GreenBus - A Generic Interconnect Fabric for Transaction Level Modelling

Wolfgang Klingauf
Robert Günzel
TU Braunschweig, E.I.S.
(Prof. U. Golze)
38106 Braunschweig,
Germany
klingauf,guenzel@
eis.cs.tu-bs.de

Oliver Bringmann
Pavel Parfuntseu
FZI, Microelectronic
System Design
(Prof. W. Rosenstiel)
76131 Karlsruhe, Germany
bringmann,parfunt@fzi.de

Mark Burton
GreenSocs Ltd.
Cambridge CB4 3ES, UK
mark@greensocs.com

ABSTRACT

In this paper we present a generic interconnect fabric for transaction level modelling tackeling three major aspects. First, a review of the bus and IO structures that we have analysed, which are common in todays system on chip environments, and require to be modelled at a transaction level. Second our findings in terms of the data structures and interface API's that are required in order to model those (and we believe other) busses and IO structures. Third the surrounding infrastructure that we believe can, and should be in place to support the modelling of those busses and IO structures. We will present the infrastructure that we have built, and indicate where our future work will head.

Categories and Subject Descriptors: B.8.2 [Performance and Reliability]: Performance Analysis and Design Aids

General Terms: Design, Performance, Verification

Keywords: On-Chip Communication, SystemC, TLM, SoC

1. INTRODUCTION

GreenBus is the project name for a collection of work aimed at providing an open source modelling framework that will enable system-on-chip (SoC) designers to exploit SystemC communication modelling techniques easily and efficiently early in the design cycle. The emphasis of the project is on model inter-operation and the results have been submitted to the Open Source SystemC Initiative Working Group on Transaction Level Modelling (OSCI TLM WG). GreenBus provides a SystemC 2.1 style port-to-port bound bus fabric which is configurable to represent any bus at a programmers view, cycle accurate and a cycle count approximate level of abstraction. It comes complete with a "native" ability to have "user API's" such that a user can choose their interface independent of the bus fabric itself.

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

The main objectives of GreenBus are:

- To provide a generic yet flexible bus and IO fabric for SystemC TLM modeling of SoC components.
- To support all levels of TLM abstraction from programmers view (PV) to cycle-callable (CC) models.
- To enable inter-operation between models of different levels of abstraction (mixed-mode), and models with different interfaces (heterogeneous components), with as little overhead as possible.
- To attain highest possible simulation performance at each level of abstraction.
- To adhere to common standards such as OSCI-TLM and SystemC-SCV, and to provide input into those standards (especially OSCI TLM).

The paper is structured as follows: In section 2 we give an overview on other works in the area of generic communication architectures, and the bus and IO structures we have analysed. In section 3 we explain the concepts of the GreenBus approach. The data structures and API's that we believe are required to model busses and IO are presented in Section 4. The following sections will provide some details of our implementation. The underlying architecture of our bus fabric will be presented in 7 and the configuration, logging and debugging capabilities are shown in 8. The achieved results including performance evaluation is given in Section 7.2. Finally, this paper concludes with a summary in Section 9.

2. RELATED WORK

Bus fabrics remain one of the most required and most argued over pieces of model IP in a SoC simulation environment. There are a number of different technological features of different proposals. They differ either in the supported levels of abstraction or the supported transportation technology. There is even argument about the requirement and scope for a ubiquitous fabric. Here we take the view that, given the number of proprietary and unpublished "generic busses" present in the industry today, the utility is unquestionable. We define the scope of our work as covering both mixed mode (different levels of abstraction) and heterogeneous (different bus interfaces) models. There is no common

use of terms, despite attempts by groups such as the OSCI TLM WG to propose a set [3] (which we adhere to).

One design goal for GreenBus is to provide better user-defined adaptability than other fabrics. The most interesting work has been done by Kogel et al. [8]. Their generic interconnect model for simulating on-chip busses and networks on chips enables architectural exploration. However, they deal with this at a very high level of abstraction supporting packet based communication only and therefore lack the possibility of providing cycle count accurate timing estimations. Other research groups identified the need of generic bus models but do not offer solutions [4, 7].

Reviews of bus protocols being used in the industry [1, 11], showed that at higher levels of abstraction (PV) there is industry wide cohesion, with blocking calls being used. [5] suggests the usage of a single transport call for PV. At lower levels of abstraction non-blocking interfaces are often preferred, but not ubiquitous. For example frameworks such as the Open Core Protocol OCP [12] offer both. The fundamental requirement is to provide a mechanism to efficiently transmit data, and timing information from initiator to target.

