Application Development Flow for On-Chip Distributed Architectures

Khalid Latif, Moazzam Niazi, Hannu Tenhunen
University of Turku, Finland
{Khalid.Latif,Moazzam.Niazi,
Hannu.Tenhunen}@utu.fi

Tiberiu Seceleanu
ABB Corporate Research
Västerås, Sweden
tiberiu.seceleanu@se.abb.com

Sakir Sezer
Queens University of Belfast, UK
s.sezer@ecit.qub.ac.uk

***Abstract*—We approach the construction of design methodologies for on-chip multiprocessor platforms, with the focus on the *SegBus*, a segmented bus platform. We study how applications can be mapped on such distributed architecture and show how to build the concrete level software procedures that will coordinate the control flow on the platform. The approach employs models developed in the Matlab-Simulink environment considering also a unified representation of both platform and application. The running example is represented by the H.264 encoder. Allocation of processing elements on the platform, structure and functionality and the eventual control code for arbiters are the main topics described here.**

I. Introduction

The tremendous technological advances of the last decade or so, in the direction of ever smaller technology figures, induced a matching increase of the complexity of modern silicon devices. It is possible today to transfer architectures previously only considered at computer levels into the boundaries of single chip systems. The intent is to implement the increasing requirements concerning design features like performance, power consumption, adaptability, reusability, apart from allowing a better understanding of the huge increase in complexity. In general, these architectures attempt to exploit at maximum the benefits from current technologies, with respect to the mentioned design characteristics, while allowing a transparency with respect to newer technological advances.

In this context, distributed on-chip architectures, or multicore, or *multiprocessor system-on-chip* (MPSOC) paradigm gains increasing support from system developers. Instances of such architectures are mostly known as *networks-on-chip* (NOC) or *segmented buses*. However, while the tasks of hardware designers seem to be, at least at higher levels of abstraction, eased by the employment of MPSOC architectures, these pose great challenges in front of application and software developers. The latter hardly see effort benefits in MPSOC designs, as the traditional software development is based on sequential, single processor design process.

One of the reasons behind the difficulties in MPSOC development is the lack of design methodologies [2]. Due to environmental and application requirements, the operation and communication characteristics of the employed devices and architectural instances may vary greatly from system to system. Performance measures are intrinsically related to the specifics of the underlying hardware platform. The lack of information availability at the higher abstraction (application) layers affects how specification requirements are reflected in the final system realization. Another important issue is the control of data transfers between different devices, as concurrent communication will certainly create conflicting situations.

At the same time, there is a sensible growth in the demand for multimedia applications performance. In order to address such issue, both performant platforms, but as well efficient design methodologies need to be developed. Employment of Intellectual Property (IP) designs is one of the high requirements in order to allow a fast deployment of new design solutions. Alternatively, hardware design languages might prove at times to be too restrictive, as only a small part of the design community has good respective knowledge. The tendency is therefore to replace, or to make transparent, whenever possible, VHDL (for instance) based design with higher level constructs, for instance C-like languages. The new challenges reside now in having a good platform representation at these higher levels, such that early evaluations are possible to perform.

The present work delves into aspects related to design methodologies for MPSOC. We describe the principles of a stepwise design methodology that targets a distributed on-chip architecture, namely the *SegBus* platform [11]. We continue the work of previous research results in the direction of raising the levels of abstraction at which such methodology is beneficial. We also take a step further in the direction of automation, by providing platform models in the framework offered by Matlab [3]. We are interested in Matlab/Simulink as a high level design environment which allows the exploration of allocation results and offers the possibility for early assessment of application - platform mapping.

Related work. Even though multiprocessing was not part of the mainstream practices, studies reflecting on this topic have been around for a long period, now, mostly considering computer networks. In recent years, however, research started to address on-chip solutions.

The most common current methods to deal with concurrency are *threads, semaphores, mutual exclusion locks*, etc. However, these approaches are intended to build *virtual* parallel environments, most often not well suited for current *heterogeneous* multi processor systems. For instance, threads are defined as sequential processes, exchanging information through shared memory resources, and several synchronization methods must be implemented in order the ensure the security

and reliability of the shared data. This is because threads are highly non-deterministic, and a immense effort is dedicated to establishing an order of execution.

