Moazzam Fareed Niazi | Tiberiu Seceleanu |
Hannu Tenhunen

An Emulation solution for the SegBus Platform

Turku Centre *for* Computer Science

TUCS Technical Report
No 958, October 2009

An Emulation solution for the SegBus Platform

Moazzam Fareed Niazi
University of Turku, Department of Information Technology
Joukahaisenkatu 3-5 B, FIN-20520 Turku, Finland
moazzam.niazi@utu.fi

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

Hannu Tenhunen
University of Turku, Department of Information Technology
Joukahaisenkatu 3-5 B, FIN-20520 Turku, Finland
hannu.tenhunen@utu.fi

TUCS Technical Report
No 958, October 2009

Abstract

The report presents an emulation solution for a multi-core segmented bus platform, SegBus, to assess the performance aspects of any specific application on a particular platform configuration, modeled in UML. We present method to transform Platform Specific Model (PSM) of application into Java source code using modeling tool and how the generated code can be utilized by the emulator program to get the execution results. The solution enables us to estimate performance aspects with different platform configurations together with the application at early stages of the design process.

Keywords: Emulator, Domain Specific Language, UML, SegBus, Model Transformation

TUCS Laboratory
Distributed Systems Design

1 Introduction

In recent years, the complexity of the digital systems has increased tremendously, along with the decreased technological figures. The time to market is also shrinking, imposing challenges for the designers to adopt new design methods. The designers must do a better job of supporting platform-based design, which is becoming the most popular approach to developing complex systems. The platform-based approach may refer to either single chip or multi-chip solution. We address here issues related to the former case.

The use of a hardware emulator for platform-based design can increase the efficiency of the development team and improve both design verification and embedded-software development at early stages of the design process. Design decisions taken at early stages of the development process, impact heavily on the quality of the eventual system implementation. Therefore, the application running on such platforms can take full benefits from all the features exposed by the platform, if it is configured optimally. The specific platform we consider in this study is the *SegBus* platform [13].

The *Unified Modeling Language* (UML) [1] has been utilized in novel design methods proposing a solution for the challenge. We continue here the work towards establishing a full functional unitary framework for platform modeling, application mapping and system (platform+application) emulation, such that performance aspects are targeted, estimated and adjusted to optimal levels in a correct and fast manner. While the main aspects of the platform modeling and application mapping has already been introduced in the form of a *Domain Specific Language* (DSL) in [10], we address here issues related to system emulation. *Model-to-text* (M2T) transformation [2] plays a key role in Model-Driven Architecture (MDA) based development [6]. The outcomes produced by M2T usually are source code files in any desired high-level programming language like Java, C++, etc.

The approach we deliver in this report is based on the activities for building an emulator program targeting the *SegBus* platform. An emulator is a program that imitates the behavior of a device/hardware (the *SegBus* platform in our case) or a program, while a simulator is a software that duplicates some real process and environment in almost all possible ways e.g. flight simulator - simulates the functionalities of an aircraft, etc. The *SegBus* emulator enables us to evaluate the performance aspects of any given application running on a specific platform configuration, defined during modeling.

In addition, the emulator will support the analysis of various *SegBus* instances that may answer, better or worse, to specific application requirements. It helps to decide at early stages of design process which platform configuration will be most suitable for any particular application before moving towards lower abstraction levels. The code generation engine, supplied by the *MagicDraw UML* [5] tool transforms PSM of the system into Java Source code. The generated source code is then employed by the emulator application to estimate the utilization of platform

elements with respect to data transfers and total execution time. After the analysis of the returned results, the designer is able to make decision at this stage whether emulated configuration will be best/optimal or not for the target application, and can change the platform configuration before moving towards lower levels of the design process.

Related work. The primary objective while designing emulator applications is to get as much as possible accuracy in estimating the execution results that we can expect from the real platform. Several research studies have been presented in recent years where the target was to achieve an emulation program for different hardware platforms, specially for the Network-on-Chip (NoC) [8], but there exists a number of emulation tools for other areas as well.

Schelle et al. [12] introduced an emulation tool - *NoCem*, for NoC exploration. The tool provides capability to emulate memory architectures, asymmetric processor configuration, special purpose offload, etc. The tool is able to deliver path latencies used for any particular transfer between processor cores and provides a true picture of the communication bottlenecks within the NoC platform.

Liu et al. [9] presented *NoCOP* - an emulation and verification framework for exploring the on-chip interconnection architecture. An instruction-set simulator and universal serial bus communicator has also been introduced to set the parameters for the emulation environment. Through the experimental results using both software and hardware, the authors proved that the proposed emulation/verification framework can speed up the simulation, preserve the cycle accuracy and decrease the usage of the resources of the Field Programmable Gate Array (FPGA).

