SAMBA-BUS: A HIGH PERFORMANCE BUS ARCHITECTURE FOR SYSTEM-ON-CHIPS*

Ruibing Lu and Cheng-Kok Koh

School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907-1285
{lur,chengkok}@ecn.purdue.edu

ABSTRACT

A high performance communication architecture, SAMBA-bus, is proposed in this paper. In SAMBA-bus, multiple compatible bus transactions can be performed simultaneously with only a single bus access grant from the bus arbiter. Experimental results show that, compared with a traditional bus architecture, the SAMBA-bus architecture can have up to 3.5 times improvement in the effective bandwidth, and up to 15 times reduction in the average communication latency. In addition, the performance of SAMBA-bus architecture is affected only slightly by arbitration latency, because bus transactions can be performed without waiting for the bus access grant from the arbiter. This feature is desirable in SoC designs with large numbers of modules and long communication delay between modules and the bus arbiter.

1. INTRODUCTION

Due to the high design complexity and time-to-market pressure, designing complex systems on a chip (SoC) usually requires stitching pre-designed IP cores together through various forms of communication links. In addition, the delay of global interconnects becomes the dominating factor of system performance as the feature size of integrated circuits scales down to nanometers. Therefore, the design of high performance global communication architecture becomes the key to successful SoC designs.

Shared-buses are among the most widely used on-chip communication architectures. The main advantages of shared-bus architectures include simple topology, low cost, and extensibility. Several companies have developed their own on-chip bus architectures, such as CoreConnect [1], AMBA [2], and OpenCore [3]. Since only one module can access the bus at any time, the bandwidth of a bus is limited when the number of modules attached to the bus is large. The bandwidth can be improved by a hierarchical bus architecture [1], in which multiple buses are connected with each other through bridges. Studies in [4, 5] propose algorithms to perform bus hierarchy optimization based on communication profiles. However, hierarchical bus architectures may suffer long communication latency for inter-bus communications. Other than those studies that focused on improving bus performance, a split shared-bus architecture is proposed in [6] to reduce power consumption.

In this paper, we propose a high performance bus communication architecture called SAMBA-bus, which is capable of providing, with a *Single Arbitration, Multiple Bus Accesses*. In the SAMBA-bus architecture, the arbiter grants only one module to access the bus as in traditional bus architectures. Each bus segment, defined as the bus connection between two neighbor modules, is used only if it is part of the communication path. The main difference between SAMBA-bus and traditional buses is that idle bus segments can be used by other pending communications automatically without introducing additional arbitration complexity. Therefore, with a single arbitration, multiple bus accesses are allowed to improve both bus bandwidth and communication latency. Experimental results show that the proposed bus architecture can have up to 3.5 times effective bandwidth improvement and up to 15 times latency reduction. Another advantage of the SAMBA-bus architecture is that its performance is much less sensitive to the arbitration latency than that of traditional bus architectures. This is particularly preferable for an SoC design that has large numbers of modules and high communication delay between modules and the bus arbiter.

2. BACKGROUND OF BUS COMMUNICATION ARCHITECTURES

In this section, we review some background of bus communication architectures. Modules connected to a bus are typically divided into two categories: masters and slaves. While a master can initiate a communication transaction, slave modules merely respond to the transactions initiated by masters. Arbitration is required for buses with multiple masters. Commonly used methods include priority based arbitration [2] and time-division multiplexing (TDMA) [3]. Randomized arbitration is also introduced in [7].

Current on-chip bus architectures typically have centralized arbitration. A module seeking for communication must first obtain bus access grant from the bus arbiter. This may introduce long delay, which includes the arbitration delay, the interconnect delay from a module to the arbiter, and that from the arbiter to the module. In an SoC design with a large number of modules, it is hard to avoid long communication delay between the arbiter and every master module. The bus clock period may have to be significantly increased due to the delay introduced by the arbitration process. Pipelined arbitration can be used to reduce the bus clock period. However, it is hard to reduce the communication latency, *i.e.*, the time spent to gain the bus access. In addition to the arbitration delay, the response of slow slaves may also cause severe performance loss, because the bus remains idle while the master is waiting for the response. Split transactions are used to avoid such performance loss. In split transactions, the bus access right of the master is released after the slave module obtains the communication request. The slave module has to initiate a new bus transaction to transfer the response. In the split transactions, the 'slave' module may also have to initiate bus transactions. Therefore, split transactions lead to an increase in arbitration complexity.

