Retargetable Generation of TLM Bus Interfaces for MP-SoC Platforms

Andreas Wieferink, Rainer Leupers,
Gerd Ascheid, Heinrich Meyr
Integrated Signal Processing Systems
RWTH Aachen University, Germany
http://www.iss.rwth-aachen.de
wieferink@iss.rwth-aachen.de

Tom Michiels,
Achim Nohl, Tim Kogel
CoWare, Inc.
CA, USA
http://www.coware.com
tom.michiels@coware.com

ABSTRACT

In order to meet flexibility, performance and energy efficiency constraints, future SoC (System-on-Chip) designs will contain an increasing number of heterogeneous processor cores combined with a complex communication architecture. Optimal platforms are obtained by customizing both computation and communication modules to the application's needs. In our design flow both kinds of SoC modules are automatically derived from abstract specifications. This work focuses on generating the communication adaptors, which are tailored to the processor as well as to the bus side. For early system simulation, the adaptors are capable of bridging an abstraction gap by implementing a bus interface state machine. The generated processor cores, adaptors and bus nodes are applied in the exemplary design of a JPEG decoding platform.

Categories and Subject Descriptors: B.8.2 [Performance and Reliability] Performance Analysis and Design Aids ; C.1.2 [Processor Architectures] Multiple Data Stream Architectures (Multiprocessors) - *Interconnection architectures* ; I.6.7 [Simulation and Modeling] Simulation Support Systems - *Environments*

General Terms: Design, Performance, Measurement

Keywords: MP-SoC, Architecture Exploration, Retargetability, TLM, SystemC, Simulation

1. INTRODUCTION

The ever increasing complexity of modern electronic devices together with the ever shrinking time-to-market and product lifetimes pose enormous SoC design challenges to meet flexibility, performance and energy efficiency constraints with a good design efficiency.

Programmable platforms are the best way of fulfilling today's and tomorrow's flexibility constraints, and tailoring them specifically for the target application domain is the key to meet the performance demands with a good energy efficiency. Designing these platforms requires a systematic methodology and suitable tooling for obtaining optimal results in a reasonable design time.

In order to identify a suitable platform for a specific application or application domain, design space exploration on a higher level of abstraction is mandatory. The better the tool support for a thorough investigation of the full design space, the more likely optimal design decisions are made in early stages of the design flow. This avoids the high costs of long redesign cycles or the risk of placing suboptimal products on the market.

CODES+ISSS'05, Sept. 19–21, 2005, Jersey City, New Jersey, USA.

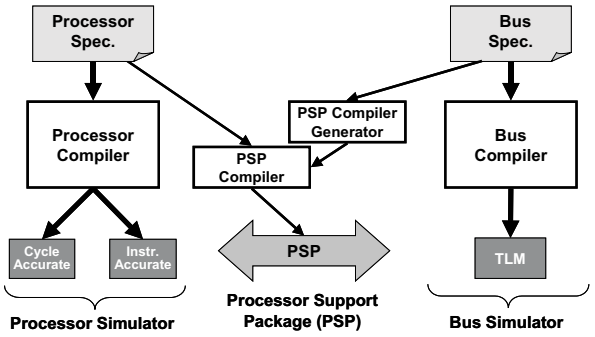

Figure 1: Simulator Generation Flow

The largest design space is opened by heterogeneous platforms with application specific instruction set processor (ASIP) cores on the computation side together with a highly optimizable interconnect structure (*Network-on-Chip*, *NoC*) on the communication side. For best possible performance and energy efficiency, in this work computation as well as communication modules can freely be specified instead of instantiating predefined IP (Intellectual Property) blocks.

As shown on the left hand side of Fig. 1, a *processor compiler* tool automatically generates a processor simulator on multiple possible abstraction levels, taking an abstract textual processor specification as input. Besides the Instruction Set Simulator (ISS), also C compiler, assembler and linker are generated to be able to run the target application on the respective processor. Analogously, as shown on the right hand side of Fig. 1, a *bus compiler* generates a simulator for the communication modules. These bus simulators apply the Transaction Level Modeling (TLM) communication paradigm, which allows very efficient but still fully cycle accurate simulation.

With P being the number of different processor cores and B being the number of different bus modules, in a heterogeneous Multi-Processor SoC (MP-SoC) generally $P > B > 1$ holds true. The processor compiler and the bus compiler take care of generating the P processor and the B bus simulators. But in order to do meaningful overall SoC exploration, these simulators have to be coupled, which at maximum leads to $P \cdot B$ combinations. During the design flow the adaptors often also have to bridge a gap in model abstraction, which calls for implementing a bus interface state machine in the adaptor. Thus, manually developing these $P \cdot B$ couplings is much too tedious and error prone.