Our approach here is based on the existence of segment and central arbiters that contain the schedule for data exchanges between devices within the same segment, or in different ones. Out of a possible group of "enabled" transfers, these devices, with a built-in policy of granting select the appropriate one. The present study builds on the work of Truscan et. al [14], and it provides an improved tooling support for the development of applications.

Lahiri et al. [8] address design optimality for a segmented bus platform similar to the *SegBus*. The segmented bus architecture [8] is, however, *memoryless*, different to our case, where the segments are separated by storage devices. Moreover, the protocols are fit to one application, and contentions can be extracted following a higher level simulation. The approach introduces a valuable simulation-based trace extraction, to indicate the communication patterns, considered consistent, after which an algorithmic solution is found to the allocation problem. Arbitration issues are not specifically addressed, and hence, possible contention problems and precedence relations are not analyzed. The intermediate arbitration tables, in our case, solve both the contention and the precedence issues.

Srinivasan et al. [12] introduce an AMBA-like hierarchy of a segmented bus. The authors employ genetic algorithms for finding optimal segmented bus allocations, but the methodology is not continued to other levels of abstraction. There is a similarity with [8], in the sense that no control procedures, either for local or inter-segment activities, is presented. The arbitration is possibly organized following AMBA protocols, but this may affect both allocation optimality and solving the conflicting task execution.

De Jong [7] elaborates a system design flow based on UML and SDL, mainly for the purpose of control, communication and synchronization refinement of both hardware and software components. As it pertains more to the area of software-hardware co-design, this study is viewed as a complementary research to the present work.

Dekeyser et al. [6] propose a "Y-chart" methodological approach to multiple SOC system design with UML. While the results are applicable to our specific platform-based approach, in general, several design steps, such as application and platform refinement, granularity, communication restrictions, are not captured in [6].

The approach we illustrate here does not impose restrictions towards other MPSOC platforms. We are the moment exploring the creation of network-on-chip [5] models in order to enlarge the basis of the solution. Considered together with earlier results [14] on high level design methodologies, we approach the realisation of a complete framework for the design of multiprocessor systems.

II. Backgound

A. Segmented Bus Architecture

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. In this case, all dynamically connected segments act as a single bus. Due to the segmentation of this shared resource, 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 Figure 1.

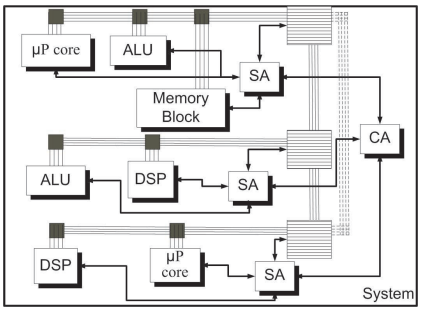

Fig. 1. Segmented bus structure.

The *SegBus* platform [11] is thought as having a single central arbitration unit (**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.

A transfer from the initiating segment k to the target segment n is represented in Figure 2. 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. 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 [13]: 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.

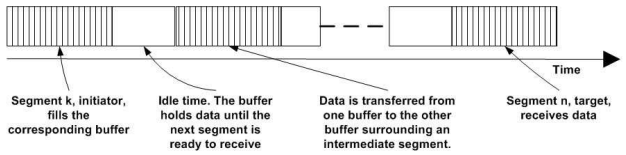

Fig. 2. Inter-segment package transfer.

In addition, it becomes necessary that 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.

Platform characteristics. The *SegBus* platform specifics consist in a set of global parameters that have a great impact on the implementation [11]: (i) topology - a linear or circular geometry; (ii) number of segments; (iii) size of the package.

III. Design Methodology

Truscan et. al [14] introduced a MDA approach to application development for the *SegBus* platform. We complement that here with additional tooling support coming in the form of Matlab / Simulink descriptions. As a running example we employ a H.264 encoder [10] - Figure 3.

