CoMPSoC: A Template for Composable and Predictable Multi-Processor System on Chips

ANDREAS HANSSON

Eindhoven University of Technology, Eindhoven, The Netherlands

KEES GOOSSENS

NXP Semiconductors, Eindhoven, The Netherlands

Delft University of Technology, Delft, The Netherlands

MARCO BEKOOIJ

NXP Semiconductors, Eindhoven, The Netherlands

and

JOS HUISKEN[1]

Silicon Hive, Eindhoven, The Netherlands

A growing number of applications, often with firm or soft real-time requirements, are integrated on the same System on Chip, in the form of either hardware or software intellectual property. The applications are started and stopped at run time, creating different use-cases. Resources, such as interconnects and memories, are shared between different applications, both within and between use-cases, to reduce silicon cost and power consumption.

The functional and temporal behaviour of the applications is verified by simulation and formal methods. Traditionally, designers resort to monolithic verification of the system as whole, as the applications interfere in shared resources, and thus affect each other's behaviour. Due to interference between applications, the integration and verification complexity grows exponentially in the number of applications, and the task to verify correct behaviour of concurrent applications is on the system designer rather than the application designers.

In this work, we propose a Composable and Predictable Multi-Processor System on Chip (CoMPSoC) platform template. This scalable hardware and software template removes all interference between applications through resource reservations. We demonstrate how this enables a divide-and-conquer design strategy, where all applications, potentially using different programming models and communication paradigms, are developed and verified independently of one another. Performance is analysed per application, using state-of-the-art dataflow techniques or simulation, depending on the requirements of the application. These results still apply when the applications are integrated onto the platform, thus separating system-level design and application design.

Categories and Subject Descriptors: B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

---

[1]Jos Huisken is currently affiliated with IMEC-NL, Eindhoven, The Netherlands
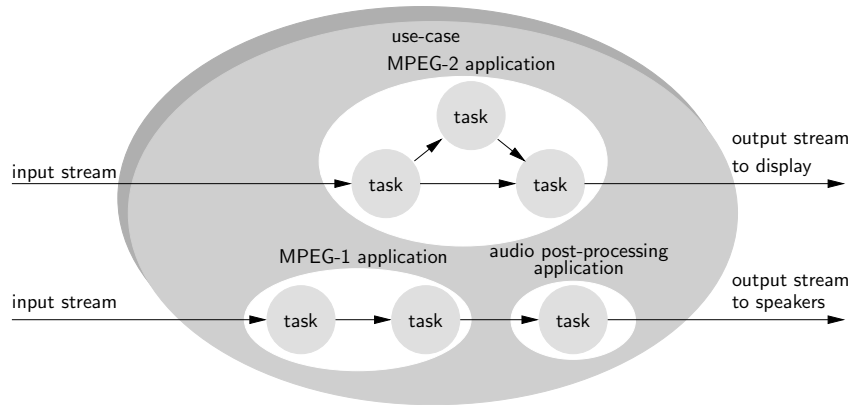

Fig. 1.   Application model.

General Terms: Design, Performance, Verification

Additional Key Words and Phrases: Composable, Predictable, Model of Computation, System on Chip, Network on Chip.

---

1.   INTRODUCTION

Systems on Chip (SoC) grow in complexity with an increasing number of independent *applications* integrated on a single chip [Dutta *et al.* 2001; Rutten et al. 2005]. The applications are realised by hardware and software Intellectual Property (IP), e.g. processors and application code, that is reused across platform generations and instances [Keutzer et al. 2000]. As exemplified in Figure 1, applications are often split into multiple *tasks* running concurrently, either to improve the power dissipation [Rowen and Leibson 2004] or to meet *Real-Time (RT) requirements* that supersede what can be provided by a single processor [Sasaki 1996].

The individual applications have different RT requirements [Buttazo 1977]. For Firm Real-Time (FRT) applications, e.g. a Software-Defined Radio [Moreira et al. 2007] or the audio post-processing filter in Figure 1, deadline misses are *highly undesirable* due to standardisation, e.g. upper bounds on the response latency in wireless standards, or steep quality reduction in the case of misses. Note that FRT only differs from *hard RT*, a term widely used in the automotive and aerospace domain, in that it does not involve *safety* aspects. Soft Real-Time (SRT) applications, e.g. the MPEG-2 decoder in our example, can tolerate occasional deadline misses with only a modest quality degradation. In addition, some applications have No Real-Time (NRT) requirements, e.g. a user interface, and must only be functionally correct.

To verify the *functional* and *temporal* behaviour of an application, *the platform*, i.e. the architecture, the low-level drivers and the middleware, together with *the mapping*, has to be modelled [Jantsch 2006; Bekooij et al. 2004] or simulated [Martin 2006; Jerraya et al. 2006]. The verification methodology is strongly coupled to the type of application. For FRT applications, the *worst-case* analysis must cover

all input streams, under all behaviours of the hardware and software components (arbiters, memories, processors, etc) used by the application. SRT applications, on the other hand, require a *probabilistic* analysis of the temporal behaviour, e.g. to reach a specific deadline miss rate or average-case performance. For NRT applications, it suffices to verify a correct functional behaviour. The system designer has to integrate *all* applications, usually developed by different suppliers, and verify *the combined behaviour* [Rumpler 2006].

Traditionally, the analysis cannot be done on applications in isolation, due to *interference* in shared resources, e.g. interconnect and memories. The interference couples the temporal, and potentially also functional, behaviours of the applications, thus making the burden of verifying external application IP the responsibility of the system designer. The task is further complicated by applications with *input-dependent behaviour* and applications that are started and stopped at run time, creating many different *use-cases*. The complexity of *monolithic* system analysis and verification is the main challenge [Jantsch 2006; Rumpler 2006], and is together with cost pressures far outgrowing the capacity of semiconductor scaling and incremental design productivity improvement [Rowen and Leibson 2004].

To cope with the complexity of system design, there exist two alternatives: *abstraction* and *partitioning* [Rumpler 2006]. With abstraction, complexity is reduced by moving to a *higher abstraction level* [Jerraya et al. 2006], which typically trades speed for accuracy, or by abstracting the system through a *Model of Computation* (MoC) [Jantsch 2006]. Unfortunately, it is often difficult if not impossible to find *one* abstraction that captures the whole system (applications, platform and mapping) and allows reasoning about relevant metrics. Often a mix of applications, using different programming models and communication paradigms, and having varying RT requirements, are mapped to the platform [Martin 2006]. For FRT applications, it suffices to have a MoC that is *monotonic* [Poplavko *et al.* 2003], i.e. free of *scheduling anomalies* [Graham 1969], and a *predictable* platform that *bounds* the interference between applications. Upper bounds on interference are, however, not sufficient to analyse SRT and NRT applications in *isolation*. The observed behaviour, e.g. deadline miss rate, depends on the other applications in the system. Therefore, in addition to abstraction, we need partitioning, i.e. the means to split the large system into a number of independent parts, each of which is easier to understand than the whole system [Rumpler 2006].

The key characteristic that we believe is necessary to mitigate integration complexity is partitioning by means of *composability* [Kopetz 1997]. We define a system as composable if the functional and temporal behaviour of an application is the same, irrespective of the presence or absence of other applications in the system. Composability is essential to the ability to effectively design systems [Ivimey-Cook 1999; Jantsch 2006; Kopetz 1997; Bekooij et al. 2004] as it does not only bound, but completely *eliminates* interference between applications and enables *incremental design, integration and verification*. Composability is a well-known concept in the automotive and aerospace industry [Avionics Application Software Standard Interface 1997], where systems are traditionally designed using *federated architectures*, with one function per Electronic Control Unit (ECU) [Rumpler 2006]. Composability is thus achieved by *not sharing* any resources between applications. This
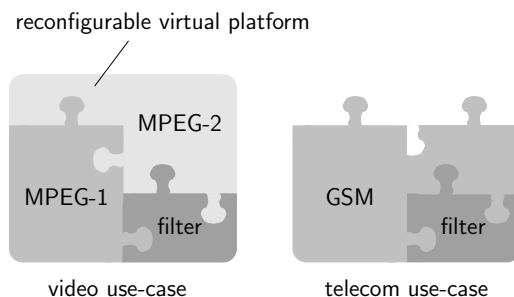
reconfigurable virtual platform

Fig. 2.   Reconfigurable composability.

approach is often too costly for the consumer-electronics domain, and also the au-
tomotive industry is moving in the direction of *integrated architectures* [Rumpler
2006], where ECUs are shared between applications. This requires architectural
building blocks that *enable composability* and tools that provide it to the applica-
tions by *assigning resources*.

