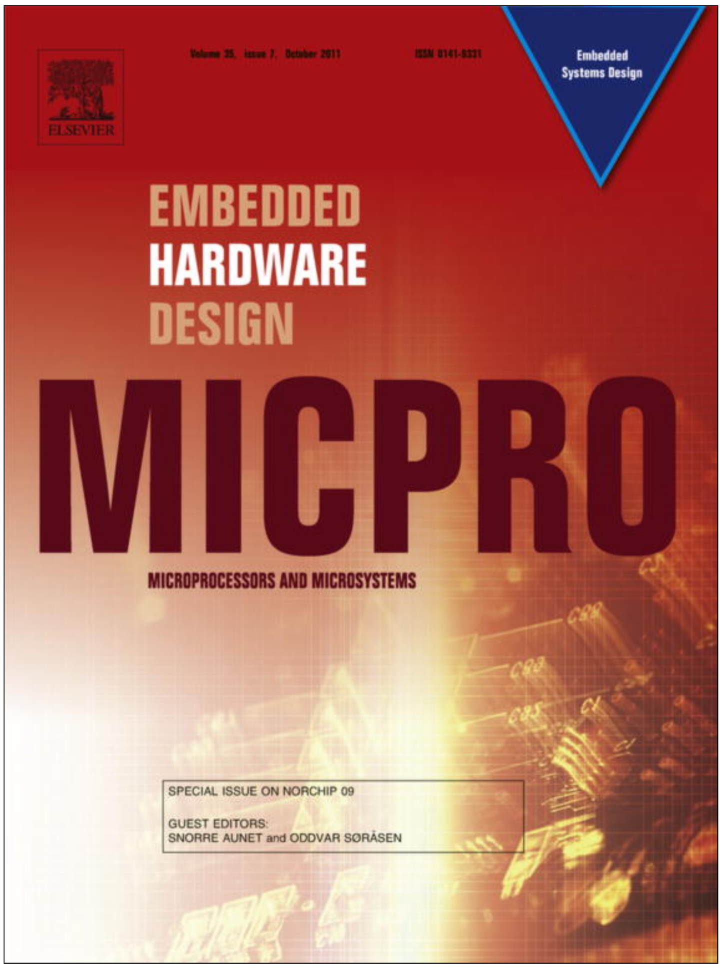

Volume 35, Issue 7, October 2011
ISSN 0141-9331
ELSEVIER
Embedded Systems Design
EMBEDDED
HARDWARE
DESIGN
MICPRO
MICROPROCESSORS AND MICROSYSTEMS
SPECIAL ISSUE ON NORCHIP 09
GUEST EDITORS:
SNORRE AUNET and ODDVAR SØRÅSEN

Microprocessors and Microsystems 35 (2011) 643–655

Contents lists available at SciVerse ScienceDirect

Microprocessors and Microsystems

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

Service based communication for MPSoC platform-*SegBus*

Khalid Latif[a,b,*], Tiberiu Seceleanu[c], Cristina Seceleanu[d], Hannu Tenhunen[a,b]

[a] *Dept. of Information Technology, University of Turku, Finland*
[b] *Turku Centre for Computer Science (TUCS), Turku, Finland*
[c] *ABB Corporate Research, Västerås, Sweden*
[d] *Mälardalen University, Västerås, Sweden*

A R T I C L E I N F O

Article history:
Available online 22 July 2011

Keywords:
MPSoC
SegBus
Multicast
Scheduling
Design methodology

A B S T R A C T

MPSoC platforms offer solutions to deal with communication limitations for multiple cores on single chip, but many new issues arise within the context. The *SegBus* platform is one of the solutions for application deployment on multi-core applications. There are many applications where identical data is transferred from the same source towards different destinations. *Multicast services* may come as a performance improving factor for the interconnection platform, together with interrupt service.

In this paper, the task is to analyze, how different services can be designed for the *SegBus* platform and observe the improvement in system performance. The designer can select the services according to the requirements. The running example is represented by the H.264 encoder. The *SegBus* platform architecture, the communication mechanism, the allocation of processing elements on the platform, the communication services and their implementation are the main topics elaborated here.

1. Introduction

Modern embedded systems consists of heterogenous components like processing elements (PEs), ASICs, programmable microprocessors, memory and FPGAs. Off-chip communication among these components is very slow, requires extra design effort and it is not efficient regarding power and area. Traditionally, it was not possible to integrate all the components on a single chip because of technology limitations. At the present, continuous technology scaling has made it possible to integrate billions of transistors on a single chip [8]. Thus, the entire system can be integrated on a single chip, hence the paradigm of Multiprocessor System-on-Chip (MPSoC). After resolving the integration issue, the next problem is to provide an efficient communication platform for communication among the processing elements. A traditional data bus can provide enough bandwidth for communication among 3–10 elements, but it does not scale to higher numbers [13]. Different on-chip communication platforms have already been proposed like Network-on-Chip (NoC) [11] and Segmented bus (*SegBus*) [24], to address this issue.

A computation mechanism comprises multiple functions and tasks generally expressed at different levels of abstraction. Individually, such functions or tasks can be considered as a service. We define here the two types of services for SoC: *computation* and *communication* services as shown in Fig. 1. A computation service means that the chip offers service(s) which will (together) complete a certain task, or execute some application. A communication service contains functions that facilitate or support the data transfer from source to destination, such as monitoring, scheduling, arbitration, multicast and SPLIT transfers. These communication services can be customized and included in MPSoC communication platforms according to the application requirements. The implementation of communication services for on-chip communications is completely different from off-chip or computer system communications, though the basic concept is exactly same. For on-chip communications, challenges are deep submicron effect, cross talk, thermal noise and many other issues, which are not the key problems for typical communication systems. Apart from that, for systems based on MPSoC platforms, the customer considerations - power consumption, area and latency, are different than for communication systems. There are few services, required for on-chip communications but not used in telecommunication systems like cache-coherence or interrupt communication. Similarly, there are services for communication systems, which are not very important for on-chip communications like security. Some services cannot be completely defined as computational or communication, for instance the thermal monitoring. Such services are found then on the border in Fig. 1.

Two well known architectures for implementation of communication fabrics for SoCs are transaction-based – *buses* and packet

* Corresponding author at: Dept. of Information Technology, University of Turku, Finland. Tel.: +358 443310316; fax: +358 23336950.

E-mail addresses: khalid.latif@utu.fi (K. Latif), tiberiu.seceleanu@se.abb.com (T. Seceleanu), cristina.seceleanu@mdh.se (C. Seceleanu), hannu.tenhunen@utu.fi (H. Tenhunen).

0141-9331/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2011.06.006

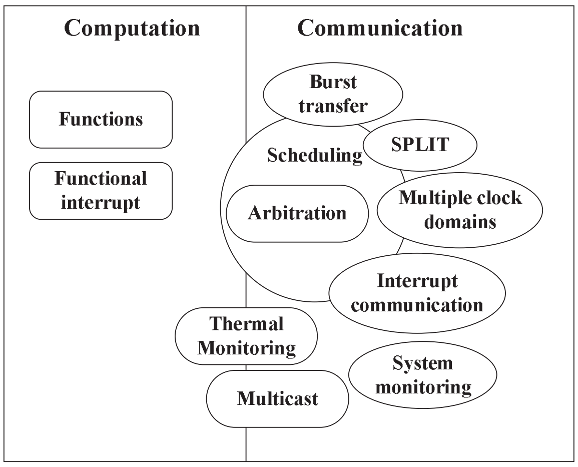


Fig. 1. Classification of on-chip services.

based – NoCs [17]. Implementing the communication services for transaction based data transfers is not economical because of the service overhead for each transaction. Communication services built on packet based transactions can show a significant improvement in system performance. Communication services have already been implemented for NoC but for bus based systems, only the obvious services like scheduling and arbitration are most commonly considered.

Congestion control, cache coherence and power consumption are the key problems for on-chip distributed architectures. A first technique to address congestion control is the avoidance of identical data re-transmission, which leads to the idea of a multicast approach in many streaming applications like H.264 codec, MP3 codec or video conference application. Multicast techniques are simple and easy to implement in bus based systems as compared to the implementations of networks on chip (NoC). Considering the later, apart from implementation, multicast introduces a serious communication overhead. For segmented bus approaches, multicast approaches bring only a negligible overhead. This makes multicast power and latency efficient for bus based systems.

We describe, in the present paper, a realization of a multicast protocol for the segmented bus platform *SegBus*. We perform this in the context of a perspective change, where activities related to the design and execution of applications on the *SegBus* platform are viewed now as *services*. While scheduling can be seen as a service at design time, arbitration, and various kinds of communication features (SPLIT, interrupt, multicast, etc.) are treated as run-time platform services. Multicast features of communication protocols are not a recent development. This comes as a performance improvement to repetitious transactions containing the same data. Lately, multicast procedures have been analyzed in the context of on-chip multi processor architectures. Similarly, interrupt communications, scheduling, task allocation mechanism and other services are discussed.

