A Performance Estimation Technique for the SegBus Distributed Architecture

Moazzam Fareed Niazi
Turku Centre for Computer Science TUCS
Department of Information Technology
University of Turku, Finland
moazzam.niazi@utu.fi

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

Hannu Tenhunen
Turku Centre for Computer Science TUCS
Department of Information Technology
University of Turku, Finland
hannu.tenhunen@utu.fi

***Abstract*—We propose a performance estimation technique for a multi-core segmented bus platform, SegBus. The technique enables us to assess the performance aspects of any specific application on a particular platform configuration, modeled in Unified Modeling Language (UML). We present methods to transform Packet Synchronous Data Flow (PSDF) and Platform Specific Model (PSM) models of the application into Extensible Markup Language (XML) schemes using modeling tool and how the generated XML schemes can be utilized by the emulator program to get the execution results. The technique facilitates us to estimate performance aspects of application mapped on a number of different platform configurations during the early stages of the design process.**

I. 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 place 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 [15].

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 [11], 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 outcome produced by M2T usually are textual artifacts from the provided models. These textual artifacts could be XML schema or source code of any high-level programming language like C++, Java, etc. The XML Schema provides means for defining the content, structure and semantics of XML documents.

The technique we deliver in this paper 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 PSDF and PSM models of the system into XML schemes. The generated XML schemes are 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. [13] 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. The tool is written in VHDL with extensive use of generics throughout the code, but the tool deals with designs at lower levels of abstraction and hence less flexible to use, unlike our approach which is easy to use and deals with designs at much higher levels of abstraction.

Liu et al. [10] 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). The design under emulation needs to be programmed onto a FPGA device and a separate host computer is responsible for initializing and managing emulation of the programmed design in the FPGA which makes it less flexible compared to our approach which is more flexible and doesn't require any FPGA device and consideration about deeper lavels of abstraction.

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.

II. Background

A. 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.

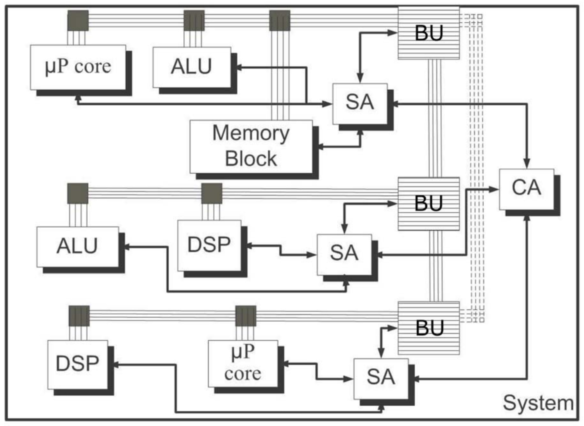


Figure 1. Segmented bus structure.

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.

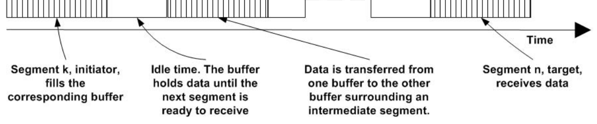

Time
Segment k, initiator, fills the corresponding buffer
Idle time. The buffer holds data until the next segment is ready to receive
Data is transferred from one buffer to the other buffer surrounding an intermediate segment.
Segment n, target, receives data

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.

B. DSL for the SegBus Platform

The *Domain Specific Language* (DSL) for the *SegBus* platform is the specification language that is used to model the *SegBus* platform at higher-level of abstraction, based on stereotypes stored in the *SegBus* UML profile [11]. The DSL provides ability to model application and 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 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.

Before the current work, the DSL was only capable of modeling application at *Platform Specific Model* (PSM) level. Here, we add capabilities to model application at the *Packet SDF* (PSDF - section III-A) level, too. We introduce three new stereotypes, that is, *InitialNode*, *ProcessNode* and *FinalNode*, in the UML profile of DSL. The profile defines the main structural elements of the platform. The new stereotyped classes related to PSDF are generalization of the metaclass *UML Standard Profile::UML2MetaModel::Classes::Kernel::Class*. We also introduced their related customization classes and set tags with suitable values. We skip here further details about tag values intentionally because of the space limitation.