The first aspect of this is how memory will be managed. The choices are to either have the initiator port allocate enough memory for both the request, and response information (from the target) - this is commonly called pass by pointer as recommended by CoWare and ST; or, all data can be transfered as data items and no memory allocation is necessary - this is often referred to as pass by value [14].

This fundamental difference at the outset of a transaction impacts the way subsequent communication is handled. Frameworks that deploy pass by pointer can then either use subsequent function calls or simply events to indicate updates to the (shared) data structure. Frameworks that pass by value must use function calls to pass updated values.

The second aspect is the timing of data transfers. In order to minimise the amount of re-calculation, some bus fabrics are designed to execute on the falling edge of the clock [13], such that all requests which need to be arbitrated will be present, and the arbitration only need take place once. Compelling as this scheme at first seems, on todays multi-bus SoC's, designers soon run out of "falling edges", and interfacing to RTL becomes very much harder.

One of our goals is to be able to support the body of existing IP, which uses the full spectrum of "bus interfaces", hence we introduce an extra requirement on our bus fabric implementation to be able to support both blocking and non-blocking interfaces, pass by pointer and pass by value. Similarly, there are different approaches taken to the data that is transported. There are in essence two different approaches taken to this subject. Some favour extensible data types, while others opt for providing a defined bus fabric onto which "all other" busses can be mapped. The latter approach is typical of bus vendors, while the former is often adopted by bus users. Hence, ST's TAC [11] favours extensible data types, enabling the user to transfer any data, while for example OCP [12] predefines the data structure (bit vectores) but offer some user definable flags. The disadvantage of the extensible data structures is, first there can be lack of consistency between implementations, second many data types do not lend themselves to extension. However, there are equal disadvantages with defined busses, i.e. its extension if (and when) a bus is being used which does not easily map onto the offered structures. In addition, from the simulation speed perspective, a pre defined "common bus" incurs a simulation overhead if modes have to be wrapped onto the predefined structures.

Our approach is to fix the data structure elements, but to allow choice of which elements to use in a bus, or interconnect fabric. The intent is to mitigate the problems of inconsistent implementations while offering the user a flexible bus framework.

Both ARM's AXI [9] protocol and IBM's CoreConnect [6] offer interfaces and protocols that can deal with most communications occuring in embedded systems. Both of these protocols map easily onto the OCP bus fabric. The details of how they map onto our framework will be detailed below.

3. GENERAL CONCEPT OF GREENBUS

In transaction level modeling, a system-on-chip is composed of various master and slave system components which are connected (often via a bus fabric). The models of the components themselves will use "convenience functions" to provide the model writer with an as flexible, intuitive, and easy to use interface as possible. As has been seen in section 2, the convenience functions (or interface) that is chosen varies between different user companies, and different bus fabrics. Thus, GreenBus follows a two-layered approach that decouples the user component's bus interfaces (convenience functions) from the GreenBus interface - which is the underlying low-level transport API.

This has several key advantages:

1. Component models can be written using the most convenient IO interface API, and can remain unchanged (even shipped as binary objects).
2. Components can be exchanged by simply changing the "user API to GreenBus low level layer".
3. The user API to low level GreenBus layer is efficiently managed and has no significant effect on simulation performance.

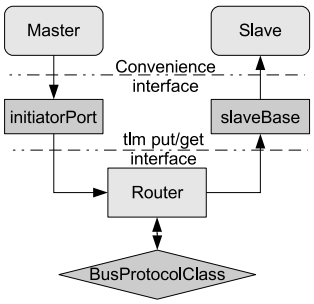

Figure 1: Simple GreenBus use case

Figure 1 shows a simple use case of the GreenBus fabric. The initiator port provides an application-specific communication API, e.g. simple read/write methods. Slave components are connected to the bus by inheriting from the `slaveBase` class. A slave implements a potentially different application-specific interface defined in the specific slave base. The underlying communication fabric is independent from the convenience API which can be chosen by the GreenBus user and can be different for each IP component in the system. The router module belongs to GreenBus and is accompanied by a bus simulation engine which is responsible

for all the bus protocol arbitration and timing estimation. In order to validate our approach, we have constructed a router, which will be examined in Section 7. Besides the two layered API, another basic concept of GreenBus is the use of transaction atoms and quarks, their relation ship to each other and how they can be used to model busses is examined next.

