (although protocol standards like AXI and OCP allow multi-threading). As a consequence, synchronisation operations and load/store operations to remote memories, i.e. another tile or an external memory, cannot be interrupted. There is consequently a strong dependency between the operating system and the NoC, and both the processor tile and operating system must take this into account in order to deliver composability.

### *3.1.1. Composability*

Composability removes inter-application interference, but not necessarily all variation caused by the platform itself. Thus, even though CoMPSoC offers composability, variations in the application behaviour may occur due to e.g. clock domain crossings, analog-to-digital converters at the inputs, the alignment of arbiters of different resources, out-of-order execution and caches. Moreover, the application itself may have input-data dependencies or timing-dependent behaviour (e.g. drop frame if approaching deadline) causing variations. All the aforementioned effects may cause variations in the temporal behaviour, but they are *not dependent on the other applications* and the platform is still composable. We refer to [7] for a more extensive discussion on the more subtle aspects of composability.

In our experiments, as later shown in Section 6, we use a deterministic simulator in the evaluation of the MPSoC netlists. Thus, the aforementioned variations are the same for repeated runs. This allows us to verify composability by looking at the difference between traces on a cycle level, although in practise this may be impossible to achieve.

## *3.2. Application software*

We assume that the applications can be represented as task graphs with *explicit communication and synchronisation.* Most applications in the multimedia domain lend themselves to implementation as tasks that communicate using FIFO buffers on a per-token basis. Traditionally, a task is implemented as a never-ending loop that reads input data, performs computation and produces output. In our case, the input and output operation is (preferably) left for the operating system, and the task is not a loop, but rather a function that executes and returns (for each invocation), as exemplified in Listing 1. This code implements a task with two inbound FIFOs using a token size of 12 bytes, and one outbound FIFO with a token size of four bytes. As we shall see in Section 5, the I/O is handled by CompOSe, but takes place in task time rather then operating system time. This ensures the overhead of the operating system

```
void Task1(const int in1[3], const int in2[3], int out[1]){
  out = process(in1, in2);
}
```

Listing 1: Task code

is kept to a minimum.

The proposed task semantics give the operating system information about the input and output dependencies of the task (the enabling condition), and also provides information about task completion when the function returns. This is not a requirement, but as we shall see in Section 5.3, these two points are essential in enabling the operating system to distribute slack and thus take full advantage of CompOSe.

Note that there is *no need for execution time characterisation* from the operating systems point of view. If a particular application requires real-time guarantees it is left to the specific application developer to verify that these guarantees are satisfied on the virtual processor assigned to the application in question. The on-chip interconnect and the memory controllers are predictable offer formal models to facilitate end-to-end verification [22, 23].

### *3.2.1. Limitations*

One of the main limitations we impose on the applications is that they must not use interrupts. This is due to the fact that the number of interrupts and the time incurred serving interrupts is difficult to bound. Rather than virtualising interrupts, we currently chose to allow only one interrupt that is used by the operating system itself to limit the length of the time slices allocated to tasks. This limitation is discussed further in Sections 4 and 5. Many applications in the signal-processing domain are not inherently relying on interrupts, and we therefore leave composable virtualisation of interrupts for future work.

In addition to interrupts, the applications must not employ any kind of resource locking, e.g. slave locking in AXI [24], of slaves shared between applications. A shared resource that is locked violates composability unless assumptions are placed on the application's use of the lock (and this contradicts our design goals). To facilitate communication and synchronisation without using locks we use a library for inter-task communication that uses polling. Next, we describe this library in more detail.

## *3.3. Inter-task Communication*

CompOSe is targeted at MPSoCs, with multiple copies of CompOSe running on different processor. Each processor runs an independent instance of CompOSe, and to enable distribution of one application across multiple processors, the platform must therefore offer some form of inter-task and possibly inter-processor communication. Although our hardware platform supports any memory-mapped communication, thus giving the programmer full flexibility, we have adopted C-HEAP [25] as our communication API. In C-HEAP, all communication takes place via dedicated buffers, using *acquire* and *release* calls in the application code, thus making all inter-task *communication explicit*, in line with the task semantics described in Section 3.2. The use of an API also makes the application code more portable and simplifies potential reuse.

The advantage of C-HEAP compared to other communication protocols is that it *does not use atomic operations* such as slave locking or semaphores that do not scale well and complicate the provision of composability. C-HEAP also does not use interrupts. A C-HEAP FIFO is implemented as a circular buffer in shared memory, which allows run-time configuration of the FIFO size. Copies of the FIFO administration, e.g. read and write pointers, are kept in the local memories of both the producer and consumer to *avoid polling remote memory locations*, which would incur the interconnect latency and decrease performance.

If size permits, the FIFO communication buffers are placed in the local memory of the processing element running the consumer task. Thereby, all inter-tile communication is *using posted (non-blocking) writes* and completely avoids reading from remote memories. This is crucial for a NoC-based MPSoC with distributed memories as the read latencies to remote memories are tens to hundreds of cycles. If a FIFO is too large to fit in the local memories of the processors tiles, e.g. the reference frames in a video decoder, the FIFO data can be mapped to one of the dedicated memory tiles, as shown in Figure 2. The FIFO administration is still mapped in the local memories, to keep the administration close to the processor for low latency reads. Using distributed memories, however, requires a *memory consistency* model that guarantees that the administration is updated only after a token is actually produced and in the FIFO memory. Furthermore, when remote memories are used to store the FIFO data and the processors access the contents more than once, e.g. when decoding an image, it is beneficial to use the data cache for increased performance. This raises the issue of *cache coherency*. The on-chip interconnect used in this work offers release consistency but no hardware cache coherency [7, 26]. We return to discuss how to solve the issues of memory consistency and cache coherency when discussing the implementation of CompOSe in Section 5.

## *3.4. Design flow*

Although CompOSe is centred around composable sharing of a single processor, it is aimed at MPSoCs and applications distributed across multiple processing elements and memories. A major design flow challenge is the application mapping and scheduling, and although it is outside the scope of this work, CompOSe plays a central role in this problem as: 1) the applications must conform with the application model in Section 3.2, and 2) the tools in the design flow should be able to reason about the outcome of scheduling decisions.

Looking at the two problems in turn, the applications could be provided by parallelisation tools like [27, 28] or by hand, as done in [29]. Many algorithms in the signal-processing domain are conveniently described in a form very suitable for the application model of CompOSe, and automatic parallelisation is an active research area.