Genko et al. [7] presented a NoC emulation platform implemented on FPGA. The NoC hardware platform has been implemented on a Virtex-II FPGA, which consists of network injection, reception and controller components. The processor core PowerPC has been integrated into the hardware platform and functions as a controller. Instead of merely being the platform where the circuit is prototyped, the method can speed up functional validation and add flexibility to the NoC configuration exploration. The major drawback in their approach is the use of processor core in the hardware to control and monitor the network at the cost of FPGA resources, already limited.

Overview of the report. In the rest of the report, we proceed as follows. In section 2, we provide a short description of the *SegBus* platform, its structural characteristics and associated Domain Specific Language. Next, in section 3, we provide description of proposed emulation solution with its all involved phases. Furthermore, in section 4 we provide an example of a simplified stereo MP3 decoder in the context of proposed solution to show the significance of the method, followed by conclusion of work in section 5.

2 Background

2.1 Segmented Bus Architecture

A segmented bus is a "collection" of individual buses (segments), interconnected with the use of FIFO like structures. 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 to establish a connection between modules located in different segments. Due to the segmentation of the bus lines, and their relative isolation, 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.

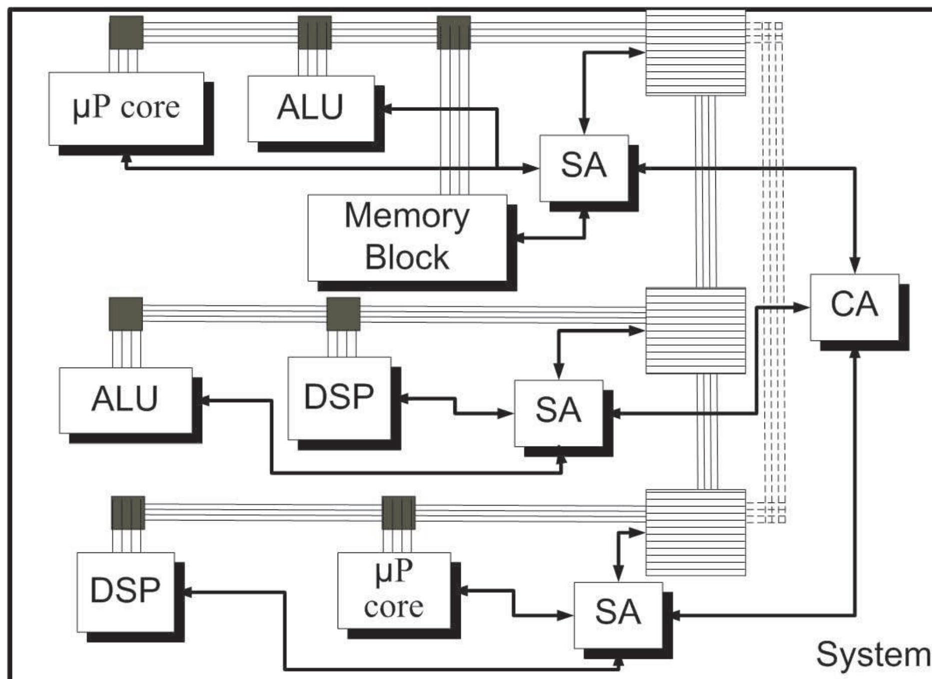


Figure 1: Segmented bus structure.

The *SegBus* communication platform is built of components that provide the necessary separation of segments - *Border units* (***BU***), arbitration units - the *Central Arbiter* (***CA***) and local, *Segment Arbiters* (***SA***). The application then is realized with the support of (library available) *Functional Units* (***FU***).

The *SegBus* platform has a single ***CA*** unit and several ***SA***s, one for each segment. The ***SA*** of each bus segment decides which device (***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" package based bus protocol, with ***SA***s arbitrating the access to local resources.

The inter-segment communication, is also a package based, circuit switched approach, with the ***CA*** having the central role. The interface components between adjacent segments, the ***BU***s, are basically FIFO elements with some additional logic, controlled by the ***CA*** and the neighboring ***SA***s. A brief description of the communication is given as follows.

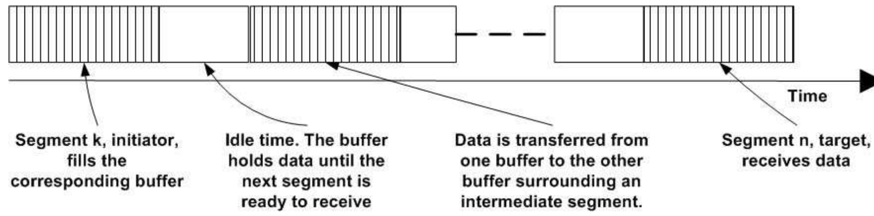


Figure 2: Inter-segment package transfer.

Whenever one ***SA*** recognizes that a request for data transfer targets a module outside its own segment, it forwards the request to the ***CA***. The later 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, and it starts filling the buffer of the appropriate bridge with the package data. Following a signaling protocol, the data is taken into account by the corresponding next segment ***SA***, which forwards it further, towards the destination. At this point, the ***SA*** of the targeted segment routes the package to the own segment lines, from where 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.