4. TRANSACTIONS, ATOMS, QUARKS

To understand the interaction of all GreenBus components the concepts of transaction atoms and quarks are crucial: We believe that every transaction can be composed from a number of so called "atoms". An atom is the smallest uninterruptible part of a transaction that once started will complete its lifecycle. We have chosen "atom" as a neutral term, others have used different names, for instance OCP refers to these as transfer phases (see section 2).

All the bus and IO structures we have seen can be represented as containing transactions with up to just three different atoms: *init* atoms, *data handshake* atoms and *finalise* atoms. The init atom carries all the transfer qualifiers, after its completion both master and slave are ready to exchange data. The data handshake atom is used to transfer write or read data and all accompanying qualifiers like byte enables or error flags. The finalise atom finishes the transfer and can carry final responses or information needed to release the connection properly. It is possible that a transaction does not use all atoms, e.g. there are busses that won't need a finalize atom, a simple IO interface may only use one atom.

Table 1: GreenBus quark sets for AXI and PLB

Quark	type	AXI	PLB
Init Atom			
address	sc_uint<64>	AWADDR / ARADDR	PLBABus
masterID	int	AWID / ARID	PLBMID
burstlenght	int	AWLEN / ARLEN	used with line reads / writes and fixed length bursts
BE	int	AWSIZE / ARSIZE	PLBBE
burstType	enum	AWBURST / ARBURST	PLBSize
busLockType	enum	AWLOCK / ARLOCK	PLBbusLock
userQuark1	n/a	AWCACHE / ARCACHE	PLBtype
userQuark2	n/a	AWPROT / ARPROT	PLBordered, PLBblockErr, PLBguarded, PLBcompress
atomValid	bool	AWVALID / ARVALID	PLBrequest
ackValid	bool	AWREADY / ARREADY	PLBAddrAck
priority	int	N/A	PLBreqPri
rNw	bool	fixed parameter of port	PLBrNw
burst	bool	always true	PLBrdBurst / PLBwrBurst

We refer to the payload carried by the atom as "quarks". A quark is nothing more than a basic data type. A fundamental principle of GreenBus is that quarks are pre-defined. Again, other bus fabrics have similar notions. We simply suggest that for all features of a bus there should be a one-to-one mapping of feature and underlying transport type. For instance, any bus capable of transporting exactly 64 bits of data should always use the same data structure to do so. This fundamental principle is the key to providing model inter-working with the minimum cost.

The "quark" data types need not be exhaustive, as bus and IO features which are really unique will always require some interpretation between IP not designed to the same interface. In this case, inter-working will always come with some cost, hence standardising on the types for unique features does not help.

As an initial set, we are persuaded that the set of types defined by OCP is relatively comprehensive, with some minor additions. To illustrate how a standardised set of quarks can be applied to different bus architectures, table 1 shows the quarks and atoms needed for IBM's CoreConnect PLB and ARM's AXI. Their common signal are mapped onto standardized quarks, uncommon features are mapped onto so called user-quarks, whose semantics are defined by the users.

5. ABSTRACTION LEVEL FORMALISM

A natural outcome of considering transactions as being composed of atoms and quarks is that we can present a formalism for the TLM abstraction layers.

Depending on the abstraction layer of a module, the points of interest differ. A PV module is only interested in the beginning and end of a transaction (it may not proceed until the transaction is complete). A bus accurate module additionally needs to know about the start and end of atoms in order to "accurately" provide timing information. A cycle accurate module requires information about changes to each quark as it may need to react. Finally, an RTL level model will need to know about the state of each quark at each clock edge (see Table 2).

Table 2: Abstraction layer/information relationship

Name	Required information
PV	Transaction completions
BA	Transactions and Atoms completions
CC	Transactions, Atoms and Quark updates
RTL	Quarks at each clock edge

A "PVT" model as the OSCI TLM WG defined it is really more a technology than an abstraction layer. It essentially works at two abstraction layers, PV, and then either BA, CC or indeed RTL.

In accordance with this, a PV master will always send the whole transaction including all atoms and quarks at once and wait for it to be completed. A BA master would send the transaction atom-wise, to keep control of phase propagation and a CC master would also send the transaction atom-wise but wont fill in all the quark a priori, but one after the other. For example if the CC master does a burst write it will send the data hand shake atom to the slave, with the first word inside, and after the slave acknowledges, the master will fill in the next, so on and so forth until all data has been transferred and the atom is finished. Figure 2 illustrated how transactions are build out of atoms and quarks and shows the points of interest (depicted as arrows) at the various abstraction levels.