This paper describes a methodology and the tooling for automatically generating these $P \cdot B$ couplings which are called Processor Support Packages (PSP). As shown in Fig. 1, this is done in a two-step approach. A PSP compiler generator generates a set of B PSP compilers, one for each bus model. A PSP compiler, in turn, generates the simulator coupling for any of the P processors to a specific bus. In the recent years, some PSP compilers for very common bus protocols have been developed manually. This already enabled automatic retargetability to the processor side. The main focus of this work is on generating a bus interface state machine, which is the main task of the PSP compiler generator. This enables automatic retargetability also the the bus side.

The rest of the paper is organized as follows: After discussing the related work in section 2, we introduce the processor compiler and the bus compiler in more detail, focusing on their integration into the design flow. The main body of the paper in section 4 presents the concept and implementation of the PSP generation chain. A case study of a JPEG decoding system is the topic of section 5. Finally, section 6 concludes this work and gives a short outlook on future research topics.

2. RELATED WORK

It is commonly agreed that RTL models are not suitable anymore to simulate or even explore the system behavior of today's and tomorrow's complex heterogeneous MP-SoC platforms. Virtually all recent work addressing the increasing MP-SoC design complexity is based on leveraging the abstraction level considered by the system designer.

Communication models on multiple levels of abstraction for early processor integration have been proposed since a long time to overcome the simulation performance and modeling efficiency bottleneck [1, 2]. This principle is now leveraged by the TLM paradigm [3] by providing standardized, system level bus-interfaces with different levels of abstraction [4, 5, 6, 7]. Based on the SystemC language as the emerging EDA standard for system-level design, SystemC TLM communication models are supported by a new generation of Electronic System Level (ESL) SoC design tools [8, 9]. However, techniques to automatically generate customized cycle accurate TLM bus models from an abstract formal specification are not published yet.

In order to efficiently design customized processor cores, abstract Architecture Description Languages like EXPRESSION [10], ISDL [11], LISA [12], MIMOLA [13] and nML [14] have become popular. Basically, those environments which generate fast simulators with good system integration capabilities could have been used for this work. We chose the LISA platform, because additionally, it provides support for C-Compiler and full RTL generation.

For integrating processor cores and communication modules into the final MP-SoC platform, most approaches follow the component-based design principle [15, 16]. In this bottom-up approach, relatively coarse grained fixed IP blocks are quickly combined to complex MP-SoCs, but this is done at the expense of flexibility losses. Even if the NoC topology is modifiable, the limited set of communication modules, protocol IP, and especially processor cores significantly narrows the design space.

Our approach is capable of solving this shortcoming by providing *modifiable* communication and processor IP. An optimization in an abstract model is relatively quickly propagated to the models and simulators on the lower levels of abstraction. The TLM communication adaptors dealt with in this work are generated using the same data base which already served for building up the involved SoC modules.

However, the main focus of this work is to automatically generate the necessary couplings on multiple abstraction levels for an efficient successive top-down refinement flow [17], which bears the best potential for designing optimal platforms.

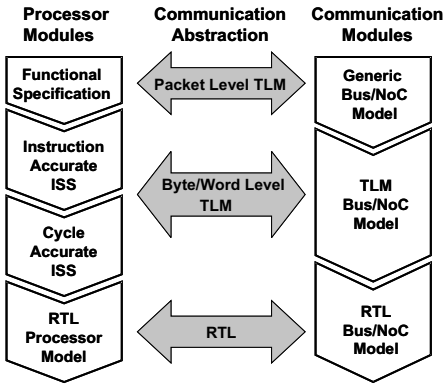

Figure 2: Abstraction Levels

3. MP-SOC DESIGN FLOW WITH RETARGETABLE SIMULATORS

For the top-down design of very complex MP-SoC platforms, models on multiple levels of abstraction are used in a successive refinement flow. On every level of abstraction, important design decisions are made to guide the implementation or the IP selection on the lower abstraction levels. In less complex designs, of course, some levels can be skipped. In Fig. 2, the refinement flow is outlined separately for the programmable computation modules and the communication modules.

During this flow, the abstract processor and bus models are refined to full cycle accuracy. This is a good starting point for directly generating optimized synthesizable RTL models automatically as well. However, these tools are far beyond the scope of this paper. In this section, only retargetable simulators for abstraction levels above RTL are considered.