Once we model the application components as PSDF, model the platform and map the application components on to the platform correctly, we apply validation process to get the correct 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 PSDF and PSM model can be transformed into XML schemes for further analysis of the desired platform configuration. We employ the generated XML schemes for emulating the performance aspects of the configured system, as described in the next section.

III. 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.

A. 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 [14]. 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 comprises mainly two elements: *processes* and *data flows*; data is, however, organized in data items, which are later transformed into packets according to package size during execution. Processes transform input data packets into output ones, whereas packet flows carry data from one process to another. A *transaction* represents the sending of one data packet by one source process to another, target process, or towards the system output. A *packet flow* is a tuple of four values, P_t, D, T and C. The P_t value represents the target process for the given transactions; the D value represents the number of data items 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; and the C value represents the number of clock ticks a process consumed before sending one package. Thus, a flow is understood as the number of data items (later transformed into packets) issued by the same process, targeting the same destination, having the same ordering number and same clock ticks require to process one individual package.

If s is the package size (number of data items in a package) in the platform configuration, then the *Packet SDF (PSDF)* of a certain system is a sequence of packet flows, $< (P_{t_x}, \frac{D_1}{s}, T_1, C_1), \ldots, (P_{t_x}, \frac{D_n}{s}, T_n, C_n) >$, where $\forall i, j, x \in \{1, \ldots, n\} \cdot \frac{D_i}{s} \neq \frac{D_j}{s}$ 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.

B. Design Methodology

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 [11] to form a DSL, specific for the *SegBus* platform. The DSL provides a graphical environment where a designer can model PSDF and PSM of the application quickly and assign pre-existing components from the *SegBus Component Library* during the modeling. The application should be already partitioned before modeling it in the PSDF form and mapping it on to the platform according to available library components. The model can be validated for possible mistakes to get the correct PSDF and PSM. Later on, we transform both PSDF and PSM of the application into XML schemes using M2T transformation supplied by the tool. The XML schemes contain information about platform elements, application components and their relative placement on different segments.

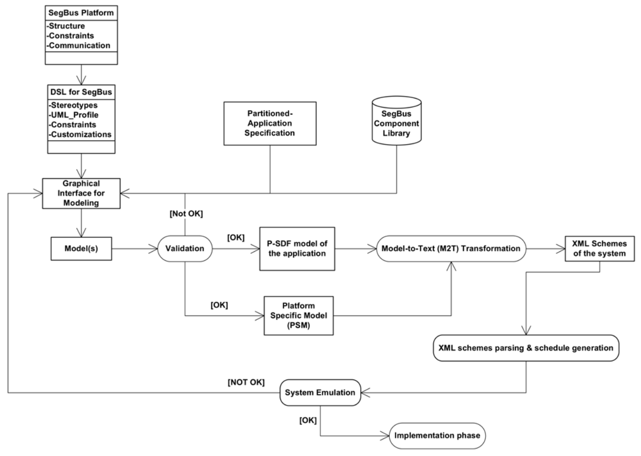

SegBus Platform
-Structure
-Constraints
-Communication
DSL for SegBus
-Stereotypes
-UML_Profile
-Constraints
-Customizations
Partitioned-Application Specification
SegBus Component Library
Graphical Interface for Modeling
[Not OK]
Model(s)
Validation
[OK]
P-SDF model of the application
Model-to-Text (M2T) Transformation
XML Schemes of the system
[OK]
Platform Specific Model (PSM)
XML schemes parsing & schedule generation
[NOT OK]
System Emulation
[OK]
Implementation phase

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

Before the execution, the emulator application reads the XML schemes of the PSDF and PSM models, package size and considers the structure (segment organization and resource allocation) from the XML schema of the PSM. Upon completion, the tool returns results of the transactions from each platform element, performed during execution. Figure 4 shows the operating flow of the emulation in the step-by-step manner. An overview of the operating principles of the emulator is given in the following sections.