Related work. Different MPSoC design models have already been proposed [5,6,18,20,29]. We approach the problem by considering the communication cost and throughput, which are dependent on each other and make the design process automated. By introducing different communication services one-by-one or in a group, communication cost can be reduced. Then, a service or the group of services with minimum communication cost and overhead is selected for final implementation.

A bus system and its variants do not scale well with the system size in bandwidth and clocking frequency [15]. However a bus platform is very efficient when considering broadcasting, since all clients are directly connected to it. A unicast transaction is in fact broadcasted to all clients in the bus segment, but read only by the destination device.

Bus snarfing is proposed as well for performance improvement of multiprocessor systems [9,10]. Broadcasting technique can be used to reduce memory latency for bus-based multiprocessor systems [4]. As with other broadcast techniques for NoC, this uses the whole interconnect platform. In the case of the *SegBus* platform, we utilize only those segments which host the destination devices, and not all the structure.

Tranberg-Hansen and Madsen [28] present a unified model for application and platform. Authors present the service model in an abstract model of a hardware component implementing the behavior of the component by offering the services. For this purpose, the service model has been proposed. We complement here by introducing the services block in our design methodology instead of having a service model. The authors also pointed out that Artemis [6] and the subproject Sesame [31] present the application and architecture model separately. We approach similarly the problem, by separating the computation and communication models.

Cornelius et al. [32] introduce the service oriented approach for NoC based communications. A useful comparison of centralized and distributed service oriented architectures is presented. In case of centralized approach, extra communication from all the nodes will be needed to coordinate with Central Coordination Node (CCN). In this case, CCN will be overloaded but it simplifies the monitoring of services. On other hand, the distributed nature of the system control promises to circumvent the hot spot around a single resource (like the CCN). Similar approaches can be adopted for any MPSoC platform but here, we adopt a hybrid approach. In the *SegBus* platform, a hierarchical solution is used, as a central arbiter works only if a some service needs cross border communication. The central arbitration unit (**CA**) deals with multicast service only when a multicasted packet is to be delivered across the border.

Zhang et al. explain the use of snoopy protocol for bus-based MPSoCs in [30]. It makes use of the broadcasting and the serialization properties of buses, resulting in a cheap solution.

Faizal et al. [21] presents the architecture of a multicast parallel pipeline router for NoC. The routing engine computes the direction from each header flit and writes it in the register of the routing table. After that, according to direction register entries, payload flits are broadcasted in multiple directions. In our case, just one or two (for both directions) packet copies are generated. This provides for power efficiency and simplicity of the implementation, compared to Samman et al. [21]. Ebrahimi et al. [12] propose the dynamic multicast routing protocol for traffic distribution in NoC. In this approach, packet prioritization service is not considered, which is a rising requirement for most of the upcoming applications. One may also infer also a power performance overhead at the source node, as all the destination addresses need to be sorted. There is no need of sorting and prioritization in our approach, where also prioritization is considered during the placement of processing elements.

Paper overview. The remainder of this paper is organized as follows. Section 2 presents the background of the *SegBus* platform, communication mechanism and previous research for on-chip communication services. Section 3 explains design methodology by tool environment and placement of processing elements. Section 4 describes the implementation details of communication services, offered by the platform. Section 5 explains the multicast mechanism and its hardware implementation. Finally, experimental results are presented – Section 6 and conclusions are drawn in Section 7.

2. Background

A segmented bus is a bus which is partitioned into two or more segments. Each segment acts as a normal bus between modules that are connected to it and operates in parallel with other segments. Neighboring segments can be dynamically connected to each other in order to establish a connection between modules located in different segments. Due to the segmentation of the bus, parallel transactions can take place, thus increasing the performance. A high level block diagram of the segmented bus system which we consider in the following sections is illustrated in Fig. 2.

The *SegBus* platform [24] is thought as having a single **CA** and several local segment arbitration units (**SA**), one for each segment. The **SA** of each bus segment decides which device, generically referred as *functional unit* (**FU**), within the segment will get access to the bus in the following transfer burst.

Platform communication. Within a segment, data transfers follow a "traditional" bus-based protocol, with **SA** s arbitrating the access to local resources. The inter-segment communication is a package based, circuit switched approach, with the **CA** having the central role. The interface components between adjacent segments, the *border units* – **BU**s, are basically FIFO elements with some additional logic, controlled by the **CA**. A brief description of the communication is given as follows.

Whenever one **SA** recognizes that a request for data transfer targets a module outside its own segment, it forwards the request to the **CA**. This one identifies the target segment address and decides which segments need to be dynamically connected in order to establish a link between the initiating and targeted devices. When this connection is ready, the initiating device is granted the bus access. This one starts filling the buffer of the appropriate bridge with the package data. The latter is taken into account by the corresponding next segment **SA** which forwards it further, until it reaches the destination. At this point, the **SA** of the targeted segment routes the package to the own segment lines, from here it is collected by the targeted device.

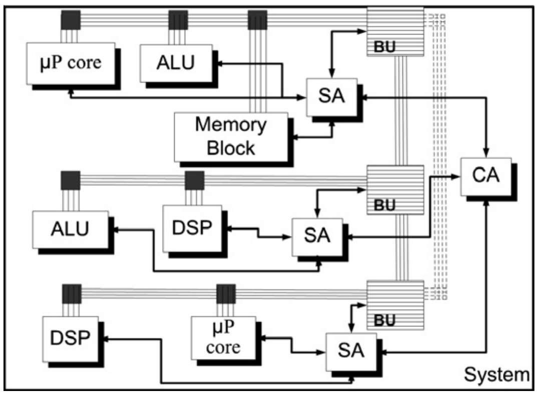


Fig. 2. Segmented bus structure.

A transfer from the initiating segment k to the target segment n is represented in Fig. 3. The packet structure shown in figure contains different fields. Destination ID field is the PE ID of packet destination. Similarly Source ID is the PE ID of packet source. Data payload is the actual data to be transferred. The segments from k to n are released for possible other inter-segment operations in a cascaded manner, from the source k to the destination, n as specified by the packet header. However, the figure stresses the relatively long duration of an inter-segment transfer: whenever the data has arrived in the **BU** FIFOs, such a transaction collides with on-going local activities. A solution in this sense, that is, speeding up the global communication, comes in the form of interrupts [22]: when a data package arrives at one **BU**, the local operations of the next segment to be traversed is interrupted, to make way for the inter-segment package.

The arbitration at **CA** level, that is, for global transfers, implements the application dataflow, with respect to these transfers. Hence, one has to implement accurate control procedures for inter-segment transfers, as possible conflicting requests must be appropriately satisfied, in order to reach performance requirements and to correctly implement applications.

The bus snooping mechanism is shown in Fig. 4. Wrapper (**W**) provides the abstraction between processing elements (**PE**) and communication platform to make the system plug-and-play. Here, the task of the wrapper is requesting the bus, reading data from bus and other control signals communication. In packet based communication, packetization and depacketization are additional tasks of the wrapper.

Tool Environment. At the time, the implementation technology for the *SegBus* platform is offered by Altera [1] devices. Hence, after application modeling and platform customization the flow is taken into the Quartus design environment, where previously defined functional units are mapped on actual devices. Following compilation, a simulation is performed within a Modelsim [2] framework.

3. Design methodology

The design methodology for *SegBus* platform has already been proposed by [23]. In this methodology, optimization of communication cost is considered according to the application and platform model. Communication cost can be further optimized by introducing different communication services like multicast, cache coherence or proper scheduling approach according to the application requirements. Fig. 5 shows the updated design methodology. We approach the problem by introducing the services block in the existing methodology. Services block offers different kind of services with a variety of architectures. The services are introduced one by one and performance improvement regarding communication cost is observed by simulation. The group of services, which provides the best performance can be selected for final implementation. The criteria to select some service for final implementation is discussed individually for each service in Section 4.

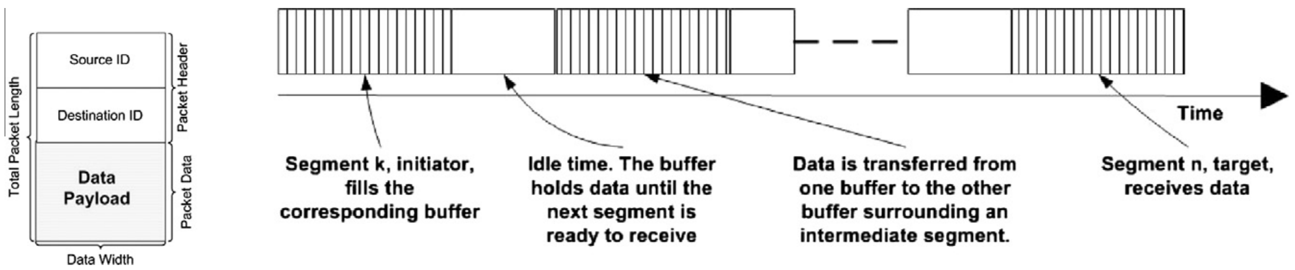

Source ID
Destination ID
Data Payload
Total Packet Length
Packet Header
Packet Data
Data Width
Time
Segment k, initiator, fills the corresponding buffer
Idle time. The buffer holds data until the next segment is ready to receive
Data is transferred from one buffer to the other buffer surrounding an intermediate segment.
Segment n, target, receives data

Fig. 3. The data packet structure and the flow of an inter-segment package transfer.