3.1 Initial Design Space Exploration

For very complex MP-SoCs, design space exploration starts with an executable functional specification of the processor modules, which communicate over a generic, parameterizable Network-on-Chip (NoC) model. On this level, the temporal and spacial task mapping to the processor modules is done on the computation side, while a suitable interconnect topology and adequate bus or network engine properties are detected on the communication side. For these examinations, no generated simulators are necessary yet. Generic models are parameterized or equipped with the target application code, executing on the simulation host with annotated timing budgets. Communication is modeled very efficiently on this level by treating a whole packet or burst transfer as a single event instead of separately simulating every word transfer or clock cycle (Packet Level TLM).

The next refinement step towards fully cycle accurate system simulation uses a new set of models which actually simulates the system model instruction-by-instruction, word-by-word, or already cycle-by-cycle instead of annotating respective budgets.

3.2 Generated Instruction Set Simulators

In order to get reliable information concerning the performance of the application software, its execution on the target processor platform must be simulated. Also, such an ISS is needed for early starting the implementation of the final application software. Since it is very tedious and error prone to develop and maintain the ISS as well as the software tools C-compiler, assembler and linker manually, abstract Architecture Description Languages (ADLs) have become popular. They allow modeling a processor architecture on a high level of abstraction; the respective tools are generated automatically.

The LISA [12] ADL allows modeling processor architectures on two main levels of abstraction: Instruction Accurate (IA) and Cy-

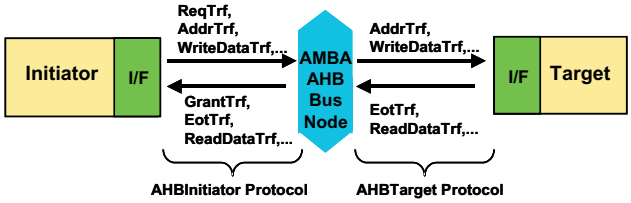

Figure 3: Cycle Accurate Bus Communication Modeling

cle Accurate (CA). The simulator generated from an IA model is already aware of the full instruction set. It executes the application code instruction-by-instruction, but without interleaving their execution, as it is typically done in the real processor's pipeline. Whenever a memory access is necessary, in an IA simulator the *ideal* or *functional* memory interface is accessed. Both interfaces basically offer two methods to the processor core: one for read and one for write access. To enable easy processor modeling on this level, these interface calls are expected to return successfully in any case. If such an IA simulator is embedded into a TLM system simulation, a call to the functional memory interface typically blocks the respective processor simulator a number of cycles until the access is fully completed.

An IA simulator is very suitable to do instruction set exploration and to start implementation of the embedded software. However, for a fully cycle accurate simulator, the ADL model has to be refined to the CA abstraction level. In this manual process, the designer adds a pipeline to the model and assigns the atomic operations of the instructions to a suitable pipeline stage. Furthermore, activation chains are defined according to the processors execution scheme. In a further step, the memory accesses are refined to access the *cycle accurate* memory interface. Since memory or bus accesses generally take more than one cycle, at least portions of this latency should be hidden in the processor's pipeline. The cycle accurate memory interface offers the possibility to invoke the different phases of a memory or bus access from within different pipeline stages. As long as a read data word is not yet available in the respective pipeline stage, for example, the processor simulator dynamically reacts on that by inserting stall cycles.

3.3 Generated Bus Simulators

On the communication side, there are basically two main degrees of freedom. The SoC designer can differentiate or tweak the communication part of the design by two means. First by changing the implementation of the communication modules or nodes, and second, by changing their interconnect topology. Already on the abstraction level of TLM models, manually exploiting this freedom becomes very tedious and error prone. Thus, again tools support the designer by reducing his work to the more creative part. The interconnect topology is mostly specified applying graphical user interfaces [8, 9, 18], while the customized creation of the communication modules most efficiently is done using a formalized abstract textual description. The generated TLM bus nodes allow cycle accurate communication modeling by exchanging small data packets, called *transfers*, with their environment every cycle during an ongoing transaction.

The nodes can be connected to active initiator modules, reactive target modules, or additional communication nodes to form crossbars or other NoC interconnect hierarchies. On every connection, information exchange happens according to a specific communication protocol. As shown in the example of an *AMBA AHB* [19] node in Fig. 3, the protocols for the initiator side and the target side of the *AHB* node at TLM can exchange a limited set of transfers in fixed directions. This transfer exchange replaces the detailed pin wiggling being simulated on RTL. Every transfer carries a set of attributes, which is information associated with this transfer. An