C. Basic Concepts

The following considerations apply in the approach to build the emulator as a close match to the *SegBus* architecture and to the application execution.

• The schedule of the application is extracted from the PSDF and implemented within the arbiters, providing the correct sequencing among processing and transfers.

• As at the moment we are not interested in the actual operational results, the ***FU***s are modeled as counters, performing for an established duration. The ranges of the counters will stand as a “processing” time associated with each ***FU***.

• The performance measurements (execution times) are established with respect to the starting moment of the emulation process. While for individual processes this might provide errors in measurement (as certain modules have to wait until data is present in order to start operating), this does not affect the overall application time performance - which is our main target in this study.

• The emulator will be equipped with an array of flags - “Process Status Flags”, each element here corresponding to one process of the application. When a process finishes the activities and related transfers, the appropriate flag is raised.

• During the execution of the application on the emulated platform, monitoring activities are executed to measure the execution times (clock ticks) of the ***FU***s, ***SA***s and of the ***CA***.

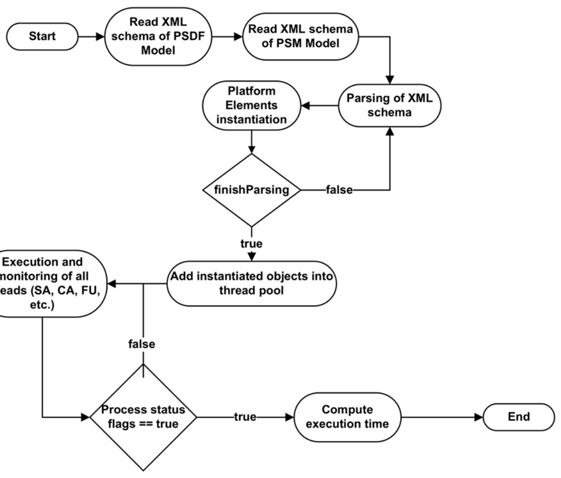
• The operation is considered finished when all the flags described above are high, and there is no activity to execute within any of the platform’s ***SA***s or ***CA***.

D. Model Transformation

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

The emulator application is written in Java language [4] due to its rich collection of classes for handling XML schemes and classes for implementing multi-threaded application (discussed in section III-F). The code generation engine of the tool does provide capability to transform model(s) into XML schema as per M2T specification [2].

A *code engineering set* needs to be introduced in the tool for each model 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 XML content we want to



Start
Read XML schema of PSDF Model
Read XML schema of PSM Model
Parsing of XML schema
Platform Elements instantiation
finishParsing
false
true
Add instantiated objects into thread pool
Execution and monitoring of all threads (SA, CA, FU, etc.)
false
Process status flags == true
true
Compute execution time
End

Figure 4. Operating flow of the emulator.

generate during transformation. We make two separate code engineering sets (one for PSDF and other for PSM) consisting of platform elements (SAs, CA, BUs, etc.) and all application components in the form of processes (P0, P1, etc.). A directory is also specified where the generated XML schemes to be saved. After applying transformation on our PSDF and PSM models, we get the required XML schemes in the mentioned directory.

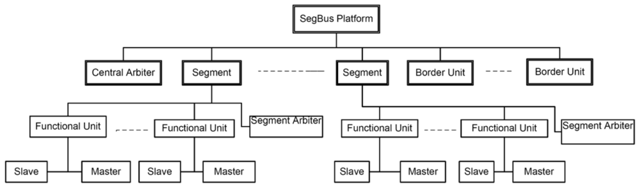


Figure 5. Hierarchical structure of the SegBus elements.

The generated XML consists of a *schema* element and a number of sub-elements, in the form of *complexType* and *element* types.