The arbitration at ***CA*** level implements the application data flow, 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.

2.2 DSL for the SegBus Platform

The *Domain Specific Language* (DSL) [10] for the *SegBus* platform is the specification language that is used to model the *SegBus* platform at higher-level of abstraction. The DSL provides ability to model platform elements in the form of high-level graphical constructs and provide methods to map partitioned application components on particular segment in a fast and correct manner.

The DSL comprises of a number of structural constraints related to the platform, written in *Object Constraint Language* (OCL) [3], to implement the correct component approach to platform design. These constraints are used to validate

our models. Upon breach of any constraint requirement during the design process, the tool provides appropriate error message, so that the designer can take proper action to make the model correct according to platform requirements.

The relationship between all platform elements are defined using *Customization* classes. The customization classes comprise of tags that store user-defined DSL customization rules. The customization rules are parsed and interpreted by the *DSL Customization Engine* (provided by the tool) to assist validation process.

Once we model the platform and map the application components on to the platform correctly, we apply validation process to get the correct *Platform Specific Model* (PSM) of the application. If there exists some errors in the model, we get error message(s) and associated model element become highlighted.

Finally, the PSM model can be transformed into source code of any desired high-level programming language (Java in our case) to further analysis of the desired platform configuration. We employ the generated source code for emulating the performance aspects of the configured system, as described in the next section.

3 The SegBus Emulator

Generally, emulation is necessary while designing applications targeting hardware devices and platforms. The huge design and manufacturing costs of such hardware platforms motivate designers to develop emulators and verify the execution results. An emulator provides the same functionality as the original hardware platform or computer program. Designing an emulator requires a thorough understanding of the target device or platform. We have developed the *SegBus* emulator to test platform configuration and estimate performance aspects before moving towards the final implementation.

Figure 3 illustrates a general overview of the *SegBus* design process employing DSL and emulation. At the top level, the transformation of the platform concepts into the high-level graphical constructs has already been done in [10] to form a DSL, specific for the *SegBus* platform. The DSL provides a graphical environment where a designer can map *Platform Independent Model* (PIM) of the application on to platform quickly and assign pre-existing components from the *SegBus Component Library* during modeling. The application should be already partitioned before mapping it on to the platform according to available library components. The model can be validated for possible mistakes to get a correct PSM. The *configuration data* is built from the PSM. The configuration data contain information about relative placement of each application components on different segments and other necessary information like IDs of all elements, placement information, etc. Finally, we transform the PSM of the application into Java source code using M2T transformation supplied by the tool for code generation. The generated source code is then configured using configuration data and compiled along with the emulator source code to get an executable application.

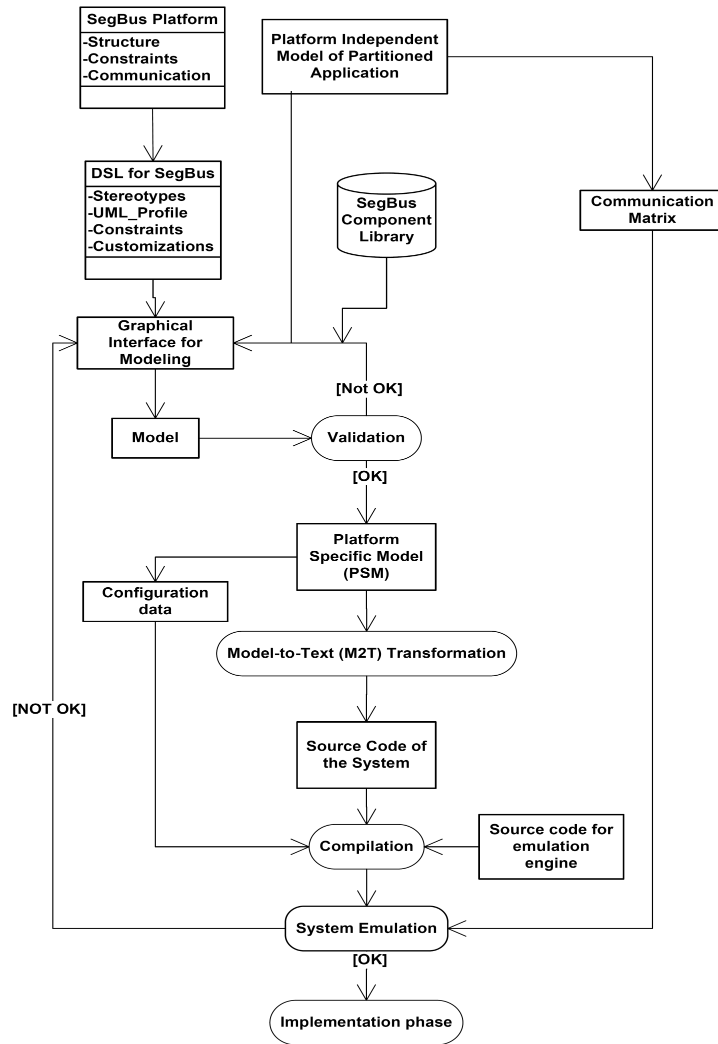


Figure 3: Design process of the SegBus platform using DSL and emulation.