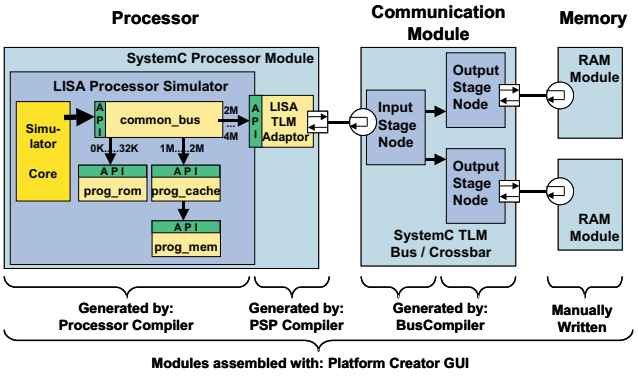

Figure 4: Simulator Structure

address transfer (*addrTrf*), for example, carries the address and an access direction flag.

The abstract textual bus model is a formal description and consists of three parts. First, a generic protocol section defines all transfers being part of any protocol of a whole bus family, with the maximum possible set and width of their attributes. Second, a protocol definition section introduces all protocols with that subset of generic transfers and those transfer attributes they comprise, respectively. And third, a node definition section formally defines the pipelines and state machines inside every node of a bus family. The generated bus simulator then allows sending and receiving the protocol transfers only in valid time slots. Simple attached target modules very efficiently declare themselves sensitive to events occurring in the bus simulator instead of modeling their own state machine.

4. BUS INTERFACE RETARGETABILITY

This section presents the new technique to automatically couple the generated simulators for early TLM system simulation.

4.1 Structure of the Platform Simulator

The system simulator structure is outlined in Fig. 4. Not only the processor simulator core, also the internal processor memory hierarchy is generated by the processor compiler according to the ADL specification. During simulation, the memory requests are directed to the *ideal*, the *functional* or the *cycle accurate* API [1] of the memory or the bus modules. Alternatively, if the respective memory location is modeled outside the processor simulator, the request is directed to an adaptor.

Automatically generating and instantiating these adaptors depends on the respective processor as well as the bus model, and it is the main focus of this work. Our approach is presented in more detail in the following paragraphs.

The bus node implementations are generated by the bus compiler, together with a definition file that provides a GUI[2] tool with the necessary information how the nodes can be parameterized and interconnected. This enables modeling complex NoC topologies at fully cycle accurate level. However, details about these capabilities are outside the scope of this paper.

The TLM memory modules and peripherals normally are relatively simple and are not generated automatically.

4.2 PSP Generation

As indicated in Fig. 4, the main task of the PSP compiler is to generate the SystemC processor wrapper module by instantiating and connecting the LISA processor simulator and one LISA TLM adaptor per TLM port. Additionally, the wrapper is equipped with the necessary code to enable powerful analysis and debugging ca-

[1] API = Application Programming Interface
[2] GUI = Graphical User Interface

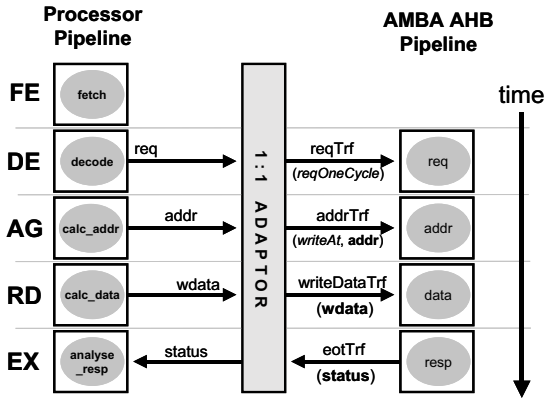

Figure 5: Pipeline Example: $STR\ @0x1000$**,** $R[0]$

pabilities for the processor core in its system environment. The PSP compiler also generates a module definition file for the GUI.

However, the TLM adaptor class itself is a fixed building block of every PSP compiler. Due to the generic LISA memory API, the adaptor does not depend on the processor side. A LISA processor core is tailored for a specific bus or memory only by distributing the generic *cycle accurate* function calls suitably over the pipeline and by issuing commands for special communication features like sub-block-access or read-modify-write.

The adaptor maps the generic LISA memory API calls to the bus specific API of the respective bus simulator. Thus, the main challenge of the PSP compiler generator is to build up these TLM adaptors for arbitrary bus protocols.

4.3 The Adaptor Concept

The task of the adaptor is to couple a processor model with a bus model, independent of whether an IA or a CA processor simulator is involved. [3]

On fully cycle accurate level, if both sides are driven by the same system clock, a simple adaptor maps the API calls from the processor pipeline one-to-one to the bus pipeline. As shown in Fig. 5 on an exemplary data memory store instruction, an optimally tailored processor pipeline can hide multiple cycles of bus latency. In the *decode* stage, the processor already requests access to the data memory, without already knowing the address or even the write data. If bus access has been granted, then the *address generation* stage calculates the write address and forwards it to the adaptor, which in turn sends a respective *addrTrf* transfer to the bus. The processor determines the write data one cycle later in the *read* stage, which is exactly the time it is expected by the bus pipeline.