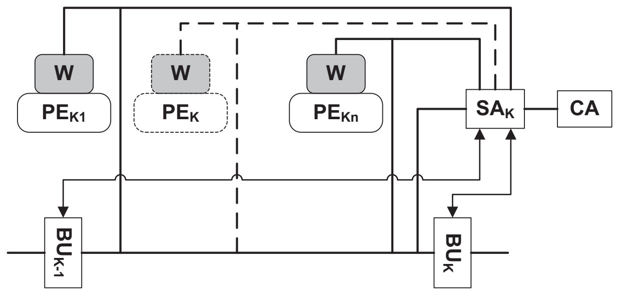


Fig. 4. Abstraction between computation and communication.

Application development. We start by analyzing the targeted application by splitting it in processes. The interaction between these is observed in terms of input–output data-flows. In subsequent steps the top-level process is decomposed hierarchically into less complex processes and the corresponding data-flows between these processes.

The decomposition process is based on designer's experience and ends when the granularity level of the identified processes maps to existent library elements or devices that can be developed by the design team. The communication between processes is organized as a *Packet SDF* diagram [23]. The PSDF representation will be used to extract the programs controlling the activity (grant distribution schedule) of the **SA** s and of the **CA** [26].

The *Packet SDF*. A PSDF comprises mainly two elements: *processes* and *data flows*; data is, however, organized in packets. Processes transform input data packets into output ones, whereas packet flows carry data from one process to another. A *transaction* represents the sending of one data packet by one source process to another, target process, or towards the system output. In [23], a *packet flow* is a tuple of two values, *P* and *T*. *P* represents the

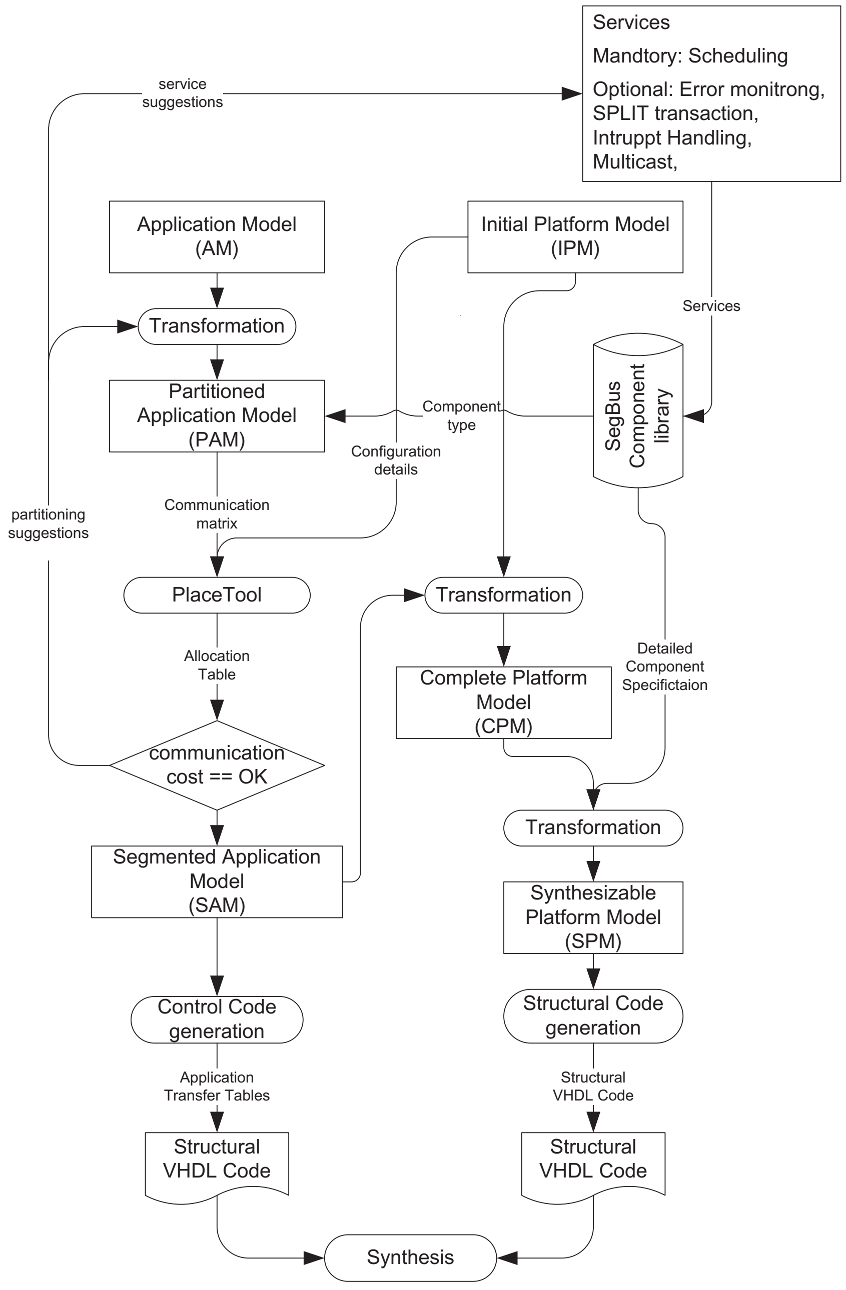


Fig. 5. *SegBus* design methodology.

number of successive, same size transactions emitted by the same source, towards the same destination; T is a relative ordering number among the (package) flows in one given system. Thus, a flow is understood as the number of packets issued by the same process, targeting the same destination, and having the same ordering number.

A third element of the PSDF tuple characterizes the *kind* of the packet. The *kind* – a natural number – identifies if a packet is to be routed to multiple destinations, thus establishing the modeling basis for multicasted or broadcasted transmissions. Packet flows having same I value carry the same content of data, from the same source towards multiple destinations.

The PSDF of a certain system becomes hence a sequence of packet flows, $\langle(P_1,T_1,I_1),\ldots,(P_n,T_n,I_n)\rangle$, where $T_1 \leqslant T_2 \leqslant \ldots \leqslant T_n$. Flows sourcing in the same node and with identical Is, will also have identical Ts, identifying a multicast packet.

The non-strictness of the relation between T values of the above definition models the possibility of several flows to coexist at moments in the execution of the system. In the case of the *SegBus* platform, this most often will describe *local* flows, that is flows where the source and the destination are situated in the same segment. However, considering a segment number larger than 3, *global* flows, where the source and the destination are in different segments, are also possible to be characterized by the same ordering number. In this case, it means that the **CA**, if possible, allows a simultaneous execution of transactions from all the "same T" global flows.

The H.264 encoder application is shown in Fig. 6. The values on the edges in Fig. 6 represent the number of transaction packets for processing of one video frame. The corresponding PSDF diagram is shown in Fig. 7. For the moment, the reader should ignore the partition in segments, which is based on developments in the next

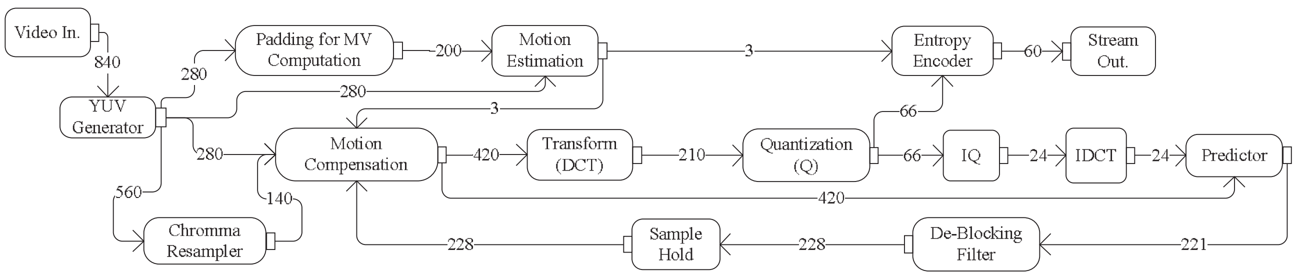


Fig. 6. The H.264 Video Encoder application.