As the main contribution of this work, we introduce a template for Composable
and Predictable Multi-Processor System on Chips (CoMPSoC), where each applica-
tion is given its own *reconfigurable virtual platform*, as shown in Figure 2. We show
how the building blocks in the shared SoC infrastructure implement composability
and predictability by employing *admission control and budget enforcement* [Rajku-
mar et al. 1998; Mercer et al. 1994]. Thereby, interference is eliminated, making
even the cycle-level behaviour of an application independent of all other appli-
cations, within and across use-cases [Hansson et al. 2007b], without placing any
requirements on the applications. On a platform instance, mapped to FPGA, we
integrate two applications, an image decoder and an audio post-processing filter,
and we show: 1) the importance of having both predictability and composability,
2) how the applications can be analysed and verified independently of one another,
using e.g. simulation-based techniques [Jerraya et al. 2006] for SRT applications
and *dataflow analysis techniques* [Sriram and Bhattacharyya 2000] for FRT appli-
cations, and 3) how the results still apply when the applications are integrated onto
the platform.

The remainder of the paper is structured as follows. We start by introducing
related work in Section 2. Next, the problem domain is described in Section 3.
Section 4 gives an overview of the CoMPSoC platform and the concepts behind it.
Thereafter, the architectural elements of the hardware platform are presented in
Section 5, followed by a description of the software platform in Section 6. We show
how these building blocks come together in our SoC design flow in Section 7. In
Section 8, we demonstrate the benefits of CoMPSoC by mapping two applications
to a platform instance. Finally, conclusions are drawn in Section 9.

2.   RELATED WORK

We review related work in a bottom-up fashion, starting from the NoC architecture,
and moving towards the system-level architecture and design flow.

Much work focus on generating, exploring, evaluating, and comparing NoC ar-

chitectures and instantiations [Benini 2006; Genko et al. 2005; Bartic et al. 2004; Moraes et al. 2004; Goossens et al. 2005; Jantsch 2006; Liang et al. 2000], a few of which have been demonstrated on FPGA [Genko et al. 2005; Bartic et al. 2004; Moraes et al. 2004; Kumar et al. 2007]. Most networks do not offer composability, as they implement only NRT services [Genko et al. 2005; Moraes et al. 2004] or employ arbitration schemes that bound but do not eliminate interference between applications [Bartic et al. 2004; Benini 2006]. Composable NoC architectures are introduced in [Jantsch 2006; Goossens et al. 2005; Liang et al. 2000]. The works, however, focus only on the NoC internals.

A large body of research presents application-specific [Vercauteren et al. 1996; Baghdadi et al. 2001] and domain-specific [Leijten et al. 2000; Nieuwland et al. 2002; Kopetz et al. 2007; Bekooij et al. 2004] multi-processor architecture templates. Of these works, a majority focus on the signal-processing domain [Vercauteren et al. 1996; Leijten et al. 2000; Nieuwland et al. 2002; Bekooij et al. 2004], while [Kopetz et al. 2007] is tailored for safety-critical applications in the automotive and aerospace domain. A specific communication model and protocol is used in [Vercauteren et al. 1996; Leijten et al. 2000; Nieuwland et al. 2002; Bekooij et al. 2004], whereas [Baghdadi et al. 2001] leaves this choice open. In addition to the hardware architecture, [Nieuwland et al. 2002] and [Baghdadi et al. 2001] present libraries for synchronisation and communication between tasks. The architecture templates in [Vercauteren et al. 1996; Leijten et al. 2000; Baghdadi et al. 2001; Nieuwland et al. 2002] are not composable, and hence introduce resource dependencies between the applications in the system.

Composable MPSoC architecture templates are presented in [Bekooij et al. 2004; Kopetz et al. 2007]. The *time-triggered* platform proposed in [Kopetz et al. 2007] includes error correction and redundancy, which requires logical synchronicity and a global notion of time. Moreover, the RT guarantees and composability requires a priori-known worst-case execution times for the tasks, and communication that is taking place at a priori-determined instants [Kopetz and Bauer 2003]. Additionally, the work uses an idiosyncratic communication protocol, and does not specify a *memory consistency model*. Similar to [Bekooij et al. 2004], we consider *event-triggered* systems, optimised for *streaming communication*, but with support for a range of communication paradigms and programming models, where *back pressure* [Wiggers et al. 2007], as a result of full buffers, is taken into account, and where we can deliver conservative performance guarantees based on dataflow analysis techniques. Extending both [Bekooij et al. 2004] and [Kopetz et al. 2007], we demonstrate an actual composable and predictable MPSoC implementation, realised on FPGA, with applications mapped to it, and show how to implement a composable architecture, without a global notion of time, and without placing any restrictions on the applications.

In this work, we introduce the CoMPSoC template and demonstrate the divide-and-conquer design methodology it enables. The work builds on an existing design flow [Kumar et al. 2007] that uses a composable NoC [Goossens et al. 2005] and a customisable processor core [Silicon Hive 2007]. In contrast to [Kumar et al. 2007] that focuses only on architecture generation, we combine the NoC and processor cores with: 1) a composable memory controller, thus forming a complete tile-based

MPSoC architecture, with 2) platform support libraries [Hansson and Goossens 2007], 3) libraries for synchronisation and communication [Nieuwland et al. 2002], and 4) tools for formal verification [Wiggers et al. 2007]. We demonstrate how the resulting CoMPSoC platform simplifies system-level design by mapping actual applications to a platform instance.

3. PROBLEM DESCRIPTION

Our aim is to provide a hardware and software platform that:

(1) enables performance guarantees for RT applications,

(2) allows independent analysis and verification of all applications,

(3) supports multiple application domains, and

(4) offers run-time reconfigurability for many use-cases.

To enable RT analysis (Item 1), we use a *predictable* architecture where it is possible to guarantee lower bounds on performance, further discussed in Section 4.1, and formal analysis using a MoC that is *monotonic* [Poplavko *et al.* 2003] and captures the applications, the platform and the mapping decisions with a good accuracy, i.e. with a tight worst case [Moonen et al. 2007].

Independent application analysis and verification (Item 2) is accomplished by a *composable* architecture that eliminates all interference between applications, as discussed in Section 4.2. Composability has stringent implications for the management of all resources that are shared by multiple applications, in our case: 1) the interconnect, and 2) shared on-chip and off-chip memories. Note that the *processor tiles are not shared* between applications in this work, a limitation that is elaborated on in Section 5.3.

The application support (Item 3) is addressed by: 1) not basing the RT guarantees and isolation on the concepts of a time-triggered architecture, but instead use an *event-triggered architecture with budget enforcement* that does not place any restrictions on the applications, 2) using *industry-standard* interfaces and programming languages, 3) offering a range of *communication paradigms*, i.e. Distributed Shared Memory, Message Passing and stream-based communication, 4) adhering to a well-known *memory consistency model*, and 5) supplying platform *middleware* for inter-processor communication. The aforementioned properties are returned to in Sections 5 and 6, where we present the building blocks of the hardware and software platform, respectively.

Run-time reconfigurability (Item 4) is offered by: 1) basing the platform on *software-programmable* hardware components, i.e. the processors and the SoC infrastructure, and 2) by allocating resources such that *an application can be added and removed without affecting the other applications* running concurrently. We return to these topics in Section 7.

4. PLATFORM OVERVIEW

In this section we introduce the concepts of predictability and composability and highlight the differences between these *orthogonal* system properties. We end this section with a discussion of the current limitations of our proposed platform.

