Resource-Aware Task Allocation and Scheduling for *SegBus* Platform

Khalid Latif[1,4], Tiberiu Seceleanu[3], Cristina Seceleanu[2], Hannu Tenhunen[1,4]
[1]Department of Information Technology, University of Turku,Finland. [2]Mälardalen University, Västerås, Sweden.
[3]ABB Corporate Research, Västerås, Sweden. [4]Turku Centre for Computer Science (TUCS), Finland.
Khalid.Latif@utu.fi, Tiberiu.Seceleanu@se.abb.com, Cristina.Seceleanu@mdh.se, Hannu.Tenhunen@utu.fi

***Abstract*—In this work, we propose an integrated task allocation and scheduling mechanism to minimize the resource contention and the processing latency for application running on the *SegBus* platform. The transactions are classified as local and cross border SPLIT transactions. The hybrid scheduling approach implemented by hierarchal arbiter code structure shows significant improvement in system performance. The interrupt scheduling has been implemented to further enhance system performance. A H.264 video encoder application has been used to verify the proposed technique, showing a large improvement in system throughput.**

I. Introduction

Modern embedded systems consist 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 not efficient regarding power and area. Continuous technology scaling has made it possible to integrate billions of transistors on a single chip [3]. Thus, an entire processing system can be integrated on a single chip, known as Multiprocessor System-on-Chip (MPSoC). After resolving the integration issue, the next problem is to provide an efficient communication platform for communication among the PEs. Traditional data bus may provide enough bandwidth for communication among 3-10 elements but does not scale to higher numbers [5].

Various on-chip communication platforms, like Networks-on-Chip (NoC) [4] and Segmented bus (*SegBus*) [8], have been proposed to deal with scalability issues. NoC resolves the scalability issue well till hundreds of processors at the expense of power consumption and resource overhead. On other hand, *SegBus* scales well in range of tens with relative complexity, power consumption and area lower than NoCs. The *SegBus* architecture and communication mechanism has been explained in [9], [8].

For better performance and proper utilization of communication resources in any MPSoC system, a suitable task allocation and scheduling mechanism is needed, according to the platform architecture and application requirements. In this paper, we address the task allocation and scheduling issues for the *SegBus* platform, to minimize the resource contention and the processing latency for application running on it.

Related Work. The problem of task graph scheduling is NP-Complete [11]. A number of task allocation and scheduling techniques for communication and resource management of MPSoC systems with homogenous and heterogenous PEs have been proposed.

Wang et al. [12] approach the issue by removing inter-core communication overhead by jointly optimizing computation and communication task scheduling. The approach is limited to streaming applications, and it has been analyzed for single bus, with only four processing cores.

Malani et al. [7] presents an application specific scheduling approach for MPEG decoder application, which completely eliminates the interprocess communication. The approach shows significant improvement in performance regarding latency with resource contention avoidance.

Suhendra et al. [11] address the optimization issue of scratchpad memory (SPM) in context of embedded MPSoC. A flexible partitioning of the SPM budget among the processor is used, showing significant performance improvement. Then, the memory optimization was combined with task scheduling for further performance enhancement. Our approach is similar, based on two phases - allocation and scheduling, but less coupled.

Anderson et al. [2] proposed a scheduling approach at thread level for synchronization and to avoid the L2 cache miss. The approach shows significant improvement but the communication platform used to justify the approach is very simple - only 4 cores used. The complexity may rise with number of cores.

II. Platform Communication Mechanism

The *SegBus* platform is a grouping of single bus sub-systems: the *segments*, connected via border units (**BU**s): FIFO structures with logic.

The *SegBus* communication can be divided into two categories: local and cross-border. The cross-border communication deals further with: local-external, external-external and external-local communications [8]. Segment arbiters (**SA**s), and a central arbiter (**CA**) control the local or external aspects of the communication, respectively.

In single bus scenarios, during a transaction, it is possible that the receiving (slave) device is not able to receive the data, or can not respond immediately. In this situation, the initiator (master) device will hold the bus and other masters can not utilize it.

A *SPLIT transfer* improves the overall bus utilization by splitting the master part of the communication from the slave part [1]. Thus by using the *SPLIT transfers* the possible 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.