Once the applications are provided, the mapping and scheduling can be done by a tool aimed at dataflow models, e.g. [30]. Even in the absence of execution times, deadlock freedom can be proven by [31] and buffers (in the network and between the tasks) sized accordingly. CompOSe does not enforce a predictable processor architecture (or application), but in the presence of worst-case execution times, the aforementioned dataflow tools can also reason about the end-to-end temporal performance together with models of the NoC [22].

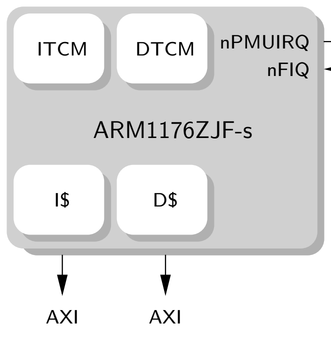
## 4. Processor tile

The primary goal of CompOSe is composability and the key idea to achieve it is: 1) the use of Time Division Multiplexing (TDM) with fixed-size (constant duration) *service units*, coupled with 2) a context-switching mechanism that guarantees a well-defined *zero state* that is independent of the applications running on the processor, e.g. with no outstanding I/O and cold caches.

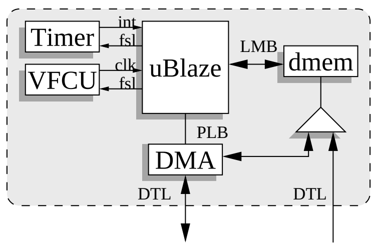
To achieve fixed-sized service units, the tasks have to be interrupted at fixed moments in time. Consider processor $p_2$ in Figure 2(c). If the task $a_2$ of application $a$ would be able to monopolise the processor for a variable duration, the composability would be compromised, i.e. the temporal behaviour of application $b$ would depend on how much $a$ executes. Note that the inability to bound the time to serve the timer interrupt is equivalent with an application monopolising the processor.

In addition to the tasks, the operating system should also execute in constant time. Naturally, the operating system execution time depends on the number of applications and task it has to schedule. Thus, if the operating system execution time is not forced to take a constant (worst-case) duration, the starting times, and implicitly the temporal behaviour of an application would depend on the presence or absence of other applications in the system, again compromising composability.

In this section we discuss the various options how to construct a processor tile well-suited for achieving this functionality. First, in Section 4.1, we discuss the options to generate and interface with a timer as required by CompOSe. Second, we look at the possibilities of clock-gating during idle periods in Section 4.2. Third, in Section 4.3 we look at how to limit the number of outstanding I/O transactions and bound their worst-case finishing time. Finally, we discuss the



(a) ARM11 tile



(b) MicroBlaze tile

Figure 3: Tile architectures.

implications of adding caches to the processor tile in Section 4.4.

As shown in Figure 3, we exemplify multiple points in this design space, with the ARM11 tile (Figure 3(a)) having an internal timer, instruction and data caches, no clock-gating, and no Direct Memory Access (DMA) functionality; and the MicroBlaze tile (Figure 3(b)) having an external timer, no caches, clock-gating-based delay and DMAs for external I/O transactions.

## *4.1. Timers*

To track the length of the service-unit time slots, CompOSe needs a timer. In the general case this is implemented with a dedicated external timer accessed via a memory-mapped peripheral bus or instruction-mapped accelerator port, as exemplified by the MicroBlaze in Figure 3(b). This approach is generally applicable to any processor architecture, and as discussed in Section 4.2, the processor can enter a low-power state during idle periods without stopping the timer.

The ARM11, shown in Figure 3(a) has an internal cycle counter that can be used as a timer. The cycle counter is a programmable 32-bit counter, counting upwards on every clock cycle and on overflow the output-pin nPMUIRQ is pulled low. No additional hardware is used and it is easy to manipulate the instruction-mapped timer. However, it is no longer possible to enter a low-power state (or gate the clock) as this stops the timer, resulting in a complete processor stop.

In both approaches, the timer generates an interrupt signal that is connected back to the processor. In the case of the ARM11 to the nFIQ port and for the MicroBlaze to the IRQ.

## *4.2. Halting*

As we shall see in Section 5, a critical step in achieving constant time service units is to delay further execution until the worst-case duration is reached. The easiest way to achieve this, which is also implemented in the ARM instance of CompOSe is to simply idle and execute NOP in a loop. The idling is also generally applicable to any processor tile architecture. In the presence of an external timer, as we have seen in the previous section, it is possible to extend the functionality with a Voltage Frequency Control Unit (VFCU). This is implemented in our MicroBlaze tile, as shown in Figure 3(b). The VFCU provides the processor clock and is able to (un)gate the clock at a future moment in time, or to immediately gate the clock [20][1].

## *4.3. Communication latency*

A bounded interrupt latency requires interruptible instructions in the processor's pipeline or a bounded, preferably short, maximum time to finish in-flight instructions. Thus, the maximum delay till serving an interrupt is the maximum time it takes to execute an instruction, which is in the order of a few cycles, except for synchronisation operations and load/store operations to remote memories, i.e. another tile or an external memory. We avoid the conventional synchronisation operations to exclude the implications of an interrupt being raised during those. Instead, synchronisation is implemented by polling checks for data and space according to Section 3.3. The I/O operations to remote memories, however, remain an issue.

Our platform is built around a composable and predictable on-chip interconnect and memory hierarchy according to Section 3.1. It is thus possible to find

[1]The VFCU is also able to change the clock frequency, but this functionality is outside the scope of this work.

an upper bound for the delay of any load/store operation. The bound does, however, depend on all applications sharing the processor, interconnect, and memories, and their resource allocations. Moreover, such a bound is quite conservative (thousands of cycles) which incurs a large performance penalty. For every context switch we have to budget for the worst-case outstanding I/O.

To reduce the impact of external I/O, each processor tile has a local instruction and data (also communication) memory. The operating system and its data structures, and the FIFO administration used in the communication API are thus always accessible from local memory, reducing the worst-case time to a few cycles. For the ARM11 the operating system is in a single cycle tightly couple memory (ITCM and DTCM) and for the MicroBlaze we use a local data memory (dmem) with similar functionality.

To reduce the bound on the interrupt latency it is possible to make all load/store operations local with the introduction of a DMA block [20]. Hence, instead of a potentially long load/store operation, the processor initiates a DMA transfer between the local and external memories, and polls until the DMA finishes the transfer. The processor is interruptible after each polling (local read) operation, thus the interrupt latency is *kept short* and is *independent* of the resource allocation of any other application. However, as all external memory access must take place via the DMA, the task data and instructions must fit in the local memory. Additionally, the DMA requires explicit access to remote memories, e.g. through the use of the API introduced in Section 3.3. By embedding the interaction with the DMA in the communication API its use is transparent to the application programmer.