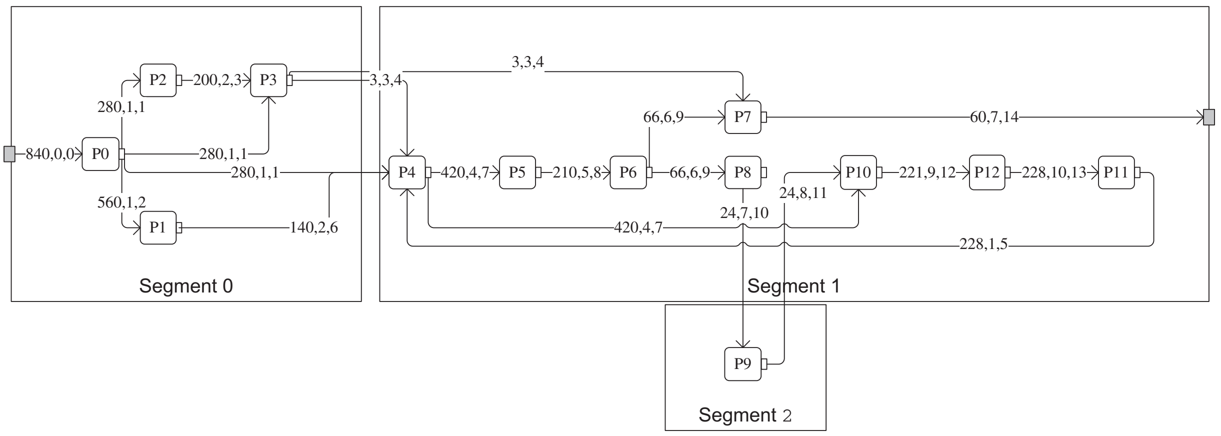
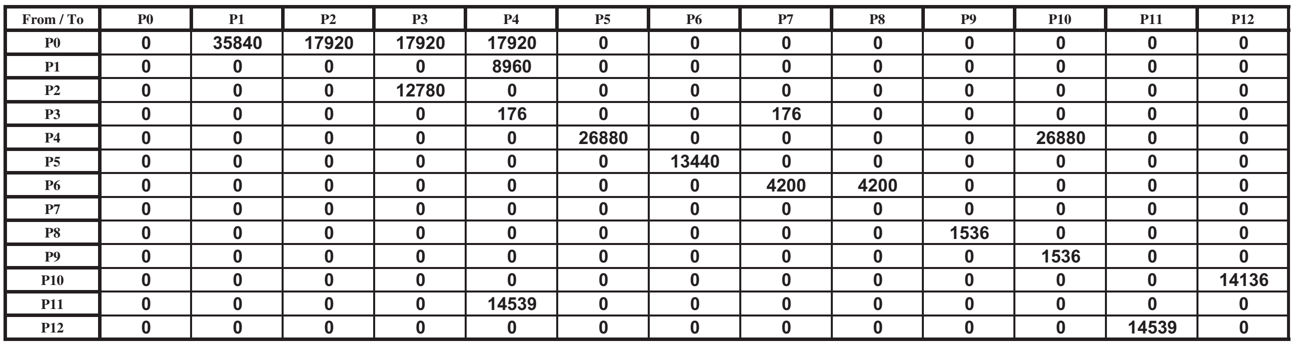


Fig. 7. PSDF application specification.



From / To	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
P0	0	35840	17920	17920	17920	0	0	0	0	0	0	0	0
P1	0	0	0	0	8960	0	0	0	0	0	0	0	0
P2	0	0	0	12780	0	0	0	0	0	0	0	0	0
P3	0	0	0	0	176	0	0	176	0	0	0	0	0
P4	0	0	0	0	0	26880	0	0	0	0	26880	0	0
P5	0	0	0	0	0	0	13440	0	0	0	0	0	0
P6	0	0	0	0	0	0	0	4200	4200	0	0	0	0
P7	0	0	0	0	0	0	0	0	0	0	0	0	0
P8	0	0	0	0	0	0	0	0	0	1536	0	0	0
P9	0	0	0	0	0	0	0	0	0	0	1536	0	0
P10	0	0	0	0	0	0	0	0	0	0	0	0	14136
P11	0	0	0	0	14539	0	0	0	0	0	0	0	0
P12	0	0	0	0	0	0	0	0	0	0	0	14539	0

Fig. 8. The communication matrix for the example.

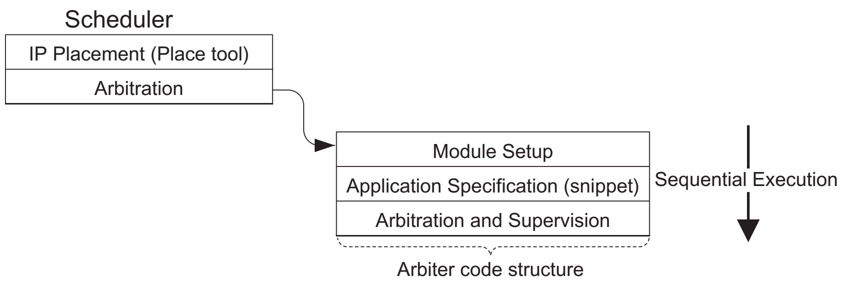

Scheduler
IP Placement (Place tool)
Arbitration
Module Setup
Application Specification (snippet)
Arbitration and Supervision
Arbiter code structure
Sequential Execution

Fig. 9. *SegBus* scheduler structure.

sections. The processing elements (*P*0, *P*1, ..., *P*12) correspond respectively to YUV generator, Chroma resampler, Motion vector estimator units, etc. Fig. 8 shows the number of bytes to be transferred between to PEs for the processing of one video frame.

4. Development of communication services

In this section, we present the current available services offered by the *SegBus* platform. The features, initially not perceived as services, provide the background for further developments.

4.1. Scheduling

Scheduling is the basic and mandatory service to use the interconnection platform. Scheduling can be divided into two steps: task allocation and arbitration as shown in Fig. 9.

Communication features of the running application are needed for the proper placement of IPs. Here, we are interested in the transaction frequency between processing units, their relative sequencing and scheduling. System performance will depend on the utilization of throughput and the balanced traffic load. With all these considerations, the *PlaceTool* [25] has been developed, to deliver the allocation cost for various scenarios.

For *SegBus*, the *PlaceTool* works as the task allocator. The communication matrix Fig. 8 is extracted from PSDF diagram and fed to the *PlaceTool* to approximate the effect of segmentation on the performance of the application (tasks)/platform mapping. After having the placement of tasks and processes, next step is the scheduling on bus, which is controlled by arbitration. For *SegBus* platform, arbitration mechanism can be further divided into three steps as depicted in Fig. 9 and explained in Section 4.1.2.

For an example, consider the arbitrary task graph shown in Fig. 10. t_A represents the processing time for task A and similarly, the processing time for other tasks is mentioned. *PlaceTool* allocates the tasks A, F, G, H, I on segment '0' and the tasks B, C, D, E on segment '1'. The scheduling on a single bus and also for the segmented bus with two segments is presented in Figs. 11 and 12 respectively. The context switching time in scheduling is considered zero. The total execution time is reduced by 21% but this value is application dependent. The detailed description of *PlaceTool* and arbitration mechanism is presented in detail in Sections 4.1.1 and 4.1.2 respectively and H.264 video encoder is used as a running example (see Fig. 6).

4.1.1. Placement of processing elements

The communication frequency for the H.264 video encoder, as captured by the communication matrix shown in Fig. 8, is obtained from running a Simulink Model of the application. The matrix is further fed into the *PlaceTool*. Results of this exercise for communication cost are presented in Section 6. The resulting segmented application model for H.264 video encoder application is shown in Fig. 7.

4.1.2. Arbitration

The **SA** s and the **CA** are VHDL defined modules, with a similar structure. The code runs with multiple parameters as required by the platform specification. We see the application as a set of correlated transactions that must be ordered in their execution by the arbiters. The specification of the schedule – as supplied by the PSDF representation, is provided by a snippet introduced in the **SA** or the **CA** codes, representing the projection of the application flow at the respective level and location [26].

The structure of the arbiters is depicted in Fig. 9. The "Module SetUP" and the "Arbitration & Supervision" blocks are concerned with application-independent procedures, such as reading the input signals, selecting the granted master, and counting the number of transactions performed in a granted activity. The middle block, "Arbitration specification", brings in the application specific requirements for scheduling grant decisions.

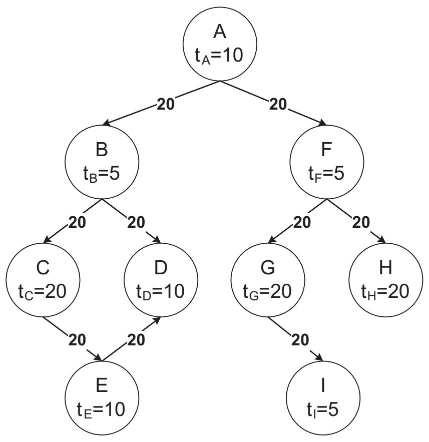


Fig. 10. Task graph.