4.1 Predictability

Predictability is needed to be able to guarantee that RT requirements are met for FRT applications, thus delivering a desired user-perceived quality or living up to, e.g. latency requirements in wireless standards. To provide predictable sharing of resources, the platform uses hardware *resource budget enforcement* [Mercer et al. 1994] to give a lower bound on resource availability, e.g. by providing a minimum bandwidth and maximum latency in a NoC [Goossens et al. 2005; Jantsch 2006]. The architecture thereby *bounds the interference* between applications. In our platform the enforcement is implemented using *preemptive arbitration* in all shared resources [Bekooij et al. 2004]. With a preemptive scheduler, applications are interrupted when their resource allotment is depleted. Note that using preemptive schedulers is not a necessary requirement for budget enforcement. It does, however, ease application integration as it removes the need for finding conservative worst cases, which is difficult and even undesirable for SRT and NRT applications [Abeni and Buttazzo 2004], and makes it impossible for a misbehaving or ill-characterised application to invalidate another application's bounds. To enable preemption, the application designers must avoid *shared slave locking* [ARM Limited 2003] (depreciated in the AXI standard), or only lock slaves for finite known times.

To determine bounds on the temporal behaviour of an application, not only the architecture, but also the tasks themselves must fit in a MoC that allows analytical reasoning about relevant metrics. Predictability thus *places limitations* on the application. In addition, the MoC must be *monotonic*, i.e. a reduced task execution time cannot result in a reduction of the throughput or latency of the application [Poplavko *et al.* 2003]. Without monotonicity, a *decrease* of the execution time on the task level may cause an *increase* on the application level [Graham 1969]. To benefit from the bounds provided by a predictable platform, the MoC must be free of these types of scheduling anomalies. Dataflow analysis [Sriram and Bhattacharyya 2000], used in this paper, is an example of a monotonic MoC.

4.2 Composability

In a predictable platform, an application can be: 1) given more resources, and 2) allocated resources at an earlier point in time, both as the result of another application's behaviour. This is, for example, the case when using traditional round-robin arbitration, common in contemporary SoC infrastructures. While the additional resources or earlier service might seem positive at first, for a general application, earlier service for an individual task may lead to reduced performance on the application level, as already discussed. Additionally, the extra budget an application might get (at many possible points in time) depends on the other applications in the system. As illustrated next, this has negative implications, both on the design of the application, and its behaviour when integrated in the system.

Consider an Independent Software Vendor (ISV) that is asked to develop a SRT video decoder for a platform with a predictable (but not composable) infrastructure. To simplify the application design process, a processor is dedicated to the application in question, but the SoC infrastructure, i.e. the NoC and memories, are shared with other applications. As a result of the sharing, the ISV must integrate also these applications to verify that *the desired deadline miss rate* of the video decoder is not

violated for a chosen set of input sequences. If the video decoder is analysed in isolation, the quality perceived during development and the quality perceived by the end user, when the application is integrated on the platform, could differ significantly. Furthermore, the analysis must cover multiple use-cases and variations due to input-dependent behaviour of other applications. The monolithic analysis both leads to an *explosion of the behaviours to cover*, and requires the entire platform to be included in the analysis, which *negatively impacts simulation speed* [Rowen and Leibson 2004]. Hence, the dependency on other applications, and their behaviours, complicates the *protection* and *concurrent engineering* of IP.

In addition to complicating the design of SRT and NRT applications, predictability alone also leads to problems after integration, as additional resources may improve the miss rate. While this might seem positive at first, a fluctuating miss rate is not necessarily good. The accelerated display of images can appear unnatural and unpleasant [Abeni and Buttazzo 2004], and rapidly changing quality levels are perceived as non-quality [Bril et al. 2001; Wüst et al. 2005].

To enable independent analysis and verification *for all applications*, RT or not, we *eliminate the interference between applications* completely. We refer to a system where applications do not influence each other as *composable*. Composability is a well-established concept in systems used in the automotive and aerospace domains [Avionics Application Software Standard Interface 1997; Rumpler 2006]. There, however, it is traditionally achieved by not sharing any resources between applications. For a system with shared resources to be composable, all those resources must guarantee that every application enjoys the same service independent of whether other applications are present and how they behave. In time-triggered architectures [Kopetz 1997], this is established by placing restrictions on the applications, i.e. only allowing applications for which *a static schedule can be derived at design time*, and the interfaces of the components are *fully specified in the value domain and in the temporal domain* [Kopetz and Bauer 2003]. This is an unreasonable assumption for many applications in the consumer-electronics domain, and instead, we completely remove the uncertainty in resource supply by using *hard resource reservations* [Rajkumar et al. 1998], where *the amount of service and the time at which an application receives its service is independent of other applications*. One illustrative example of composable resource sharing is the Time Division Multiplexing (TDM) arbitration used in several NoCs [Liang et al. 2000; Goossens et al. 2005; Jantsch 2006].

In the resulting platform: 1) It is impossible to create cyclic resource dependencies between applications that would lead to deadlocks, as shared resources are always made available after a known and finite time that is independent of other applications. 2) There is no potential for denial-of-service attacks between applications, as one application can never negatively (or positively) affect the service given to other applications. This also makes it impossible to observe changes in behaviour caused by other applications. 3) Probabilistic analysis, e.g. average-case performance or deadline miss rate, during the application design gives a good indication of the performance that is to be expected after integration, as the virtual platform, i.e. the resources available, are the same during both applications design and integration. 4) Design and debugging of applications can be done in isolation,

with higher simulation speed and reduced debugging scope. This is possible as only the virtual platform assigned to the application in question has to be included in the simulation. Everything that is not part of the application can be excluded. 5) IP of different ISVs does not have to be shared, nor the input stimuli. This also follows from the fact that the applications sharing the system cannot affect each other.

Note that predictability and composability are *orthogonal* properties. For an illustrative example, consider a system with two processors and a bus that connects them both to a shared memory, with two different applications mapped to the processors. If the memory port is shared using composable arbitration, as later described in Section 5.2, but the processors are running a non-RT Operating System (OS), then the platform is *composable and not predictable*, because the applications do not influence each other, but the OS makes it difficult, if not impossible, to derive *useful bounds* on the worst-case behaviour. In contrast, if the two processors are both predictable VLIW cores without any caches and operating systems, as exemplified in Section 5.3, but the memory port is shared using round-robin, then the platform is *predictable and not composable*, because the applications influence each other in the shared resource, but useful bounds on the interference can be computed.

Composability, unlike predictability, places no requirements on the applications, and there is no need to characterise the application behaviours. Once resources are reserved, the CoMPSoC platform guarantees isolation, placing no restrictions on how the various applications use their share of the resources. As a consequence of the isolation, the capacity unused by one application cannot be given to another one. The allocation of resources to applications is, however, configurable, and allows adaptation to requirements at the moment an application is started or enters a new mode of operation. Additionally, slack can be distributed within one application, thus distinguishing inter-application and intra-application arbitration [Hansson et al. 2007a]. It remains a problem to determine an appropriate amount of resources to reserve, especially in the presence of varying requirements. We elaborate further on this in Section 8.1, when discussing the mapping of applications to a platform instance.

4.3 Limitations

In contrast to [Jantsch 2006; Kopetz 1997], that *limit rather than remove interference*, our definition of composability does not allow any interference at all. The reason for our strict definition is that, in the general case, it is not possible to say what effects a small perturbation on the task level would have on the complete application. In our future work we aim to relax the current restrictions and investigate composability on higher levels of abstraction.

In our current platform, the *processing elements are not shared* between applications. Note, however, that this is due to the architecture of the VLIW cores that are used, and not a fundamental limitation.

The platform, as demonstrated in this paper, *does not contain caches*. This is also a result of the choice of processor architecture. Caches introduce two problems that must be addressed, namely *composable sharing* between applications, and *cache coherency*. The platform currently supports software-based cache coherency [Brand,
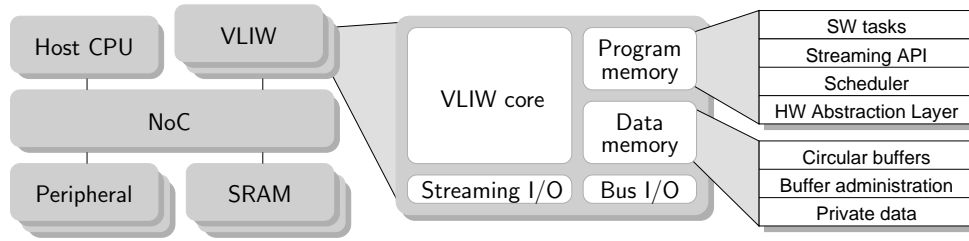
Fig. 3.   Architecture template.

van den and Bekooij 2007], and we envision that future generations of our platform will include processors with both instruction and data caches. As long as a cache is not shared by applications, it can be used without further considerations in the current platform. Sharing of caches, however, offers more challenges. In conclusion, *the inclusion of caches and sharing of caches and processors are outside the scope of this work*, and will be addressed in future work.