*This work was supported in part by NSF CCR-9984553

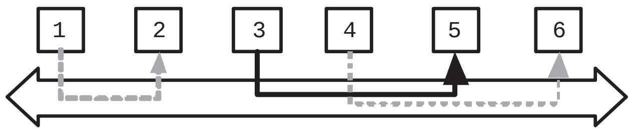

Figure 1: An example of bus communications.

Traditional bus architectures have several limitations. The communication ability of a traditional bus is strongly limited by the arbitration policy that only one module can access the bus at any time. Therefore, as the number of modules increases, the bandwidth becomes more limited. At the same time, one bus transaction consumes all bus resources, resulting in a waste of bus resource. A bus with six modules in Figure 1 for example, has three pending communication requests and the arbiter grants the bus access to the data transfer from module 3 to 5. Clearly, the only necessary bus segment for such a communication is between modules 3 and 5. The propagation of data from module 3 to other parts of the bus disallows the data transfer from 1 to 2. Moreover, the contention and the arbitration latency may prolong the waiting time for a bus access. Furthermore, the implementation of a fast arbiter become more complex as the number of modules on the bus increases and the communication delay between modules and the arbiter becomes large. Although split transactions may avoid the performance loss due to slow responses, arbitration complexity is further increased, leading to increased arbitration delay.

3. SAMBA-BUS ARCHITECTURE

In this section, we proposes the new SAMBA-bus architecture, which has large improvements over both bus effective bandwidth and communication latency. At the same time, it can also reduce the penalties on bus performance due to long arbitration latency. Our bus architecture supports modules that can both initiate bus transactions and respond to bus transactions initiated by other modules. In other words, such modules can function as both masters and slaves. We believe that such modules can reduce the bus traffic and improve the communication performance in SoC designs. The reason is that master-to-master communication can be performed in one bus transaction compared to the two different transactions required in a traditional bus architecture. In addition, it is easier to implement split transactions for such modules. One disadvantage of this approach is the increase of arbitration complexity. As we will discuss later however, the performance of SAMBA-bus is affected only slightly by the arbitration latency. The implementation difficulty of the arbiter can be mitigated by an increase in the arbitration latency. In order to focus on the bus structure, we first assume that the response data to any bus transaction is always available immediately after the slave obtains the communication request. More general cases for slow slaves are considered later.

3.1. Overview

The SAMBA-bus architecture requires that the addresses of modules from one end of the bus to the other end are in increasing or decreasing order. As we shall see later, this assumption is reasonable because not only can the assignment of in order addresses to modules along the bus be easily implemented, but any predesigned out of order module addresses can also be converted to in order addresses with minimal hardware cost.

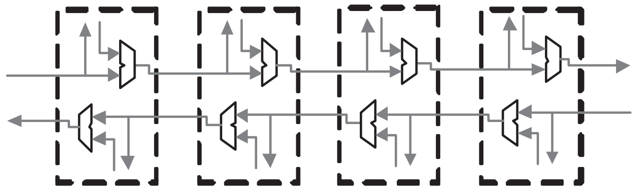

Figure 2: Structure of SAMBA-Bus.

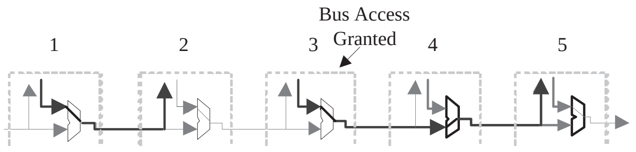

Figure 3: Multiple bus accesses with single arbitration.

The overall structure of SAMBA-bus architecture is shown in Figure 2. It consists of two sub-buses with opposite signal propagation directions. Each of the two sub-buses is used for the transmission of data in one direction. We refer to the sub-bus transferring data from low-address modules to high-address modules as the *forward sub-bus*, and call the other one the *backward sub-bus*. A module is attached to the bus through an interface unit, which can communicate with other interface units through both forward and backward sub-buses.

There are two bus operation phases in each bus cycle: *request phase* and *response phase*. In the request phase, bus transaction initiators send the communication requests to the destination modules on both the forward sub-bus and the backward sub-bus; the destination modules respond to the received communication requests in the following response phase. For any bus transaction, communication request is sent through one sub-bus in the request phase, and the other sub-bus is used for the response data in the following response phase. As the bus architecture is symmetric for the forward sub-bus and the backward sub-bus, we focus only on the bus transactions initiated on the forward sub-bus.

