Embedded Software Development on Top of Transaction-Level Models

Wolfgang Klingauf, Robert Günzel, Christian Schröder

Technical University of Braunschweig, Dept. E.I.S.
Mühlenpfordtstr. 23, 38106 Braunschweig, Germany
{ klingauf, guenzel, schroeder } @ eis.cs.tu-bs.de

ABSTRACT

Early embedded SW development with transaction-level models has been broadly promoted to improve SoC design productivity. But the proposed APIs only provide low-level read/write operations via a TLM interconnect. SW developers have to implement platform-specific communication procedures and handshake protocols to access HW functions, which requires a deep understanding of both the HW interfaces and the TLM fabric used. In this paper, we propose our concept of *hardware procedure calls* (HPC) with which HW services are provided to SW processes as remote methods *on top* of transaction-level communication. To this end, a lightweight HPC protocol is presented, and we propose a method to generate the infrastructure for HPC communication from a straightforward input description. Our experiments show that with HPC, embedded SW development is considerably made easier, and that also the effort to create the transaction-level model itself is reduced.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Design Tools and Techniques – *Modules and interfaces; Object-oriented design methods*

General Terms: Design, Languages

Keywords: Hardware-Software Communication, SystemC, TLM, Embedded Software, Middleware, SoC, HPC

1. INTRODUCTION

Transaction-level modeling (TLM) has widely been adopted for system level design. One ultimate goal is to allow for early embedded SW development based on a virtual prototype of the HW platform. Much work has been done in embedded SW generation from a transaction-level description [1, 2, 5, 9, 12, 15]. Typical to these approaches is that the SW developers are obliged to use a dedicated TLM API in order to access the functionality of HW IP. Predominantly, a blocking message-passing API based on FIFO channels is proposed. During the software generation process, the communication channels are replaced by behaviorally equivalent channel implementations and device drivers from a target RTOS library.

CODES+ISSS'07, September 30–October 3, 2007, Salzburg, Austria.

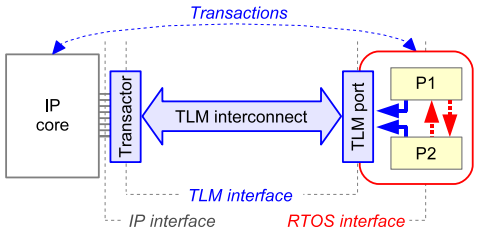

Figure 1: TLM HW-SW model

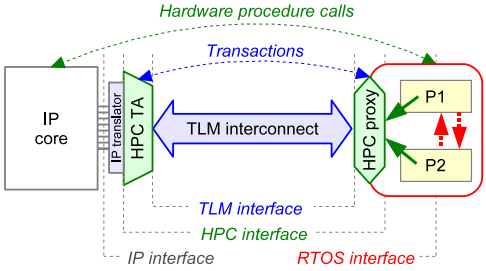

Figure 2: HPC HW-SW model

While from the HW developer's point of view TLM channels describe communication at a high level of abstraction, for SW developers the proposed TLM fabrics are rather inconvenient. Fig. 1 shows a typical transaction-level model of a simple system-on-chip (SoC) with a HW core and a SW subsystem. In such models, communication between concurrent SW processes is modeled by using an RTOS API [2, 5, 10, 15]. This is indicated by the dotted arrows between the SW processes P1 and P2.

In contrast, SW-to-HW communication takes place through the TLM interconnect. As a result, the code in the SW model written for SW-HW communication highly depends on the TLM API used and the HW handshake protocol. If during subsequent model refinement a HW core is replaced by another version, or a HW service is moved into the SW part of the system, tedious and error-prone code refactoring will be required. Switching to another TLM fabric (e.g., to explore different communication architectures) also might result in a lot of effort.