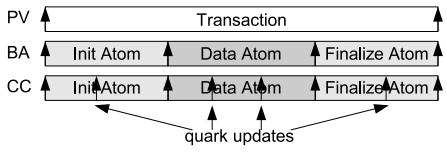

Figure 2: Transactions - Atoms - Quarks

Since the atoms are the basic blocks used at BA and CC abstraction layers, the lifecycle of atoms is of extreme importance. The "life" of an atom that is to be transferred over a bus starts with it requesting access to the bus. After t_{grant} the atoms is granted access to the bus, after $t_{deliver}$ the atom arrives at its destination, after t_{accept} the atom is accepted by the target and finally after $t_{terminate}$ this atom will be terminated and the master gets informed about this, so it knows the transfer of this atom has been finished. Figure 3 shows the life cycle of such atoms, regardless of the type of the atom. It is important to note that in this life cycle, the initiator is the only entity that holds the atom from it's conception till its demise. It is therefore the only entity that can adequately handle any memory management issues.

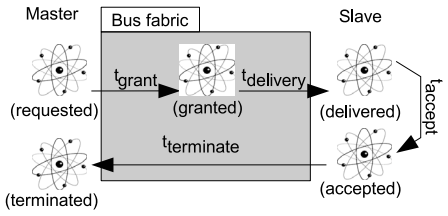

Figure 3: An atom's life cycle

6. LOW LEVEL API

As we have seen, the unit of transport, be it atoms or quarks, has a close relation to abstraction level. We propose here an API which must reflect this, and must fulfil the requirement to be able to construct a set of convenience functions at little or no cost.

We propose two orthogonal interfaces. First, to reflect the PV levels requirement to pass entire transactions we propose a single blocking[1] function call. This is in common with ST's TAC (see Section 2) . The semantic contract for this function call is that it will return when the entire transaction is complete. There are issues with synchronisation between masters within the same system. These are not dealt with within this paper (see [5]).

Second, to cover all other lower levels of abstraction we propose a single non-blocking API. This function call takes an atom, and notifies all other (interested) components in the system of the atoms presence. There are then a number of events that can be generated by various system components with respect to the atom. For instance, and most useful at the BA level of abstraction, atom terminate and accept events are generated by the target ports. In a BA bus model, no other events are generated. Though at a cycle accurate level of abstraction it is required to generate and listen to events on each quark or at least on each data cycle completion (for performance improvement).

This simple interface allows blocking and non-blocking user APIs to be constructed with little or no overhead.

The data structures at all abstraction layers are the same. There is a transaction object, containing all required atom objects. Those atom objects contain all required quark objects. A PV master is able to validate all atoms and all quarks at once, a BA master is able to validate an atom at once and a CC master is only able to validate some quarks at a time, but all will create the whole structure at the beginning of the transaction.

[1]this means the function call *may* call wait.

The disadvantage of this is that data structure members which are timing specific are present (but unused) at a PV level of abstraction. The implication is that even at a PV level of abstraction some decision about the eventual nature of the bus that will be used has been made. This is not always the case, but where no such decision has yet been made, some sort of fabric is still required. Our recommendation is that a bus fabric as near to the final one be chosen, because the internal IP will need to know, and handle some features of the bus - which if absent will need to be accounted for later in "wrapping" layers.

7. ROUTER ARCHITECTURE

In order to validate our approach, we have constructed an entire system, including the principle part of any bus fabric, the router and arbitration mechanism itself. Again, the approach we have taken keeps as much of the fabric re-usable as possible. In GreenBus the router and the bus protocol class form the actual bus fabric. The router is the generic part, that can be reused without change for any bus. In contrast the bus protocol class contains all the bus specific information. So the router in connection with an AXI bus protocol class forms an AXI bus functional model. In connection with a PLB bus protocol class it forms a PLB bus functional model. This decoupling of routing, which is common to all busses and bus behaviour, which is very specific, is possible with the help of the previously described atom concept.

From our review of busses (see Section 2), the most important requirements for GreenBus are:

1. Support multiple simultaneous, outstanding and active transactions.
2. Support and profit from transaction's phase structures.
3. Support fixed and dynamic delays.
4. Events must mark rising signal edges to enable wrapping onto RTL.
5. Support clocked and combinatorial arbitration.

