Transaction level control for application execution on the SegBus Platform

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

Ivica Crnkovic, Cristina Seceleanu
Mälardalen University
Västerås, Sweden
{*ivica.crnkovic, cristina.seceleanu*}*@mdh.se*

***Abstract*—We define here a simple, low level control procedure definition, to support application implementation on a particular multiprocessor platform, namely the *SegBus* segmented bus. The approach considers communication as data package transactions from one device to another. It takes into consideration the platform characteristics and requires details of application partitioning and mapping on platform resources. The dependency between operations are extracted from a SDF-like representation, and the actual control code is produced as "application-dependent" VHDL code, grouped in so-called *snippets*, application and platform instance dependent. The obtained code is inserted in a specific section of a (segment or central level) arbiter. We illustrate the application of our approach on a small implementation example.**

I. Introduction

The available transistor technologies made possible the transition into the on-chip multiprocessing era. Alternative system architectures are considered in order to cope with the tremendous advances provided by ever smaller technology figures. Distributed on-chip architectures, or multi-core, or *multiprocessor system-on-chip* (MPSOC) paradigm gains increasing support from system developers. MPSOC is seen as one of the major means through which performance gains are still to be sustained even after Moore's law may become decrepit [1]. Today, there is a relatively large set of MPSOC platforms that answer, in their own way, to multiple challenges raised by technology. Common interconnect structures are *network-on-chip* (NOC) [4], and *segmented bus* platforms [6], [9].

To fully benefit from the features of MPSOC platforms, has been a challenge. While there are still important issues to be solved at the silicon levels, the most important impact of the MPSOC era is projected on the application, and, subsequently on the software designer. One of the reasons behind the difficulties in MPSOC development is the lack of design methodologies [1]. 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. Regardless of platform, the *optimality* of the design, in the sense of application-platform matching, is always an issue. Platform specific characteristics must be taken into consideration for each application, in order to offer a good match.

The present work analyzes aspects related to design methodologies for MPSOC, in the (restricted) context of the *SegBus* platform [9]. The main question that the research addresses regards the definition of control structures and their realization, in order to successfully implement a given application on the distributed platform at hand. This is especially necessary as the platform does not require (or benefit) from an operating system solution. The answer is based on a basic transaction specification that captures the expected scheduling and arbitration policies. Segment and application specific VHDL *snippets* are developed to be included as modules in the arbiter specifications. They support the granting process and provide mutual exclusion mechanisms for intra- or inter-segment transactions. We build both *virtual* and *actual* parallel computing environments by means of application-specific arbitration / scheduling policies. A certain acceptable level of non-determinism can be observed, contributing, however, to the actual implementation of parallelism. While the approach is expected to be further improved with automated procedures, we offer here the basic principles concerning the creation and (application) semantics of the code.

II. 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. 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 fig. 1.

The *SegBus* platform [9] 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 within the segment will get access to the bus in the following transfer burst.

A. Platform communication.

Within a segment, the **SA**s arbitrating the access to local resources. The inter-segment communication is a circuit

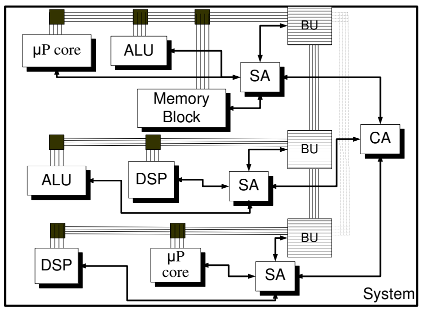
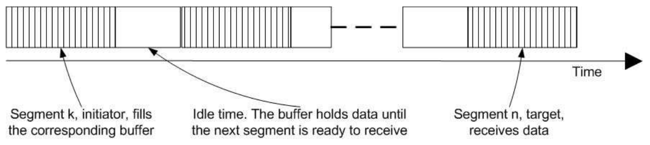


Figure 1. Segmented bus structure.

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 additional logic, controlled by the **CA**. The platform communication is packet based. 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 for establishing a link from source to destination. When this connection is available, the initiating device is granted the bus access. It 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. When the package reaches its destination segment, the respective **SA** routes the package to the own segment lines, to be collected by the targeted device.

A transfer from the initiating segment k to the target segment n is represented in fig. 2. 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. Here, the inter-segment transfer has to await the end of the local communication.



Time
Segment k, initiator, fills the corresponding buffer
Idle time. The buffer holds data until the next segment is ready to receive
Segment n, target, receives data

Figure 2. Inter-segment package transfer.