A. Local SPLIT Transactions

We consider here that both the master and the slave are placed in same bus segment. The master requests the bus ownership by raising the *req* signal to the local **SA**. In two situations, the **SA** does not grant the master the access to the bus: (i) if another transaction is under completion on the bus; (ii) if the target slave is busy and cannot respond the request. If the bus is busy, the **SA** assigns the bus ownership to the requesting master later at some time according to the arbitration scheme. If the target slave is not able to serve the request, the **SA** grants the bus to another requesting master. A similar approach has been proposed by [1].

B. Cross border SPLIT transactions

A SPLIT transaction for cross-border communication builds on the existing SPLIT mechanism. Consider the situation that a master module from segment0 requests to send a data packet to a slave in segment2. The segment1 will be used as well because it is on the way of transaction. It is not efficient to allocate all the bus segments required by 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 by the respective **SA**s. A local SPLIT mechanism will be followed for each segment traversal.

The bus request comes along with target slave address. **SA** checks, if the request can be served in current segment or not. If the request targets a slave in other segment, the request is forwarded to the **CA** with target address. In the meanwhile, no other external request are served by the **SA**- only local ones may be serviced.

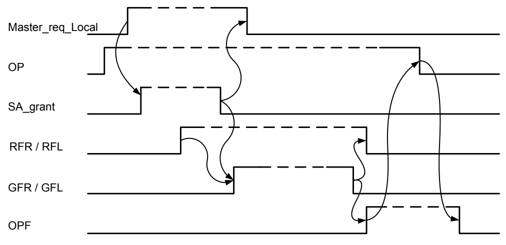

Fig. 1. Inter segment transfer control.

Whenever the **CA** is able to serve the request, it informs the **SA**s, from the initiator to the target segment, of the imminent transfer (signal OP - operate). When the current operation finishes in the initiator segment, that **SA** grants the requesting master to access the necessary **BU**.

Upon filling up the FIFO, the **BU** informs further the next segment that data is waiting to be transferred, via certain signals (RFR or RFL)- Fig.1. 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). Hence, the package waits in the **BU** the time required for the current local transfer in the next segment to end. When this operation completed, the **CA** receives the OPF (operation finished) signal from the corresponding **SA**, and answers by lowering the respective OP line. When OPF is also reset, the segment is ready for a new inter-segment transfer.

In a cascaded manner, the above scenario repeats all the way to the target segment, providing significant transfer delay.

III. Task Allocation and Scheduling

Scheduling is the basic and mandatory service to use the inter-connection platform. Scheduling can be divided into two steps: task allocation and arbitration as shown in Fig. 2. Communication features of the running application are needed for the proper placement of IPs. Here, we are interested in the transaction frequency between PEs, 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* [10] has been developed, to deliver the allocation cost for various scenarios.

The *PlaceTool* works as the task allocator. A *communication matrix* is extracted from the Packet based Synchronous Data Flow (PSDF) [6] diagram and fed to the *PlaceTool* to generate the task allocation or placement of PEs. After having the placement of tasks and processes, the next step is scheduling, controlled by arbitration (Fig. 2).

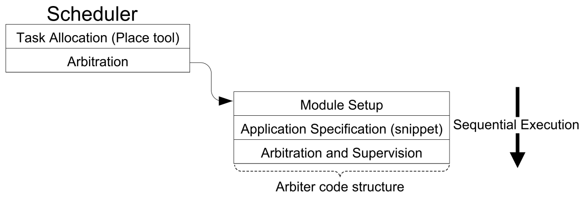
Scheduler
Task Allocation (Place tool)
Arbitration
Module Setup
Application Specification (snippet)
Arbitration and Supervision
Arbiter code structure
Sequential Execution

Fig. 2. Segbus scheduler structure.

We elaborate further based on an arbitrary task graph as shown in Fig.3. The *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 Fig. 4. The context switching time in scheduling is considered zero. The detailed description of *PlaceTool* and arbitration mechanism is presented in detail in sections III-A and III-B respectively and a H.264 video encoder is used as a running example.

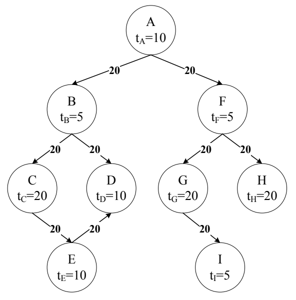

Fig. 3. Task graph.