The proposed design flow is illustrated in Figure 4.

A. Tool Environment

Matlab / Simulink. Matlab Simulink Environment [3] is a tool commonly used for modeling, simulation, analysis and profiling of multi domain systems. These systems range from a simple adder to complex application like Video coding, transceiver synchronization in communication systems or control system design. It comprises of different block sets, libraries and programming functionalities. After the application specification, a working Simulink model can be modeled and application algorithm can be verified using different configurations and random as well as normal inputs.

Here, we use the "Video and image processing" blockset from Simulink to model the H.264 Encoder application. This blockset provides a variety of functions that can be used for modeling of Image and Video applications.

The Matlab Simulink environment also supports obtaining the communication matrix necessary to compute the optimal allocation scheme for the platform.

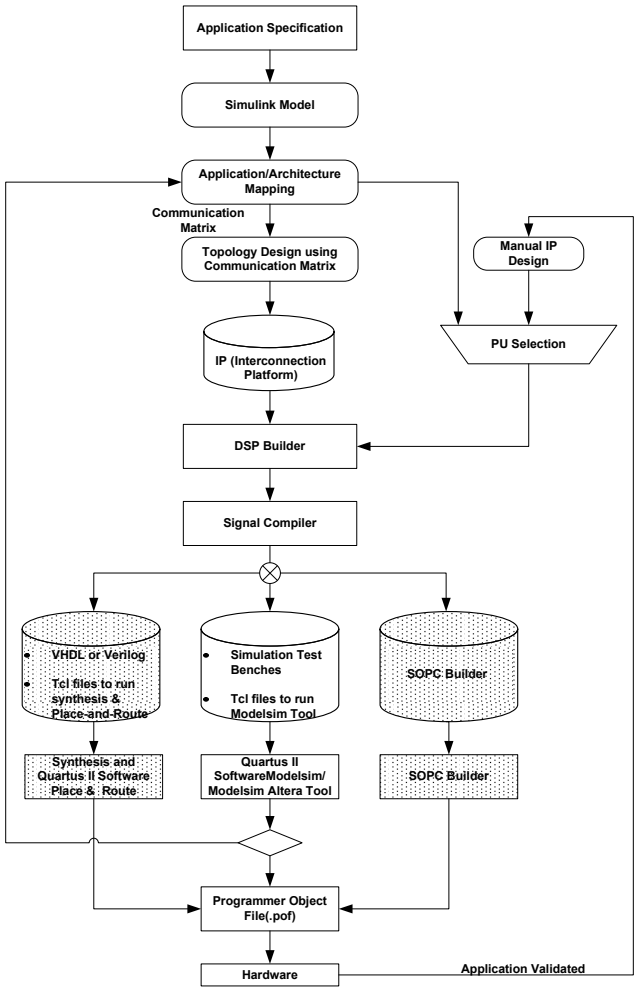

Fig. 4. *SegBus* Design Process

Altera. 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 [4] framework.

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 [14].

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

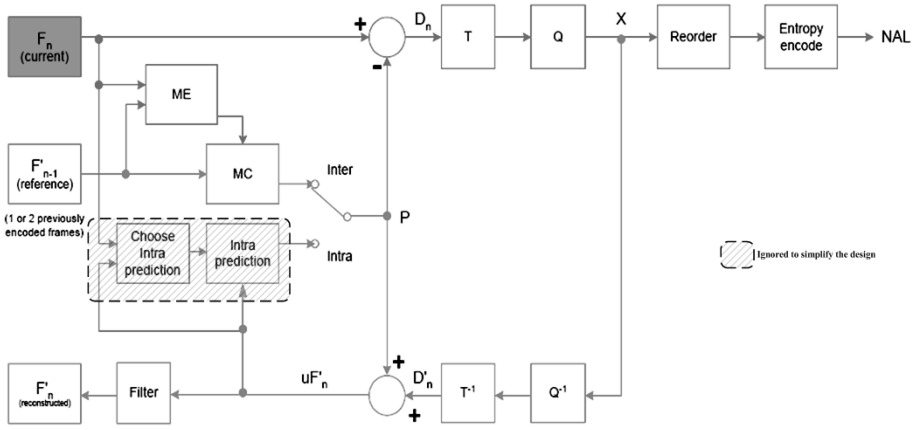

Fig. 3. The H.264 application specification

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. A *packet flow* is a tuple of two values, P and T. The P value represents the number of successive, same size transactions emitted by the same source, towards the same destination; the T value 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.