Each complex type represents a platform element (***CA***, ***SA***, etc.) or application component (P0, P1, etc.). The *name* attribute of each complex type shows the name of the element. Furthermore, each complex type may contain sub-elements. 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*** contains at least one *Master* or one *Slave*. Following, we show an XML snippet of the PSDF model after transformation, consisting of process *P0*, *P1* and their relative transfers to other processes.

```
<xs:complexType name="P0">
     <xs:sequence>
         <xs:element name="P1_576_1_250" type="P1"/>
         <xs:element name="P8_576_1_250" type="P8"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="P1">
     <xs:sequence>
         <xs:element name="P2_540_2_250" type="P2"/>
       <xs:element name="P3_36_3_250" type="P3"/>
   </xs:sequence>
</xs:complexType>
```

Below is the piece of XML snippet of PSM model after transformation, representing the *SegBus* platform instance (*SBP* with three segments as child-elements) and "*Segment 1*" element with its child-elements.

```
<xs:complexType name="SBP">
   <xs:all>
      <xs:element name="segment0" type="Segment0"/>
      <xs:element name="segment1" type="Segment1"/>
      <xs:element name="segment2" type="Segment2"/>
      <xs:element name="ca" type="CA"/>
      <xs:element name="bu12" type="BU12"/>
      <xs:element name="bu23" type="BU23"/>
   </xs:all>
</xs:complexType>

<xs:complexType name="Segment1">
   <xs:all>
      <xs:element name="buRight" type="BU23"/>
      <xs:element name="buLeft" type="BU12"/>
      <xs:element name="p5" type="P5"/>
      <xs:element name="p6" type="P6"/>
      <xs:element name="p7" type="P7"/>
      <xs:element name="p11" type="P11"/>
      <xs:element name="p12" type="P12"/>
      <xs:element name="p13" type="P13"/>
      <xs:element name="p14" type="P14"/>
      <xs:element name="arbiter" type="SA1"/>
   </xs:all>
</xs:complexType>
```

E. Setup for emulation

The next phase of the design methodology is to parse the generated XMLs and build required structure of platform and allocation of resources on it within the emulator application. The *DocumentBuilderFactory* and *DocumentBuilder* classes from the *javax.xml.parsers* package has been utilized in order to create *XML document* for further parsing. The *parse* method of the *DocumentBuilder* class returns *XML Document*, when we supply generated XML files.

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. The emulator program builds the matrix by extracting transactions between processes in PSDF model. Based on the matrix, the *PlaceTool* application [16] finds the optimal device allocation solution, given the platform specifics (the number of segments).

The emulation process is based on both PSDF and PSM. The PSDF model provides information about interaction between application processes with required data items and other useful parameters, while the PSM model represents the placement of each application process on different segments of the platform. Hence, the emulator program parses XML of both models to be later used for emulation. During the parsing process, the emulator extracts following information from the PSDF model:

- Number of application processes.
- Data transfers from each process.
- Ordering of transfers.
- Clock ticks to be consumed by each process while processing one package.

The emulator stores above information in temporary variables and arrays inside the program. Figure 7 show the PSDF model of the example application (discussed in section IV). For instance, the name attribute from one of the *element* from *P0*, that is, "P1_576_1_250" represents a transfer from process *P0*. The "_" character serves as the separator between the entities. The first entity "*P1*" represents the target process of this transfer; the second entity "576" is the number of data items to be transferred; the third entity "1" is the sequencing order and the last entity "250" is the number of clock ticks a process needs to consumed before sending each package.

Furthermore, the emulator extracts following information from the PSM model and stores in a number of variables and arrays inside the emulator, too:

- Number of segments in the platform.
- Number of border units based on platform geometry.
- Placement of application processes on different segments.
- ..

When the parsing process is finished for the XML of PSDF model, the emulator iterates in the previously populated arrays, instantiates the required ***FU***s and pass them necessary information. This necessary information contains number of data items to be transferred, destination processes, relative ordering, clock ticks a process needs to be consumed before sending a package and placement in the specific segment. The *contructor* method of the ***FU*** class analyzes the passed information to it and instantiates the required number of objects of *masters* and *slaves*, which later run as threads during emulation.