B. Present design methodology

In the following, we give a brief description of the *SegBus* design methodology [12] with the help of a (simplified) stereo mp3 decoder (layer III) [8] application.

The *Packet SDF*. The specification of the application itself starts with a *Packet SDF* (PSDF) model. PSDF is a customized version of Synchronous Data Flow diagrams [7]. The approach is intended to facilitate the mapping of the application to the architecture due to the similarity between the operational semantics of the PSDF and that of the *SegBus* architecture, thus allowing us to cope in a more detailed manner with the communication characteristics of our platform.

A PSDF description comprises two elements: *processes* and *data flows*; data is organized in packets. Processes transform input data packets into output ones, and 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 *transaction count*, that is, the number of successive, same size transactions emitted by the same source, towards the same destination; the T value is the *transaction index*, that 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 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.

Application modelling. The specification starts with the context diagram of the application, where the interactions between the application (depicted as a process) and the external environment are modeled 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. We adopt the activity diagrams of UML (ver. 2.0) to represent the PSDF. The mp3 example is given in fig. 3. In brief, process $P0$ represents frame decoding, $P1/P8$ - scaling on the left / right channel, $P2/P9$ - dequantizing left / right, etc. The represented flows consider packets of 36 data items

The application is further *partitioned* (processes to run as software or hardware), and we obtain the *partitioned application model* (PAM). At the same time, decisions on

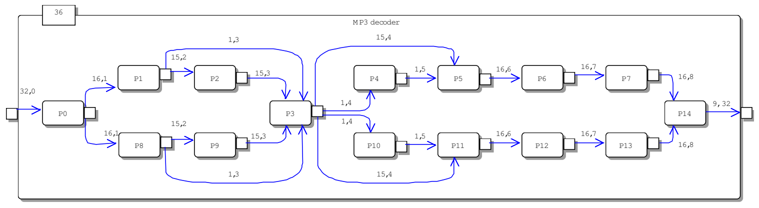


Figure 3. Application specification diagram (PAM).

the platform characteristics are taken (number of segments, topology, etc) and grouped into the *initial platform model* (IPM). Having an application model and the *SegBus* configuration at our disposal, next, the PAM is mapped onto the IPM. Considering a device-to-device communication matrix, we use a dedicated utility, the *PlaceTool* [10], in order to optimally place processing elements (one process is identified here with one device) on the IPM. The result is a *segmented application model* (SAM), where all the devices are assigned to a given segment.

The *complete platform model* (CPM) represents the SAM mapped onto the IPM. Following this, and a selection of library units, one finally reaches the *synthesizable platform model* (SPM), a “ready to deploy” stage.

However, the platform is not yet ready to execute the application. This, as the arbitration policies are not yet specified. Such specification follows a *code generation process* which we describe in the following section.

III. Arbitration via VHDL snippets

We remind the reader here that the *SegBus* platform has a two level arbitration mechanism. The segment level is controlled by the **SA**s, while the inter-segment communication is directed by the **CA**. Without considering details, the control flow of both **SA**s and of the **CA** is represented in Fig. 4.

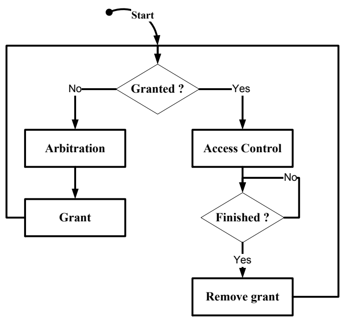


Figure 4. Arbiter control flow.

Arbiter structure. The **SA**s and the **CA** are VHDL defined modules, with a similar structure. The code implements the operational flow of Fig. 4, running 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.

The intended structure of the arbiters is depicted in fig. 5. 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. Our intention here is to develop the middle, “Arbitration specification” block, in such a way that it will bring in the application specific requirements for scheduling grant decisions. The resulting snippet will characterize the given application as mapped on a given instance of the platform.

The 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.

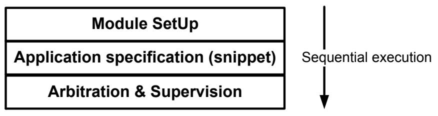

Module SetUp
Application specification (snippet)
Arbitration & Supervision
Sequential execution

Figure 5. Arbiter code structure.

A. **SA** *level arbitration*

The segment level arbitration is similar to any traditional 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**, otherwise it proceeds to granting it (or not).

Code generation. The deliverable of the code generation process is the *application control code* (ACC) which will drive the *SegBus* communication strategy at runtime. 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. Next, we describe the generation of the ACC content.