The *Packet SDF (PSDF)* of a certain system is a sequence of packet flows, $< (P_1, T_1), \ldots, (P_n, T_n) >$, where $\forall i, j \in \{1, \ldots, n\} \cdot P_i \neq P_j$ and $T_1 \leq T_2 \leq \ldots \leq T_n$.

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 number" global flows.

An additional and optional third dimension is added to the definition of the packet flow - the package *kind*. This (another number) identifies flows from the same source, but with different destinations, where the content of the data is the same. This will allow a simplification of the overall *PSDF* scheme and is the basis for development of services such as *message broadcasting*.

For the H.264 encoder, the corresponding 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 sections. The processing elements ($P0, P1, \ldots, P12$) correspond respectively to YUV generator, Chroma resampler, Motion vector estimator units, etc.

B. Application Partitioning

We consider that the application is already partitioned and mapped on the available devices as described in Fig. 7. In general,this means also that all possible software procedures are already mapped within the hardware devices. However, this is not the case in Fig. 7, where all the devices are hardware elements.

At this moment, we can extract the communication features, that is, the frequency with which the various devices communicate with each other. We group these frequencies in the so-called "communication matrix". For the application at hand, this matrix is illustrated in Fig. 5. The matrix was obtained by using the signal dimension option in Simulink.

The matrix is fed into the *PlaceTool* programme which delivers the allocation costs for various scenarios [9]. The results of the exercise are given in Fig. 6. While we can observe that already 2 segment platform will deliver the best of the performance improvement. The improvement diminishes with the number of segments, due to the additional communication overhead. However, for the sake of exemplification, we select the solution with 3 segments in our case.

The resulting segmented application model is obtained as in Fig. 7.

C. Code Generation

The segmentation process, while providing premises for a more performant execution, it raises the complexity related to finding a (good) schedule for both the processing tasks, but, mostly for the data transfers. The communication matrix is just a means to obtain an as optimal as possible allocation of resources with respect to global (inter-segment) transfers, but in order to implement the application functionality, both local (intra-segment) and global transfers must be appropriately scheduled.

The *PSDF* representation helps in creating such a communication & processing schedule. This is applied in two turns, once at the segment level and once at the platform level, in

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. 5. The communication matrix for the example

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	4 5 6 7 8 10 11 12 || 0 1 2 3 || 9	-41%
4	143100	9 || 8 || 4 5 6 7 10 11 12 || 0 1 2 3	-39%

Fig. 6. The allocation and associated cost results.

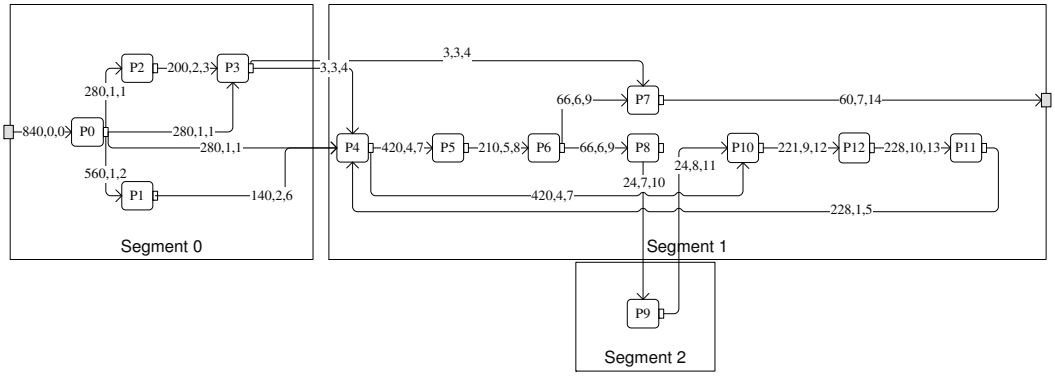

Fig. 7. PSDF application specification

order to obtain the *programme* that will coordinate the activity of the segment and central arbiters, respectively.