The DMA offers improved performance, but places strict requirements on the communication model and requires the tasks to fit in the local memories. It is also possible to not use a DMA and thus avoid the aforementioned restrictions. The drawback is reduced performance, due to the potentially large bound on outstanding transactions. As a major advantage, the absence of the DMA allows a more flexible placement of data and instructions and also enables the use of caches, about which more presently.

### *4.4. Caches*

Caches have the ability to significantly improve processor performance. However, caches present a problem to CompOSe as the applications affect the cache state as they execute, but must not interfere with each other. From a composability point of view, we distinguish between two types of cache interference [32]:

- Intra-task (intrinsic) interference occurs when a task overwrites its own cache lines, mainly because of the relatively small size of the cache as compared with the tasks memory demands. Intra-task interference occurs on both single- and multi-tasking execution platforms.
- Inter-task (extrinsic) interference occurs when in a multitasking environment context switches swap out cache contents of previous applications, often resulting in a burst of cache misses.

Both intra- and inter-task interference make it hard to calculate worst-case execution times. To achieve composability, however, there is no need to bound execution times, and only the inter-task interference must be removed. In addition to the issues concerning composability, the inclusion of caches also raise the issue of cache coherency. CompOSe is tailored for a NoC-based MPSoC platform without hardware support for cache coherency. As a consequence that responsibility is shifted to the software. We show in Section 5.2 how composable cache sharing and software cache coherency is accomplished in CompOSe.

## 5. CompOSe

After having described the hardware architecture, we now describe the implementation of our proposed operating system. Note that each processor runs an *independent instance* of CompOSe, without any knowledge of the other processors in the system. Thanks to the NoC, each processor can run on its own clock and be completely decoupled from other processors and memories. Each scheduler takes local decisions, and is not aligned or synchronised with any other scheduler in the system. All task communication and synchronisation is using C-HEAP and is composable thanks to the NoC.

We start by describing the data structures used by CompOSe in Section 5.1 and continue by looking at the functionality in Section 5.2.

## *5.1. Data structures*

The key data structure elements of CompOSe are shown in Figure 4. At the top we have the Processor Control Block (PCB), followed by the Application Control Block (ACB), Task Control Block (TCB) and FIFO Control Block (FCB). The data structure for each processor is dynamically allocated on the heap (in the local memory of the processing element) during system initialisation (or reconfiguration). Note that there is *no system level* in the data structure. In other words, each processor is unaware of tasks or applications running on other processors, even tasks belonging to the same application. Consider for example Figure 2(c) where the $p_1$ is only aware of $a_1$ despite $a_2$ being part of the same application ($a$).

As seen in Figure 4 the application-level scheduler is hardcoded to TDM, with the period, slot length and schedule being part of the PCB. The PCB also holds the slack-distribution matrix, about which more in Section 5.3. The PCB also points to a circular linked list of applications. On the application level we see that each ACB has a function pointer, thus allowing it to have a per-application choice of task scheduler. Note, however, that the scheduler runs in the operating system execution time unit, which we return to in the following section, and must hence be trusted (and characterised). The ACB also holds information about all the FIFOs and tasks that reside on the processor in question.

The TCB contains pointers to instructions, stack and heap start of the task (in remote and possibly cached memory). Each TCB also points to its input and output FIFOs, so that CompOSe knows what conditions must be satisfied for the task to run. In addition to the tasks created by the user, there is always a default task, `os_idle`, that is connected to each application. This task, when

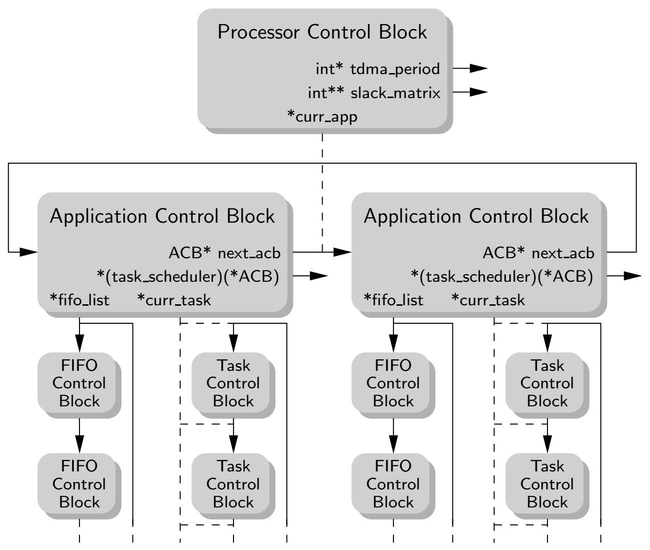


Figure 4: Data structure organisation.

invoked, immediately finishes (by returning). As we will see, the idle task is important for the slack management.

It is possible for a host processor to create and modify the data structure by manipulating the memory locations directly. We have also implemented a reconfiguration application that executes on each processor and receives reconfiguration messages from one or more hosts (currently for the ARM platforms only). We discuss this further in Section 6.

## *5.2. Functionality*

In this section we describe the functionality of CompOSe and how it makes use of the processor tile and data structures. The core of CompOSe is the functional loop shown in Figure 5. As seen in the figure, it consists of two major parts, the *Operating System (OS) unit* and the *service unit*. The operating system unit is responsible for saving the context of the previous task on its stack, to schedule a new application and along with it a new task. The service unit is where the task is allowed to execute. In the following sections we traverse the complete cycle and explain the individual steps.