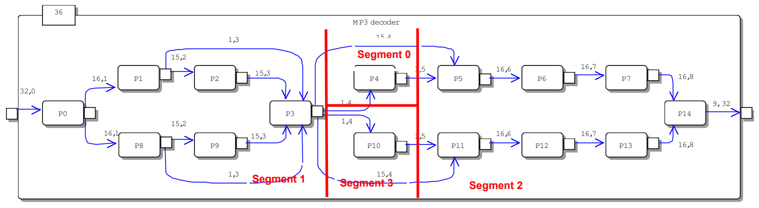


Figure 6. Segmented application model (SAM), four segment platform.

We start by considering each transfer operation as an "execution line". Such line is one element in the ACC, and the number of lines corresponding to an application is passed as a parameter (*nrLines*) to the arbiter. The transfers that can be executed in parallel (they are independent of each other - such as $P1 \rightarrow P2$ and $P8 \rightarrow P9$) are identified by the second figure in the PSDF description.

Of importance for every granting action is the information on request source (the master), the destination (the slave and its segment number), and the number of packets the transfer is to be repeated. We can extract the specified information from the previously obtained SAM model.

Thus, we can already identify a few elements of the ACC, as follows (all the variables are of a natural type).
- *source*. Identifies the requesting master.
- *dest*. Identifies the target slave.
- *dest_seg*. Identifies the target slave's segment.
- *count*. The number of packets the master has to send to the specified slave (the first number in the PSDF description).

Considering the above, the schedule of the transfers in the first segment (Segment 1 - fig. 6) looks like this:

execution line (program_index)	source	dest	dest_seg	count
0	0	1	1	16
1	0	8	1	16
2	1	2	1	15
3	8	9	1	15
4	1	3	1	1
5	8	3	1	1
6	2	3	1	15
7	9	3	1	15
8	3	10	3	1
9	3	11	2	15
10	3	5	2	15
11	3	4	0	1

We model each execution line as values of a multidimensional vector *program* as in the VHDL snippet:

```
-- SA segment 1 snippet_1
program(0) <= (source => 0, dest => 1,
                    dest_seg => 1, count => 16);
program(1) <= (source => 0, dest => 8,
                    dest_seg => 1, count => 16);
program(2) <= (source => 1, dest => 2,
                    dest_seg => 1, count => 15);
...
```

The arbiter then "scans" the snippet and tries to select a single line that will offer support in granting the bus. At this stage, the only basis on which it can take this selection is to check if the source master has requested the bus access, towards the specified destination. If the master specified in one of the execution lines above is granted access, the arbiter decreases the *count* value of the respective line.

Observe however, that multiple masters may request at a given moment the access to the bus. For instance, the arbiter is not able to differentiate from the program lines 1 and 2 above, as both masters 0 and 1 may have their request signals raised. Additional information is thus necessary in order to enforce the selection of a single master.

As a solution, we add an *enabling / disabling* mechanism, materialized by additional execution line values:
- *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*.

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

execution line (program_index)	guard	source	dest	dest_seg	count	enables
0	0	0	1	1	16	2
1	0	0	8	1	16	3
2	1	1	2	1	15	4
3	1	8	9	1	15	5
4	1	1	3	1	1	6
5	1	8	3	1	1	7
6	1	2	3	1	15	8
7	1	9	3	1	15	8
8	2	3	10	3	1	9
9	1	3	11	2	15	10
10	1	3	5	2	15	11
11	1	3	4	0	1	12

The VHDL code corresponding to the above table is:

```
-- SA segment 1 snippet_2
program(0) <= (guard => 0, source => 0, dest => 1,
    dest_seg => 1, count => 16, enables => 2);
program(1) <= (guard => 0, source => 0, dest => 8,
    dest_seg => 1, count => 16, enables => 3);
--------------------------------------------------
program(2) <= (guard => 1, source => 1, dest => 2,
    dest_seg => 1, count => 15, enables => 4);
program(3) <= (guard => 1, source => 8, dest => 9,
    dest_seg => 1, count => 15, enables => 5);
--------------------------------------------------
...
--------------------------------------------------
program(8) <= (guard => 2, source => 3, dest => 10,
    dest_seg => 3, count => 1, enables => 9);
program(9) <= (guard => 1, source => 3, dest => 11,
    dest_seg => 2, count => 15, enables => 10);
program(10) <= (guard => 1, source => 3, dest => 5,
    dest_seg => 2, count => 15, enables => 11);
program(11) <= (guard => 1, source => 3, dest => 4,
    dest_seg => 0, count => 1, enables => 12);
```

Observe that, at several moments in the execution of a snippet, multiple ACC lines may be enabled. The arbiter will consider the first one matching the condition, after which exits the selection process. For instance, the firsts two program lines above can be concurrently selected. In practice, we see an interleaved execution of the two lines.