The router conforms to the GreenBus Architecture, providing two interface methods, a blocking and a nonblocking one. The blocking method (`blockingPut(transaction)`) takes a transaction and transfers it as a whole to the targeted slave. This method is only used by PV masters. The non-blocking interface method puts atoms into the router (`put(atom)`) and the router target port will generate an event signalling the termination of the atom.

The slave's non-blocking interface is the counterpart of the master's interface. There's a method that puts an atom into the slave and the slave will trigger an event to signal the acception of the atom. If the master uses the blocking put, the router will do a `decodeAddress(transaction.initAtom)` call back to the bus protocol over its bus protocol port and call the blocking put of the slave base. No transactional timings or delays are applied, since this is not necessary for the PV abstraction layer.

If the master uses a non-blocking put, the router's main task is to receive atoms from initiator ports and to deliver them to the targeted slave bases. Thereby the router must apply all the delays introduced in section 4. This has to be done in a generic manner, so that the router can be used for every conceivable bus. To this end, every time the router

has to apply a delay it does a callback into the bus protocol class, which is responsible for calculating the delay (in our implementation, the delay is realized in the router itself, so as to maintain control of the thread).

There can be simultaneously incoming atoms competing for access to the bus. To solve such conflicts, the router must do another callback to the bus protocol class, in order to discover which atom can be granted. In this way the router utilizes the arbitration scheme of the simulated bus.

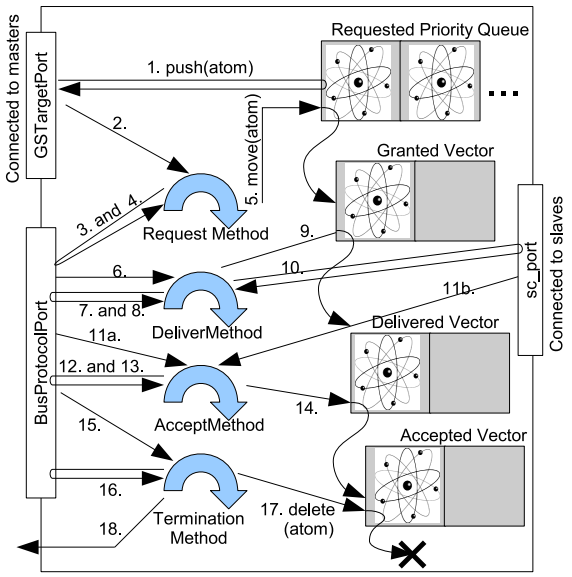

Figure 4: Router's internal structure

Figure 4 shows the router's internal structure and its functionality for the non-blocking put. An atom transfer always starts by the master putting it to the router (1.). Thereby the atom gets "enqueued" into the router's internal priority queue. The ordering mechanism of this queue is specified in an external bus specific class and sorts the incoming atoms according to the arbitration scheme of the bus. Afterwards the request method of the router gets triggered (2.). This method does two call backs to the bus protocol class. The first one (3.) `getAtom` returns the next atom that can be granted (hence the bus protocol accesses the priority queue) and the second one (4.) `scheduleDelivery` schedules the start of the deliver method. Then the router moves the atom it got from the `getAtom` call back to the granted vector (5.). The deliver method starts (6.) at the time that was defined by 4. and again, does two call backs. The first one identifies the atom of the granted vector that has just been delivered (7.,`atomIsDelivered`) and the second one (8., `scheduleAcception`) schedules the start of the accept method. This scheduled start can be used to model a timeout. Afterwards the deliver method moves the recently delivered atom from the granted vector into the delivered vector (9.) and puts the atom to the targeted slave (10.). The port to which the targeted slave is connected is retrieved by a `decodeAddress(atom)` callback to the bus protocol. Now accept method starts either at the scheduled acception time (11a.) or when the slave accepts the atom (11b). Again this is followed by the accept method doing two callbacks to the bus protocol. The first one (12.) `atomIsAccepted` returns the atom that was recently accepted (in case of 9b happened) or that just timed out (9a.). The second one is (13.) `scheduleTermination`, which schedules the point of time at which the transaction can be terminated. Finally the accept method moves the atom from the delivered vector to the accepted vector (14.). As soon as the terminate method starts (15.) the final call back to the bus protocol is made (`atomIsTerminatable()` (16.)) which returns the atom that can be terminated. Now the termination method removes the atom from the accepted vector (17.) and informs the master about the end of the atoms life cycle (18.). (This actually goes through the target port, but was drawn seperately to increase the figures clarity). To summarise this, the call backs `scheduleDelivery` (4.) and `scheduleTermination` (13.) are used to determine $t_{deliver}$ and $t_{terminate}$, while `getAtom()` (3.) and the external bus specific priority queue ordering class determine the arbitration scheme and t_{grant}. It is important to notice that t_{accept} is bus protocol dependend because of `scheduleAcception` (8.) and slave dependend because the acception can also be triggered by the slave. The router implementation enables the user to apply this kind of dual sensitivity to all other methods. This way even the delivery or the termination can be triggered from external modules. The router's implementation contributes to the previously given requirements by making use of the atom concept (requirement 2), by handling multiple atoms at once (requirement 1), by supporting dual sensitivity (requirement 3), by triggering events at the positive edge of an signal (requirement 4) and by using a re-evaluation mechanism for combinatorial arbitration (requirement 5, not part of this paper).