Our current CoMPSoC instance uses TDM arbiters to achieve predictability and composability. This is, however, not a requirement. As long as all arbiters can be characterised as *Latency-Rate servers* [Stiliadis and Varma 1998] it is possible to eliminate the influence of other applications, i.e. make the arbitration composable, by *delaying the notification that an action is complete*. Thus, it is possible to use e.g. rate-controlled priority-based arbiters [Akesson et al. 2007], where unlike TDM it is possible to distinguish between the allocated latency and rate. We consider the inclusion of such arbiters and delay mechanisms future work.

5.   HARDWARE PLATFORM

Our platform template, depicted in Figure 3, is built around processor and memory tiles [Culler et al. 1999], interconnected by a NoC. The SoC infrastructure, i.e. the NoC and the memory tiles, are covered in Sections 5.1 and 5.2, where we also show how these building blocks enable application composability and predictability. The processor tiles, are further detailed in Section 5.3, where we describe how they enable predictability. In addition to the aforementioned blocks, the system also comprises peripheral I/O functionality and a host tile. The functionality of the latter is further covered in Section 6.2 where we discuss the host software.

5.1   Network

The NoC interconnects all the tiles in the system and is thus shared by multiple applications. Sharing takes place in the network interfaces (NI), in the routers, their buffers and the network links. In the Æthereal NoC, predictability and composability is provided on the level of *connections* that realise the communication channels between two IP ports[2]. The IPs present transactions to the NIs, that are serialised and packetised before they are sent over the router network. The arbiters in the NoC are preemptive on the level of *flow control units* (flits), thus enabling

---

[2]The fine grain of composability and predictability is sufficient, but not necessary. It is possible to distinguish composable *inter-application* and predictable *intra-application* arbitration to reduce the average-case latency within applications [Hansson et al. 2007a].
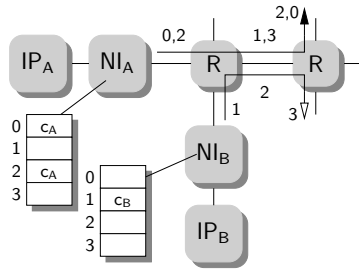
Fig. 4.   Contention-free routing.

guarantees independent of other connections. The injection of flits is regulated by a
TDM table in the NI such that no flits contend [Goossens et al. 2005], as illustrated
in Figure 4. In the figure, there are two IP cores, IP_A and IP_B, that communicate
over the network of routers (R), using the communication channels c_A and c_B, in-
dicated by a solid and an open-headed arrow, respectively. Channel c_A has slots 0
and 2 reserved in the table, and channel c_B has slot 1 reserved in the table. The
TDM table has the same period throughout the NoC, in this case 4, and works on
a fixed slot size. End-to-end flow control is used to avoid buffer overflows in the
NIs.

The Æthereal NoC has three important properties that makes it suitable for
CoMPSoC. First, and foremost, *it enables composability and predictability* by em-
ploying a flit-based TDM scheduling mechanism [Goossens et al. 2005]. Second,
the architecture is *run-time re-programmable*, and allows different partitioning of
the resources for the different use-cases [Hansson and Goossens 2007]. Resources
are reserved such that applications can be added and removed *without affecting the
other applications* running concurrently [Hansson et al. 2007b]. Third, to enable
programmers to reason about ordering when using multiple memories, the NoC im-
plements *release consistency* [Gharachorloo et al. 1990], using responses to assert
that (barrier) operations are complete. That is, a master is notified when an oper-
ation has been executed by the receiving slave and can wait for these responses to
realise a specific consistency model. This enables *both Distributed Shared Memory
and Message Passing.*

5.2   Memory tiles

Like the network, the memory tiles are shared by multiple applications. Sharing
takes place in the *slave port* of the memory controller, as illustrated in Figure 5.
Each *shell* in the figure is the end point for a connection to the memory tile,
originating from a remote master port, i.e. of a processor tile or the host tile. In
the memory tiles, predictable and composable sharing is thus provided between
master ports by the bus that connects the shells to the controller.

To achieve *composable and predictable sharing* of the memory port, the bus ar-
biter employs a TDM-based arbitration scheme, with time slots assigned to the
different masters (shells). Note, however, that the arbitration in a memory tile
is completely decoupled from the arbitration in the NoC (and any other memory
tiles). Furthermore, in contrast to the NoC, where the unit of preemption is always
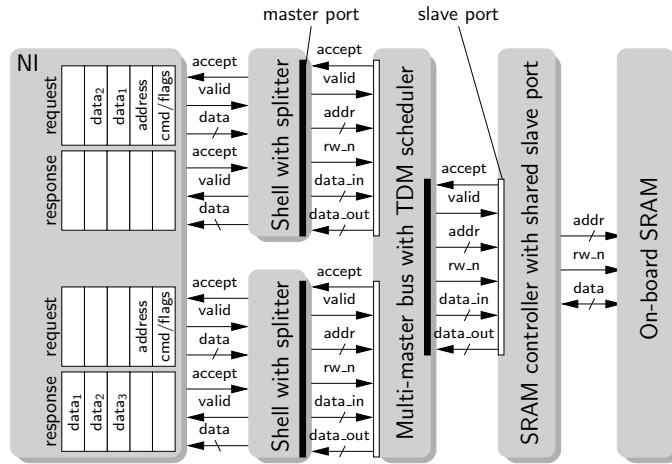
Fig. 5.   Memory tile architecture.

a fixed-size flit, the masters connected to the memory issue transactions ranging from pure Memory Mapped I/O (MMIO), with a granularity of one word, to Memory Mapped Block Data (MMBD), with a block size limited only by the number of bits used in the physical interface. Sizing the time slots for the worst-case burst size leads to long latencies, large buffering requirements [Coenen et al. 2006], and under-utilisation of the memory port if that burst size is not used by all the masters that share the memory. To overcome these issues, the shells *break multi-word MMBD transactions into multiple one-word MMIO accesses*, thus splitting and re-assembling MMBD bursts transparent to the masters using the memory. The shells also determine when requests can be executed *in a non-blocking fashion*. That is, when data and space is available in the NI, for a write and read respectively. This way, it is guaranteed that *a transaction finishes in a known and finite time* once it is presented to the bus arbiter.

The proposed scheme has three limitations. First, each master need a dedicated shell at the slave side. The area overhead is only around 2500 μm^2 per shell in a 90 nm CMOS technology, but the maximum amount of masters simultaneously accessing the memory is fixed at design time. This limitation is further discussed in Section 7 when describing the design flow. Second, while minimising the amount of buffering and the service latency, interleaving transactions potentially violates the atomicity of the protocol used by the IPs. In multi-threaded protocols, e.g. AXI [ARM Limited 2003] and OCP [OCP International Partnership 2007], the parallelism between connections can be made explicit via *connection and thread identifiers* in the interface. For DTL [Philips Semiconductors 2002], however, that offers no such functionality, we assume that ordering and atomicity of transactions is assured by higher-level protocols, such as the C-HEAP API discussed in Section 6.1. Third, the proposed architecture is tailored to slaves that enable word-level interfaces with fixed access times. This fits nicely with the Zero-Bus Turnaround (ZBT) SRAM, available in our experimental platform. SDRAM controllers, however, typically require larger bursts and more sophisticated arbitration schemes to

achieve predictability with a reasonable memory efficiency [Akesson et al. 2007]. We consider it future work to add shared SDRAM to the CoMPSoC template.