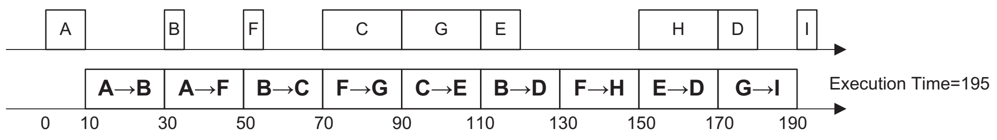


Fig. 11. Single bus scheduling.

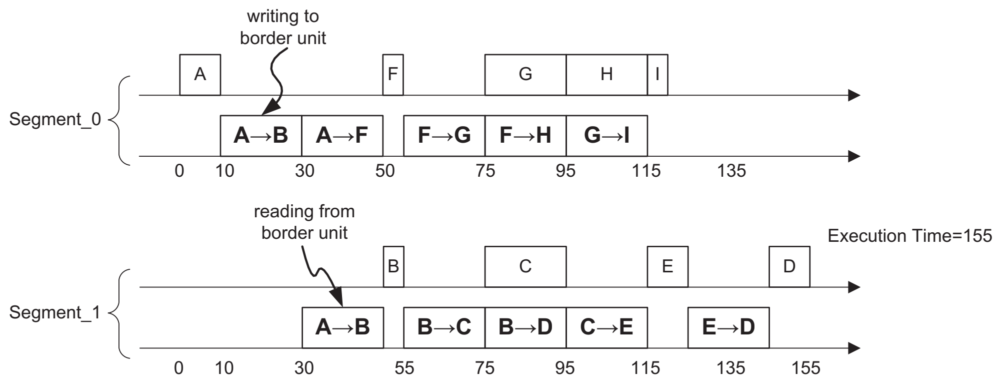


Fig. 12. Two segment *SegBus* scheduling.

The application snippet is part of the actual arbiter VHDL code, and, as such, will be executed. The addressed variables will be read or written by the other arbitration code blocks.

SA level arbitration. The segment level arbitration is similar to any single segment bus situation. Activities in the segment are sequential, the **SA** deciding which device can access the bus lines. Any attached **BU** behaves like a local master, but the respective requests will have the highest priority. A master willing to transfer data on the bus raises the request line, while it also specifies the segment to which it wants to communicate. The **SA** identifies the target and, if it is outside the own segment, it forwards the request to the **CA**. If the request target is within the own segment, it proceeds to granting it.

These activities are collected in the *application control code* (ACC) which will drive the *SegBus* communication strategy at runtime [26]. The ACC is basically a binary matrix where each line controls the granting algorithm such that the "right" master obtains the access to the bus. The code is parsed at every arbitration execution, and it contains *nrLines* lines of code – a parameter of the arbiter module. One line of code, assimilated to a *program line* (an array) has the following field structure (see also Fig. 13), Where the destination (*dest*) field has more than one values for multicast purpose.

- *PC*. This is the Programme Counter, providing reference to the lines of instructions possible to be accessed from other instructions. It ranges from 0 to *nrLines* − 1.
- *source*. Identifies the requesting master's ID.
- *dest*. Identifies the target slaves. The number of maximum targets for one transmission is a parameter of the arbiter module (*max_dest*). If one of the *dest* sub-fields equals the ID of the *source*, the content is ignored.
- *dest_seg*. Identifies the target slave's segments. The number of maximum targets for one transmission is a parameter of the arbiter module (*max_segs*). This in compliance with the allocation results and the *dest* field content. The sub-fields ignored for the *dest* specification will also be ignored here.
- *count*. Identifies the number of packets the master has to send to the specified targets. It corresponds to the first number in the PSDF description.
- *guard*. When *guard* = 0, the respective line is *enabled*, that is, the arbiter may consider it for selection. When *guard* > 0, the line is *disabled*, that is, it cannot be considered in the arbitration. The arbiter marks a line as *executed* whenever the respective *count* value reaches *0*, by establishing *guard* = *nrLines.*
- *enables*. Whenever a line is marked *executed*, the **SA** will *enable* the line specified by this field, by subtracting 1 from it's current *guard* value. In order to become enabled, a line with an initial *guard* > 1 will require that several previous operations (execution lines) to have finished. If, for a given line, *enables* = *nrLines*, then the arbiter does not try to enable any other line, when the current one is marked *executed*. One line may enable multiple downstream lines. The number of maximum enable targets for one line is a parameter of the arbiter module (*max_enable*). If one of the sub-fields equals the current line number, the information is ignored by the arbiter.

Example:

PC	Guard	Source	Destination			Dest_Seg	toGrant	count	enables		
0	0	0	1	X	X	0	0	560	2	X	X
1	1	0	2	3	4	0	1	280	3	4	5
---	---	---	---			---	---	---	---		
5	1	4	10	5	X	1	4	420	7	11	X
---	---	---	---			---	---	---	---		

Fig. 13. Program line example, with parameters: *max_dest* = 3, *max_segs* = 3, *max_enable* = 4.

The VHDL code corresponding to the table in Fig. 13 is:

```
-- SA segment 0 snippet
program(0) <= (guard => 0, source => 0,
    dest1 => 1, dest2 => 0, dest3 => 0,
    dest_seg => 0, count => 16,
    enables1 => 1, enables2 => 0, enables3 => 0);
program(1) <= (guard => 1, source => 0,
    dest1 => 2, dest2 => 3, dest3 => 4,
    dest_seg => 0, count => 16,
    enables1 => 3, enables2 => 4, enables3 => 5);
...
```

The application execution ends when all the lines are marked *executed*. That is, we have $PC = nrLines - 1$ and, for all lines, $guard = nrLines$. This triggers the arbiter to restore the initial values of the ACC content.

A similar approach is taken at the level of the **CA** for request–grant activities (containing only info about segment requests).

4.2. SPLIT transactions

Either the transfer is local or across the border, it is possible that the slave is not able to receive the data or can not respond the master with required data immediately. In this situation, master will hold the bus and other masters can not utilize the bus, even when it is free. This reduces the degree of bus utilization. A *SPLIT transfer* improves the overall bus utilization by splitting the operation of the master providing the address to a slave from the operation of the slave responding with the appropriate data [3]. Thus by using the *SPLIT transfers* idle bus cycles can be used for other transactions.

The basic criteria to have SPLIT transaction service depends on bandwidth requirements of application and the packet size. If packet size is very small, SPLIT transaction overhead to arrange the SPLIT mechanism will not be economical and even might reduce the utilization. According to the design methodology discussed in Section 3, SPLIT mechanism can be introduced and improvement in communication cost can decide to either use or do not use the SPLIT service.

The *SegBus* communication operation situations can be divided mainly into two categories: Local situation and Cross border situation. Cross border situation can be further divided into three situations [24]: Local–External, External–External and External–Local. In the following sections, we detail the *SPLIT transaction* approach for mentioned communication situations.

4.2.1. Local SPLIT transactions

In this situation, both the communicating modules are placed in same bus segment. The initialing master requests the bus ownership by raising the *req* signal to the corresponding local **SA**. There are two reasons, when bus ownership cannot be granted to the requesting master for local transaction. First, if another transaction is under completion on the bus. Second, the target slave is busy and cannot respond the request. If the bus is busy, **SA** assigns the bus ownership to the requesting master later at some time. If the target slave is not able to serve the request, **SA** assigns the bus ownership to another master until the slave is not ready to serve, which is exactly the same mechanism as employed in AMBA busses [3].

4.2.2. Cross border SPLIT transactions

SPLIT transaction for cross border communication is the enhancement of existing SPLIT mechanism. Consider the situation that a master module from segment0 requests to send the data packet to a slave located in segment2. In this transaction, segment1 will be used as well because it is on the way of transaction. It will not be an economical option to allocate all the bus segments for this transaction for the time span of generating the packet at source and delivering it to the destination. To enhance the bus utilization in this scenario, we split the transaction into the number of steps equal to the number of segments including the source and destination segment on its way. Neighboring **BU** s are considered as local modules for the corresponding **SA**. Thus the SPLIT mechanism discussed in Section 4.2.1 will be followed for each segment traversal.

The bus request come along with target slave address. **SA** checks, if the request can be served in current segment or not. If the request cannot be served in current segment (inter-segment request), request is forwarded to the **CA** with target address. In the meanwhile, no other external request is served by the **SA**, until the pending operation is completed. Here, the bus can stay idle for a number of cycles. To deal with this situation, interrupt communication service was introduced as discussed in Section 4.3. In this section, we address another issue of bus utilization.

Whenever the **CA** is able to serve the request, it informs the **SA** s, from initiator to target of the imminent transfer (signal SOP – operate). As soon as the current operation finishes in the initiator segment, that **SA** grants the requesting master to access either the left, or the right **BU**. In a circular set-up, the **CA** selects the shortest possible distance from the initiator segment to the final one.