1.1 Hardware Procedure Calls

In this paper we propose our concept of *hardware procedure calls* (HPC). The goal is to abstract away the platform-dependent details of TLM communication by providing a flexible middleware solution for embedded SW modeling *on top* of transaction-level models. With HPC, HW services are

made available to SW processes by high-level service access methods. These access methods are independent of both the HW handshake protocol and the underlying TLM fabric. HPCs transport an arbitrary number of arguments in the form of complex datatypes and are invoked by a simple method call. Communication between service providers (HW IP cores) and service accessors (SW processes) is established by a lightweight HPC protocol which is performed over the TLM interconnect.

The main contribution of this work is a concept for HW-SW communication in transaction-level modeling, that

- provides high-level access to HW functions in a service oriented manner,
- is independent of the used TLM fabric,
- conceals HW interfaces and handshake protocols from the SW implementation.

In particular, HPC enables the use of a single SW model throughout the whole design process. Based on our experiments with SystemC we show that the infrastructure that needs to be in place to enable HPCs in a transaction-level model can be generated automatically from a straightforward input description. Thus, our methodology also eases the integration of heterogeneous IP into a transaction-level model, since less lines of code are required to create IP-specific TLM transactors.

2. RELATED WORK

TLM-based HW-SW co-design with a general-purpose system level design language (SLDL) such as SystemC or SpecC is generally seen as the state-of-the-art in embedded system design. By raising the level of abstraction, design productivity can be considerably increased [4].

Several approaches to synthesize embedded SW from transaction-level models have been proposed. [1, 5, 9, 12] consider C++ code generation for a target RTOS from the SystemC language. In [15], ANSI C code generation from SpecC models is discussed. TLM channel synthesis is focused in [1, 5, 15]. In [5] and [15], the authors propose to map the TLM API onto an RTOS API. In [1], the use of configurable communication coprocessors and a hardware abstraction layer (HAL) is proposed. The flurry of publications on this topic shows that SW generation from transaction-level models is a promising solution to system level HW-SW codesign. But all these proposals have in common that the SW developer must adhere to a specific shared-memory or message-passing TLM API to establish SW-HW communication, with the above mentioned disadvantages.

To overcome this issue, in [3], a service-oriented architecture based on hybrid elements is proposed. These elements provide a high level SW interface on the one side and a low level HW port on the other side. Thus, they act as a transactor between different levels of TLM abstraction. This concept is close to our HPC approach, but the examples in [3] are limited to low-level FIFO communication and the authors do not present a solution to automated transactor generation. An approach for the automatic generation of transactors from an abstract input description is presented in [14]. In contrast to our approach, here communication must be modeled using a predefined semantic based on message passing, shared memory accesses and events.

We propose to raise the level of abstraction for SW-HW communication modeling even further. Our objective is to provide the same ease of use for HW services in a MPSoC model as Sun-RPC and Java-RMI do for remote services in distributed systems. To this end, we developed a high level HW service access protocol to be used on top of virtually any TLM fabric and also in the final SoC. For the experiments we considered the open-source TLM framework GreenBus [8] for SystemC, which supports different TLM APIs in one TLM fabric, including OCP [11], OSCI-TLM [13], and ST's TAC package [4]. Also, different levels of TLM abstraction are supported. Thus, our results can be applied to a broad range of TLM components.

3. GENERAL CONCEPT OF HARDWARE PROCEDURE CALLS

In TLM, a SoC is composed of various HW and SW system components which are connected via a communication fabric. Each component implements a part of the SoC's functionality. Looking at the functionality provided by HW components, we here adopt the concept of services [3]. A service describes a procedure with input and/or output values which is executed by a component on request of another component. For example, a JPEG HW core could provide an `encodeJPEG` service. It describes a complete JPEG image compression procedure. Input values could be a compression rate and a block of raw image data, the output would be JPEG-compressed image data. Different granularity classes for HW services can be considered. Taking up the JPEG example, another IP core may give more control over the image compression procedure by providing the services `quant`, `dct`, and `huffman`. Ideally, for each HW service in a SoC model there would be one SW method which can be called with a set of parameters.