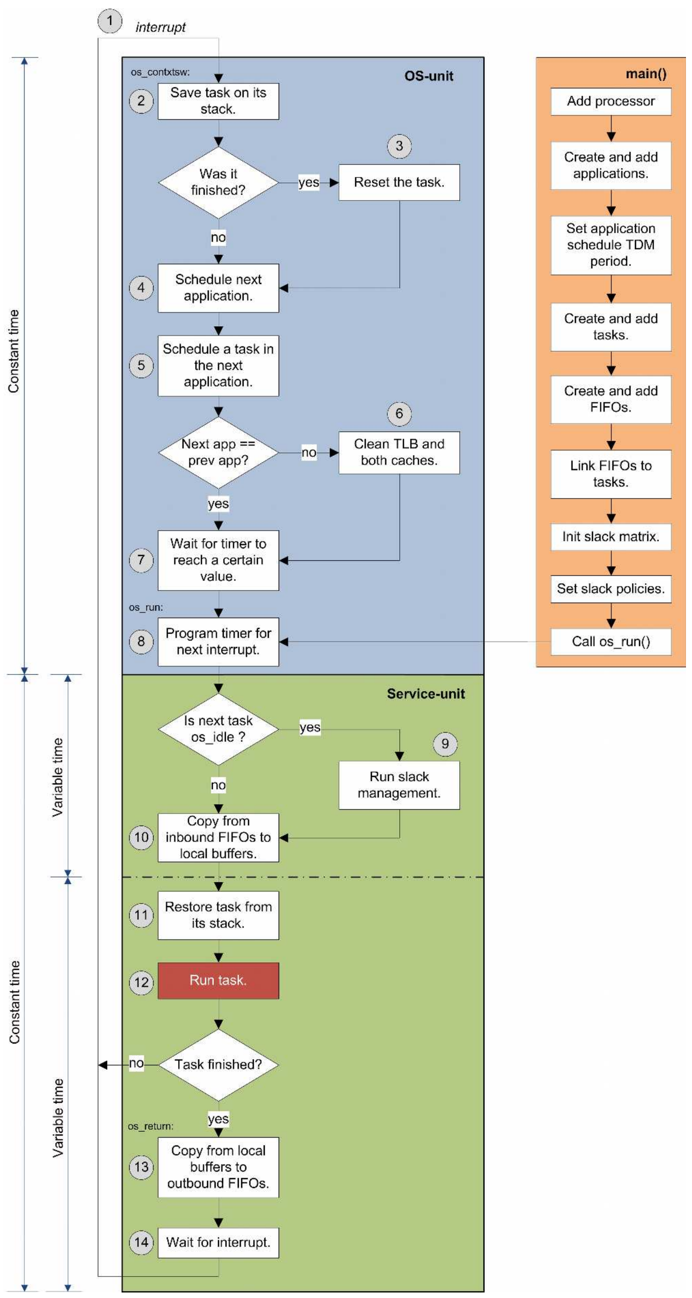


Figure 5: Functional flowchart.

### *5.2.1. Interrupt handling*

An interrupt from the system timer marks the start of a new cycle, marked with (1) in Figure 5. The processor switches to the corresponding execution mode, stores the current program counter, disables further interrupts and starts execution from the relevant exception vector. On our platform the execution mode entered is the fast interrupt mode for the ARM7 and ARM11. This is due to the system timer implementation explained in Section 4.1. The MicroBlaze uses normal IRQs. The exception vector contains a branch instruction forcing the program counter to the beginning of the `os_contxtsw` function.

### *5.2.2. Context saving and task reset*

In `os_contxtsw` (2), the context of the interrupted task is saved onto its stack and the stackpointer saved in the TCB. The task is reset to its original state (3), if it is marked as finished. The original state is defined as the state that the task was in when the system was first started. This implies resetting all the task registers, which are now located on its stack.

### *5.2.3. Application and task scheduler*

Next, the application-scheduler selects an application (4), using the TDM schedule of the PCB. The `curr_app` pointer in the PCB, is updated to point out the scheduled application. When the application-level scheduler has decided what application to run next, the task-level scheduler takes over (5).

The task scheduler is a per-application selectable algorithm that can use any scheduling strategy. The task- scheduler is specified via a function pointer, and takes a pointer to the calling application as argument. Three algorithms are implemented; Round-Robin, TDM and Credit Controlled Static Priority [33]. Round-Robin, TDM are compared in Section 6. When deciding on a task, the scheduler only considers tasks that are eligible to execute, i.e. tasks that *have data available in all input FIFOs and space available in all output FIFOs.* A task without FIFOs is always considered eligible. Note that all the available schedulers guarantee a minimum rate. A task that is eligible will consequently be scheduled eventually (in contrast to purely priority-based arbitration). Progress is thereby guaranteed on the task level (as well as on the message and packet level in the interconnect), thus ensuring *deadlock freedom* if all buffers are sized properly [7, 31].

### *5.2.4. Clean cache*

If the processor is using caches, they have to be dealt with to achieve composability. The easiest way of including caches, and the method CompOSe implements, is to clean and invalidate the instruction cache, data cache and Translation Look-aside Buffer (TLB) (6). This gives the tasks cold (empty) caches upon each activation, thus removing any influence of previous applications. It does, however, result in a burst of cache misses, lowering the execution speed. More complex ways to achieve composability while the caches are activated include cache partitioning [34] and cache locking. The official ARM compiler does not support cache partitioning, and cache locking restricts the number of tasks to the number of cache-ways (on our chosen ARM11 that gives a maximum of four tasks). Moreover, when the task does not entirely fit in the cache, intra-task interference will take place and the application composability is lost.

To reduce the performance impact of the cache invalidation we make it conditional and *do not clean the caches if the previous application scheduled is the same as the next one*. This removes the bursts of cache misses in this case, while keeping the composability among applications.

### *5.2.5. Constant execution time*

As mentioned, constant operating system execution time (7) is crucial for achieving composability. This worst-case operating system duration must accommodate all time required to service the interrupt, reset the task, run the application scheduler, run the task scheduler.

A possible way to remove the variation in duration, is to halt or clock gate the processor after the operating system execution, up to its worst-case duration.

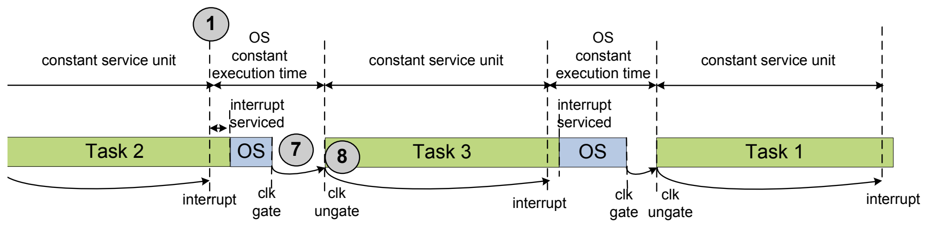


Figure 6: MicroBlaze service units and task switching time line

This is the approach used for our MicroBlaze processing element. When a halt instruction is not available, the processor can poll on a timer as in Listing 2. In both cases, i.e. end of halt instruction or the timer value reached, there are still variations due to the fact that the reading of the timer, and the loop that performs the check does not run in zero (or even constant) time. The processor might leave the loop up to seven cycles after the desired value has been reached. This *polling window* is not dependent on the application, but rather the uncertainty of the platform and does not have to be eliminated to achieve composability. However, to verify our implementation we choose to also remove this effect and thus enable us to demonstrate composability by looking at the cycle-level behaviour, as shown in Section 6.

It is very important that the code is located in a memory with zero wait-states and that the targeted processor executes NOP instructions in a single cycle. It is also important that the system timer runs at the same speed as the processor, otherwise the cycle accurate control is lost.