Upon filling up the FIFO, the **BU** informs further the next segment that data is waiting to be transferred. The corresponding **SA** allows for the current operation to end, after which it will grant the transfer from one segment border to the other (by setting the granting lines (GFL or GFR, respectively). Hence, the package waits in the FIFO the period of time required to end the current local transfer in the next segment. When this operation completed, the **CA** receives the OPF (operation finished) signal from the corresponding **SA**, and answers by lowering the respective SOP line. When OPF is also reset, the segment is ready for a new inter-segment transfer. The whole mechanism is shown in Fig. 14. In a cascaded manner, the above scenario repeats all the way to the target segment (as also illustrated in Fig. 3).

4.3. Interrupt communication

As mentioned in Section 4.2.2, a packet may have to wait in **BU** for number of clock cycles during the cross border transactions. By using an *interrupt service*, the delay in **BU** can be significantly reduced [22]: an inter-segment transfer, when reaching one of the **BU** s on the road from source to destination, will preempt the local activities of the next segment to be crossed.

The local **SA** is the controller that supervises any activity within the segment. The moment of interruption, with respect to the completion of the running local transfer, while the data package is waiting in the intermediate **BU** FIFO, is of prime importance with highest criticality value. Hence, the decision to interrupt, or continue the current local activity will fall into the attributions of the **SA**. The interrupt transaction will be non-preemptive from start to the completion of execution.

In every clock cycle during the execution of a local activity, the corresponding **SA** monitors if an external request for inter-segment data transfer is raised. When such request is detected, the local grant is put down in the subsequent clock cycles. The whole process from detection of interrupt to the resetting of local grant takes four clock cycles. The ID of the master that has just been interrupted is saved by the **SA** and it will be granted again access, immediately when the inter-segment transaction completes. The

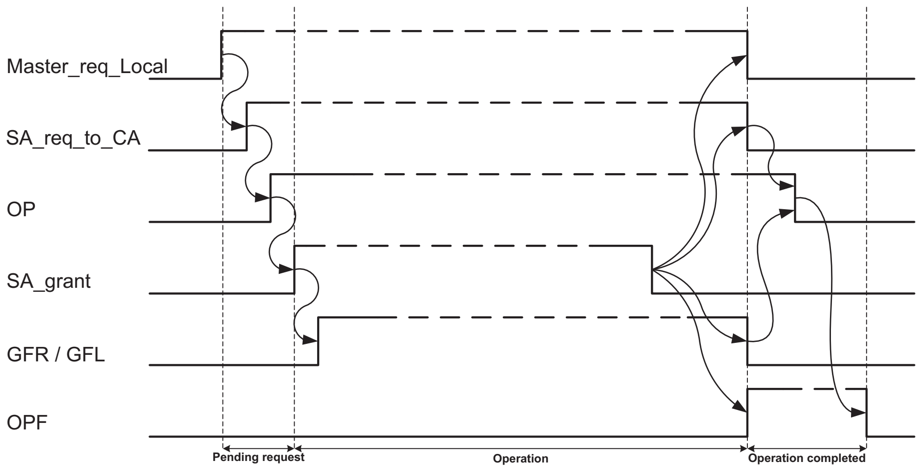


Fig. 14. Inter segment transfer control.

respective master then continues to send the information remaining from the interrupted operation.

To illustrate the interrupt mechanism, consider the task graph shown in Fig. 10. Suppose that it is the graph of a streaming application like audio/video codec and all these tasks execute repeatedly. As shown in Fig. 15, after completing the local transaction G → I on segment 0, next transaction will be again cross the border (A → B). After filling the **BU**, an interrupt will be generated by border control unit to the **SA** of segment 1, which will preempt the current transaction and will read the buffered data from **BU** for transaction A → B. The context switching time is supposed to be zero for to make the explanation simple. After reading the complete data packet from **BU**, the interrupted transaction (E → D) will be resumed. Now consider the situation that there is no interrupt service available. In that situation, not only the transaction A → B will be delayed but the rest of the processing and transactions on segment 1 will be delayed as well and this delay value will go on increasing because of previous delay. Thus after few application cycles, segment 1 will be lagging too much behind segment 0. In this way, interrupt communication enables the pipelining of tasks on *SegBus*.

The selection criteria of interrupt service to use for final implementation is data dependency and urgency. It depends on the application the how urgent, the data packet stored in **BU** is needed by the destination node. Another issue is data dependency: how many nodes will have to wait directly or indirectly due to the delay of the packet in **BU**. In Fig. 15, the data packet for transaction A → B will not delay only the processing on node B but all of the processing elements on segment1. Thus, improvement in communication cost will be the parameter to favor interrupt service to be used for final implementation.

5. Multicast transactions

We introduce here an additional communication feature, namely *multicast transactions*, meant to further improve performance aspects of the platform, in the situations when a single device must send the same data packet to multiple destinations. Such a situation can be observed for the application at hand, in Fig. 7: *P*0 has to send the same packet no less than three times, to *P*2, *P*3 and *P*4, respectively.

Without the multicast feature, *P*0 has to execute three requests and send, in some sequence, the data to the necessary destinations. It is natural that a single transaction, if possible, would dramatically reduce thus the communication load, at least in this context.The multicast service will show improvement in communication cost, only if, there is a big fraction of identical data in the application.

In the following sections we illustrate the impact of providing such multicast feature on the activities performed by the local and central arbiters.

5.1. Implementation

For multicast transactions, the source device request to the corresponding **SA** is accompanied by the destination IDs without

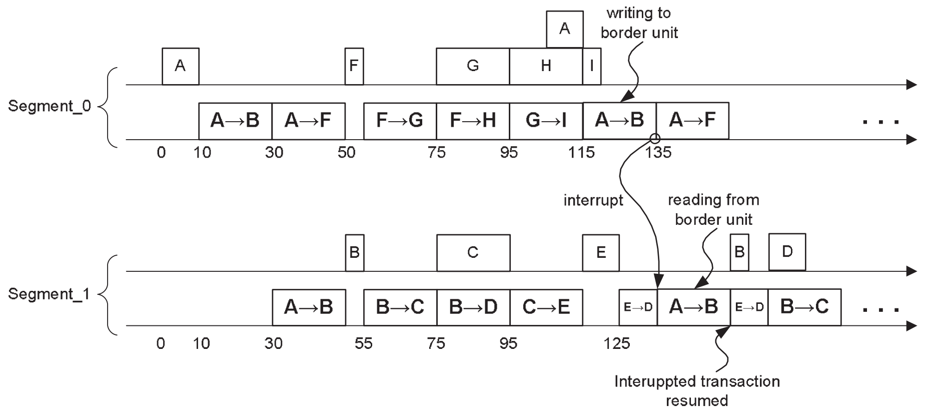
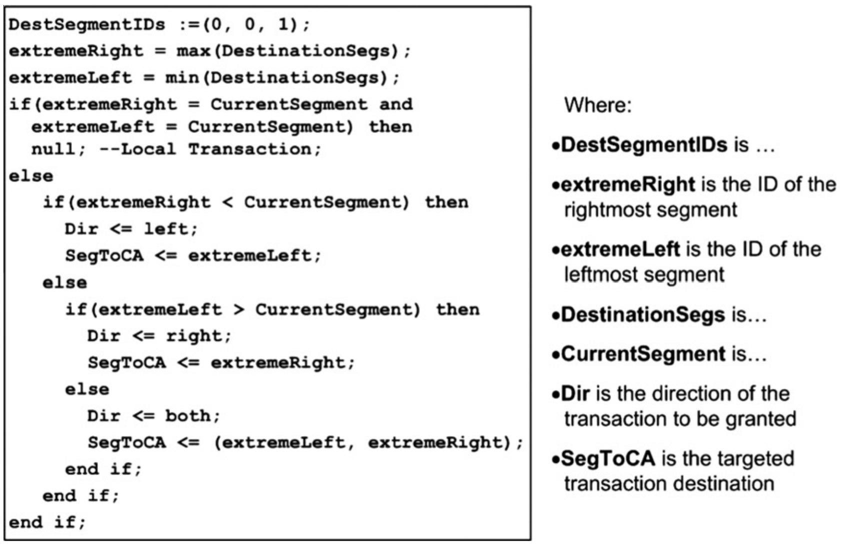


Fig. 15. Interrupt scheduling.



```
DestSegmentIDs :=(0, 0, 1);
extremeRight = max(DestinationSegs);
extremeLeft = min(DestinationSegs);
if(extremeRight = CurrentSegment and
  extremeLeft = CurrentSegment) then
  null; --Local Transaction;
else
   if(extremeRight < CurrentSegment) then
     Dir <= left;
     SegToCA <= extremeLeft;
   else
     if(extremeLeft > CurrentSegment) then
       Dir <= right;
       SegToCA <= extremeRight;
     else
       Dir <= both;
       SegToCA <= (extremeLeft, extremeRight);
     end if;
   end if;
end if;
```

Where:

- **DestSegmentIDs** is …
- **extremeRight** is the ID of the rightmost segment
- **extremeLeft** is the ID of the leftmost segment
- **DestinationSegs** is…
- **CurrentSegment** is…
- **Dir** is the direction of the transaction to be granted
- **SegToCA** is the targeted transaction destination

Fig. 16. Packet read mechanism.

having any information of their relative placement. It is the task of the **SA** to identify the respective destination segment. A local table is available to all **SA**s, indicating the placement of resources as from the *PlaceTool* selection. **SA** will read the requested slave IDs from the program line to compute the direction and destination segment or segments (for both directions). The table is modeled by the assignment:

```
--(Processing Unit) => (Segment ID)
Segmentation <= (0 =>0, 1 =>0, 2 =>0, 3 =>0, 4 =>1, 5 =>1, 6 =>1,
                 7 =>1, 8 =>1, 9 =>2, 10=>1, 11=>1, 12=>1);
```

In case of a broadcast transaction, the requested slave ID (the destination) is a universal code "11…1" (which must not be assigned to any processing element). The width of this requesting code will be equal to the data bus width. Broadcast is a special case of multicast transaction. Multicast is used more often in todays on-chip applications. So, for implementation details, we focus on multicast transactions.