The *communication matrix* is the specification of device-to-device transactions between application components. Each entity in the communication matrix describe how many data items need to be transfered from one device to any other device. Based on the matrix, the *PlaceTool* application [14] finds the optimal device allocation solution, given the platform specifics (the number of segments).

Before the emulation, the emulator application reads the communication matrix and considers the structure (segment organization and resource allocation) from the DSL description. Upon completion of the emulation, the tool returns results of the transactions from each platform element, performed during execution. Figure 4 shows the flow of activities involved in the proposed design process. The major phases involved between modeling in DSL and emulation are: *Model Transformation* and *Setup for emulation*, which we briefly discuss below.

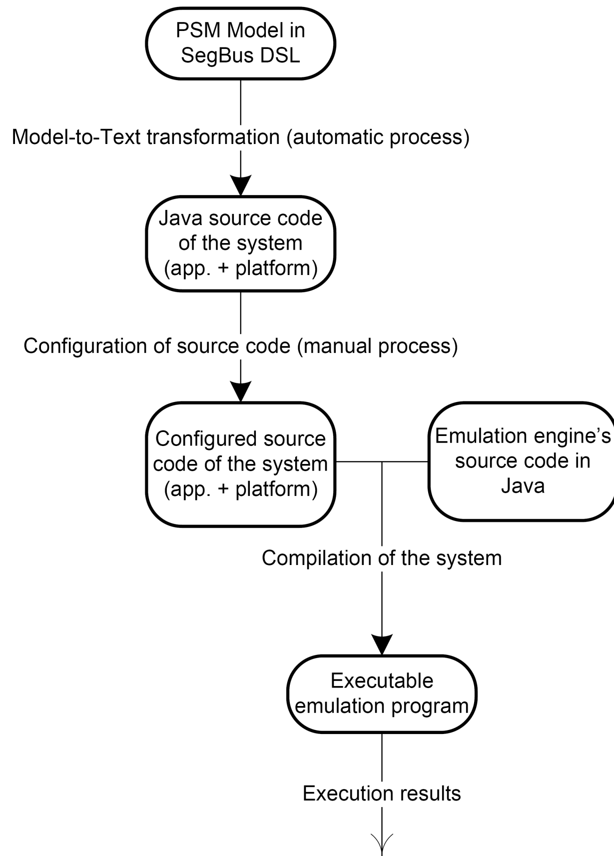


Figure 4: Flow of activities in the design process.

3.1 Model Transformation

The first phase for performing emulation on any modeled *SegBus* configuration in DSL is to transform the model into source code so that the configuration can be used by the emulator program for further analysis.

The emulator application is written in Java language [4]. We choose Java as a target language for the source code generated from the model to make it compatible with the emulator program. The code generation engine of the tool does provide capability to transform model into source code as per M2T specification [2].

A *code engineering set* needs to be introduced in the tool where we specify required type of transformation i.e. Model-to-Model, Model-to-Text (as in our case), etc. The code engineering set consists of a set of model elements whose source code we want to generate during transformation. We make a code engineering set consisting of platform elements (SAs, CA, BUs) and all application components in the form of processes (P0, P1, etc.). A directory is also specified where the generated source code to be saved. After applying transformation on

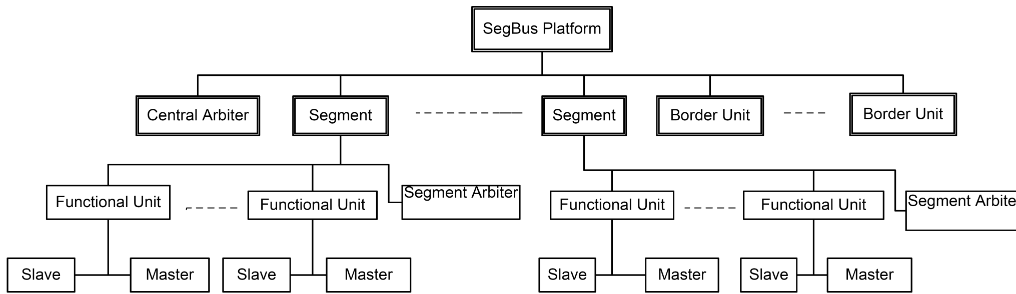


Figure 5: Hierarchical structure of the SegBus elements.

our model, we get the required source code in mentioned directory.

The generated source code contains a set of files. Each file corresponds to a particular platform element or application component. Figure 5 shows the hierarchical structure of the platform elements. At the top level is the *SegBusPlatform* itself composed of *Segment*(s) and exactly one ***CA***. Every segment is composed of at least one ***FU***, and exactly one ***SA***. Each segment is connected with other neighboring segment through ***BU***. One ***FU*** may contain *Master*(s) and *Slave*(s). The generated code also follow the same hierarchy as depicted in Figure 5. The ***FU***s and ***SA*** always instantiate in the segment's source code as class attributes. The source file of *SegBusPlatform* instantiates required number of segments, matching number of ***BU***s and exactly one ***CA*** in the form of class attributes. The scope of all attributes is kept *public* for the ease in programming.