The emulator has been programmed in a way to exhibit the behavior of an 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. The method also counts how many clock ticks have been consumed for any particular transfer at different stages of the operation.

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.

During the parsing process of XML for the PSM model, the emulator application first looks for the *SegBus* platform instance in the XML document, analyzes its structure by counting how many segments and ***BU*** it contains as child nodes. It instantiates an object of platform instance, ***CA***, required number of ***BU***s and saves the references (discussed below). Later on, it looks for the elements in XML document, which represent segments. It analyzes the structure of each segment, instantiates one ***SA*** and required number of ***FU***s associated with any particular segment and pass the reference of segment to left/right ***BU***(s).

The emulator application maintains a number of lists each for different communication (***CA***, ***SA***, ***BU***, etc.) and application (***FU***) components. Whenever it encounters specific element in the XML document, it instantiates an object of the relevant class and adds it to corresponding list. For instance, if the emulator program finds an element representing a ***BU*** in the XML document, it instantiates an object of class *BU* by calling the constructor and passing the necessary values and adds the object to a list that holds only ***BU*** objects.

F. 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 is to transform the parallel behavior of hardware elements associated with platform into some special form that can be run on the microprocessor and exhibit the correct characteristics of the hardware devices. *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 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 (emulator engine and source files related to platform) run as threads during execution. Each class implements the *Runnable* interface from *java.lang* package by introducing a specific *run()* method. The method executes when emulation starts and performs dedicated functionality.

Class descriptions. In Figure 6, we illustrate the (simplified) class diagram of the emulator program with the most important classes and their relationships. We simplify the diagram by omitting class attributes and methods, for the purpose of clarity and to save the space.

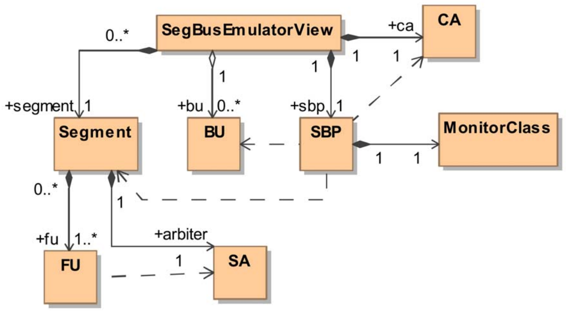


Figure 6. Class diagram of the emulator application.

Apart for the platform modeling classes, the most important ones are the *SegBusEmulatorView* and the *MonitorClass*, which both control the execution of the application.

The *SegBusEmulatorView* class performs the core functions of the emulator program. It contains methods to read the communication matrix and PSM model, and to set-up the emulation process. The class also contains a number of methods for parsing the XML schema.

The *AddToThreadPool()* method from *SegBusEmulatorView* 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 items in all lists that has been populated during parsing phase. We add objects (in the form of items) from all lists into the thread pool before emulation.

An *MonitorClass* object acts as a thread during execution. This class is responsible for analyzing the status flags for all ***FU***s and monitors the activity within other platform elements. When the object of this class detects no communication activity within the platform, it sets particular flag to inform the emulator application about the end of emulation.

During the execution of the emulator, all the 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 setup. In general, application processes communicates with each other at different time instant after performing specific computation on the supplied data. The emulator extracts execution sequence from the PSDF and forward them to relevant application processes. During emulator development, we skip some timing factors that are less important in estimating performance. For instance, we did'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 compute 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.

After we supply the XML schemes to the emulator, the tool parses the models, build the communication matrix, instantiates the threads corresponding to platform elements, supply particular value from communication matrix to each ***FU*** and starts the emulation process. Upon completion, the emulator returns results from 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 receive 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 increment 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 increment 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.