7.1 Initiator port and slave base

The initiator ports and slave bases are used to wrap the APIs of the user modules to the GreenBus underlying transport mechanism. A piece of IP that conforms to our proposal will use initiator ports that are capable of either generating blocking or non-blocking calls, and, if non-blocking, will handle events generated on the atoms. Likewise, target ports must provide *both* blocking and non-blocking interfaces. The semantic contract for a blocking function call is that it only returns when the transaction completes (or there in an error, for instance, the IP simply can not process the transaction). It is the responsibility of the target port slave base (within a slave, or within a BA level bus fabric for instance) to guarantee the semantics.

There are two scenarios. First, a blocking function call may be requested on a piece of IP built at an BA or CC level of abstraction. In this case, the slave base must insure the correct sequence of atoms is played out into the IP. Second, a non-blocking call may be made to a piece of IP written at a PV level of abstraction. In this case, the slave base must insure that the entire transaction is assembled before calling the IP. On successful completion of the transaction by the IP, the slave base must generate the correct events back to the rest of the system.

7.2 Experimental results

We implemented the router, and configured it to simulate a core-connect PLB 100% accurately in accordance with [6] at a BA level of abstraction. To demonstrate both the heterogeneous, and multi-mode abilities of the fabric, we combined this with a piece of TAC IP at PV level and a piece of proprietary 3rd vendor IP at a BA level of abstraction. Building the slave bases and initiator ports for the 2 different user APIs took about 2 day each, even for the more complex BA level API. Building the bus protocol class for core connect

PLB took a further day. The entire system, with 3 different protocols involved at 2 different abstraction levels (PV, BA) was built and tested in under one week. The combined simulation runs with a performance of 250,000 transactions per second on our machine.

Also we implemented OCP-tl1 initiator and target ports, which turned out to be much more difficult than BA/PV ports and so this took us about a week. While simulation speed itself is vital, the time taken to construct the model is also vital, and a key success of our project. Therefore we would like to automate this process still further.

To evaluate the performance and the overhead of GreenBus with OCP-tl1 ports, we compared it to IBM's CC SystemC PLB-models [2]. Table 3 shows some of the measurements (64 Byte line write, 128 Byte maximum fixed length burst, 2/5 kByte arbitrary length burst example).

Table 3: GreenBus performance with OCP-tl1 ports

Transaction size	Transactions per Second			
	undelayed Ack		5 cycle delayed Ack	
	IBM	GreenBus	IBM	GreenBus
64 Byte	31,850	30,120	12,310	16,340
128 Byte	22,620	18,180	6,980	8,790
2 kByte	2,360	1,410	495	605
5 kByte	1,000	560	204	240

As a result of the wrapping overhead in the GreenBus initiator and target ports, the IBM model scales better in terms of burst length, while GreenBus scales better with the slave delaying the data acknoweledgement, since IBM's model updates its state every cylce, while GreenBus is just waiting for an event.

These experiments where carried out on a 2.8 GHz / 512 MB athlon machine under linux 2.6.