5.3 Processor Tiles

The processor tiles are based on Silicon Hive [Silicon Hive 2007] Very Large Instruction Word (VLIW) processing cores. The cores are customisable, making it possible to adapt the costs and the performances of the various computation nodes to a given application, as advocated in [Jerraya et al. 2006; Rowen and Leibson 2004]. The core architecture has a number of characteristics that make it appropriate for the CoMPSoC template. First, the processor core *has no caches*. Second, unlike a super-scalar processor, the VLIW *has no complicated bypassing or hazard detection mechanisms*. As we will see in Section 8.2, the aforementioned properties enable us to easily determine the execution time of tasks by analysis of the instruction schedule of the VLIW. Third, instructions are executed from a local memory thus removing the latency-critical reads from external memories. Fourth, the core does not only offer traditional bus-based interfaces, but also FIFO streaming ports. These two characteristics enables a more efficient mapping to our NoC-based SoC infrastructure. A more in-depth discussion on these properties follows.

The Silicon Hive cores use a memory-mapped architecture, with support for multiple Load Store Units (LSU). The master interface on the processor enables reads and writes to memories external to the processor. As shown in Figure 3, every processor also has its own private program and data memory. Having memory distributed over the different processing devices provides higher bandwidth with lower latency, which results in a higher performance at a lower power consumption [Soudris et al. 2000]. Moreover, the memories are accessible through a slave port on the processors bus interfaces, forming a distributed memory together with the dedicated memory tiles. Arbitration between multiple external master ports is implemented similar to the memory tiles, as described in Section 5.2.

In addition to the traditional LSUs, the processor template also has Send-Receive Units (SRU), that act as instruction-mapped First-In First-Out (FIFO) streaming interfaces [Vercauteren et al. 1996; Leijten et al. 2000]. For applications that use FIFO-based communication, e.g. an audio subsystem that works on streams of samples, it is thus possible to match the architecture to the application, as advocated in [Jerraya et al. 2006]. The address-less communication has two additional benefits. First, only data is communicated across the NoC and no commands, flags or addresses [Radulescu et al. 2005]. This removes the need for protocol shells and uses the NoC more efficiently. Second, it becomes impossible to corrupt memory contents of another tile.

The major limitation of the Silicon Hive core is that the processor *does not support preemptive multi-tasking*. It is hence not possible to enforce budgets in either a predictable, nor composable way. As a result, in this work, we do not allow multiplexing of task belonging to *different applications* on the same processor. Note, however, that we allow processor sharing between tasks belonging to the same application, e.g. by a static-order scheduling strategy, where the order in which the tasks execute are determined before the application is started [Bekooij et al. 2004]. An additional limitation, albeit with important positive consequences, is the absence of caches, as already discussed in Section 4.3.

6. SOFTWARE PLATFORM

While the hardware components play an important role, the platform also contains software. First, *application middleware*, i.e. libraries that facilitate inter-processor communication, further discussed in Section 6.1. Second, *host software* that enables a system designer to orchestrate the applications running on the platform and the resources allocated to them. The latter is discussed in Section 6.2. Third, and most importantly, the *design flow* that helps in defining, realising and verifying the system. This software component is discussed in Section 7.

6.1 Application Middleware

To facilitate synchronisation and communication between tasks running on the processors, an implementation of the C-HEAP [Nieuwland et al. 2002] protocol[3] is offered as a part of the CoMPSoC platform. The protocol specifies an API for *token-based communication*, built around shared memory. Synchronisation is done using pointers in a memory-mapped FIFO-administration structure.

C-HEAP fits well with CoMPSoC for several reasons. First, it does not use interrupts or slave locking for synchronisation. With interrupt-based synchronisation, it is difficult or even impossible to bound the frequency of interrupts and the execution time incurred by them [Kopetz and Bauer 2003]. Second, the local memories of the cores are suitable for mapping communication buffers to, enabling low-latency access to data, random access in acquired data and space, and changes in element size and FIFO length even after final silicon realisation. Third, by keeping two copies of the FIFO administration, both at the producer and consumer side, *all read operations are local* and *only posted write operations traverse the NoC* (also known as the *information push paradigm* [Kopetz and Bauer 2003]). The advantage of only writing and never reading remotely is the reduced impact of the interconnect latency. We exemplify the use of the C-HEAP implementation in Section 8.2.

6.2 Host software

In our platform, all administration of the processor tiles and the control registers of the communication infrastructure is done by a central *host tile*, as shown in Figure 3. The host functionality is implemented in portable C libraries [Silicon Hive 2007; Hansson and Goossens 2007], to be executed on e.g. an embedded ARM processor. On the instance exemplified in Section 8 this software runs on a PC, interfacing with the SoC through an on-board USB interface. The key characteristics of the administration hardware and software that are important for CoMPSoC are: 1) *Only the host has the ability to configure the NIs and the memory arbiters.* Hence, a design fault or a hardware fault in the IP cannot affect the service given to any of the applications in the system. Thus, the host in our system corresponds to the *trusted network authority* in the terminology of [Kopetz et al. 2007]. 2) The configuration data itself is carried by the network, enjoying the same isolation as the normal application traffic [Hansson and Goossens 2007]. The configuration connections are efficiently implemented using the concept of channel trees [Hansson et al. 2007a].

---

[3]C-HEAP in its entirety, is not only a protocol for cooperation and communication between tasks, but also a top-down design methodology and an architectural template [Nieuwland et al. 2002].
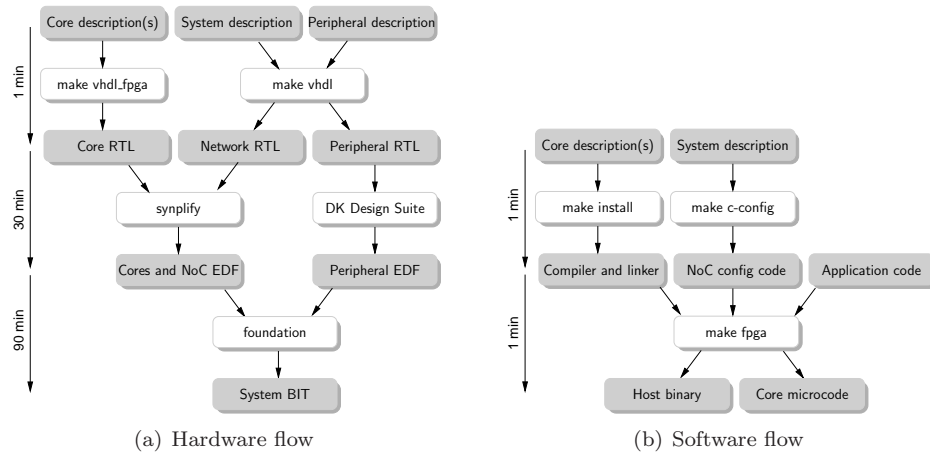
(a) Hardware flow   (b) Software flow

Fig. 6.   Overall design flow.

7. DESIGN FLOW

In this section, we describe how the hardware and software building blocks are instantiated from high-level platform descriptions and how they come together in a CoMPSoC instance. In CoMPSoC, in contrast to best-effort architectures, the design flow has a particularly prominent role, as the hardware platform, i.e. the VLIW cores, the NoC and the bus arbiters, *move the complexity of resource allocation from run time to design and compile time.* Using the tool flow, depicted in Figure 6, complete MPSoC designs are generated in a matter of hours.

The following properties of the design flow are particularly important for CoMPSoC's design methodology: 1) the *RT requirements are described per application*, 2) and *on a level that is understood by the application designer*, e.g. through constraints on the periodicity of sources and sinks rather than what TDM slots to use in an arbiter or what link to use in the NoC, and 3) it is the responsibility of the design flow to find a resource allocation with *seamless transitions between all valid use-cases.* A brief description of the individual parts of the flow follows, with more details to be found in e.g. [Goossens et al. 2005; Hansson et al. 2007b; Kumar et al. 2007; Silicon Hive 2007].