IV. Example using the Emulator program

We demonstrate our approach with an example of modeling a simplified stereo MP3 decoder [12] on the *SegBus*

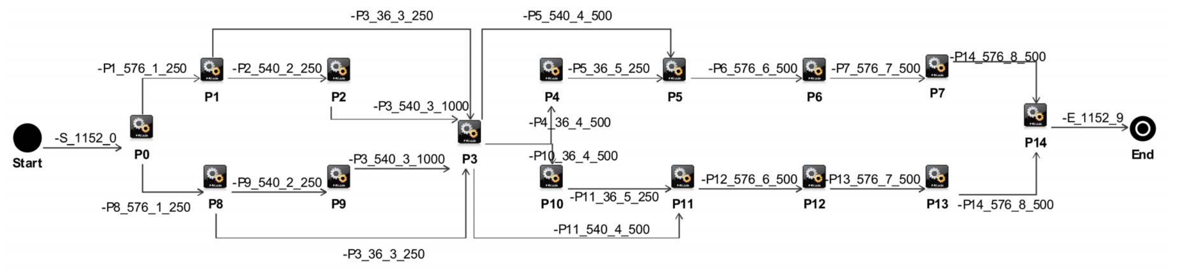
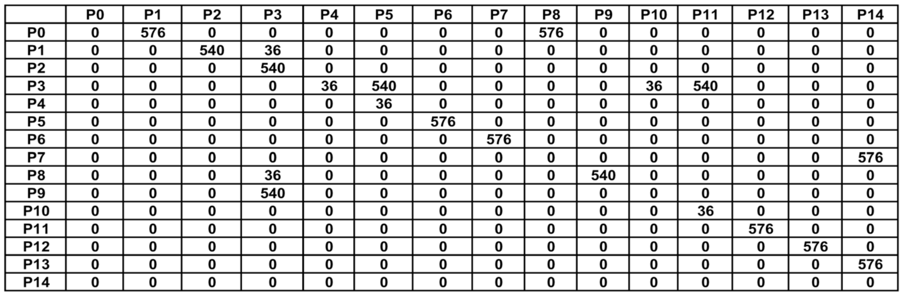


Figure 7. PSDF model of the MP3 decoder.



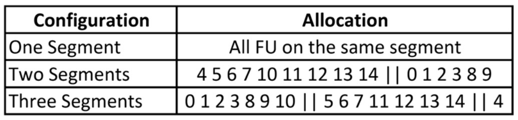
	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14
P0	0	576	0	0	0	0	0	0	576	0	0	0	0	0	0
P1	0	0	540	36	0	0	0	0	0	0	0	0	0	0	0
P2	0	0	0	540	0	0	0	0	0	0	0	0	0	0	0
P3	0	0	0	0	36	540	0	0	0	0	36	540	0	0	0
P4	0	0	0	0	0	36	0	0	0	0	0	0	0	0	0
P5	0	0	0	0	0	0	576	0	0	0	0	0	0	0	0
P6	0	0	0	0	0	0	0	576	0	0	0	0	0	0	0
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 8. The communication matrix for the example.

platform and associated emulation results. The modeling is done using DSL [11], and the application has already been partitioned up to a right granularity level [17].

Here, we model the example application as PSDF, and for PSM, we map application processes in three different platform configurations, using one, two and three segments, with linear topology in all configurations. The package size is set to 36 data items in each package. Figure 9 illustrates the allocation of application processes on each platform configuration, where segment borders are marked as '||'. Figure 7 shows the PSDF model of the example application. In brief, process *P0* represents frame decoding, *P1/P8* - scaling on the left/right channel, *P2/P9* - dequantizing left/right channel, etc. The communication matrix is generated from the PSDF model (see Figure 7) and is exposed in Figure 8. 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. The emulation results of 3 segments platform configuration are given below, where: '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 9. Allocation of processes on different platform configuration.