Hardware procedure calls provide such high level access methods to HW services in a transaction-level model. The implementation of an HPC splits up into two parts: a *HPC proxy* and a *HPC transactor*. Both modules are connected to the TLM fabric over which they implement an HPC protocol. This is outlined in fig. 2. Each HPC proxy in a SoC model provides one ore more service access methods. When a service access method is invoked, e.g.:

```
jpegimg = jpegProxy.encodeJpeg(rawimg, 8);
```

the proxy forwards the service request to the target transactor over the TLM interconnect. Upon reception, the transactor reassambles the service access method and calls it into the connected IP. If the IP does not natively provide the service access method, an IP translator has to be used to translate the service access method into the API/interface of the connected IP. After completion, the return values are signaled back over the TLM interconnect, so that the proxy's service access method can return.

3.1 HPC protocol

Proxies and transactors use a high-level HPC protocol to perform HW service requests over a TLM fabric. The design of this protocol plays a key role in the HPC concept. On the one hand, it must be simple enough to ensure that HPCs can be implemented on top of virtually any TLM fabric, independent of their APIs, the transaction semantics, and the design languages used. On the other hand, the protocol should be flexible enough to support all kinds of HW services typically used in SoC models. Finally, it must be reliable and should produce as less communication overhead as possible, since it

will be used in both the transaction-level model and the final chip.

To meet these requirements, two major aspects need to be addressed: First, how handshake between proxies and transactors is performed. Second, transport of the input and output values. In this section, an HPC protocol is specified based on these considerations.

3.1.1 Service classes

Reviews of IP cores showed that four general classes of HW services can be considered (table 1). Typically, services of class 1 are provided by functional IP cores that perform a transformation of input values into output values. Predominantly, such cores can be found in the multimedia domain, e.g. audio/video compression and digital signal processing. Class 2 and 3 services often are provided by peripheral cores. A typical example is an Ethernet MAC with a send and a receive queue. Finally, services without input and output values are specified by class 4. For example, imagine a panel with LEDs and a beeper which the SW programmer would like to control by HPCs `blinkLed` and `beep`.

Table 1: HPC service classes

Class	Service specification
1	Input and output values (e.g., `encodeJpeg`)
2	Input values only (e.g., `playAudio`)
3	Output values only (e.g., `getEthernetFrame`)
4	Neither input nor output values (e.g., `blinkLed`)

By setting either input or output value length (or both) to zero, classes 2, 3, and 4 can be treated as special cases of class 1. However, the advantage of distinguishing four different HW service classes is performance optimization, because the number of transactions per service invocation can be minimized. This becomes clear when we map each of the four service classes to a sequence of transactions.

3.1.2 Mapping service classes to transactions

Before we specify the HPC protocol, we first take a brief look on the notion of transaction to justify our decisions. Transaction is a term of wide comprehension. Basically, the only common ground that can be found when comparing TLM fabrics is that there is cohesion, with transactions being write or read transfers from an initiator (master) to a target (slave) [6, 4, 11, 13]. However, as a consequence of different objectives, the details differ significantly. For example, ST's TAC TLM fabric only supports fixed length transactions. I.e., the number of data bytes transferred in a transaction (read or write) is set up by the initiator and cannot be modified by the target. In contrast, the IBM CoreConnect SystemC TLMs offer the possibility to make the slave decide on its own initiative after how many bytes the transaction shall be terminated.

For the HPC protocol we considered a minimal subset of transaction features which we think is supported by (or can be mapped to) any TLM fabric. This set of features adheres to the following attributes:

- Transactions are always initiated by HPC proxies: this presumes that SW processes are connected to the interconnect by an initiator interface, which for processor cores usually is the case (typically they act as a bus master).
- Transactions are either read or write transfers.