Fully integrating the bus interface protocol into the processor pipeline is a very tedious and error prone task and prevents the designer from early system simulation. What is necessary is an adaptor that implements an own bus interface state machine. By that, *cycle accurate* API calls that are not optimally distributed over the pipeline yet, or even abstract calls to the *functional* API lead to a well working communication. Also, independent clocks for the processor core and the bus node require a more intelligent adaptor.

The bus interface state machine automatically buffers data arriving too early from one side, and vice versa, stalls a pipeline if a data item arrives too late. An IA processor model applying the *functional* API is blocked until the bus protocol has completely finished the requested transaction.

Either this bus interface state machine later on is synthesized to an RTL module, or, more efficiently, the timing statistics of the TLM adaptor guide the designer in fully integrating the bus protocol manually into the processor pipeline.

[3] Or, to be more precise, if the *functional* or the *cycle accurate* memory API is applied.

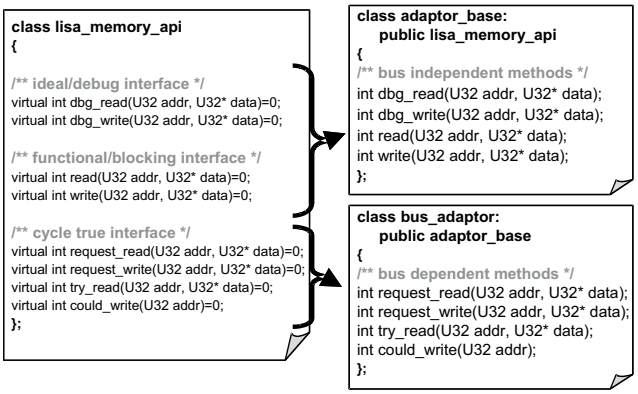

Figure 6: Implementing the ADL API

4.4 Automatically Generating the Adaptors

Basically, the TLM adaptor must provide an implementation for the *ideal*, the *functional* and the *cycle accurate* parts of the generic ADL API. These are the methods the generated processor simulators need to access. Fig. 6 displays a simplified version of the respective API methods.

The implementation is done in two parts. In a first API class derivation, the bus independent methods are implemented. Since the ideal interface basically bypasses the bus, its implementation is done once, using the generic access methods of the simulation environment. The functional interface itself also is implemented once for all generated bus protocols. For the bus specific parts, it uses the methods of the cycle accurate interface, which are virtually defined in the base class and thus accessible [20].

In the second implementation part, the bus specific functions are implemented in a further class derivation. Basically, the functions define how to feed data into the bus interface state machine and how to get the results back. This C++ class is generated automatically from the abstract bus definition. It makes sense to keep the flexibility of also implementing alternative C++ or SystemC APIs for accessing a bus node, e.g. the standardized OCP abstraction levels TL2 and TL1 [4] or the Programmer's View (PV) and Architect's View (AV) [6] APIs. Thus, the properties of the bus specific API are defined in a special section of the bus specification: the interface specification.

4.5 Bus Interface Specification

The same formal syntax which is used to specify the bus nodes and their protocols also is used to specify the C/C++ interface functions to be generated. One interface specification defines adaptors for all initiator protocols of the whole bus family.

```
busInterface LISA_AMBA,
  connect = [AHBInitiator,AHBLiteInitiator,APBInitiator]
{
  stateMachine, ...
  { ...
    sequence seq_single_read, ...
    catch burst_continue, ...
  };
  ...
  function request_write, ...
  { ...
  };
};
```

This example interface *LISA_AMBA* generates adaptors for the protocols *AHBInitiator*, *AHBLiteInitiator* and *APBInitiator* of the *AMBA 2.0* [19] bus family. Basically it consists of two parts. First, a characterization of the state machine. Here, the state sequences to traverse as well as code to execute additionally to the default behavior of a state can be defined. The second part specifies the API functions the processor calls in order to feed data into the state machine or get information back. The following fragment defines the

signature of the `request_write()` method and specifies how it feeds a write burst into the state machine.

```
function request_write,
  returnType = int
{
  parameter addr,
    type = U32;
  parameter data,
    type = U32*,
    value = 0;
  parameter n,
    type = int,
    value = 1;
section feed_data
{
  buffer,
    type = allocateNew,
    size = n,
    behaviorFailed =
        [ return = -1; ];
  ...
```