Three Segments configuration. In this configuration, processes (and the respective devices) are organized as shown in Figure 9. Following an execution of the emulator application, we reach the following results ("TCT" = total clock ticks).

```
P0, Start Time = 10989ps, End Time = 75307617ps
P8, Start Time = 75098826ps, End Time = 137758104ps
...
P7, Start Time = 401435564ps, End Time = 459394284ps
P14 received last package at  460435092ps

CA TCT = 54367
Execution time = 489792303ps @ 111.00MHz

BU12:
                Total input packages = 32,
                Total output packages = 32
    Package Received from Segment 1 = 32,
            Package Transfered to Segment 1 = 0
    Package Received from Segment 2 = 0,
            Package Transfered to Segment 2 = 32
    TCT = 2336

BU23:
                Total input packages = 2,
                Total output packages = 2
    Package Received from Segment 2 = 1,
            Package Transfered to Segment 2 = 1
    Package Received from Segment 3 = 1,
            Package Transfered to Segment 3 = 1
    TCT = 146

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:        TCT = 34764,
                Total intra-segment requests = 124,
                Total inter-segment requests = 32
                Execution Time = 382021596ps @ 91.00MHz

SA2:        TCT = 46031,
                Total intra-segment requests = 137,
                Total inter-segment requests = 0
                Execution Time = 469700324ps @ 98.00MHz

SA3:        TCT = 35884,
                Total intra-segment requests = 0,
                Total inter-segment requests = 1
                Execution Time = 403156740ps @ 89.01MHz
```

Calculation of the execution time. The total execution time is calculated when all ***FU***s finish their jobs (setting the respective "Process Status Flag"), all packages are transmitted to its relevant destination and grant signal of all arbiters are *clear*. All these events are somehow identified with either activities of the platform's ***SA***s or pf the ***CA***.

Consider the total time consumed by ***SA***x (in this case, $x \in \{1, 2, 3\}$) to finish all associated jobs as t_{SA_x}. t_{SA_x} is calculated by multiplying total clock ticks with the associated segment's clock period.

Then, 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}).

Emulation results. Figure 10 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. For instance, the process *P0* finishes the package transfers to process *P1* and *P8* at 75.30 μs.

Computed as defined above, in the given configuration, the estimated total execution time for the application is 489.79 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 that we obtain from the emulator are 95% accurate.

Next, we keep the same platform configuration, but we change the package size to 18 data items. The result shows an estimated execution time of 560.16 μs. The actual figure is 600.02 μs, giving us a precision of around 93%.

Further, we change the platform configuration by shifting process *P9* from segment 1 to segment 3. We keep the rest of the configuration stable, and the package size with 36 data items. The emulation estimated execution time of updated configuration is 540.4 μs, while the actual execution time is 570.12 μs, giving a precision of just below 95%.

Discussion. Based on our experiments, the accuracy of the emulator seems to be settled at around 95%. The errors are caused, mostly, by the not so accurate modeling of the timing figures of the ***BU*** to ***SA*** control communication, the synchronization between clock domains, the granting activity of the ***SA***s, etc.

However, firstly, these figures are very low (2 to 3 clock ticks), compared to the used size of a package (36 data units). Secondly, most of these operations do overlap with each other, or with the data transfers. A clear identification of such events is not possible, hence we should accept the resulting errors. It becomes though clear that, the higher the data package, the less impact of these figures should be observed in the estimation results of the emulator. This is due to the lower number of transfers, and hence, of synchronization, granting, etc. actions of the ***SA***s.

Due to one of the considerations described in section III-C, the timing information illustrated in figure 10 are not exact. This is due to the (variable) leading period of time during which each process awaits for data to be present at the input. However, as already mentioned, this does not have an impact on the overall application performance estimation, which, of course, includes such periods of time.

The tool helps us observe the communication bottlenecks expressed here as the time one package has to wait in one of the ***BU***s until it can be delivered to the next segment. The *useful period* (UP) of any given ***BU*** is expressed as the time (in clock ticks) required to load and then unload the data package, and it amounts to twice the size of a package. However, once a package is loaded, before unloading, the ***BU*** has to wait for a grant signal coming from the next segment - the *waiting period* (WP). As discussed and formalized in [15], WP is a non-deterministic value which may reach, at a maximum, the package size. An average value for WP ($\overline{WP}$) over the number of transfers executed by a certain ***BU*** can easily be computed given the data offered by the emulator (corresponding TCTs).