Arbitration programmes. A programme at the level of the **SA**s illustrates the schedule that must be implemented by the respective arbiter unit. For a given (single) application, this is simply the order in which the masters of the segment can be granted access to the local or global bus lines for data transfers. However, a simple ordering will render ineffective the operation of bus splitting. The arbiters should allow, whenever possible, a certain kind of parallelism (interleaving) in data transfers. This, superimposed over the simultaneous processing activities of (some of) the masters within the segment will offer the benefits in performance.

The programme for both the **SA**s and the **CA** is a grouped collection of VHDL statements placed in the controlling process of the arbiter's specification. Through specific mechanisms (described in the further paragraphs) the sequential execution of VHDL statements within a process is improved with a non-deterministic interleaved execution model. This gives the possibility for several lines to be perceived as executed in parallel, whenever appropriate.

A *programme line* for a **SA** is a VHDL statement that can be interpreted as an instruction with several fields - Fig. 8, with the following meaning.

1) *PC*. This is the *Programme Counter*, providing reference to the lines of instructions possible to be accessed from other instructions.
2) *Guard*. The *Guard* signals if the respective line is possible to be selected for execution. This is to enforce the necessary order of data transfers. Devices not part of a granted transfer may, meanwhile, proceed with their processing tasks. The value of the guard is a natural number: 0 means the line is *enabled* and any number larger than this will mark the line as *disabled*. In the first case, the arbiter checks the "rest" of the instruction for a possible granting operation, if additional conditions are met; in the latter, the arbiter operation ignores the information. Several lines with guards evaluated to 0 are potentially selectable for granting operation. However, only one of the instructions can be actually "executed".
3) *Source*. This field contains the address of the requesting master - the initiator of a transfer request. Devices on the platform (masters, slaves) are identified by an unique number.
4) *Destination*. This field contains the address of the targeted device - the slave.
5) *Dest_Seg*. This field contains the address of the segment where the *Destination* is located.

6) *toGrant*. This is the instruction for the arbiter to grant the requesting master. At this moment the specification is obsolete, but the field is preserved for future developments.
7) *Count*. This is represented as a natural number containing the number of consequent packages to be transferred by a master. Every time the master is granted and performs the transfer, this number is decreased. When it reaches 0, the line cannot be anymore selected for execution, even if the *Grant* field is also 0.
8) *enables*. *Disabled* lines will become *enabled* during the execution of the programme. The *enables* field (one per instruction) specifies which line can be moved towards enabledness at the end of the current transfer. This is achieved by subtracting 1 from the present value of the *Guard* field of the respective line.

	PC	Guard	Source	Destination	Dest_Seg	toGrant	count	enables
Example:	5	0	2	5	0	2	200	6
	6	1	6	3	0	6	120	9

Fig. 8. The structure of the programme line, with two examples.

The programme construction considers also requests coming from **BU**s as events to be part of schedules. In this case, as an actual example, it may be interesting to observe the whole code describing the operation of the **SA** for segment 2 of the H.264 application, given as follows.

```
program(0) <= (guard => 0, source => RFL, dest => 9,
        dest_seg => 2,  togrant => RFL, count => 24,
        enables => 1);
program(1) <= (guard => 1, source => 17, dest => 5,
        dest_seg => 1, togrant => 17, count => 24,
        enables => 0);
```

In the above, the "RFL" term stands for "request from left". In brief, and in correlation with the flow described in Fig. 7, the **SA** of segment 2 waits first that a transfer is received from left (segment 1), after which a transfer from the local device ($P9$ - Fig. 7) is able to be executed, targeting a device in segment 1.