After having declared the function name, the return type and the function parameters, the buffer is defined the implementation is working on. This function allocates a new buffer with one entry per burst item.

```
behavior = [
  buffer[current].reqTrf.reqMode = reqUntilUnreq;
  buffer[current].addrTrf.type = writeAtAddress;
  buffer[current].addrTrf.address = addr + 4*current;
  buffer[current].writeDataTrf.writeData = data[current];
  buffer[current].sequence =
        if (n==1)                then  seq_single_write
        elseif ( current == 0)   then  seq_first_write
        elseif ( current == n-1) then  seq_last_write
        else                           seq_burst_write;
  return = 0;  ]  }; };
```

In the corresponding behavior section, the buffer is filled for every burst item. This function already provides attribute values for the *AMBA 2.0* transfers *reqTrf*, *addTrf* and *writeDataTrf*, which are stored in the buffer. If the API does not specify properties, fixed default values can be set here. Additionally, a state sequence is selected the state machine has to use the succeeding cycles in order to process the request. The state sequences are defined in the central section of the interface definition.

```
stateMachine,
  extraStates = [ burstContState, finishState  ]
{
  sequence seq_single_write,
    value = [ reqTrf , addrTrf , writeDataTrf ,
              eotTrf , unreqTrf , finishState ];
  sequence seq_first_write,
      value = [ reqTrf , addrTrf , writeDataTrf ,
                finishState ];
  sequence seq_burst_write,
      value = [ burstContState ; addrTrf ,
                writeDataTrf , finishState ];
  sequence seq_last_write,
      value = [ burstContState ; addrTrf , writeDataTrf ,
                eotTrf , unreqTrf , finishState ];
};
```

The first state sequence is to be used for a single word write access. The other three sequences are for the first burst item, the middle burst items, or the last item of a write burst, respectively.

Every bus transfer has an associated state. If a state is entered during simulation, then the respective transfer is sent to or received from the bus. The attribute values are taken from or written to the buffer, and the next state according to the sequence definition is entered. If a transfer could not yet be exchanged with the bus, then the generated adaptor will try it again the next cycle.

In the right hand side of Fig. 5, a simplified write sequence is depicted. Here, the state sequence to traverse is *reqTrf*, *addrTrf*, *writeDataTrf*, *eotTrf*. Independent when exactly the processor model delivers the write address or the write data, the respective transfers will be exchanged in the correct time slots. Information arriving too early will be buffered in the adaptor, information arriving too late will cause a stall of the bus pipeline.

Additional states can be defined that do not have an associated transfer, and thus do not forward automatically. In the example, the *burstContState* is the state burst items have as long as they are waiting for their predecessor having finished the first access phase. In order to define the behavior of these additional states, or to alter the default behavior of the transfer states, catch statements can be inserted into the state machine.

```
catch burst_continue,
  state = addrTrf,
  condition = (buffer[next].state == burstContState),
  behavior = [  buffer[next].state = addrTrf; ];
```

Since the state sequences for all items of a burst except the first start with the passive *burstContState*, they are explicitly forwarded to the *addrTrf* state as soon as the preceding burst item succeeded sending the *addrTrf*. In a similar manner, a burst can completely be re-sent if the target reported a *retry* in the *eotTrf* response transfer.

After the bus interface state machine went through the state sequences, the result can be returned. The implementation of the `try_read()` function, for example, works on a buffer of type *searchFinished* and checks if the address matches and if the corresponding state sequences have succeeded. In case of success the read data is copied into the target array.

Similarly, by using a buffer of type *searchProcessing*, an API function can supplement an already running write transaction with the write data. The adaptor will not sent the *writeDataTrf* until the write data has been provided.

4.6 Advantages

This single condensed interface specification [4] is suitable to customize the generation of the bus interface for all initiator protocols of a bus family. Given information can be verified automatically, e.g. if the state sequences are a valid path through the state graph of the respective bus nodes. Information excessive for a specific bus protocol is skipped. In *AMBA 2.0*, for example, all information concerning the bus request phase is used for generating the *AHBInitiator* interface, but is ignored for generating the *AHBLiteInitiator* and the *APBInitiator* bus interfaces.

Manually implementing these state machines specifically for every bus protocol would be a very tedious and error prone task. Using the new approach, working adaptors are obtained very quickly. Additional features like bursts and sub-block access are added successively by refining the initial specification. Further annotations indicate the applicability of specific optimizations. Using them, the simulation speed of platforms with generated and handwritten adaptors is roughly the same [17].