The execution flow of the arbitration activity, considering the ACC, is illustrated in fig. 7.

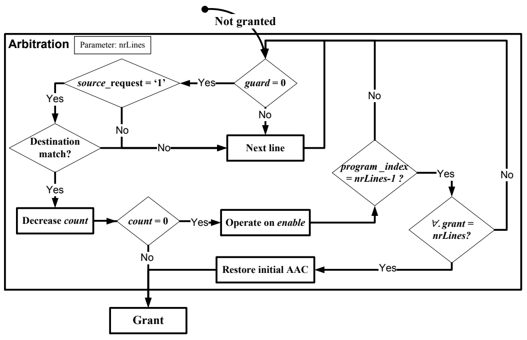


Figure 7. Arbitration flow.

As an additional exemplification, we show below the ACC of the segment 3. Here, *RFL* stands for *Request From Left*, that is, a request coming from the left **BU**, identified as a local master.

```
-- SA segment 3 snippet
program(0) <= (guard => 0, source => RFL, dest => 10,
    dest_seg => 3, count => 1, enables => 1);
program(1) <= (guard => 1, source => 10, dest => 11,
    dest_seg => 2, count => 1, enables => 2);
```

B. **CA** *level arbitration*

A similar approach is taken with respect to the VHDL code to be generated for the **CA** operations. The difference is that, instead of considering as source and destinations the actual devices, the **CA** code only needs information regarding the initiating segment and the target segment. Hence, the *source* field identifies the requesting segment, and the *dest* field is not necessary.

Consider a platform level allocation as in Fig. 8.

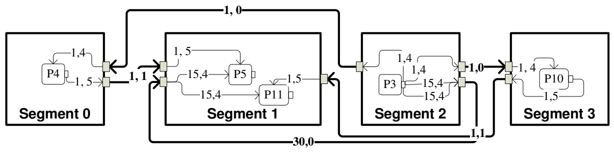


Figure 8. Global transfers view.

Following the specifications described in the previous section, we obtain the **CA** ACC code as follows.

```
-- CA snippet
program(0) <= (guard => 0, source => 1,
    dest_seg => 3, count => 1, enables => 4);
program(1) <= (guard => 0, source => 1,
    dest_seg => 2, count => 30, enables => 5);
program(2) <= (guard => 0, source => 1,
    dest_seg => 0, count => 1, enables => 3);
--------------------------------------------------
program(3) <= (guard => 1, source => 0,
    dest_seg => 2, count => 1, enables => 5);
program(4) <= (guard => 1, source => 3,
    dest_seg => 2, count => 1, enables => 5);
```

C. *Discussion*

When the platform is composed of four or more segments, the **CA** may provide opportunities for actual parallel executions. In our example, once the first ACC line above is marked *executed*, the fifth line becomes *enabled*. This offers the real possibility that the **CA** allows for two simultaneous transfers to execute, from segment 1 to segment 0 (line 3) and from segment 3 to segment 2 (line 5).

The above is supported by the snippet parsing activity performed by the **SA** arbitration module(s) and the **CA** arbitration module. In the first case, once an ACC line is considered for arbitration, the parsing stops, and the read information is used by the procedures in the arbitration block. As a consequence, at the level of segments, we can only have *virtual* parallelism, created by the alterante selection of simultaneously enabled ACC lines. In the case of the **CA**, the parsing does not stop after the first opportunity is established; instead, additional possibilities are analyzed until the whole code is parsed. As discussed, this allows for *actual* parallel executions (inter-segment transfers).

Observe further that ACC lines with the same value (0) for the *grant* fields can be used for arbitration. However, the selection is not determinisitic. Moreover, lines corresponding to transfers with the same transaction index (from the PSDF description) can be written in any order. This is the case with the lines 8 to 11 of *SA segment 1 snippet_2*, for instance. While this does not raise issues from an application execution flow point of view, the designer may want to provide a (possibly) more performance oriented solution, based on the platform characteristics. In the mentioned case, we have selected the described order such that the longer and then the larger inter-segment transfers are favored.

The "snippet-based" approach to the control of transactions on the platform can be (intuitively, at this moment) extended to cover multiple applications to be deployed on an instance of the platform. One can imagine information from different application SAMs being interleaved in a "system ACC". Application dependent constraints will be further necessary to define, wherever applicable, a priority scheme, either at the segment (**SA**) or at the platform (**CA**) levels.