The hardware flow, depicted in Figure 6(a), takes its starting point in a *System description* file. This file is used to generate the appropriate protocol-conversion shells for the NIs [Radulescu et al. 2005], generate the HDL for the memory tile(s) and peripherals (*Peripheral description*), and instantiate the Silicon Hive cores used in the design according to their individual descriptions (*Core description(s)*). Additionally, the flow constructs a NoC instance, with NIs and routers, and sizes the TDM wheels of the NoC and the bus arbiters [Hansson et al. 2007b]. For the last step, the maximum number of concurrent connections to and from all the IP ports in the system is assumed to be known at design time. The number of hardware resources is thus fixed, but the actual assignment to applications is reconfigurable. The entire HDL description is generated from a high-level specification in a few minutes, together with a transaction-level simulation model for each component,

making an accurate system model available early in the design schedule, so detailed Very Large Scale Integration (VLSI) and software implementations can proceed in parallel, as advocated in [Rowen and Leibson 2004].

The starting point of the software flow, depicted in Figure 6(b), is the same description of the cores, from which the corresponding assembler and linker are generated. These are then used to compile the source code that is to be run on the processors.

In parallel with the core microcode, the code needed to configure the cores and the network is produced and linked together with the Hive Run-Time (HRT) and Æthereal Run-Time (ART) APIs [Hansson and Goossens 2007]. The NoC configuration software is derived based on a description of the communication requirements of the various applications and the *valid application combinations* [Hansson et al. 2007b]. The requirements are given per connection, specifying communication between a master port and a slave port, the required (minimum) bandwidth and the (maximum) allowed latency. In this work, the various configurations are determined at design-time, based on the concept of *partial dynamic (re)configuration* [Hansson and Goossens 2007], with *seamless transitions between use-cases* [Nieuwland et al. 2002; Hansson et al. 2007b]. Note that it is possible to change the NoC configuration after the hardware is fixed (at compile time rather than design time), and thus accommodate new or modified applications, assuming that the requirements do not exceed the resources that are available.

8. EXPERIMENTAL RESULTS

In this section, we demonstrate how two applications, a *JPEG decoder* and an *audio post-processing filter*, are developed and integrated on an instance of the CoMPSoC platform. The JPEG decoder has a data-dependent behaviour, which is typical for compression and decompression functions [Leijten et al. 2000]. Thus, the amount of data produced or consumed, and the processing delay varies over time. The decoder has no RT requirements but *should produce output as fast as possible* given its allocated resources. The audio filter, on the other hand, has FRT requirements, as failure to consume and produce samples at 48 kHz causes noticeable clicks in the audio output.

The platform instance to which we map the two applications is depicted in Figure 7(a) and contains:

—Three VLIW cores, each with 32 kB program memory and 32 kB data memory.

—An instance of the Æthereal NoC, with guaranteed services only.

—A Cirrus Logic audio codec, connected via hardware FIFOs.

—A single SRAM memory controller.

—An LCD display controller.

—A host interface.

The platform instance has only one shared memory. While this is common, either for cost reasons or due to a limited number of pins [Leijten et al. 2000], a single shared memory is inherently non-scalable, as the performance is directly affected by the amount of applications sharing it [Jantsch 2006]. It should be noted that our

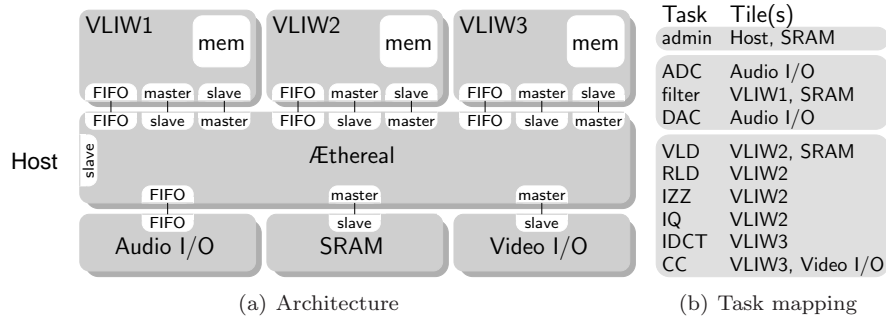(a) Architecture   (b) Task mapping

Fig. 7.   Experimental platform instance.

template does not constrain the amount of memories, but currently only supports SRAMs, as already discussed in Section 5.2.

8.1 Application mapping

The starting point for the JPEG decoder is sequential C code that we split into multiple tasks: Variable-Length Decoding (VLD), Run-Length Decoding (RLD), Inverse Zig-Zag (IZZ), Inverse Quantisation (IQ), Inverse Discrete Cosine Transform (IDCT), and Colour Conversion (CC). Communication between the tasks is implemented using the C-HEAP API, with FIFO data mapped to circular buffers in the local memories of the processor tiles. The original code is easily ported to the platform, requiring only an explicit mapping of variables and communication buffers to memories. The VLD, RLD, IZZ and IQ are mapped to one processor, as shown in Figure 7(b), occupying 24.3 kB program memory and 6.7 kB data memory (not counting the C-HEAP FIFOs). A second processor executes the IDCT and CC, requiring 13.4 kB of program memory. The encoded input image is read from background memory by the first core, and written to the frame buffer by the second core, both using shared memory communication. The JPEG decoder relies solely on the bus interfaces of the processor, with four network connections realising the communication: one each for accesses to the SRAM and Video I/O, and two for the inter-processor communication.

The audio post-processing application, shown in Figure 8(a), comprises three tasks. First, the source Analog to Digital Conversion (ADC), periodically producing signed 16-bit Pulse Code Modulated (PCM) stereo samples. Second, the actual filter task, executed on one of the cores. Third, the Digital to Analog Conversion (DAC), which acts as a periodic sink. The filter task receives input samples from the ADC via the NoC and adds a two-tap reverberation effect by mixing in past samples. The output is then sent both to the DAC and stored in the background memory for future mixing with the input. The processor uses FIFO streaming for the communication with the ADC and DAC, and shared memory communication for reading and writing reference samples in the background memory. The filter application thus requires three connections, and has one slot allocated in the memory tile.

Network resources are allocated to the two different applications using the UMARS tool [Hansson et al. 2007b]. The amount of resource needed for the filter applica-

tion is determined based on its FRT requirements and strictly periodic behaviour (as described in Section 8.2). For the JPEG decoder, the amount of resources requested are based on rough estimates of how much data needs to be communicated *on average* between the different tasks. The performance of the decoder can thus be improved by asking for additional resources, a trade-off left for the application designer. Additional resources are allocated to enable the host to load the program memories of the cores, configure the NoC and memory controller, and load encoded JPEG images into the background memory.

The complete architecture is mapped to a Xilinx Virtex4 LX-160 FPGA, using Synplify 8.8 and Foundation 9.1. The resulting design occupies 96 (33%) block RAMs, 25 (26%) DSPs, 18397 (13%) flip flops, and 57682 (42%) LUTs. The obtained maximum clock frequency is 48 MHz, and the tools estimate a total equivalent gate count of roughly 7.8 Mgates. In addition to logic on the FPGA, two on-board 8 MB SRAM modules are used, one for the shared background memory, and one for the frame buffer. The memories, the audio codec, the USB μcontroller and the display driver are all integrated on the board level, with only wrappers integrated on the actual FPGA.

8.2   Performance analysis and verification

With the mapping given, we proceed to analyse the performance of two applications. The analysis is done independently for the two applications, first looking at the JPEG decoder, and then the filter application. This is made possible by the composability of CoMPSoC, and is a *major qualitative difference* with existing MPSoC platforms.

The *average-case performance* of the JPEG decoder is analysed by instrumentation on the actual FPGA implementation. For various 1 Mpixel images, we experience execution times ranging from 1.4 s, for a monochrome image, up to 2.8 s. The mapping to processors is such that the execution time is largely determined by the number of (stalling) read accesses to background memory and hence the size of the encoded image. Increasing the size of the FIFOs between the tasks therefore only has a negligible impact on the execution time, saturating at 10% improvement.

Next, we formally verify the RT performance of the audio filter application. The worst-case analysis of the complete application is made possible by the predictability of CoMPSoC. We choose to model the filter applications and its mapping to the architecture as a Homogeneous Synchronous Dataflow (HSDF) [Sriram and Bhattacharyya 2000] graph. The choice of a MoC is up to the designer, but many signal-processing applications can be represented by dataflow graphs and the expressivity of HSDF is sufficient for the filter application.