Multicast transaction. Fig. 7 shows that the third value of the tuple is the same for communication links from source P0 to destinations *P*2, *P*3 and *P*4. The payload of these packets is the same. In this case, a single packet can be transmitted instead of sending three different copies of identical packets. So, P0 will request the processing element with ID "1111", the broadcast code. **SA** will read *dest* from the respective programme line after receiving the request from P0. *dest* will provide the destination slave IDs 2, 3 and 4. **SA** then obtains the corresponding destination segment IDs from *Segmentation* (0, 0 and 1 respectively).

```
program(0) <= (guard => 0, source => 0, dest => 2,3,4, togrant => 0,
               count => 280, enables => 4,5,6,7);
```

Using the requested segment IDs, only one or two segments will be selected for transmitting packets either in left, right or in both directions. The selection of segments and direction for current transaction is made as illustrated in Fig. 16.

This example code corresponds to the multicast transaction initiated in segment '0'. After computing the destination segment IDs, the **SA** will decide if the transaction is local or across the border. If the transaction is local, **SA** will check the status of segment and make proper signaling to initiate the transaction. If the transaction is across the border, **SA** will forward the request to **CA** with destination segment IDs. After receiving the signal *InS* from **CA**, **SA** will allow the requester to start the transaction.

Then packets are transmitted according the new packet format, shown in Fig. 17. The packet header contains the operational code with destination address because some processing elements may offer more than one operations. Source opCode is the operation done by source element for current packet generation.

Once the communication link is established and the packet is injected into the platform, the slave will read the packet according to the mechanism shown in Fig. 18. When the packet header arrives and *granted* is inserted by **SA**, all slaves start snooping the data bus. If destination ID is matched in packet header to the current slave, *Slave_Acq* is raised high. Slaves snoops the bus, even after this event because more than one operational code may be assigned to one slave for a single packet. *Slave_Acq* enables *Slave_Data_Read* at the end of packet header.

In NoC, packet prioritization requires extra processing for broadcast or multicast communication. For the *SegBus* platform, no prioritization is required because processing elements are placed with the consideration of communication requirements and packets are transmitted towards the extreme destination segments. By inspecting the bus lines, each destination will receive the requested traffic. In the same way, a very economic cache coherence snoopy protocol can be implemented.

6. Experimental results

The output results of the *PlaceTool* for H.264 encoder application (Section 4.1.1) are shown in Fig. 19, where || represent the segment borders. It can be observed that performance may go down by increasing the number of segments due to increase in communication overhead. In this case, a two segment platform delivers the best performance; however, we decide to select a three segment platform, in order to analyze a more complex structure as explained in [14].

The suggested multicast service was applied to the H.264 video encoder application, with different number of segments. Results of around 24% reduction in traffic load show a significant reduction in latency and communication overhead. This comes in comparison

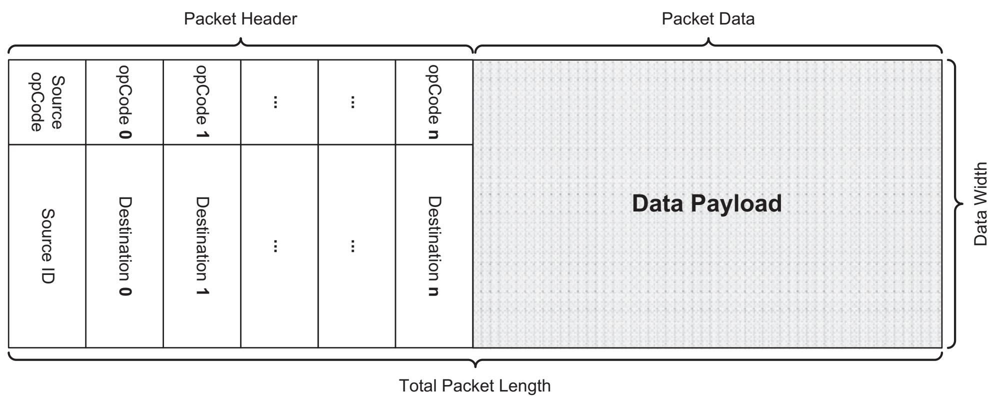


Fig. 17. Packet format.

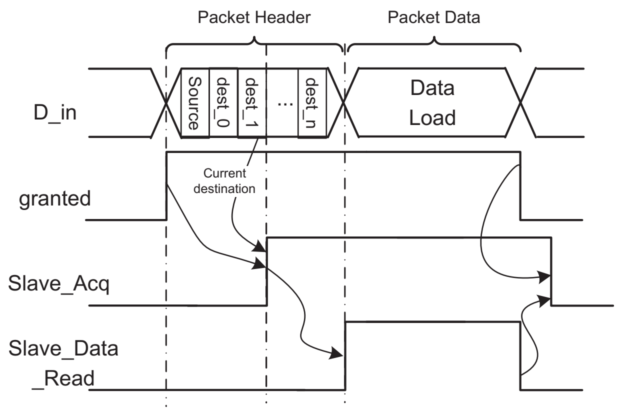


Fig. 18. Packet read mechanism.

to the original, "not multicast enabled" *SegBus* platform. These results are application dependent because multicast service will show significant improvement when a big fraction of identical traffic is injected to the platform. The best performance is achieved when all multicast destinations are placed in one direction with reference to source processing element. However, this is a difficult arrangement to reach.

The interrupt approach brings further improvements to the platform performance. The communication load with and without multicast service for each platform component is shown in Fig. 20. It can be observed that the individual segment load values ($S0, S1$) show significant reduction in the presence of the multicast service. But there is no significant improvement on border unit loads ($BU0, BU1$), due to an efficient placement by the tool – as indicated, the *PlaceTool* already tries to minimize the cross border transactions.

Reduction in the total system power is proportional to the reduction in traffic load. However during broadcast, all destination elements read the bus, thus increasing the capacitive load on the bus. Still, the reduction in power due to the reduction in traffic load dominates the power consumption overhead due to capacitive load. Power consumption results for *SegBus* platform with and

Nr. Segs	Cost	Allocation	Improvement
1	233000	0 1 2 3 4 5 6 7 8 9 10 11 12	100%
2	132000	4 5 6 7 8 9 10 11 12 || 0 1 2 3	-43%
3	137400	0 1 2 3 || 4 5 6 7 8 10 11 12 || 9	-41%
4	143100	9 || 8 || 4 5 6 7 10 11 12 || 0 1 2 3	-39%

Fig. 19. The allocation and associated cost results.

Segbus Element	S0	S1	S2	BU0	BU1
Transactions without multicast	1746	2333	48	426	48
Transactions with multicast	1183	1844	48	423	48
reduction in number of transactions	32%	21%	0%	0.7%	0%

Fig. 20. Communication cost with and without multicast service for H.264 application with three segments.

SegBus Platform	without Multicast service	with Multicast service	Improvement
Static Power Consumption	755.64 mW	755.72 mW	-0.01%
Dynamic Power Consumption	137.15 mW	134.20 mW	2.15%

Fig. 21. Platform power consumption with and without Multicast service.

without multicast service are presented in Fig. 21. The results have been extracted by Altera *PowerPlay* tool [1].

7. Conclusions

We have illustrated here a new perspective on the *SegBus* platform, based on services. Each service is considered individually, but relations between them can be also noted (however, at this moment, not quantifiable). We have also introduced a solution that provides multicast services on the *SegBus* platform, and shown the impact on arbitration and scheduling, down to the VHDL code. Our intuition about the performance enhancement was proved correct by the implementation results as exercised on the H.264 encoder application, where a further improvement in performance has been observed. The broadcasting feature is implemented with a minimal overhead in terms of arbitration computation, and in terms of data packet size. While the former impact is overcome by the parallel activities of the arbiter and of the functional modules, the latter can be seen as a small price for a possibly very large improvement in performance, when multiple destinations are required.