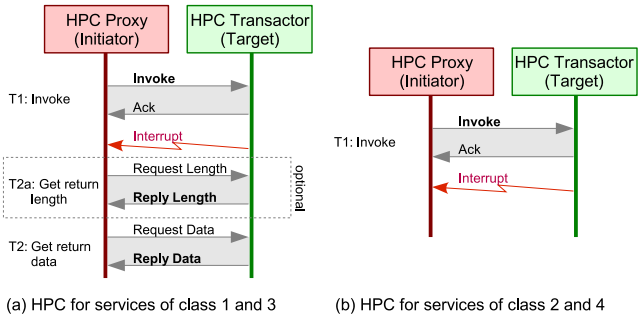

Figure 3: HPC handshake procedures

- HPC proxies and transactors can be configured to use either fixed-length or arbitrary-length bursts.
- No further special TLM fabric features or transfer qualifiers shall be used.

Based on these considerations we specified two handshake protocols. They are shown in fig. 3. Each pair of arrows in the sequence charts represents one transaction. The protocol variant in fig. 3a is used to implement class 1 and 3 HPCs. For class 2 and 4 HPCs, the protocol variant in fig. 3b is sufficient.

3.1.3 Service invocation

For all four service classes, the first transaction $T1$ transports two pieces of information: (1) the HW service that is to be invoked, (2) the input values.

For the first argument, we utilize the fact that in all TLM fabrics known to the authors (both bus and NoC simulation frameworks) the target of a transaction is specified by an address parameter. Thus, for each transactor in an HPC model we specify a range of addresses under which it is reachable. HPC addresses are assigned by the following scheme:

$$hpc\_addr = transactor.base\_addr + service\_id$$

Each transactor can provide an arbitrary number of HW services. The service IDs are assigned ascending, so that the transactor for a HW core that provides n services will allocate the address range:

$$[transactor.base\_addr, transactor.base\_addr + n).$$

The input values for a HW service invocation are transferred (if there are any) as data payload of $T1$. When the transaction arrives at the target HPC transactor, it acknowledges its reception and reads the target address and data payload to extract the service ID and input values. Then it communicates with the HW core over its proprietary interface to activate the HW procedures whose execution is necessary to render the requested service.

After the HW procedures have been finalized an interrupt signal is generated to inform the HPC proxy that it can return control to the SW process. The class 2 or 4 HPC is finished.

3.1.4 Return value transmission

There are two reasons why we chose to transmit the return value of a service invocation in a separate transaction. First, it is not known how long the execution of a HW service will take. Thus, if $T1$ would back transfer the return value, it could block the SoC's communication architecture for a long

time, suspending other data transfers. Second, for class 1 services a combined write/read transaction would be necessary which is not supported by any TLM fabric. Hence, for services of class 1 and 3, a read transaction $T2$ is necessary after the interrupt to retrieve the return value.

Transaction $T2a$ in fig. 3 is not always required. It is necessary if (1) the TLM fabric only supports fixed-length transactions, and (2) the return value length of the invoked HW service is unknown. The latter often is the case for signal processing cores, e.g. a JPEG encoder.

When $T2$ has been finalized, the class 1 or 3 HPC is finished.

3.1.5 Data representation

In the transaction-level model, the input and output values of HPCs may be arbitrary complex datatypes such as classes and structs. Before they are transported via the TLM interconnect, they need to be serialized into a stream of bytes. This is necessary for a number of reasons: (1) to achieve accurate simulation results regarding the communication time consumed by HPCs, (2) because some TLM fabrics may require data fragmentation (e.g. a packet-based NoC), (3) to enable HW-SW interface synthesis from an HPC model, because in the final chip, byte-wise SW-HW communication is inevitable.

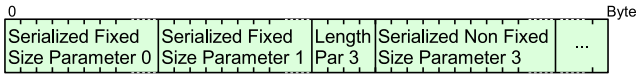

Figure 4: HPC invocation object

Serialization is part of the data representation used in the HPC protocol. Fig. 4 shows the basic structure of an HPC invocation object. It is assembled by the HPC proxy and put into the data payload field of the transaction data structure provided by the TLM fabric. For the data representation, the platform-independent 'network byteorder' is used which in C can be generated by the `ntoh` and `hton` macros. The return value of an HPC is transported in a similar object. Since the receiving HPC transactor knows which HPC is getting currently invoked due to the provided service ID encoded in the address, it knows (1) the number of expected parameters and (2) the size of the parameters if they are fixed in size. Only if a parameter's size is not fixed (e.g., std::vector) a length identifier is included in the serial stream in front of the serialized parameter. Consequently the overhead of the HPC data representation compared to the data that gets transmitted without using HPC is zero when using fixed length paramters and it's only 4 bytes per non fixed size parameter, since we believe parameters do not usually extend sizes of 4 GB.