In case the timer is external and the processor clock can be controlled (e.g. in the MicroBlaze tile) instead of polling on the timer, the operating system variations can be removed by gating the clock of the processor till an absolute moment in time. Figure 6 presents the time line with the main events in task switching (using the enumeration from Figure 5: (1) timer interrupt raise, (7) waiting up to operating system worst-case duration, and (8) program timer for next interrupt, as described in the functional flow.

As the last step of the operating system unit the timer is programmed for

```
    // Wait for timer to reach r1
    LDR r1, =0x2000
loop
    BL read_timer
    CMP r0 , r1 // timer value is returned in r0
    BLS loop

    // Do NOP instruction to hide the polling window.
    SUB r1, r0, r1
    MOV r1 , r1 , LSL #2
    ADD pc, pc, r1
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP

    // Write next interrupt to timer
    BL reset_timer
    BL os_get_time_base
    BL init_timer
    // Next tasks service unit starts now
```

Listing 2: Timer poll

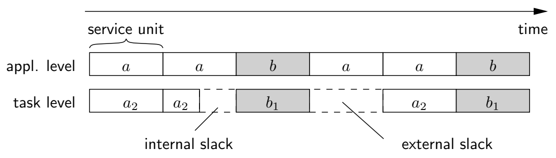


Figure 7: Schedule with internal and external slack.

the next interrupt (8) and the execution continues with the service unit.

## *5.3. Slack management*

One of the drawbacks with the fixed-size time slices (central to the ability to provide composability) is that slots might be left unused. For this reason, we provide slack management as an optional addition to CompOSe. At the start of the service unit, the slack manager is invoked (9) if the next task is `os_idle`. In CompOSe we distinguish between two types of slack: internal and external. Internal slack arises when a task finishes its work (firing) before the end of a service unit. In Figure 7 $a_2$ in application $a$ finishes in the middle of the service unit, leaving the processing element idle until the next scheduling decision. External slack, on the other hand, is introduced when an application is scheduled, but has no eligible tasks to execute. As a prerequisite, the operating system must be aware of the eligibility, i.e. firing rules of the task. Figure 7 also illustrates this case, where a complete service unit is spent idle due to the lack of input data or output space for $a_2$.

CompOSe is able to detect and distribute the external slack using a slack distribution graph, as shown in Figure 8. The slack distribution graph defines which application can give slack which to other applications, and it is determined at application initialisation, with information from the application designer. A key observation for slack management in a composable system is that interference is an *unidirectional relation.* The application that offers its unused resources is not affected by other applications. Conversely, the execution of the application that receives the slack suffers from interference from the slack-donating applications. Hence it is possible to have a system where some

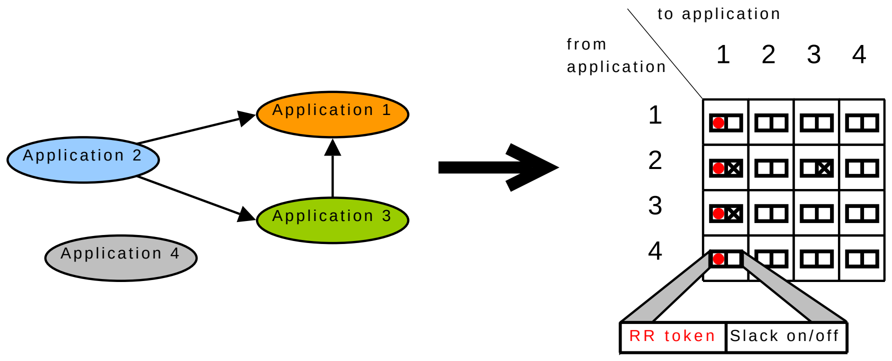


Figure 8: Slack distribution graph and slack matrix with Round-Robin (RR) tokens.

applications are composable (i.e. have no interference from other applications), and other applications are not composable, as they receive slack.

The distribution of internal slack to another task (potentially belonging to another application) would imply an extra scheduling and slack management decision (OS invocation) at task iteration finish. To respect the fixed size time slots, this OS invocation should not take place if the internal slack is not larger than the OS slice. However, the amount of internal slack is known only at run time, thus managing the internal slack may complicate the application timing analysis for which the number of OS invocation should be known as design time. Moreover, extra OS invocations lead to extra overhead. Hence CompOSe does not manage internal slack similarly to the external slack. However, for a processor with an external timer and controllable clock frequency unit (as in our MicroBlaze based platform) the processor can be clock gated to utilise internal slack to save power.

As a result of the slack manager CompOSe may schedule an eligible task in the current service unit instead of idling.

### *5.3.1. Buffer management and task execution*

Before executing the task, CompOSe potentially copies input data from remote locations to local buffers (10). This step is optional and is a user choice. It is possible to use remote buffers cached (we invalidate cache lines when acquiring data) or non-cached, but the best performance is achieved if the buffers fit in local memory and no additional copying is necessary.

Once the input data is available, the task is restored (11) and execution from its previous state (12). This is where the actual user task code is run.

If the task returns before the timer interrupt, then CompOSe optionally copies the output data from local memory to the physical location of the output FIFOs. Once the data is written to the target memory, the local and remote FIFO administration is updated. If both the data and the remote administration are in the same memory the ordering of data and synchronisation transaction is guaranteed. This is, however, not the case when data and synchronisation have different QoS budgets in the interconnect [35]. For this purpose we include an ARM *Data Memory Barrier (DMB) operation* in the release call, or a read back of the last written value for processors without such functionality. This ensures that all outstanding explicit memory transactions (i.e. to the FIFO buffer) complete before any following explicit memory transactions begin (i.e. to the FIFO administration). As our platform implements cache coherency in software we also flush cache lines when data is released. The implementation is hidden in the communication API, and is transparent to the user.

Once the I/O is complete CompOSe continues to wait for the interrupt (14) marking the end of the service unit.

## 6. Experimental results

In this section we put CompOSe to the test and demonstrate three different instances using a range of processors and tile architectures, as introduced in Section 4. First, in Section 6.1 we present an ARM7 single-processor system illustrating the effects of choosing different task-level schedulers. Next, we continue with an ARM11 multi-processor system with caches in Section 6.2 and show how CompOSe delivers composability on a cycle level. Finally in Section 6.3 we show similar results for an FPGA implementation built around MicroBlazes.