3.2 Setup for emulation

The next phase of the design methodology is to generate the source files and make them ready for execution with the emulator application. The emulator has been programmed in a way to exhibit the exact behavior of actual platform instance. The functionality and behavior of each platform element (***SA***, ***CA***, ***BU***, etc) are programmed and stored in individual java source files. A number of monitoring statements are introduced in different section of ***SA***, ***CA*** and ***BU*** codes. These monitoring statements count clock ticks involved in any transfer, either intra-segment or inter-segment. The *arbitrate* method in ***CA*** and ***SA*** source code performs arbitration and called by the emulator application several times during execution.

At the ***SA*** level, we put statements in *arbitrate* method to count requests coming from the application processes. Separate counters are also put to count both kinds of requests (intra and inter-segment). These statements help us later to analyze the configured system and provides means to take optimal decision according to needs. In case of inter-segment transfers, there exist separate counters to count how many packages transfered to left and right side ***BU***.

At the ***CA*** level, monitoring statements in *arbitrate* method count the number

of clock ticks ***CA*** consumed while *setting* and *resetting* related grant signal in response to inter-segment requests. The monitoring statements at ***BU*** level counts how many packages received from, and transfered to, left and right-side segment. It also counts total number of clock ticks during all transfers.

When we get the source code after model transformation, the emulator application performs *setup* operation on the generated source code to make it ready to work with existing emulator application. The steps involved in this setup process are:

- Copy the functionality (source code) of each platform element to its associated file.
- Introduce *Constructor* method in each source file.
- Setup each file to be executable in a multi-threaded environment (discussed in section 3.3) by providing proper interface (*Runnable*) and method (*run()*).

Manual configuration is also required before emulation. The tasks associated with manual configuration are:

- Setup of thread pool in the platform's class.
- Instantiation of threads in proper segment class.
- Setting IDs for processes.
- Assignment of all threads to the Java's *ExecutorService* - a class that handles the execution of threads.

3.3 Implementation approach

The microprocessor in a personal computer (PC) has the characteristics to run computer program instructions in sequential order. On the other hand, the hardware devices have the characteristics to run in parallel with other devices. The main challenge in emulator development for us to transform the parallel behavior of hardware elements associated with platform into some special form that can be run on microprocessor and exhibit the correct characteristics of hardware. *Multi-threading* is not a new idea and is exists since many years. Generally, every running program in a PC is called a *process*. Multi-threading is the task of creating a new *thread* of execution within an existing process rather than starting a new process to begin function. All the threads in a process share the same allocated memory. The parallel execution of threads within the same process is often considered as a more efficient use of the resources of the PC. Multi-threading employs time-division multiplexing to executes threads in parallel. Threads are

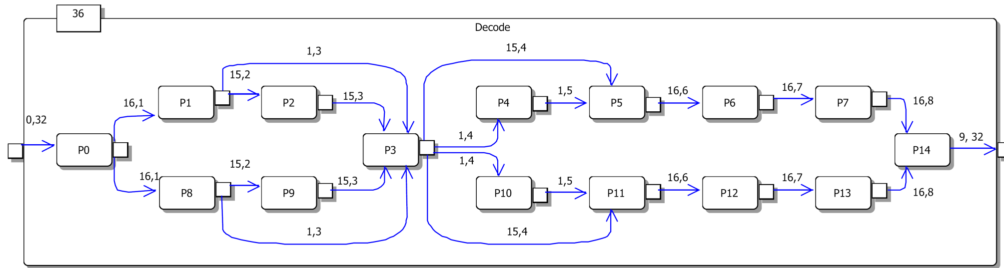


Figure 6: Partitioned application model of MP3 decoder.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14
P0	0	576	0	0	0	0	0	0	576	0	0	0	0	0	0
P1	0	0	540	36	0	0	0	0	0	0	0	0	0	0	0
P2	0	0	0	540	0	0	0	0	0	0	0	0	0	0	0
P3	0	0	0	0	36	540	0	0	0	0	36	540	0	0	0
P4	0	0	0	0	0	36	0	0	0	0	0	0	0	0	0
P5	0	0	0	0	0	0	576	0	0	0	0	0	0	0	0
P6	0	0	0	0	0	0	0	576	0	0	0	0	0	0	0
P7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	576
P8	0	0	0	36	0	0	0	0	0	540	0	0	0	0	0
P9	0	0	0	540	0	0	0	0	0	0	0	0	0	0	0
P10	0	0	0	0	0	0	0	0	0	0	0	36	0	0	0
P11	0	0	0	0	0	0	0	0	0	0	0	0	576	0	0
P12	0	0	0	0	0	0	0	0	0	0	0	0	0	576	0
P13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	576
P14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 7: The communication matrix for the example.

obtained from the pool of available ready-to-run threads and run on the available microprocessor(s).