3.1.6 Concurrent HPCs

In general, one HPC cannot be executed twice at the same time. Thus, if two SW processes try to invoke the same HPC concurrently, the SW process whose transaction $T1$ arrives first will win the race. The HPC requested by the second SW process will be aborted by the transactor. How this is implemented depends on the underlying TLM fabric. Usually, a NACK (non-acknowledge) will be replied.

However, HPC transactors that provide more than one services should be able to handle different HPCs in parallel. For example, a SW process could send data to a network controller, while another SW process receives data from the same controller. To enable this kind of 'allowed' concurrency but to also rule out unsupported constellations of parallel HPCs (most IP cores do not support parallel operations, or concurrency is limited), HPC transactors must implement a concurrency management. It should support the following features:

- Accept any HPC if state is idle.
- Reject HPC for service A if A is currently executed.
- Accept/reject HPC for service A if another service B is currently executed and there is no/a mutual exclusion rule for A and B.

3.1.7 Error handling

Several kinds of errors can occur during an HPC. We can classify them into two categories. First, the TLM fabric can produce transfer errors such as bus timeout or denial of service due to high load. Second, the HPC transactor can reject a service request due to its concurrency rules. In both cases the transaction will abort. Depending on the capabilities of the used TLM API it may not be possible to get a clear report on the error reason. Thus, HPC proxies should retry an HPC several times before they give up. If nevertheless the HPC fails, this indicates a severe deadlock problem in the SoC model (e.g. due to permanent bus overload or inappropriate arbitration schemes). This should be reported in the system level simulation.

3.1.8 DMA

The above considerations imply that all IP cores providing HW services in an HPC model boast a target (slave) interface. Using memory mapped registers in the IP core, SW processes send the input values of an HPC actively to the HW. For many tasks, this kind of 'push' communication is sufficient.

However, cores built for high data throughput normally use DMA transfers to reduce CPU load. In this case, HPCs cannot (and should not) be used to transfer user data *directly* from the SW to the IP core. Instead, HPCs for DMA cores should *signal* that a transfer is to be started. To this end, a class 2 HPC carries the base address and size of the memory region that is to be processed, so that the HW core or a dedicated DMA controller can start DMA operation. The HPC's return value indicates completion and points to the memory address where the output values can be found.

One could argue that for systems with DMA controllers this approach is inconvenient as it implies that the DMA controllers are explicitly considered in the HPC model, thus lowering the level of abstraction. However, all conceivable 'higher level' solutions would require some technique for automatic DMA controller synthesis out of the HPC model, which is not in the scope of this paper.

3.2 SystemC implementation

We have created a SystemC implementation of the HPC protocol on top of the GreenBus [8] TLM fabric. To this end, we took several SoC models including a video processor and a cellphone system[1], and converted them into HPC models by writing adequate HPC proxies and transactors. The SoC models are comprised of various HW components at different levels of abstraction. GreenBus supports the composition of

[1]The HPC implementation for SystemC and test models are available for download at www.greensocs.com/GreenBus

mixed-mode models by its 'transaction container' (TC). The TC provides a generic representation of transaction phases ('atoms') and transaction data ('quarks') so that interaction between different TLM APIs and abstraction levels is made possible. Our HPC proxies and transactors adhere to this scheme and can perform HPCs over any GreenBus-compliant communication architecture. To this end, we used the GreenBus 'generic API' to implement transaction-level communication. The interrupts that are part of the HPC protocol (see fig. 3) were implemented with SystemC events (`sc_event`).