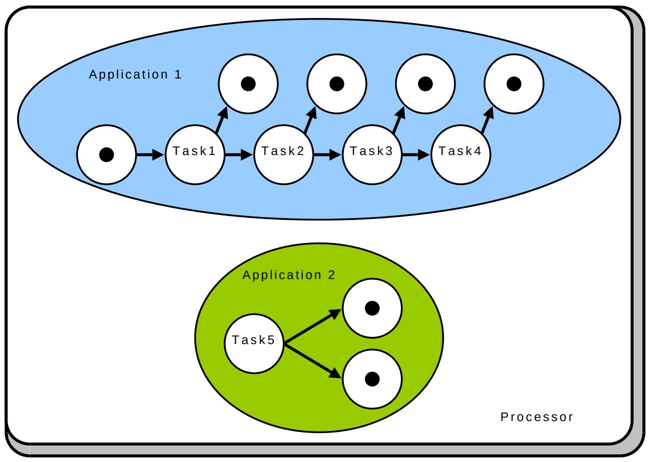


Figure 9: Two applications mapped on to the ARM7 processor.

### *6.1. Single-processor board implementation*

The microcontroller used in this experiment is a NXP LPC2129, including one ARM7 core together with a variety of peripherals, e.g. 256 KB on-chip Flash ROM and 16 KB RAM, vectored interrupt controller, real-time clock and general purpose I/O pins. The rather big ROM in the LPC2129 can hold task code together with the complete CompOSe. This memory is not single cycle, thus by scatter gathering time critical code (the operating system and the schedulers) are executed from RAM. The initialisation code copies the time critical code from ROM into RAM before calling main.

A small CompOSe console application has been developed. It allows editing of the application schedule, and also the slack-matrix. The console application uses a serial-port connection for communication with the outside world, allowing a PC to be used as a host. The console application can be mapped into a free TDM slot without interfering with any other applications, but we choose to make it a strict best-effort application and let it run purely on slack. For this to work there has to be a sufficient amount of slack that the console application can utilise.

```
void Task1(int out[1]){
  int workload = read_adc();
  out = workload;
  LED = 1;
  // Simulated work
  while(workload > 0){workload--};
  LED = 0;
}
```

Listing 3: Task code

### *6.1.1. Scheduler comparison*

Here, two simple task-schedulers are presented, using two applications as illustrated in Figure 9. Application 1 is a LED application where the first task reads a potentiometer, letting the value change the workload, as shown in Listing 3. The on-board LED corresponding to the task is switched on and upon completion the LED is switched off again. The workload is communicated to the following task in the pipeline that switches on the corresponding LED and passes the value along to the next task, etc. Application 2 is a simple sound generator that toggles two I/O ports to drive an on-board buzzer. We can thus change the workload of Application 1 (through the potentiometer) and observe the frequency of the sound omitted, and this way hear if there is any change in Application 2.

In addition to merely listening to the effects of the schedulers, the traces from tasks 1 through 4 in Figure 10 and Figure 11 illustrate the state of the LED variable from Application 1. The LED variable will stay at 1, even if the task is pre-empted. The traces shows how this workload is transported through a pipeline among the four tasks. The difference between the two simulations is the task-scheduler used in application 1. Notice that task 5 in application 2, is not affected by the switch of task-scheduler in application 1. The smaller latency introduced by the round-robin scheduler can be observed in the graphs.

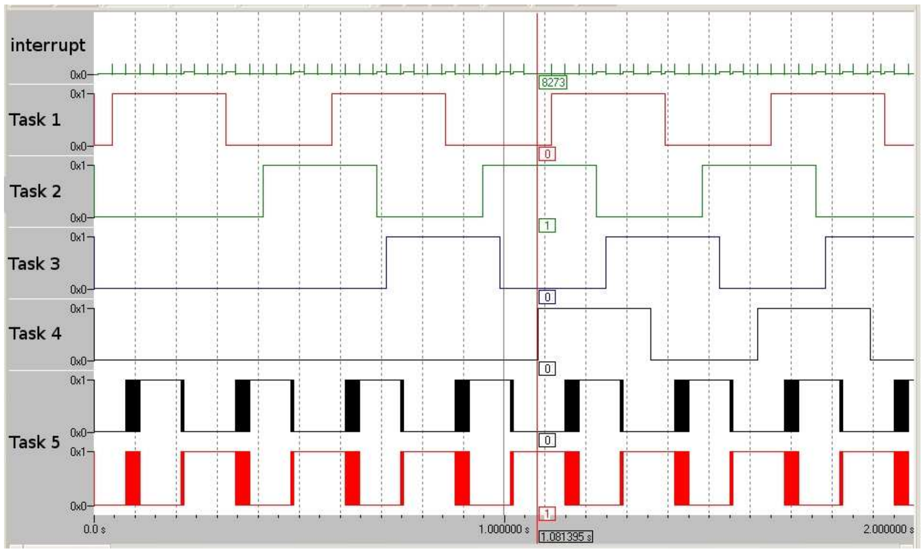


Figure 10: Task invocation using TDM task scheduler

### *6.1.2. Performance*

The CompOSe overhead is processing time consumed by the operating system itself, i.e context switching and scheduling decisions, including the time required to wait until the worst-case in-flight instructions finish and resetting the processor state. A context switch, without slack management, takes about 1600 ARM assembly instructions. The small overhead is due to the local memories (small worst-case time for in-flight instructions) and the absence of caches and TLBs. On the ARM7 implementation, running at 60MHz, the overhead for using CompOSe when using a 100 Hz system tick is only 0.3%. Raising the clock to 1kHz gives, due to linear scaling, 3% overhead.

## *6.2. Multi-processor system netlist*

In addition to the single-processor implementation, CompOSe is evaluated on a ARM11-based MPSoC, for which the netlist is available. The system contains three ARM11 processors and a large external memory. In contrast to the single-processor system, this evaluation also includes the inter-processor communication through C-HEAP. To verify composability we map two applications