We employ Java's multi-threading feature in our emulator application. All classes related with emulator application, including the generated source code from the model run as threads during execution. Each class implements *Runnable* interface from *java.lang* package by introducing *run()* method. The method executes when emulation starts and performs dedicated functionality. The platform class creates a thread pool using an instance of *ExecutorService* class from the *java.util.concurrent* package. The size of the thread pool depends on the number of ***FU***s and platform elements (***BU***s, ***SA***s, etc). We add all ***FU***s and platform elements into the thread pool before emulation. During execution of emulator, all threads execute in parallel to depict intrinsic characteristics of hardware.

Emulation and estimation. The final step of the design methodology is to emulate the platform configuration after file setup. In general, application processes communicates with each other at different time instant after performing specific computation on the supplied data. We assume here that all processes try to communicate with each other as soon the emulation starts. During emulator development, we skip some timing factors that are less important in estimating performance. For instance, we didn't include the time necessary to synchronize between two adjacent clock domains, converging at the ***BU***s. This time is parameterized, but a value of two clock ticks is usually considered, at the translation of any signal across two clock domains. We also did not computed the time necessary for

the ***SA***s to set the grant signal for a particular request and corresponding master responds, due to a similarly low value, which is also overlapping in time with the on-going activities within the segments.

We compared the estimated results that we get from the emulator with the application running on the real platform and we conclude that the estimated results are more than 90% accurate in the absence of previously mentioned timing factors.

After we supply the communication matrix to the emulator, the tool instantiates the threads correponding to platform elements, supply particular value from communication matrix to each ***FU*** and starts the simulation process. Upon completion, the emulator returns results from the platform elements' execution. Some of the results are listed below:

- Total clock ticks consumed for the operation of the ***CA*** and each of the ***SA***s.
- Total inter-segment requests received by ***CA*** and by each of the ***SA***s.
- Total clock ticks consumed by each of the ***BU***s.
- etc.

The clock tick's counter is incremented in ***SA*** and ***CA*** at various moments. Each ***SA*** has its own counter for counting clock ticks and the execution time for each device is computed separately (discussed in next section). For instance, the ***SA*** increments the clock tick's counter while checking the incoming requests from ***FU***s in the segment. It increments the counter when it receives intra or inter-segment transfer request from one of the ***FU*** in the segment. If the request is for inter-segment transfer, it forwards the request to ***CA*** and increments the counter. While setting and resetting grant signal in response to any request, it also updates the clock tick's counter. During the time limit for any transfer, the ***SA*** always increments the clock tick's counter continuously till the time limit ends. The ***CA*** increments the clock tick's counter every time when it checks for any incoming inter-segment transfer request from a ***SA***. It increments the counter while setting and resetting grant signal for any inter-segment transfer request. Furthermore, when one of the segment finishes its job in an inter-segment transfer, the ***CA*** resets the necessary signal associated with particular segment and increments the clock tick's counter.

4 Example using the Emulator program

We demonstrate our approach with an example of modeling a simplified stereo MP3 decoder [11] on the *SegBus* platform and associated emulation results. The modeling is done using DSL [10]. The application has already been partitioned up to a right granularity level [15]. Here, we map application processes in three different platform configuration, using one, two and three segments, with linear

topology in all configuration. The package size is set to 36 data items in each package. Figure 8 illustrates the allocation of application processes on each platform configuration, where segment borders are marked as '||'. The communication matrix is generated from the partitioned-application model (see Figure 6). Figure 7 illustrates the communication matrix associated with the example application. For instance, the transaction between *P0* and *P1* consists of 576 data items, packed into 16 packages. We emulate each configuration on the *SegBus* emulator to analyze the performance aspects. We intentionally skip here the emulation results of one and two segments configuration. But, the emulation results of 3 segments platform configuration are given below in which 'CA' represents the central arbiter of the platform, 'Segment X' represents the segment and X denotes the ID (1,2,3,..), 'SAn' represents the segment arbiter associated with segment n', 'BUxy' represent the border unit between segment 'x' and segment 'y'. We set clock frequency of segment 1, 2, 3 and central arbiter as 91MHz, 98MHz, 89MHz and 111MHz respectively.

Configuration	Allocation
One Segment	All FU on the same segment
Two Segments	4 5 6 7 10 11 12 13 14 || 0 1 2 3 8 9
Three Segments	0 1 2 3 8 9 10 || 5 6 7 11 12 13 14 || 4

Figure 8: Allocation of processes on different platform configuration.