Data serialization is supported by an `HPC_EXTENSIONS` macro. It is used as a wrapper around datatypes, equipping them with a `serialize` and a `deserialize` method. These methods are used by the HPC proxies and transactors to pack and unpack transaction containers. Our `HPC_EXTENSIONS` macro covers all standard C++ and SystemC datatypes.

Since GreenBus allows for timed simulations at both a bus accurate and a cycle-count accurate level of abstraction, HPC models can be used for comprehensive communication architecture exploration. The precision of the simulation results does not differ from the precision of non-HPC SoC models.

4. TRANSACTION-LEVEL MODEL GENERATION

Based on various experiments with the HPC concept, we have created a set of generic HPC proxy and transactor fragments from which application-specific proxies and transactors can be generated automatically. These specialized proxies and transactors provide exactly the functionality necessary for the supported HPCs.

To create an HPC model, the following preparations are necessary: For each IP core with services to be made available by HPCs, a set of service execution methods needs to be in place. These methods must communicate with the IP core over its interface/API to execute the HW procedures that are to be invoked by the corresponding HPCs. Thus, they are absolutely IP-specific, and there are no limitations to the techniques used for the implementation of these methods. We refer to this set of methods with the term *IP translator*.

When the IP translators are in place, the HPC model generation can begin (see figure 5): For each target module with one or more services to be invoked from SW, the developer adds HPC annotations to the methods to allow automatic understanding of the SystemC code. Using our SystemC analysis framework DUST [7], the annotations can be written to an XML file that also contains information about the design structure. Automatic processing of this data is possible without big effort. A Java tool can generate model-specific HPC proxies and transactors by simply assembling them from our library of code fragments to specialized proxies and transactors for each method that was annotated as an `HPC_METHOD`.

In order to use the proxies and transactors they can be included in a HW and SW model with two simple commands: `#define HPC_CONFIG_SLAVE_<SlaveName>` selects a proxy-transactor-pair and `#include "hpc.h"` makes them available. That followed the transactor can be instantiated in the HW module with `hpcAPISlave_transactor_<SlaveName> myTransactor` and the proxy can be instantiated in the SW module with `hpcAPISlave_proxy_<SlaveName> myJpegProxy`. The SW makes HPCs on the proxy, e.g.

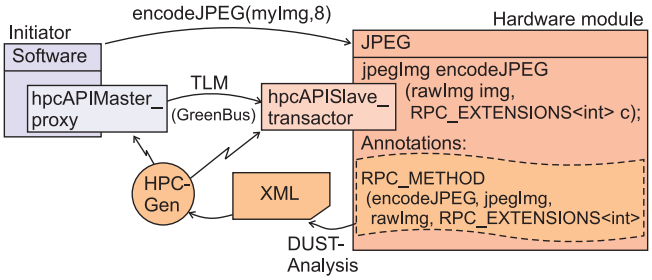

Figure 5: HPC concept

`myJPEGImg = myJpegProxy.encodeJpeg(myImg, 8);`. The HPC will be transferred through the TLM interconnect to the HW module, where the transactor instance will call the proper IP translator method.

While the proxy and transactor generation can be performed fully automatically, the implementation of the HW access methods in the IP translator is the only 'tricky' part of the HPC model generation. Here, a full understanding of the HW-specific handshake protocols and interfaces is necessary. However, this also would be the case in any 'normal' non-HPC transaction-level model, although with the difference that there the HW access must be implemented in the SW processes. Thus, the HPC concept moves this implementation from the service accessor (SW) to the service provider (HW). Moreover, the code that accesses the HW's interfaces is completely separated from the code that connects it to the TLM interconnect. The latter can be generated fully automatically with our methodology. As a result, the TLM fabric can be exchanged or modified anytime during the design process, without requiring adjustments to the IP translators.

5. A CASE EXAMPLE

To investigate how and to what extent the HPC concept can aid the designer two models of the same video processor system were designed independently. This system consists of various IP blocks that reside at different abstraction layers and use different APIs/interfaces. All cores were available out of our IP library, and only the software that controlled the functionality of the IP blocks was developed from scratch.