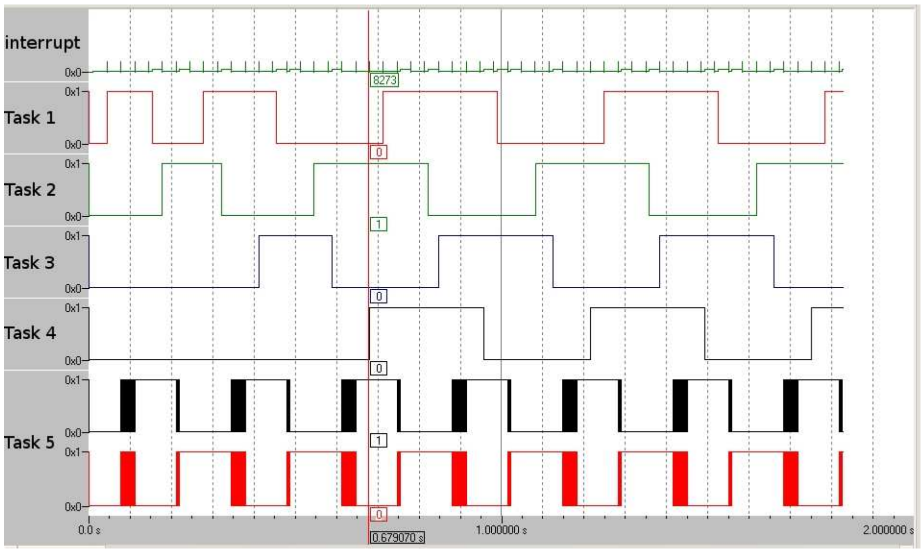


Figure 11: Task invocation using round-robin task scheduler

to the three processors, similar to what is shown in Figure 2(c). Next we run two simulations, where the second application is modified between the two runs. Traces in Figure 12 show the interface of the middle processor, executing tasks from both applications. Traces from two different simulations are overlaid. The diagonally striped (red) area indicate cycles that differ between the two. The nFIQ signal indicates where the service cycle starts and stops. The comparison between the two traces clearly shows that the only differences take place in the time slots of the changed applications (third and fourth service cycle) and in the operating system unit when accessing the data structure (in the beginning of each service cycle). Seeing that the behaviour on a cycle level remains the same clearly indicates that temporal composability is achieved by CompOSe and the NoC (and memories).

## *6.3. Multi-processor system FPGA implementation*

We implement CompOSe also on a MicroBlaze-based FPGA prototype. This experimental platform consist of two processor tiles as described in Section 4, communicating through an Æthereal network on chip [10]. The workload ex-

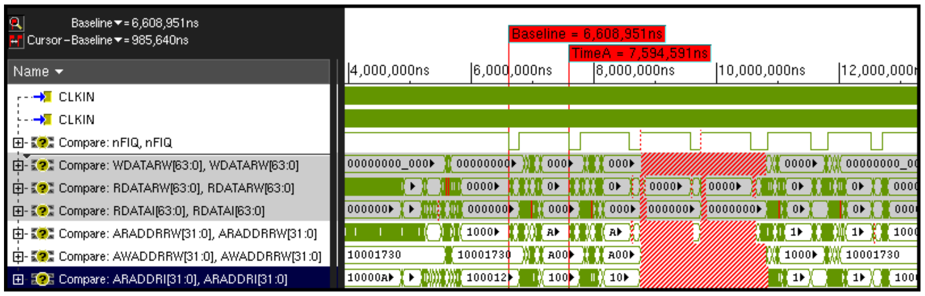

Baseline = 6,608,951ns
Cursor – Baseline = 985,640ns
Name
CLKIN
CLKIN
Compare: nFIQ, nFIQ
Compare: WDATARW[63:0], WDATARW[63:0]
Compare: RDATARW[63:0], RDATARW[63:0]
Compare: RDATAI[63:0], RDATAI[63:0]
Compare: ARADDRRW[31:0], ARADDRRW[31:0]
Compare: AWADDRRW[31:0], AWADDRRW[31:0]
Compare: ARADDRI[31:0], ARADDRI[31:0]
Baseline = 6,608,951ns
TimeA = 7,594,591ns
4,000,000ns
6,000,000ns
8,000,000ns
10,000,000ns
12,000,000ns

Figure 12: Signal trace ARM11.

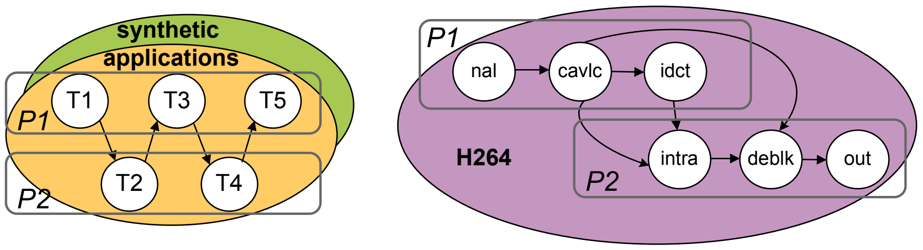


Figure 13: Applications executed on the MicroBlaze-based platform.

ercised on this platform consists of 3 applications: 2 synthetic ones (*A1*, *A2*), having the same task graph structure but different execution times, and a parallel H264 decoder (*H264*) obtained using PNGen [36]. Figure 13 presents the task graphs of these applications and their processor mapping. *H264* and *A2* are scheduled using TDM, and *A1* is scheduled using Round-Robin.

We measured the execution time of the *H264* tasks in two cases: (1) the *H264* executing alone on the platform, and the *H264* executing together with *A1* and *A2*. We observed that the *execution times* of each one of the *H264* tasks are *identical*, regardless the presence or absence of *A1* and *A2* in the system. This suggests that temporal composability is achieved.

Similar simulation traces as in the previous subsection are compared also for the MicroBlaze platform in Figure 14. We compare 5 signals of one MicroBlaze core in two different runs. The diagonally striped (red) area indicate cycles that differ between the two. In first run *A1* is scheduled using Round-Robin, and in the second one it is scheduled with TDM. The `int_out` signal indicates

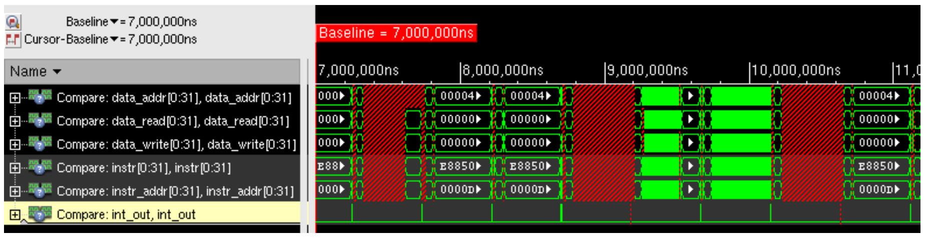

Baseline = 7,000,000ns
Cursor-Baseline = 7,000,000ns
Name
Compare: data_addr[0:31], data_addr[0:31]
Compare: data_read[0:31], data_read[0:31]
Compare: data_write[0:31], data_write[0:31]
Compare: instr[0:31], instr[0:31]
Compare: instr_addr[0:31], instr_addr[0:31]
Compare: int_out, int_out
Baseline = 7,000,000ns
7,000,000ns
8,000,000ns
9,000,000ns
10,000,000ns

Figure 14: Signal trace MicroBlaze

where the service cycle starts and stops. One can see that some of the service units completely differ because in those units different tasks of *A1* execute in the two different runs. However, the rest of the service units are cycle-level identical, as *A2* and *H264* have the same scheduling policy over the two runs. Thus this traces comparison suggests that the system is temporally composable at cycle-level.