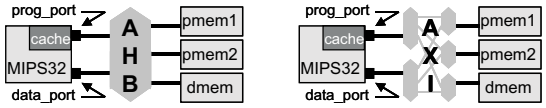

Figure 7: Platform Topologies with AHB resp. AXI

5. CASE STUDY

The generators have been used to build the components of a JPEG decoding platform. Several incarnations of the MIPS32 architecture access the program and data memory modules over a generated AMBA bus architecture. For the experiments, an AMBA AHB bus node and an AMBA AXI multistage matrix have been applied, respectively (Fig. 7).

The application is the same for all system simulations: A JPEG decoding algorithm, as it is freely available from the JPEG group's web page [22]. Our indicator for a platform's performance is the number of cycles it takes to decode a small 150x100 sized bitmap.

In this section, several platform alternatives are evaluated.

5.1 Processor Variants

The initial MIPS32 4K core [23] has a 5-stage pipeline: *Instruction Fetch* (I), *Execution* (E), *Memory Fetch* (M), *Align/Accumulate* (A) and *Writeback* (W). The instruction fetch takes place between the I and E stage; data accesses occur between stage M and A. Thus, in both cases only one cycle of memory access latency can

[4] 500...800 lines of code for the AMBA 2.0 [19] and AMBA AXI [21] bus families

program cache size	MIPS32 AHB	pipeline stages / data mem cycles IA	5/1	6/2	7/3	8/4	5/1+	6/2+	7/3+	8/4+
	no cache	23.9	19.5	17.7	16.5	16.1	17.7	15.9	15.0	15.0
	1x2 lines	23.1	21.8	21.1	20.6	20.5	20.2	19.5	19.4	19.7
	1x32 lines	16.0	15.2	14.4	14.0	13.8	13.5	12.8	12.8	13.3
	2x256 lines	14.5	13.8	13.0	12.6	12.4	12.1	11.3	11.3	11.9

Table 1: MIPS Performance with AMBA AHB Bus (MCycles/Image)

program cache size	MIPS32 AXI	pipeline stages / data mem cycles IA	5/1	6/2	7/3	8/4
	no cache	29.9	21.5	19.5	19.5	18.2
	1x2 lines	30.5	22.7	21.8	21.5	21.4
	1x32 lines	16.4	16.4	15.4	14.9	14.7
	2x256 lines	13.5	15.0	13.9	13.4	13.2

Table 2: MIPS Performance with AMBA AXI

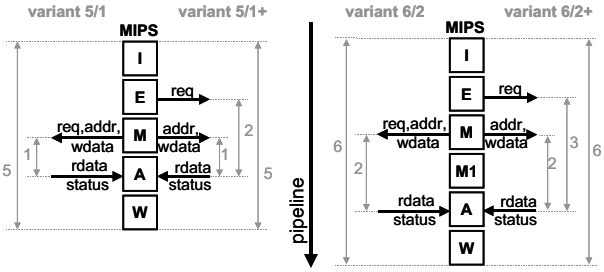

Figure 8: Data Memory Access from the MIPS Pipeline

be hidden in the pipeline. Since an AMBA AHB or AXI bus features a latency of 3 cycles or more, a lot of stall cycles are executed when directly connecting such a bus to the MIPS pipeline.

In order to reduce the instruction fetch latency, a program cache is applied. In case of a cache hit, the full latency can be hidden in the pipeline, otherwise a burst line containing 16 words needs to be fetched over the bus. To avoid cache consistency problems in multi-processor systems, the data memory is not cached. Instead, additional stages have been inserted into the MIPS pipeline. As shown in Fig. 8 for the initial 5-stage and the 6-stage pipeline (variant 5/1 respective 6/2), additional access latency cycles can be hidden this way. The MIPS bypass mechanism takes care that no data hazards occur. Only if necessary, the pipeline is stalled to ensure data consistency. The assembly application does not need to be modified because of this.

A further optimization is to invoke the bus request already one stage earlier (see variants 5/1+ and 6/2+). Many buses can already start the arbitration phase without knowing the address yet.

5.2 Results

In this section, several implementation alternatives are compared against each other by integrating the automatically generated modules into a SoC simulation. In the rows of Table 1 and 2, different program cache sizes have been evaluated, which optimize the instruction fetch. The columns contain different pipeline variants, which influence the data access. The values are million cycles consumed to decode the small bitmap.

In the first column of both tables, the simulation results of an abstract instruction accurate (IA) processor model are given, which does not model the processor pipeline at all. Its cycle count normally is higher since the IA simulator blocks until a whole cache line is loaded. In contrast, the cycle accurate models as well as the real processor already continue execution as soon as the questionable words are available in the cache.