The first model was created using a common TLM design flow in which the heterogeneous IPs were connected to a TLM framework (GreenBus), and were therefore equipped with appropriate TLM transactors translating the GreenBus API into the IP's interface. The SW thread invokes the IP's services using the TAC interface, which is translated into the GreenBus API using a TLM transactor (see figure 6).

The second model was created using the HPC concept, so the IP cores were now treated as service providers, e.g. the video digitizer was now supposed to provide a service named `getYUVFrame`. To this end, we implemented IP translators that provide the HPC functions to the environment and translate them into the API of a specific IP block, that can then be connected to this module. For example the function `getYUVFrame` of the IP translator for the video digitizer executes a complete OCP-TL1 communication sequence in order to instruct the IP core to grab and send a video frame using the YUV color format. The HPC proxies and transactors were automatically created as described in section 4, such that the software thread can use the IP services as plain function calls (see figure 7).

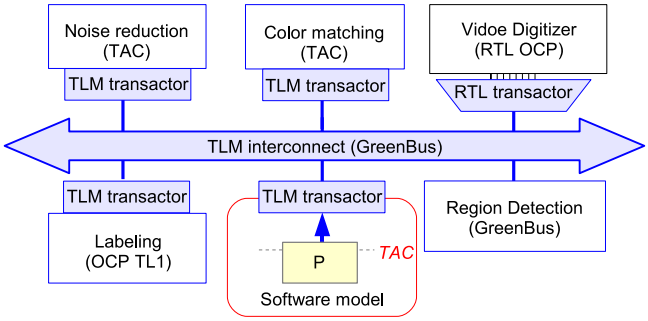

Figure 6: Original video processor TLM model

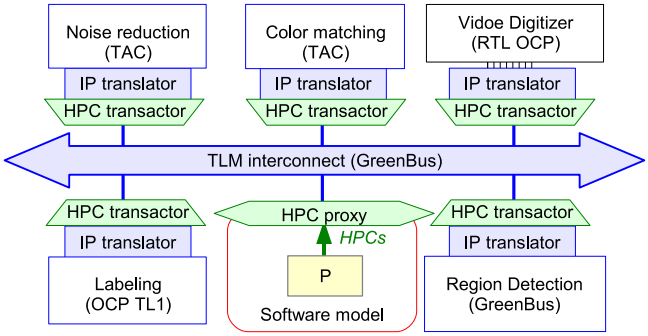

Figure 7: Video processor HPC model

As can be seen in table 2 the number of lines of code (loc) required to write the TLM transactors is usually greater than the number of loc needed to write the IP translation modules. Both basically should need the same number of loc to communicate with the IP, but the TLM transactor additionally contains 'general-purpose' code to deal with incorrect accesses from the TLM framework, which the HPC protocol renders unnecessary. The number of loc needed in the SW is also larger if using TAC, since the complete TAC protocol including error handling and retry mechanisms has to be implemented, while an HPC can be done by a simple method call. An important row in table 2 is row five. There the IP is directly connected to the TLM framwork which does not require a TLM transactor, but of course the aforementioned locs in the SW to access the TLM framework. In this case the use of HPC introduces a code overhead concerning the TLM transactor, but still simplifies the SW code significantly.

In summary, the use of HPC reduces SW complexity, and thereby development time, as the use of HW services is now done by single function calls instead of TLM API specific communication sequences. The SW model and the TLM fabric are decoupled. Hence, using automatically generated HPC proxies and transactors the software can now be used on top of different TLM APIs and frameworks, thus simplifying architecture exploration and refinement. The example demonstrates that HPC eases the assembly of heterogeneous systems. Only in case of IP and TLM fabric using the same interface a small overhead is introduced. This drawback is compensated by the fact that the HPC-enabled HW core can be reused with other TLM fabrics.