On the MicroBlaze, the worst case execution time of CompOSe (when scheduling 3 applications, each having at most 5 tasks) is 1300 cycles, representing an overhead of 6.5%, when the service unit is 20000 cycles long, as in the experiments of this subsection.

Our empirical evidence does not *prove* the ability to provide composable processor sharing. However, by having multiple different hardware and software instances, our experiments cover a large space of compilers, processor architectures, and applications. The many design points together serve as a *strong indication* that our goals are achieved.

## 7. Conclusions

In this work we introduce CompOSe, an operating system that enables composable sharing of processors, extending an existing network-based composable hardware platform with hardware and software support. With a temporally composable system, on both a hardware and a software level, we reduce the design and verification effort by a divide-and-conquer approach. The need for verification is reduced from the system level, down to an application level.

CompOSe uses a novel concept based on scheduling of fixed-size *service units*, implemented by means of pre-emptive scheduling using a *budget-enforcing scheduler*. In contrast to many other operating systems, an application need not be characterised, only adhere to the task interface with explicit communication. CompOSe also provides slack management, and uses a novel two-level arbitration scheme to separate inter- and intra-application arbitration. We demonstrate CompOSe on a range of processor architectures and show its applicability in network-based multi-processor systems with release consistency, software cache coherency and distributed memories. Our experiments, using netlist simulation and an FPGA prototype, suggest that temporal composability is achieved.

## References

[1] M. Azimi, N. Cherukuri, D. Jayashima, A. Kumar, P. Kundu, S. Park, I. Schoinas, A. Vaidya, Integration challenges and tradeoffs for tera-scale architectures, Intel Technology Journal 11 (3).

[2] R. Obermaisser, Integrating automotive applications using overlay networks on top of a time-triggered protocol, LNCS 4888.

[3] S. Dutta *et al.*, Viper: A multiprocessor SOC for advanced set-top box and digital TV systems, IEEE Design and Test of Computers.

[4] B. Smith, ARM and Intel battle over the mobile chip's future, Computer 41 (5).

[5] S. Gal-On, Multicore benchmarks help match programming to processor architecture, in: MultiCore Expo, 2008.

[6] J. Owens, W. Dally, R. Ho, D. Jayasimha, S. Keckler, L.-S. Peh, Research challenges for on-chip interconnection networks, IEEE Micro 27 (5).

[7] A. Hansson, K. Goossens, M. Bekooij, J. Huisken, Compsoc: A template for composable and predictable multi-processor System on Chips, ACM Trans. on Design Automation of Electronic Systems 14 (1).

[8] A. Jantsch, Models of computation for networks on chip, in: Proc. ACSD, 2006.

[9] Avionics Application Software Standard Interface, ARINC Specification 653 (January 1997).

[10] A. Hansson, M. Subbaraman, K. Goossens, aelite: A flit-synchronous network on chip with composable and predictable services, in: Proc. DATE, 2009.

[11] B. Akesson, A. Hansson, K. Goossens, Composable resource sharing based on latency-rate servers, in: Proc. DSD, 2009.

[12] G. C. Buttazo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Kluwer Publishers, 2004.

[13] J. Sales, Symbian OS Internals, John Wiley & Sons, 2005.

[14] C. L. Liu, J. W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, Journal of the ACM 20 (1).

[15] M. Dertouzos, Control robotics: The procedural control of physical processes, in: Proc. IFIP Cong., 1974.

[16] [link].
URL `http://www.virtuallogix.com`

[17] G. Heiser, The role of virtualization in embedded systems, in: IIES '08: Proceedings of the 1st workshop on Isolation and integration in embedded systems, ACM, 2008, pp. 11–16.

[18] M. Steine, M. Bekooij, M. Wiggers, A priority-based budget scheduler with conservative dataflow model, in: Proc. DSD, 2009.

[19] S. Sriram, S. Bhattacharyya, Embedded Multiprocessors: Scheduling and Synchronization, CRC Press, 2000.

[20] A. Molnos, A. Milutinovic, D. She, K. Goossens, Composable processor virtualization for embedded systems, in: Proc. CAOS, 2010.

[21] K. Goossens, A. Hansson, The aethereal network on chip after ten years: Goals, evolution, lessons, and future, in: Proc. DAC, 2010.

[22] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, M. Bekooij, Enabling application-level performance guarantees in network-based Systems on Chip by applying dataflow analysis, IET Computers and Design Techniques.

[23] B. Akesson, A. Hansson, K. Goossens, Composable resource sharing based on latency-rate servers, in: Proc. DSD, 2009.

[24] ARM Limited, AMBA AXI Protocol Specification (2003).

[25] 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, Design Automation for Embedded Systems 7 (3).

[26] A. Hansson, K. Goossens, An on-chip interconnect and protocol stack for multiple communication paradigms and programming models, in: Proc. CODES+ISSS, 2009.

[27] T. Bijlsma, M. Bekooij, P. Jansen, G. Smit, Communication between nested loop programs via circular buffers in an embedded multiprocessor system, in: Proc. SCOPES, 2008.

[28] S. van Haastregt, B. Kienhuis, Automated synthesis of streaming C applications to process networks in hardware, in: Proc. DATE, 2009.

[29] A. Hansson, B. Akesson, J. van Meerbergen, Multi-processor programming in the embedded system curriculum, ACM SIGBED Review 6 (1).

[30] S. Stuijk, M. Geilen, T. Basten, A predictable multiprocessor design flow for streaming applications with dynamic behaviour, in: Proc. DSD, 2010.

[31] M. J. G. Bekooij, G. J. M. Smit, Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs, in: Proc. DAC, 2007.

[32] F. Sebek, The state of the art in cache memories and real-time systems, Tech. rep., Märdalen University, Sweden (2001).

[33] B. Akesson, L. Steffens, E. Strooisma, K. Goossens, Real-time scheduling using credit-controlled static-priority arbitration, in: Proc. Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), IEEE Computer Society, Washington, DC, USA, 2008, pp. 3–14. doi:http://doi.ieeecomputersociety.org/10.1109/RTCSA.2008.21.

[34] A. Molnos, M. Heijligers, S. Cotofana, J. van Eijndhoven, Compositional, efficient caches for a chip multi-processor, in: Proc. DATE, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 2006, pp. 345–350.

[35] E. Larsson, B. Vermeulen, K. Goossens, Distributed architecture for checking global properties during post silicon debug, in: Proc. European Test Symposium (ETS), 2010.

[36] S. Verdoolaege, H. Nikolov, T. Stefanov, PN: a tool for improved derivation of process networks, EURASIP J. Embedded Syst. 2007 (1).