Future work. A very necessary step is to introduce new services apart from the already supported ones. The most critical issue is system monitoring for complex applications running on MPSOC platforms. In the current research, the focus was to introduce services at interconnection level. Another aspect is to introduce services at core level for monitoring purposes. Core temperature, for instance, can be monitored by **SA** s, that may power-down or up the respective cores, as required in order to perform the assigned tasks. Temporary out-of-use situations would be notified to the **CA**, for further considerations. Thus thermal, leakage or other monitoring service can also be integrated in the existing arbitration architecture.

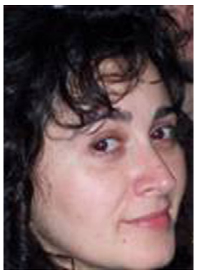
In the larger context of employing the *SegBus* platform as an infrastructure for embedded systems, we also plan to integrate the current design methodology and features into a higher level design framework. This will be provided by the *REMES* framework [27], offering the possibility to access its the real-time, component-based and formal verification capabilities.

References

[1] Stratix III Device Handbook 2007, Altera, 2007.
[2] Modelsim. <http://www.model.com>.
[3] ARM AMBA Specification and Multilayer AHB Specification (rev 2.0). <www.arm.com>.
[4] C. Anderson, J.-L. Baer. *Two techniques for improving performance on bus-based multiprocessors.*, in: HPCA '95: Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture, 1995, pp. 256–275.
[5] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, Metropolis: an integrated electronic system design environment, Computer 36 (4) (2003). pp. 45-52+4.
[6] A. Bakshi, V.K. Prasanna, A. Ledeczi, MILAN: a model based integrated simulation framework for design of embedded systems, SIGPLAN Not. 36 (8) (2001) 82–93.
[8] S. Borkar, Designing reliable systems from unreliable components: the challenges of transistor variability and degradation, IEEE Micro 25 (6) (2005) 10–16.
[9] S. Cho, G. Lee, Reducing coherence overhead in shared-bus multiprocessors, Euro-Par II (1996) 492–497.
[10] F. Dahlgren. Boosting the performance of hybrid snooping cache protocols, in: 22nd Annual International Symposium on Computer Architecture (ISCA), 1995, pp. 60–69.
[11] W. Dally, Route packets not wires: on-chip interconnection networks, DAC – Des. Autom. Conf. (2001) 684–689.
[12] M. Ebrahimi, M. Daneshtalab, M.H. Neishaburi, S. Mohammadi, A. Afzali-Kusha, J. Plosila, H. Tenhunen, An Efficent Dynamic Multicast Routing Protocol for Distributing Traffic in NOCs, Des., Autom. Test Eur. (2009) 1064–1069.
[13] A. Jantsch, H. Tenhunen (Eds.), Networks on Chip, Kluwer Academic Publishers, Boston, 2003. ISBN 1-4020-7392-5.
[14] K. Latif, M. Niazi, T. Seceleanu, H. Tenhunen, S. Sezer. Application development flow for on-chip distributed architectures, in: Proceedings of the 21st IEEE International SoC Conference (SOCC), September 2008, pp. 163-168.
[15] Z. Lu. Design and Analysis of On-Chip Communication for Network-on-Chip Platforms. Ph.D. thesis. Royal Institute of Technology, March 2007.
[17] C. Nicopoulus, V. Narayanan, C.R. Das, Network-on-Chip Architectures: A Holisitic Design Exploration, Springer, 2009. ISBN 978-90-481-3030-6.
[18] M.F.S. Oliveira, a. Eduardo W. Bri F.A. Nascimento, F.R. Wagner, Model driven engineering for MPSoC design space exploration, in: Proceedings of the 20th Annual Conference on Integrated Circuits and Systems Design. ACM, New York, NY, USA, 2007, pp. 81–86.
[20] A. Pimentel, L. Hertzbetger, P. Lieverse, P. van der Wolf, E. Deprettere, Exploring embedded-systems architectures with Artemis, Computer 34 (11) (2001) 57–63.
[21] F.A. Samman, T. Hollstein, M. Glesner, Multicast parallel pipeline router architecture for network-on-chip, Des., Autom. Test Eur. (2008) 1396–1401.
[22] A.D. Swaminathan, T. Seceleanu, Interrupt Communication on the SegBus platform, in: Proceedings of the IEEE International System on-chip Conference, Austin, TX, USA, September 2006, pp. 229–232.
[23] D. Truscan, T.Seceleanu, H. Tenhunen, J. Lilius. A Model-Based Design Process for the SegBus Distributed Architecture. 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS), 2008. pp. 307–316.
[24] T. Seceleanu, The SegBus platform – architecture and communication mechanisms, Journal of Systems Architecture 53 (4) (2007) 151–169. http://dx.doi.org/10.1016/j.sysarc.2006.07.002.
[25] T. Seceleanu, V. Leppänen, O. Nevalainen, Improving the performance of bus platforms by means of segmentation and optimized resource allocation, EURASIP J. Embed. Syst. 2009 (2009) 14, doi:10.1155/2009/867362. Article ID 867362.
[26] T. Seceleanu, I. Crncovik, C. Seceleanu. Transaction level control for application execution on the SegBus Platform, in: Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC) 2009, doi:doi:10.1109/COMPSAC.2009.78.
[27] C. Seceleanu, A. Vulgarakis, P. Pettersson, REMES: A Resource Model for Embedded Systems, in: Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009), IEEE Computer Society, June, 2009.
[28] A.S. Tranberg-Hansen, J. Madsen. A Service Based Component Model for Composing and Exploring MPSoC Platforms, in: The Proceedings of IEEE International Symposium on Applied Sciences in Bio-Medical and Communication Technologies (ISABEL), 2008, pp. 1–5.
[29] A.L. Varbanescu, H. Sips, A. Van Gemund PAM-SoC: A Toolchain for Predicting MPSoC Performance, Lecture Notes in Computer Science, vol. 4128, LNCS, 2006, pp. 111–113.
[30] Y. Zhang, Z. Lu, A. Jantsch, L. Li, M. Gao. Towards hierarchical cluster based cache coherence for large-scale network-on-chip, in: Proceedings of the 4th IEEE International Conference on Design and Technology of Integrated Systems in Nanoscale Era, 2009.
[31] A. Pimentel, C. Erbas, S. Polstra, A systematic approach to exploring embedded system architectures at multiple abstraction levels, IEEE Trans. Comput. 55 (2) (2006) 99–112.
[32] Claas Cornelius, Hendrik Bohn, Dirk Timmermann Service-oriented Approaches for the Operation of large on-chip Networks, in: 24th NORCHIP Conference, ISBN: 1-4244-0772-9, Linköping, Schweden, 2006, pp. 183-186.

Khalid Latif is currently working towards his Ph.D. degree at the University of Turku, Finland. He has a M.Sc. degree in Electrical Engineering with major in System-on-Chip design from Royal Institute of Technology (KTH), Sweden and B.Sc. degree in Electrical Engineering with major in Electronics and Communication from the University of Engineering and Technology (UET), Lahore, Pakistan. His research interests include On-Chip Interconnection platforms, Computer Architecture and FPGA based design.

Tiberiu Seceleanu received the MSc (1994) and Lic.Sc (1995) degrees from the Polytechnic University in Bucharest, Romania. He got his Dr.Tech degree from Åbo Akademi in Turku, Finland (2001). He has around 60 international conference, journal and book contributions in the areas of digital system design, platform based design, synchronous/asynchronous (formal) modeling and implementation. With ABB Corporate Research Centre in Västerås, Sweden since 2008, he is now a principal scientist on embedded systems architectures and methodologies. Main activity topics relate to the development of hardware-software co-design methodologies for industrial control applications, targeting multicore and FPGA technologies. He is member in the program committee of several first-class conferences and reviewer for numerous other conferences and a board member and reviewer for journals on system-on chip and embedded systems.



Cristina Seceleanu is a senior lecturer at Mälardalen University, Västerås, Sweden, Embedded Systems Division. She received a MSc. in Electronics from Polytechnic University of Bucharest, Romania, in 1993, and a Ph.D. in Computer Science from Åbo Akademi and Turku Centre for Computer Science, Turku, Finland, in 2005. Her research focuses on developing formal models and verification techniques for constructing predictable real-time embedded systems. She currently is and has been involved as organizer, co-organizer and chair for relevant conferences and workshops in computer engineering. She is part of the Editorial Board of the International Journal of Electrical and Computer Engineering Systems, and the International Journal of Embedded and Real-Time Communication Systems.

Hannu Tenhunen is professor of nanoelectronics at University of Turku. He has an extensive background on submicron and nanoscale CMOS system and circuit implementation issues. He was one of the originators and founders for interconnect centric design paradigm, globally asynchronous locally synchronous systems, and network-on-chip paradigm. Currently Prof. Tenhunen is active in Artemis Technology Platform frame to bring multicore processing and NoC to EU research agenda as a central technology for future embedded systems. His current research is focused on trade-offs between 2-D and 3-D integration, especially for mixed signal systems and NoCs. For mitigating process and architectural variability he is also actively developing agent oriented system design approach including fault tolerance and robustness enhancements.