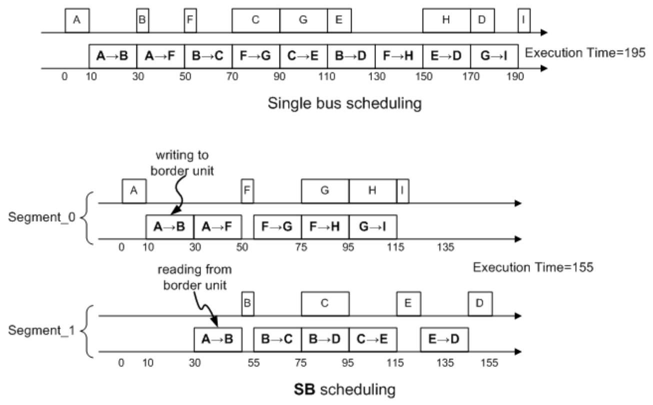

Fig. 4. Bus scheduling.

A. Task Allocation

The communication frequency for the H.264 video encoder is obtained from running a Simulink Model of the application shown in Fig. 5 - the PSDF representation.

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 [6]. The resulting segmented application model is also visible in Fig. 5.

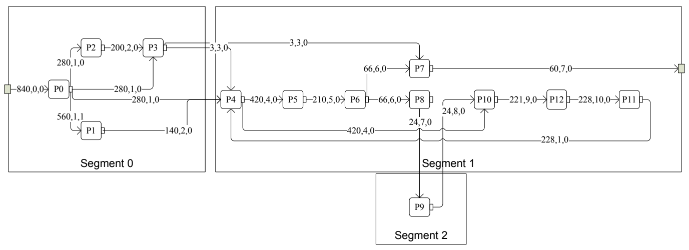

Fig. 5. PSDF application specification

B. Scheduling

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. Using *clock-driven* approach, 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 [13].

The arbiter structure is depicted in fig. 2. The "Module SetUP" and the "Arbitration & Supervision" blocks are concerned with application-independent procedures, such as reading the input signals, selecting the granted master, counting the number of transactions performed in a granted activity, etc. The middle block, "Arbitration specification", brings in the application specific requirements for scheduling grant decisions. Thus at "Arbitration specification" level, scheduling approach can be decided like *round-robin* or *priority-driven* according to the requirements.

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 [13]. 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. 6), 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 *dest* field content. The sub-fields ignored for *dest* specification will also be ignored here.