Related work. From a software perspective, Sutter and Larus [11] observe that the most important obstacle in mapping applications on MPSOC seems to be the difficulty to reason about concurrency. Granularity of parallelism, new language designs and better suited abstractions will be strongly necessary in order to support MPSOC approaches. However, the view stops somewhere high above the underlying hardware platform, hence does not address optimality of the solution. Moreover, homogeneous (and even "general use") processing elements are somehow implied.

Van der Wolf et al. [13] introduce the concept of abstract hardware - software interfaces, as a means to accommodate application models on MPSOC architectures. The mentioned study defines several types of such abstractions, suitable for different kinds of application requirements. While the proposed solution addresses well heterogeneous MPSOC designs, the means of control are embedded in the module code, thus hindering the use of off-the-shelf IPs. In comparison, our approach places the control into the two layers of arbiters, and builds a "glueing" control block that can be adapted to any application, while the platform may employ various IP block (the common requirements being functionality and platform suitability).

Lahiri et al. [6] address design optimality for a segmented bus platform similar to the *SegBus*. The architecture 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, precedence relations and controlling aspects are not analyzed. The VHDL snippets, in our case, solve both the contention and the precedence issues, offering control over the platform level operations.

Bouchebaba et al. [5] deal with program complexity (loop parallelization), targeting optimization of memory accesses. As, most probably, memory blocks are to be accessed from devices placed on the same segment, this may become a "local" optimization issue and not platform specific. Thus, our focus is on wider platform control challenges.

IV. Conclusions

We have hereby introduced the control procedures that implement applications on a distributed platform, the SegBus. The solution comes in the form of VHDL code snippets that provide the transfer schedule, such that arbiters at segment and platform levels organize the execution following the application specification. The snippets are viewed as the application-dependent part of the arbiter structure and are based on PSDF representations.

The approach allows for virtual (interleaved) parallelism at segment level, and actual parallelism at platform level. Concurrency is modeled by *enabling / disabling* mechanisms. Multiple applications can be modeled in this way, and deployed on the same platform. Their corresponding ACC snippets must be correlated such that individual application flows are followed and timing constraints fulfilled.

Future work. We are at the moment building an automated tool to provide the ACC specification out of UML based application representations. Additional activities will focus on the construction of a formal support for application mapping and scheduling, considering the transaction based specifications introduced here. This is will be further useful in the analysis of multiple application deployment on the same platform solution.

References

[1] *International Technology Roadmap For Semiconductors.* 2007 Edition.

[2] www.omg.org. *UML Superstructure Specification, v2.0.*

[3] *The Intl. Forum on Application-Specific Multi-Processor SoC.* http://www.mpsoc-forum.org/

[4] A. Jantsch and H. Tenhunen (Eds.) *Networks on Chip* Kluwer Academic Publishers, 2003.

[5] Y. Bouchebaba et al. *MPSoC Memory Optimization Using Program Transformation.* ACM Trans. Des. Automat. Electron. Syst. 12, 4, Article 43 (September 2007)

[6] K. Lahiri, A. Raghunathan, S. Dey. *Design Space Exploration for Optimizing On-Chip Communication Architectures.* IEEE Trans. on Computer-aided Design og Integrated Circuits and Systems, Vol. 23, No. 6, June 2004. pp. 952-961.

[7] E.A. Lee and D.G. Messerschmitt. *Synchronous data flow.* IEEE Proceedings, Sept. 1987.

[8] C. Park, J. Jung, S.Ha. *Extended Synchronous Dataflow for Efficient DSP System Prototyping.* Design Automation for Embedded Systems Journal,Springer, Vol. 6, N. 3, 2002, pp. 295-322.

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

[10] T. Seceleanu, V. Leppänen, O. Nevalainen. *Improving the Performance of Bus Platforms by Means of Segmentation and Optimized Resource Allocation.* To appear (2009), in the EURASIP Journal of Embedded Systems.

[11] H. Sutter and J. Larus. *Software and the concurrency revolution.* ACM Queue, Vol. 3, No. 7, pp 5462, September 2005.

[12] D. Truscan, T.Seceleanu, H. Tenhunen, J. Lilius. *A Model-Based Design Process for the SegBus Distributed Architecture.* The 15th Annual IEEE Intl. Conference on the Engineering of Computer Based Systems, 2008. pp. 307 - 316.

[13] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, G. Essink. *Design and programming of embedded multiprocessors: an interface-centric approach.* The Intl. Conference on Hardware/Software Codesign and System Synthesis, 2004, pp.206- 217.