The code for the **CA** has a similar structure, with the exception of the *Destination* field.

IV. Experimental results

We have applied the illustrated techniques for the implementation of a H.264 model on a traditional single bus platform and on a 3 segment *SegBus* platform, both on the same Stratix III device. The *SegBus* solution is characterized by a linear topology (as in Fig. 7) and 66 words package - similar for the single bus. The first one run at a clock frequency of 100 MHz, while the *SegBus* solution utilizes four clock domains (one for each segment - 100MHz, 60MHz, 50MHz and one for the **CA** - 30 MHz).

The performance (throughput) results came close (within 1%) to the ones anticipated by the Fig. 6 for the respective solution. Intuitively, this also means an approximated 40% reduction in power. What is even more satisfying was a further 12% improvement in power consumption, as approximated by the Altera's *PowerPlay Power Analyzer* tool, both in a vectorless and in a toggle-rate based approach. While the core dynamic power dissipation was in the favor of the single bus solution (due to the additional switching activity of the **BU**s), the I / O and the total power dissipation go in the favor of the *SegBus* platform.

V. Conclusions

The methodological chain used in this study (Matlab-DSP Builder-Quartus-Modelsim) proved to offer a suitable framework for the application development on the *SegBus* platform. We have described the employment of arbiter programmes for scheduling with a mostly static characteristic, but with a certain degree of (useful) non-determinism in practice.

The approach showed improvements over previous results, even in the context of a much more complex application. The power estimates are encouraging for further optimization / tool based approaches.

Future work. A very necessary step is in the continuation of automation with respect to the design flow. UML-based solutions may be one way to support a more straightforward integration of the process while also providing at least guidance for tool development.

Apart from supported services (preempted transfers), dynamic scheduling, broadcasting, and possibly virtual channeling are future topics for analysis and implementation in the context of the *SegBus* platform.

References

[1] *Stratix III Device Handbook* 2007, Altera.
[2] *International Technology Roadmap For Semiconductors.* 2005 Edition. Design.
[3] Matlab/simulink, http://www.mathworks.com.
[4] Modelsim, http://www.model.com.
[5] A. Jantsch and H. Tenhunen (Eds.) Networks on Chip Kluwer Academic Publishers, 2003.
[6] J.-L. Dekeyser et al. *Model Driven Engineering for Regular MPSOC Co-design.* Reconfigurable Communication-centric SoCs (ReCoSoC), 2006.
[7] G. de Jong. *A UML-Based Design Methodology for Real-Time and Embedded Systems.* Design, Automation and Test in Europe Conference and Exhibition, 2002, pp. 776-778.
[8] K. Lahiri, A. Raghunathan, S. Dey. *Design Space Exploration for Optimizing On-Chip Communication Architectures.* IEEE Transactions on Computer-aided Design og Integrated Circuits and Systems, VOL. 23, NO. 6, JUNE 2004. pp. 952-961.
[9] V. Leppänen, T. Seceleanu, O. Nevalainen. *Communication Scheduling for the SegBus Platform.* Proceedings of the IEEE International SOC Conference (SOCC), Sept. 2007.
[10] I.E.G. Richardson, J. E. Richardson H.264 and MPEG-4 Video Compression. Wiley, John & Sons, Inc. October 2003.
[11] T. Seceleanu. *The SegBus Platform - Architecture and Communication Mechanisms.* Journal of Systems Architecture (2006), doi:10.1016/j.sysarc.2006.07.002
[12] S. Srinivasan, L. Li, N. Vijaykrishnan. *Simultaneous Partitioning and Frequenct Assignment for On-chip Bus Architectures.* Proceedings of DATE 2005, pp. 218-223.
[13] A.D. Swaminathan, T. Seceleanu. *Interrupt Communication on the SegBus platform.* Proceedings of the IEEE International System on-chip Conference, Austin, TX, USA, Sept. 2006, pp. 229-232.
[14] D. Truscan et. all. *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.