The first step in constructing the model is to transform the task graph, in Figure 8(a) to a dataflow graph. In our case this is a one-to-one transformation, and the resulting graph has exactly the same topology as the aforementioned task graph. In the dataflow graph, the nodes, called *actors*, correspond to the tasks, and edges show data dependencies between them. Every edge can carry an infinite number of tokens between two actors, and can contain *initial tokens* (present on the edges at start time). Actors are started after *sufficient input data and output space* is available, such that they can finish their execution without having to wait for additional input data or output space. A self-edge with one initial token is used to
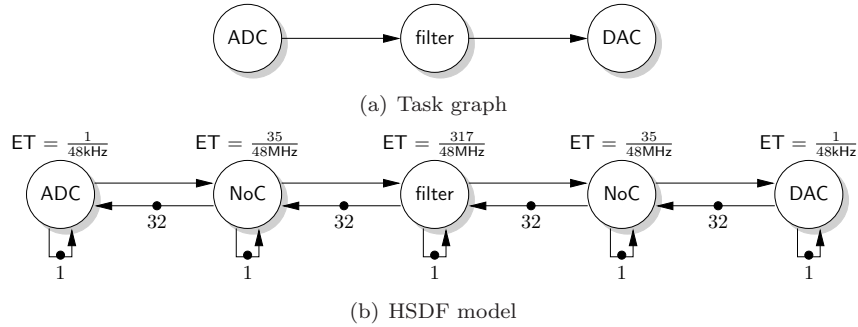
(a) Task graph

(b) HSDF model

Fig. 8.   Task graph and dataflow model of the audio post-processing filter.

model that the previous execution must be finished before the next execution can start. In Figure 8(b), the original dataflow graph is extended with actors and edges that model: 1) the network connections between the tasks, here modelled by two additional actors, and 2) the finite buffers that connect the different actors, seen as 32 initial tokens on the backward edges. For simplicity, the NoC is modelled as a constant forward latency communication channel. A more detailed (and less conservative) dataflow model of the Æthereal NoC is presented in [Hansson et al. 2008].

The next step, with the topology of the dataflow graph in place, is to determine the Execution Time (ET), i.e. the maximum time between start and finish of one execution, of the actors. The ET of an actor does not include the time a task has to wait for input data and output space. As an example, the ET of the ADC and DAC actors is determined by the sampling rate, which in our case is 48 kHz. The ET of the NoC actors is determined by: 1) the size of the sample which in our case is a single word, 2) the latency of the NI, which for streaming data is 1 cycle, in both the sending and receiving end 3) the worst-case waiting time for a TDM slot, here 8 slots, each of size 3 (the flit size), and 4) the latency of the router network, in our case 3 hops, with a flit size of 3. This accumulates to $1 + 8 \times 3 + 3 \times 3 + 1 = 35$ cycles, at 48 MHz, for every sample sent. To conservatively model the NoC, we also take the effects of end-to-end flow control into account. In this case, the 35 cycles are sufficient to guarantee that all flow control credits are returned before the succeeding sample arrives.

The ET of the filter task is determined by: 1) the processor on which the task code executes, 2) the NoC, transporting reference samples to and from background memory, and 3) the memory tile itself, and the time required to read and write the required samples. Hence, all three must enable the derivation of conservative execution times, as discussed in Section 5. The time spent executing instructions on the processor is determined by analysis of the program flow [Chen et al. 2001]. For every iteration, 83 processor cycles are spent performing arithmetic operations and accessing local memory. For every execution, two stereo samples (since the filter relies on a two-tap delay line) are read, and one output sample is written to the background memory. To determine the execution time of the task we also need to include the time required to read and write those samples.

In the memory tile, an eligible request, as discussed in Section 5.2, can potentially wait for the requests from two other connections to be served (from the host and the JPEG decoder). Hence, after a setup time of 1 cycle and a complete TDM revolution of 3×3 cycles, an eligible request of one word (one stereo sample) is guaranteed to be served. The two read requests and the write request also traverse the NoC. A read request is serialised into 2 words (command/flags and address) by the NI, and hence requires 1 flit to be sent across the network. With request serialisation of 2 cycles, one TDM slot reserved, and a 3-hop path this amounts to $2 + 8 \times 3 + 3 \times 3 + 1 = 36$ cycles per request. Read responses carry only one word of data and also have a latency of 36 cycles. Similarly, a write request occupies 3 words and therefore requires 2 flits, once the packet header is included. With a serialisation latency of 3 cycles, this conduces to a total time of $3 + 2 \times (8 \times 3 + 3 \times 3) + 1 = 70$ cycles. The aforementioned latency is sufficient to guarantee that all flow control credits are returned before succeeding memory operations start. In total a conservative ET of the filter is $83 + 2 \times (36 + 10 + 36) + 70 = 317$ cycles.

Using the constructed HSDF graph, the algorithm presented in [Wiggers et al. 2007] constructs a *conservative periodic schedule*. The existence of such a schedule confirms that the sink and source tasks of the filter application can indeed execute strictly periodically. Due to monotonicity, a decrease in execution time or start time can only lead to earlier token production times, and therefore only to an earlier actor enabling. Indeed, observations of the FPGA implementation confirm that the ADC and DAC do not suffer from overflow or underflow. We observe this behaviour irrespective of the presence and input of the JPEG decoder. Moreover, in HDL simulations, the cycle-level behaviour of the audio filter is *identical* irrespective of the images applied to the input-dependent JPEG decoder. Albeit not a formal proof, this shows that the *results of the analysis still apply* when the applications are integrated onto the platform.

In conclusions, the two applications are independently analysed, looking at different metrics, and using different methodologies. Should additional applications be added to the platform, they are assigned virtual platforms that do not interfere with the JPEG decoder and the filter applications. Thus, the conclusions we have drawn, about the average-case behaviour of the JPEG decoder, and the worst-case behaviour of the filter, are still valid.

9. CONCLUSIONS AND FUTURE WORK

The complexity of system verification and integration is exploding due to the many applications integrated on a single chip, and their interactions in shared resources. With a growing part of the functionality implemented in software, real-time requirements, and increasing run-time dynamism, it is crucial to offer an architecture that enables independent analysis of all applications, and reuse of the analysis across use-cases. We propose a Composable and Predictable Multi-Processor System on Chip (CoMPSoC) platform template that eliminates interference between applications through resource reservations. The architectural components are constructed to offer composability and run-time reconfiguration. On an instance of this hardware and software platform, we demonstrate how composability allows applications to be analysed and verified independently of one another. Although CoMPSoC is

aimed at a specific application domain, the presented template supports a wide range of programming models, and the lessons learnt from its development serve as a first step towards a system-level design method.

It still remains to include caches and off-chip SDRAM in the template, further extending the application domain.

REFERENCES

ABENI, L. AND BUTTAZZO, G. 2004. Resource reservation in dynamic real-time systems. *Real-Time Systems 27,* 2, 123–167.

AKESSON, B., GOOSSENS, K., AND RINGHOFER, M. 2007. Predator: A predictable SDRAM memory controller. In *Proc. CODES+ISSS*.

ARM Limited 2003. *AMBA AXI Protocol Specification*. ARM Limited.

Avionics Application Software Standard Interface 1997. *ARINC Specification 653*. Avionics Application Software Standard Interface.

BAGHDADI, A., LYONNARD, D., ZERGAINOH, N., AND JERRAYA, A. 2001. An efficient architecture model for systematic design of application-specific multiprocessor SoC. *Proc. DATE*, 55–63.

BARTIC, T., DESMET, D., MIGNOLET, J.-Y., MARESCAUX, T., VERKEST, D., VERNALDE, S., LAUWEREINS, R., MILLER, J., AND ROBERT, F. 2004. Network-on-chip for reconfigurable systems: From high-level design down to implementation. In *Proc. FPL*. Springer Berlin / Heidelberg.

BEKOOIJ, M., MOREIRA, O., POPLAVKO, P., MESMAN, B., PASTRNAK, M., AND VAN MEERBERGEN, J. 2004. Predictable embedded multiprocessor system design. *Lecture notes in computer science*, 77–91.

BENINI, L. 2006. Application specific NoC design. In *Proc. DATE*.