The next four columns of Table 1 show the performance of the MIPS processor variants without early bus request being connected to the AMBA AHB bus node. The optimal pipeline length is a trade-off between hiding memory access delay and avoiding data hazards. Obviously, the 8-stage pipeline does not significantly improve performance any more. In the right hand side columns of Table 1, the MIPS processor variants with early bus request are evaluated. The performance is even better compared to an elongated pipeline. This is the case because no additional data hazards occur due to the constant pipeline length. As can be seen on the 8/4+ variant, making the pipeline too long results in worse results.

Without modifying the processor models at all, the generator also can build a coupling to the AMBA AXI multistage communication infrastructure (Table 2). It does not feature a request phase, thus the early request MIPS versions do not have an advantage against the standard versions any more. In case of our single processor experiments, the performance is even worse compared to the AHB nodes. The AXI bus modules are optimized concerning throughput, but not concerning latency. When moving to a multi-processor platform, an AXI infrastructure scales much better.

6. SUMMARY

In order to exploit the enormous potential of customized MPSoC modules, a suitable tooling needs to unburden the designer from the tedious and error prone tasks during design space exploration. This paper presented a technique to automatically integrate application specific processor cores and customized bus nodes into the SoC. The adaptors allow bridging an abstraction gap to enable early system simulation on several abstraction levels. The bus interfaces generated in this work cover abstraction levels down to fully cycle accurate TLM. Future work will reuse the condensed interface specification to also generate a coupling on RT-Level.

7. REFERENCES

[1] K. Hines, G. Borriello. Dynamic communication models in embedded system co-simulation. In *Proc. of the Design Automation Conference (DAC)*, 1997.

[2] J.A. Rowson and A. Sangiovanni-Vincentelli. Interface-Based Design. In *Proc. of the Design Automation Conference (DAC)*, 1997.

[3] T. Grötker, S. Liao, G. Martin, S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[4] A. Haverinen, M. Leclercq, N. Weyrich, D. Wingard. *White Paper for SystemC based SoC Communication Modeling for the OCP Protocol, http://www.ocpip.org/data/systemc.pdf*, 2003.

[5] A. Cochrane, C. Lennard, K. Topping et al. *AMBA AHB Cycle Level Interface (AHB CLI) Specification*, 2003.

[6] B. Vanthournout, S. Goossens, T. Kogel. Developing Transaction-level Models in SystemC. White Paper, CoWare Inc., August 2004. www.coware.com.

[7] SystemC TLM Library. *Open SystemC Initiative (OSCI), http://www.systemc.org*.

[8] CoCentric System Studio. *Synopsys, http://www.synopsys.com*.

[9] ConvergenSC. *CoWare, http://www.coware.com*.

[10] P. Mishra, P. Grun, N. Dutt, and A. Nicolau. Processor-memory co-exploration driven by an architectural description language. In *Intl. Conf. on VLSI Design*, 2001.

[11] G. Hadjiyiannis, S. Devadas. Techniques for Accurate Performance Evaluation in Architecture Exploration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2003.

[12] A. Hoffmann, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002.

[13] R. Leupers. HDL-based Modeling of Embedded Processor Behavior for Retargetable Compilation. In *Proc. of the Int. Symposium on System Synthesis (ISSS)*, Sep. 1998.

[14] A. Fauth and J. Van Praet and M. Freericks. Describing Instruction Set Processors Using nML. In *Proc. of the European Design and Test Conference (ED&TC)*, Mar. 1995.

[15] P. Paulin P. Magarshack. System-on-chip beyond the nanometer wall. In *Proc. of the Design Automation Conference (DAC)*, 2003.

[16] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. Jerraya, M. Diaz-Nava. Component-Based Design Approach for Multicore SoCs. In *Proc. of the Design Automation Conference (DAC)*, 2002.

[17] A. Wieferink, T. Kogel, G. Braun, A. Nohl, R. Leupers, G. Ascheid, and H. Meyr. A System Level Processor/Communication Co-Exploration Methodology for Multi-Processor System-on-Chip Platforms. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Paris, France, Feb 2004.

[18] Magillem. *Prosilog, http://www.prosilog.com*.

[19] AMBA Specification, Rev. 2.0. *ARM, http://www.arm.com*.

[20] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.

[21] AMBA AXI Protocol, v1.0, Specification. *ARM, http://www.arm.com*.

[22] Official JPEG homepage. *http://www.jpeg.org*.

[23] MIPS32 4K Processor Core Family. *MIPS, http://www.mips.com*.