• **Three Segments configuration**: In this configuration, processes (0,1,2,3,8,9,10) are on segment 1, processes (5,6,7,11,12,13,14) are on segment 2 while process 4 is on segment 3. Border unit (BU12) is between segment 1 and 2, while border unit (BU23) is between segment 2 and 3.

```
P4 finished its task at 2808750 ps
P10 finished its task at 2747250 ps
P11 finished its task at 81632000 ps
P13 finished its task at 81632000 ps
P12 finished its task at 81632000 ps
P2 finished its task at 82417500 ps
P7 finished its task at 163264000 ps
P5 finished its task at 163264000 ps
P6 finished its task at 163264000 ps
P9 finished its task at 164835000 ps
P1 finished its task at 175824000 ps
P8 finished its task at 175824000 ps
P0 finished its task at 351648000 ps
P3 finished its task at 457142400 ps

CA total ticks = 52465
Time consumed = 472657185 ps @ 111MHz

BU12:
Total input packets = 32,
Total output packets = 32
    Packet Received from Segment 1 = 32,
```

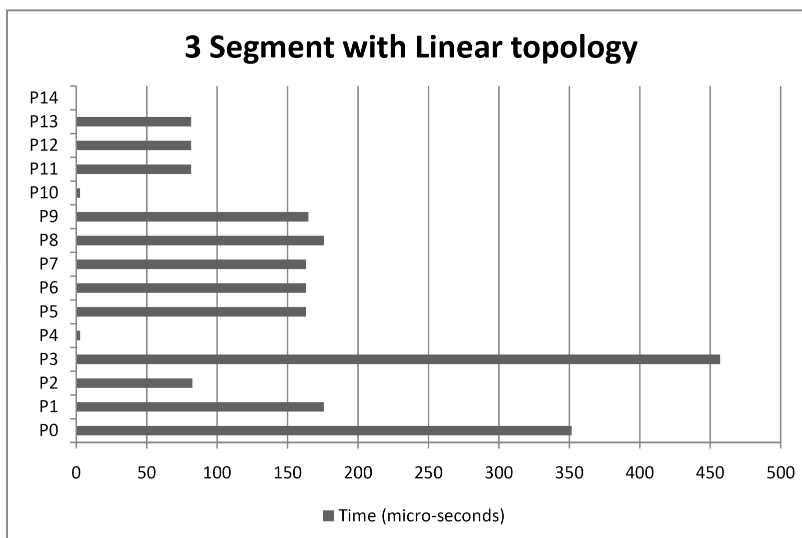


Figure 9: Progress on time of each application process in 3 segment, linear topology with package size of 36 data items configuration.

```
     Packet Transfered to Segment 1 = 0
    Packet Received from Segment 2 = 0,
     Packet Transfered to Segment 2 = 32
    Total Clock Ticks = 4362

BU23:
Total input packets = 2,
Total output packets = 2
    Packet Received from Segment 2 = 1,
     Packet Transfered to Segment 2 = 1
    Packet Received from Segment 3 = 1,
     Packet Transfered to Segment 3 = 1
    Total Clock Ticks = 149

Segment 1:
Packets transfered to Left = 0,
Packets transfered to Right = 32

Segment 2:
Packets transfered to Left = 0,
Packets transfered to Right = 0

Segment 3:
Packets transfered to Left = 1,
Packets transfered to Right = 0

SA1 total ticks = 42120,
Total intra-segment requests = 123,
Total inter-segment requests = 32
Time consumed = 462856680 ps @ 91MHz

SA2 total ticks = 48527,
Total intra-segment requests = 96,
Total inter-segment requests = 0
Time consumed = 495169508 ps @ 98MHz

SA3 total ticks = 43699,
```

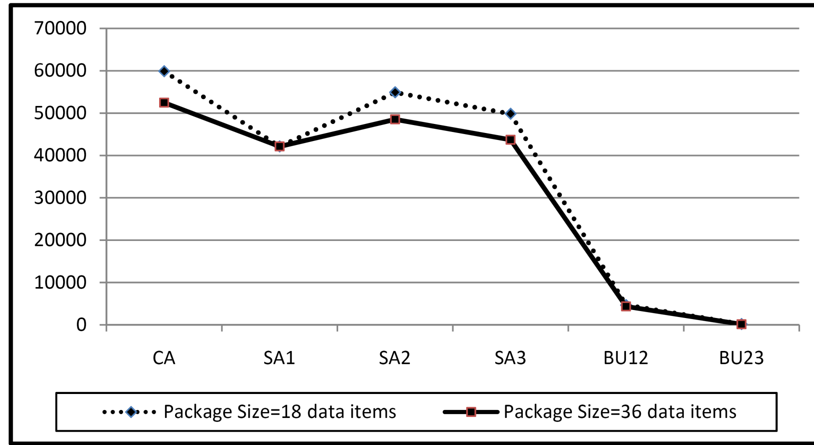


Figure 10: Activity graph of different platform elements in 3 Segment and linear topology configuration for 18 and 36 bit package sizes.

```
Total intra-segment requests = 0,
Total inter-segment requests = 1
Time consumed = 490958265 ps @ 89MHz
```