8. CONFIGURATION, LOGGING AND DEBUGGING

In order to use GreenBus for architecture exploration, comprehensive configuration and logging support is vital. To this end, configurable properties for GreenBus can be declared either static or dynamic. Static configuration is done using a configuration file, e.g. XML or Java-style properties. Dynamic properties can be modified at runtime using a front-end connected to GreenBus. Typical front-ends include a command line shell and a Java-based GUI for the Eclipse IDE, which currently is under development. Since SystemC 2.1 comes with the `SC_REPORT` logging framework, which provides an easy to use API and useful features such as severity levels, filter rules, and user-registerable callback functions, we decided to extend `SC_REPORT` with transaction recording capabilites. However, the current release of `SC_REPORT` only supports string-based messages. A more sophisticated logging framework for C++ is `log4cxx` [10]. `log4cxx` uses so called *appenders* to export log output to different targets such as files (XML, HTML, raw text), network sockets or the command line.

In GreenBus, we use a combination of `SC_REPORT`, `log4cxx`, and the SystemC Verification Library (SCV) to provide the *superlog* function `gs_log`. This function allows for both string-based message logging and enhanced transaction recording plus data introspection. Depending on the contained data structures, `gs_log` can either print a brief object summary or record transaction data in a database which can be analyzed by visualization tools.

Based on this approach, any IP with a vendor-specific debugging interface can be connected to the `gs_log` framework using wrapper functions. The debug API wrappers can be either passive or active. Passive wrappers are sensitive to debug output of the IP core and forward the received information to the `gs_log` framework. Active wrappers use an `SC_THREAD` to poll the IP core for debug information, using a configurable polling interval or events that can be specified by the user. For example, a `transaction_start` event from the GreenBus router can be used to activate polling.

9. CONCLUSION

In this paper we presented three aspects of our work on transaction level modelling. We gave an overview of several approaches to "generic buses" in SystemC, and identified from commercial bus fabrics the requirements for those approaches. We suggested both data structures and interface API's which need to be standardised in order to make interoperation a reality. We introduce terms for these aspects of our system (atoms, quarks, low level API). Finally, using these interfaces we demonstrated a working system, taking mixed, heterogeneous components and producing a system with high levels of performance and accuracy.

The key advantages of our system are:

1. Clear distinction of standards from user code
2. User and low level API are separated, and the low level API allows efficient user level convenience functions.
3. Clear formalism for abstraction levels.
4. Single "bus accurate" level router (with PV bypass four arbitration) can be used efficiently for PV, PVT BA and CC levels of abstraction. This presents the possibility of automatically generating the bus fabric from a description of the bus features.

10. REFERENCES

[1] Intel Corp. Aztalan TLM bus infrastructure. *Intel Corp.*, 2005.

[2] IBM. IBM PowerPC 405 Evaluation Kit with CoreConnect SystemC TLMs. *IBM Corp.*, 2006. *http://www.ibm.com*

[3] A. Donlin and M. Burton. Transaction Level Modeling : Above RTL design and methodology. *internal OSCI TLM WG document*, February 2004.

[4] A. Gerstlauer, D. Shin, R. Doemer, and D. Gajski. System-Level Communication Modeling for Network-on-Chip Synthesis. *ASP-DAC*, 2005.

[5] F. Ghenassia. Transaction-Level Modeling with SystemC : TLM Concepts and Applications for Embedded Systems. *Springer*, November 2005.

[6] IBM. The CoreConnect Bus Architecture. *IBM*, 1999.

[7] W. Klingauf and R. Guenzel. From TLM to FPGA: Rapid Prototyping with SystemC and Transaction Level Modeling. *Proc. FPT*, 2005.

[8] T. Kogel, M. Doerper, A. Wieferink, R. Leupers, G. Ascheid, and H. Meyr. A Modular Simulation Framework for Architectural Exploration of On-Chip Interconnection Networks. *Proc. CODES+ISSS*, 2003.

[9] ARM limited. AMBA AXI Protocol V1.0. *ARM limited*, March 2004.

[10] C. Arnold M.Catanzariti and Ch. de Vienne. log4cxx Project. *http://logging.apache.org/log4cxx/*, May 2004.

[11] ST Microelectronics. TAC: Transaction Accurate Communication. *http://www.greensocs.com/TACPackage*, 2005.

[12] OCP-IP. Open Core Protocol Specification 2.0. *OCP International Partnership*, 2003.

[13] M. Janssen R. Hilderink and H. Keding. Simple Version of an Abstract Bus Model. *SystemC 2.0 package*, January 2002.

[14] A. Rose, S. Swan, J. Pierce, and J.M. Fernandez. Transaction Level Modeling in SystemC. *OSCI TLM-WG*, 2005.