• *count*. Identifies the number of packets, master has to send to the specified targets. It corresponds to the first number in 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. The VHDL code corresponding to the table in Fig 6 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);
...
```

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

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).

C. Interrupt Scheduling

As mentioned in section II-B, a packet may have to wait in **BU** for number of clock cycles for the cross border transactions. By using interrupt service, delay in **BU** can be significantly reduced [14]. An inter-segment transfer, when reaching one of the **BU**s on it way, must preempt 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.

For real hardware, 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 respective master then continues to send the information remaining from the interrupted operation.

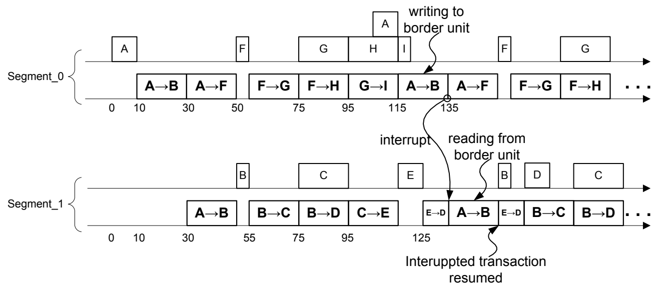

Fig. 7. Interrupt scheduling.

To illustrate the interrupt mechanism, consider the task graph shown in Fig.3. Suppose that this is the graph of streaming application like audio/video codec and all the tasks execute repeatedly. As shown in Fig.7, 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 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 rest of the processing and transactions on segment 1 will be delayed as well. This delay will go on increasing because of previous delay. After few application cycles, segment 1 will be lagging too much behind segment 1. Thus, 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. 7, data packet for transaction A→B will not delay only the processing on node B but for 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.

IV. Experimental results

The communication frequency extracted from PSDF model of H.264 video encoder with static task allocation shown in Fig. 5 is fed into the *PlaceTool*. The results of this exercise in the form of task allocation with different number of segments and corresponding communication cost are shown in Fig. 8. Allocation with three segments was used for final implementation on Stratix III device with linear *SegBus* topology and 66 words package size. With the same parameters, the single bus system was simulated as well.

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. 8. The allocation and associated cost results.

Using the *Clock-driven* scheduling approach, scheduling decisions were made before execution according to the interprocess communication requirements. The scheduling decisions for local transactions within the segment were stored on corresponding **SA** while the cross border transaction scheduling was stored on **CA**. The single bus allocation was simulated at a clock frequency of 100MHz, while the *SegBus* solution with three segments utilizes four clock domains(one for each segment - 100MHz, 60MHz, 50MHz and one for the CA - 30 MHz). The throughput results of segmented application came close to the value of 40% as delivered by *place tool*. Which means the reduction of power with same proportion. The main contribution in reduction of power come from multiple clock domains. The core dynamic power favors the single bus solution because of the switching power of **BU**s, while the total power supports the *SegBus* platform implementation.

Conclusions: The systematic and integrated approach for task allocation and scheduling for *SegBus* platform proved to offer a suitable framework for resource management and throughput enhancement. Arbiter code structure provides the flexibility to use the hybrid approach for task scheduling. The approach showed significant improvement in platform throughput and reduction in power consumption.

Acknowledgements

This work is supported by the Nokia Foundation and DOMES project (123518/2008) funded by the Academy of Finland.

References

[1] ARM AMBA Specification and Multilayer AHB Specification (rev 2.0). www.arm.com.

[2] J. H. Anderson and J. M. Calandrino. *Parallel real-time task scheduling on multicore platforms* In RTSS 06, pp. 89-100, 2006.

[3] S. Borkar. *Designing reliable systems from unreliable components: The challenges of transistor variability and degradation* IEEE Micro,volume 25(6), pp. 10-16, 2005.

[4] W. Dally. *Route packets, not wires: on-chip interconnection networks* Design Automation Conference, 2001. pp. 684-689.

[5] A. Jantsch and H. Tenhunen (Eds.) *Networks on Chip.* Kluwer Academic Publishers, Boston; 2003. ISBN 1-4020-7392-5

[6] K. Latif, M. Niazi, T. Seceleanu, H. Tenhunen, S. Sezer. *Application development flow for on-chip distributed architectures.* The 21^{st} IEEE International System on-chip Conference 2008, pp. 163-168.

[7] P. Malani, Y. Tan, Q. Qiu. *Resource-aware High Performance Scheduling for Embedded MPSoCs With the Application of MPEG Decoding* IEEE International Conference on Multimedia and Expo (ICME), 2007, pp.715-718.

[8] T. Seceleanu. *The SegBus Platform - Architecture and Communication Mechanisms.* Journal of Systems Architecture (2006), doi:10.1016/j.sysarc.2006.07.002

[9] T. Seceleanu. *Communication on a Segmented Bus Platform* The IEEE International System on-chip Conference, 2004. pp. 205-208.

[10] T. Seceleanu, V. Leppänen, O. Nevalainen. *Improving the Performance of Bus Platforms by Means of Segmentation and Optimized Resource Allocation.* EURASIP Journal on Embedded Systems, vol. 2009, Article ID 867362, 2009. doi:10.1155/2009/867362.

[11] V. Suhendra, C. Raghavan, and T. Mitra. *Integrated scratchpad memory optimization and task scheduling for MPSoC architectures* The International Conference on Compilers, Architecture and Synthesis For Embedded Systems, 2006. pp.401-410.

[12] Y. Wang, D. Liu, M. Wang, Z. Qin, Z. Shao. *Optimal Task Scheduling by Removing Inter-core Communication Overhead for Streaming Applications on MPSoC* The 16^{th} IEEE Real-Time and Embedded Technology and Applications Symposium, 2010. pp.12-16.

[13] T. Seceleanu, I. Crncovik, C. Seceleanu. *Transaction level control for application execution on the SegBus Platform.* The 33^{rd} Annual IEEE International Computer Software and Applications Conference, 2009. pp. 537-542

[14] A.D. Swaminathan, T. Seceleanu. *Interrupt Communication on the SegBus platform.* The IEEE International System on-chip Conference, 2006, pp. 229-232.