The total time consumed by ***SA*1** to finish all associated jobs is t_{SA_1}. This time can be calculated by multiplying total clock ticks with the associated segment's clock period. The execution time is calculated when all ***FU***s finish their jobs, all packages are transmitted to its relevant destination and grant signal of all arbiters are *clear*. Each ***FU*** inform the emulator about its status by setting a flag in the emulator. Hence, the total execution time of the application can be calculated by taking the maximum of time consumed by central arbiter and all segment arbiters that is *max*(t_{SA_1}, t_{SA_2}, ..., t_{CA}). Figure 9 shows the progress of each ***FU*** on time line using 3 segments, linear topology with package size of 36 data items. The figure shows the time instant on which any specific process finished its dedicated job. Process *P3* takes the longest time of 457.14 μs to complete its task in the given configuration, while the estimated total execution time for the application is 495.17 μs. After running the same partitioned-application on the real platform instance, we get the actual execution time as 515.2 μs. So, the estimated results we get are 96% accurate. When the same platform configuration is emulated with reduced package size that is 18 data items in a package, we get the estimated execution time of 560.16 μs.

In addition, we change the platform configuration by shifting process *P9* from segment 1 to segment 3 with the previously mentioned design process to estimate the execution results. While, rest of the configuration will remain same with package size of 36 data items. The emulation result shows that the estimated execution time of updated configuration is 540.4 μs.

Figure 10 illustrates the activity graph of 3 segment, linear topology configuration with different package sizes (18 and 36 data items). Based on emulation

results, it's the job of the designer to decide which configuration would be best suited for final implementation. Such decisions at early stages of design process not only improve the quality of eventual system in terms of performance, but also improves power consumption up to some extent. The granularity level of application components can also be adjusted to eliminate the traffic congestion introduced by some ***FU***s that will further improve the overall performance.

5 Conclusions

The report presented methods for specifying, modeling and implementing multicore embedded systems using UML-based methodology. We introduced emulation technique for estimating performance aspects of desired *SegBus* configuration. We described how the source code can be generated from the models, specified in DSL, and introduced mechanism to emulate the modeled configuration in early stages of the development process.

The emulation-based solution enables us to analyze any platform configuration with respect to performance. The design decisions to get optimal performance from the platform has now become easy using the proposed methodology. The methodology allows a designer to adjust the high-level design in a way to take full benefits from the features exposed by the platform.

References

[1] *Unified Modeling Language (UML) Superstructure Specification, version 2.0.* . http://www.omg.org

[2] *Eclipse Modeling - Model-to-Text Transformation.* . http://www.eclipse.org/modeling/m2t/

[3] OMG. *Object Constraint Language (OCL) 2.0 Revised Submission, version 1.6.* January 2003.

[4] Java Programming Language. http://java.sun.com

[5] MagicDraw UML. http://www.magicdraw.com

[6] Model-Driven Architecture. http://www.omg.org/mda/

[7] N. Genko, D. Atienza, G. D. Micheli, L. Benini. *Feature-NOC emulation: a tool and design flow for MPSoC.* IEEE Circuits and Systems Magazine, vol. 7, 2007, pp. 42-51.

[8] A. Jantsch, H. Tenhunen. *Networks on Chip.* Kluwer Academic Publishers, 2003.

[9] P. Liu, C. Xiang, X. Wang, B. Xia, Y. Liu, W. Wang, Q. Yao. *A NoC Emulation/Verification Framework.* In Proceedings of 6th International Conference on Information Technology: New Generations, 2009, pp. 859-864.

[10] M. F. Niazi, K. Latif, T. Seceleanu, H. Tenhunen. *A DSL for the SegBus Platform.* In proceedings of 22nd IEEE International System-on-Chip Conference (SOCC), 2009, pp. 393-398.

[11] C. Park, J. Jang and S. Ha. *Extended Synchronous Dataflow for Efficient DSP System Prototyping.* Journal Design Automation for Embedded Systems, Springer Netherlands, vol. 6, no. 3, 2002, pp. 295-322.

[12] G. Schelle, D. Grunwald. *Onchip Interconnect Exploration for Multicore Processors utilizing FPGAs.* 2nd Workshop on Architecture Research using FPGA Platforms, 2006.

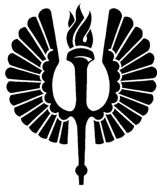
[13] T. Seceleanu. *The SegBus Platform - Architecture and Communication Mechanisms.* Journal of Systems Architecture (2006), doi:10.1016/j.sysarc.2006.07.002

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

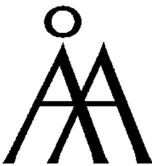
[15] D. Truscan, T. Seceleanu, J. Lilius, H. Tenhunen. *A Model-based Design Process for the SegBus Distributed Architecture.* In Proceedings of the 15th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS), 2008, pp. 307-316.

Turku Centre *for* Computer Science

Joukahaisenkatu 3-5B, FIN-20520 Turku, Finland | www.tucs.fi



University of Turku
- Department of Information Technology
- Department of Mathematics



Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 978-952-12-2340-2
ISSN 1239-1891