BRAND, VAN DEN, J. AND BEKOOIJ, M. 2007. Streaming consistency: a model for efficient MPSoC design. In *Proc. DSD*.

BRIL, R. J., HENTSCHEL, C., STEFFENS, E. F., GABRANI, M., VAN LOO, G., AND GELISSEN, J. H. 2001. Multimedia QoS in consumer terminals. *IEEE Workshop on Signal Processing Systems*, 332–343.

BUTTAZO, G. C. 1977. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Publishers.

CHEN, K., MALIK, S., AND AUGUST, D. 2001. Retargetable static timing analysis for embedded software. *Proc. ISSS*, 39–44.

COENEN, M., MURALI, S., RĂDULESCU, A., GOOSSENS, K., AND DE MICHELI, G. 2006. A buffer-sizing algorithm for networks on chip using TDMA and credit-based end-to-end flow control. In *Proc. CODES+ISSS*.

CULLER, D. J., SINGH, J. P., AND GUPTA, A. 1999. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers.

DUTTA *et al.*, S. 2001. Viper: A multiprocessor SOC for advanced set-top box and digital TV systems. *IEEE Design and Test of Computers*.

GENKO, N., ATIENZA, D., MICHELI, G. D., MENDIAS, J., HERMIDA, R., AND CATTHOOR, F. 2005. A complete network-on-chip emulation framework. In *Proc. DATE*.

GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A., AND HENNESSY, J. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the Int'l Symposium on Computer Architecture*.

GOOSSENS, K., DIELISSEN, J., GANGWAL, O. P., GONZÁLEZ PESTANA, S., RĂDULESCU, A., AND RIJPKEMA, E. 2005. A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design and verification. In *Proc. DATE*.

GOOSSENS, K., DIELISSEN, J., AND RĂDULESCU, A. 2005. The Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test of Computers*.

GRAHAM, R. 1969. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics 17,* 2.

HANSSON, A., COENEN, M., AND GOOSSENS, K. 2007a. Channel trees: Reducing latency by sharing time slots in time-multiplexed networks on chip. In *Proc. CODES+ISSS*.

HANSSON, A., COENEN, M., AND GOOSSENS, K. 2007b. Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In *Proc. DATE*.

HANSSON, A. AND GOOSSENS, K. 2007. Trade-offs in the configuration of a network on chip for multiple use-cases. In *Proc. NOCS*.

HANSSON, A., WIGGERS, M., MOONEN, A., GOOSSENS, K., AND BEKOOIJ, M. 2008. Applying dataflow analysis to dimension buffers for guaranteed performance in networks on chip. In *Proc. NOCS*.

IVIMEY-COOK, R. 1999. Legacy of the transputer. In *Architectures, Languages and Techniques*, B. M. Cook, Ed. IOS Press.

JANTSCH, A. 2006. Models of computation for networks on chip. In *Proc. ACSD*.

JERRAYA, A., BOUCHHIMA, A., AND PÉTROT, F. 2006. Programming models and HW-SW interfaces abstraction for multi-processor SoC. *Proc. DAC*.

KEUTZER, K., MALIK, S., NEWTON, A. R., RABAEY, J. M., AND SANGIOVANNI-VINCENTELLI, A. 2000. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Trans. on CAD of Integrated Circuits and Systems 19,* 12.

KOPETZ, H. 1997. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers.

KOPETZ, H. AND BAUER, G. 2003. The time-triggered architecture. *Proceedings of the IEEE 91,* 1.

KOPETZ, H., OBERMAISSER, R., SALLOUM, C. E., AND HUBER, B. 2007. Automotive software development for a multi-core system-on-a-chip. *Int'l Workshop on Software Engineering for Automotive Systems (SEAS).*

KUMAR, A., HANSSON, A., HUISKEN, J., AND CORPORAAL, H. 2007. An FPGA design flow for reconfigurable network-based multi-processor systems on chip. In *Proc. DATE.*

LEIJTEN, J., VAN MEERBERGEN, J., TIMMER, A., AND JESS, J. 2000. Prophid: a platform-based design method. *Journal of Design Automation for Embedded Systems 6,* 1, 5–37.

LIANG, J., SWAMINATHAN, S., AND TESSIER, R. 2000. aSOC: A scalable, single-chip communications architecture. *Proc. Int'l Conference on Parallel Architectures and Compilation Techniques (PACT),* 37–46.

MARTIN, G. 2006. Overview of the MPSoC design challenge. In *Proc. DAC.*

MERCER, C. W., SAVAGE, S., AND TOKUDA, H. 1994. Processor capacity reserves: Operating system support for multimedia systems. In *Proc. IEEE International Conference of Multimedia Computing and Systems.* IEEE Computer Society Press, 90–99.

MOONEN, A., BEKOOIJ, M., VAN DEN BERG, R., AND VAN MEERBERGEN, J. 2007. Practical and accurate throughput analysis with the cyclo static data flow model. In *Proc. MASCOTS.*

MORAES, F., CALAZANS, N., MELLO, A., MÖLLER, L., AND OST, L. 2004. HERMES: an infrastructure for low area overhead packet-switching networks on chip. *Integration VLSI J. 38,* 1.

MOREIRA, O., VALENTE, F., AND BEKOOIJ, M. 2007. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *Proc. EMSOFT.*

NIEUWLAND, A., KANG, J., GANGWAL, O., SETHURAMAN, R., BUSÁ, N., GOOSSENS, K., PESET LLOPIS, R., AND LIPPENS, P. 2002. 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.

OCP International Partnership 2007. *OCP Specification 2.2.* OCP International Partnership.

Philips Semiconductors 2002. *Device Transaction Level (DTL) Protocol Specification. Version 2.2.* Philips Semiconductors.

POPLAVKO *et al.*, P. 2003. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In *Proc. CASES.*

RADULESCU, A., DIELISSEN, J., GOOSSENS, K., RIJPKEMA, E., AND WIELAGE, P. 2005. An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming. *IEEE Trans. on CAD of Int. Circ. and Syst..*

RAJKUMAR, R., JUVVA, K., MOLANO, A., AND OIKAWA, S. 1998. Resource kernels: A resource-centric approach to real-time systems. *Proc. SPIE/ACM Conference on Multimedia Computing and Networking,* 150–164.

ROWEN, C. AND LEIBSON, S. 2004. *Engineering the Complex SOC: Fast, Flexible Design with Configurable Processors.* Prentice Hall PTR.

RUMPLER, B. 2006. Complexity management for composable real-time systems. In *Proc. Int'l Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC).*

RUTTEN, M., POL, E.-J., VAN EIJNDHOVEN, J., WALTERS, K., AND ESSINK, G. 2005. Dynamic reconfiguration of streaming graphs on a heterogeneous multiprocessor architecture. *IS&T/SPIE Electron. Imag. 5683.*

SASAKI, H. 1996. Multimedia complex on a chip. *Proc. Int'l Solid-State Circuits Conference (ISSCC),* 16–19.

Silicon Hive 2007. Silicon hive. Available from: http://www.siliconhive.com.

SOUDRIS, D., ZERVAS, N. D., ARGYRIOU, A., DASYGENIS, M., TATAS, K., GOUTIS, C., AND THANAILAKIS, A. 2000. Data-reuse and parallel embedded architectures for low-power, real-time multimedia applications. *IEEE International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS),* 243–254.

SRIRAM, S. AND BHATTACHARYYA, S. 2000. *Embedded Multiprocessors: Scheduling and Synchronization.* CRC Press.

STILIADIS, D. AND VARMA, A. 1998. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networking 6,* 5.

VERCAUTEREN, S., LIN, B., AND DE MAN, H. 1996. Constructing application-specific heterogeneous embedded architectures from custom HW/SW applications. *Proc. DAC,* 521–526.

WIGGERS, M., BEKOOIJ, M., JANSEN, P., AND SMIT, G. 2007. Efficient computation of buffer capacities for cyclo-static real-time systems with back-pressure. In *Proc. RTAS.*

WÜST, C., STEFFENS, L., VERHAEGH, W., BRIL, R., AND HENTSCHEL, C. 2005. QoS control strategies for high-quality video processing. *Real-Time Systems 30,* 1, 7–29.