Considering the example at hand, for BU12 and BU23, we have the following values (clock ticks), respectively: UP12 = 2304, TCT12 = 2336, and $\overline{WP}$12 = 1; UP23 = 144, TCT23 = 146, $\overline{WP}$23 = 1.

Further, the Figure 11 illustrates the activity graph of 3 segment, linear topology configuration with different package sizes (18 and 36 data items).

V. Conclusions

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

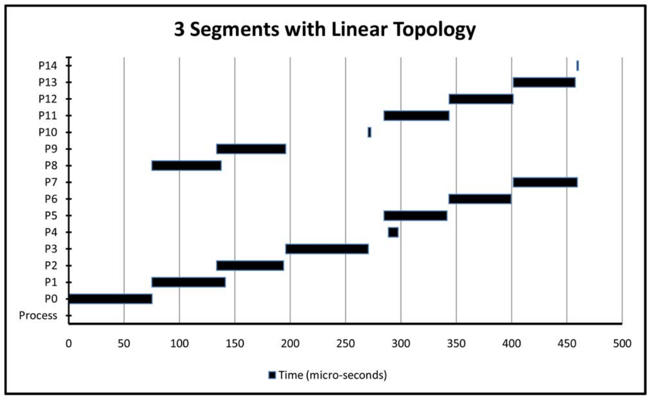


Figure 10. Progress on time of each application process in 3 segment, linear topology with package size of 36 data items configuration.

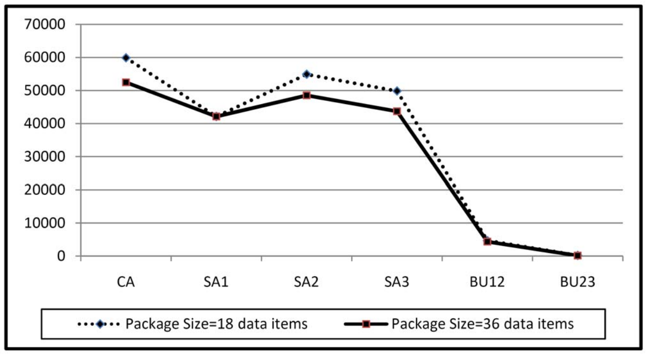


Figure 11. Activity graph of different platform elements in 3 Segment and linear topology configuration for 18 and 36 bit package sizes.

The emulation-based solution enables us to analyze any platform configuration with respect to performance figures. Based on emulation results, it's the job of the designer to decide which configuration would be best suited for the final implementation. Such decisions in the 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 [9]. The granularity level of application components can also be balanced in order to eliminate the traffic congestion located at certain ***BU***s, that will further improve the overall performance. Thus, 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.

Future work will necessarily address more application models to be tested on the emulator platform. In addition, extended support is expected to come in the form of arbiter code generation, for the implementation of the application schedules.

Acknowledgments

The present work is supported by the DOMES project funded by the Academy of Finland, project number 123518/2008.

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.* Jan. 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] K. Latif, M. Niazi, T. Seceleanu, H. Tenhunen, S. Sezer *Application Development Flow for On-Chip Distributed Architectures.* In Proceedings of the 21^{st} IEEE International System-on-Chip Conference (SOCC), 2008, pp. 163 - 168.

[10] P. Liu et. al. *A NoC Emulation/Verification Framework.* In Proceedings of 6^{th} International Conference on Information Technology: New Generations, 2009, pp. 859-864.

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

[12] 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.

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

[14] E. A. Lee and D. G. Messerschmitt. *Extended Synchronous Dataflow.* IEEE proceedings, September 1987.

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

[16] 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.

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