6. CONCLUSION

In this paper we discussed a high-level middleware approach for transaction-level MPSoC models that makes on-chip HW functions available to SW processes by simple method calls in a service oriented manner. The proposed methodology introduces almost no communication overhead (see section 3.1.5) and the experiments show that it considerably eases SW development, assembly of heterogeneous systems and design space exploration by insulating SW code and HW service access from the used TLM framework. The feasibility of automatic HPC proxy and transactor generation was shown for the TLM framework GreenBus, but can be put into practice for every TLM framework that supports fixed length, adressable read/write transactions (see section 3.1.2).

Our ongoing work focusses on defining a set of generic pin-level service access interfaces for IP cores and based on this we examine how HPC-enabled communication co-processors and appropriate RTOS drivers can automatically be generated to allow for full HPC model synthesis.

Table 2: Video processor modeling effort

IP core	Abstr. level	Interface	TLM trans. + SW loc	HPC descr. + SW loc
Video digitizer	RTL	OCP-pin	77+11	52+3
Color matching	PV	TAC	48+16	11+3
Noise reduction	PV	TAC	48+16	11+3
Labeling	BA	OCP-tl1	56+16	32+3
Region detection	BA	GreenBus	0+21	56+3

7. REFERENCES

[1] W. Cesario and A. Jerraya. Component-Based Design for Multiprocessor Systems-on-Chip. *Multiprocessor Systems-on-Chip*, Morgan Kaufmann, 2005.

[2] J. Chevalier, M. de Nanclas, L. Filion, O. Benny, M. Rondonneau, G. Bois, and E. M. Aboulhamid. A SystemC Refinement Methodology for Embedded Software. *IEEE Design & Test of Computers*, pages 148–158, 2006.

[3] P. Gerin, H. Shen, A. Chureau, A. Bouchhima, and A. A. Jerraya. Flexible and Executable Hardware/Software Interface Modeling for Multiprocessor SoC Design Using SystemC. *Proc. ASP-DAC*, 2007.

[4] F. Ghenassia. *Transaction-Level Modeling with SystemC.* Kluwer Academic Publishers, 2006.

[5] F. Herrera, H. Posadas, P. Sanchez, and E. Villar. Systematic Embedded Software Generation from SystemC. *Proc. DATE*, 2003.

[6] IBM. IBM PowerPC 405 Evaluation Kit with CoreConnect SystemC TLMs. IBM, September 2005. *Available at http://www.ibm.com.*

[7] W. Klingauf and M. Geffken. Design Structure Analysis and Transaction Recording in SystemC Designs: A Minimal-Intrusive Approach. *Proc. FDL*, 2006.

[8] W. Klingauf, R. Günzel, O. Bringmann, M. Burton, and P. Parfuntseu. GreenBus - A Generic Interconnect Fabric for Transaction Level Modelling. *Proc. DAC*, 2006.

[9] M. Krause, O. Bringmann, and W. Rosenstiel. Target software generation: an approach for automatic mapping of SystemC specifications onto real-time operating systems. *Design Automation for Embedded Systems*, 10(4):229–251, 2005.

[10] R. Le Mogine, O. Pasquier, and J.-P. Calvez. A Generic RTOS Model for Real-time Systems Simulation with SystemC. *Proc. DATE*, 2004.

[11] Open Core Protocol International Partnership. A SystemC OCP Transaction Level Communication Channel. *Available at http://www.ocpip.org*, February 2007.

[12] S. Ouadjaout and D. Houzet. Generation of Embedded Hardware/Software from SystemC. *EURASIP Journal on Embedded Systems, Article ID 18526*, 2006.

[13] A. Rose, S. Swan, J. Pierce, and J. M. Fernandez. Transaction Level Modeling in SystemC. *OSCI TLM Working Group*, 2005.

[14] D. Shin, A. Gerstlauer, J. Peng, R. Dömer, and D. Gajski. Automatic generation of transaction-level models for rapid design space exploration. *Proc. CODES+ISSS*, 2006.

[15] H. Yu, R. Dömer, and D. Gajski. Embedded Software Generation from System Level Design Languages. *Proc. ASP-DAC*, 2004.