Before a module performs a bus transaction, its interface unit first decides which sub-bus should be used based on the destination address, and an bus access request is sent to the arbiter for that sub-bus. At the same time, it monitors the bus activities and the arbitration result. The arbiter broadcasts the arbitration winner address to all interface units; therefore, every interface unit knows which is granted the bus access. Note that it is possible that one unit is the arbitration winner of the forward sub-bus, while a different one is granted access for the backward sub-bus. As shown in the Figure 2, multiplexers are used to combine all signal sources, and each interface unit has a multiplexer on the forward bus. Through the multiplexer, either the address/data information received from the previous unit or the pending address/data of this unit will be propagated to the next interface unit. *The main difference between the SAMBA-bus architecture and traditional bus architectures is that a unit may access the bus even if it is not the arbitration winner.* Instead, as long as the communication paths of these bus transactions do not have common bus segments, they can be performed simultaneously. When a unit has pending communications for the *forward sub-bus*, it can initiate a bus transaction on the forward sub-bus if any of the following three sets of conditions is met:

1. This unit wins the arbitration and obtains the forward sub-

bus access right.

2. (a) The bus transaction destination of this unit is not after the arbitration winner, *i.e.*, the address of the destination unit is lower than or equal to that of the arbitration winner; and (b) no units before this unit perform bus transactions with units after this unit.
3. (a) This unit is after the arbitration winner; and (b) no units before this unit perform bus communication with units after this unit.

It is obvious that the winner unit can access the forward sub-bus in the first situation. The second situation is for those bus transactions that can be finished in the portion of the bus before the arbitration winner. Condition 2(b) avoids the bus access conflict with transactions from other units before the arbitration winner, whereas condition 2(a) avoids the conflict with the bus transaction from the arbitration winner. The third set of conditions is for those bus transactions performed on the bus segments after the arbitration winner. Since this unit and its communication destination are all after the arbitration winner, there is no need to consider the conflict with the arbitration winner as long as condition 3(b) is satisfied.

Figure 3 shows an example for simultaneous multiple bus accesses in the SAMBA-bus architecture. The bold lines show the data propagation paths in the request phase. Unit 3 is the arbitration winner, therefore, the bus transaction from 3 to 5 should be performed. The destination unit of the pending communication of unit 1 is unit 2, not after the arbitration winner, and no data is being transferred through unit 1. These satisfy conditions 2(a) and 2(b). Therefore, unit 1 can perform its communication on the bus in the same bus cycle.

SAMBA-bus architecture has the following two advantages over traditional bus architectures: high bandwidth and low latency. Bus bandwidth is improved due to the ability to deliver multiple transactions in one bus cycle. The improved bandwidth helps to reduce the communication latency. The reduction of communication latency however, is not only from the increased bandwidth. In traditional bus architectures, long arbitration delay may introduce extra latency to bus communications, even if the bus is idle at that time. The communication latency in the SAMBA-bus architecture is less affected by long arbitration latency, because communications can be performed through automatic compatible transaction detection, without waiting for the bus access grant from the arbiter.

3.2. Interface Unit Implementation

Figure 4 and Figure 5 show the logic functions of bus access controllers, which play a central role in an interface unit. Note that these access controllers are for interface units capable of both initiating and responding to bus transactions. The operation and structure of this kind of bus access controllers can be easily adapted for slave-only or master-only interface units. The functions of bus access controllers include the detection of whether the interface unit is the destination of the received bus transaction, and the determination of what should be propagated to other units. An interface unit should have two access controllers for both forward and backward sub-buses, because one access controller is required for each sub-bus.

Figure 4 shows the logic operation of an access controller for the forward sub-bus in the request phase. In this phase, transaction initiators send communication requests to their destination modules. Signal $fValidOut_{i-1}$ indicates whether the data/address

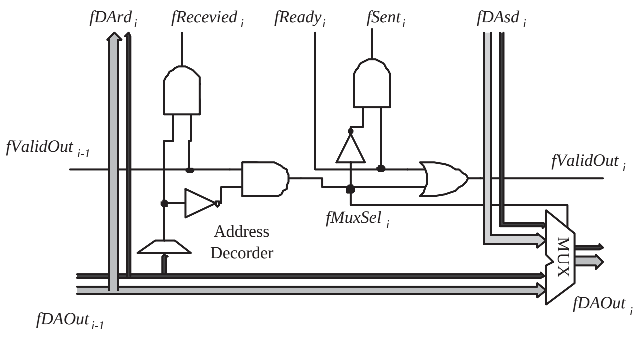

Figure 4: Logic operation in request phase of a forward access controller.

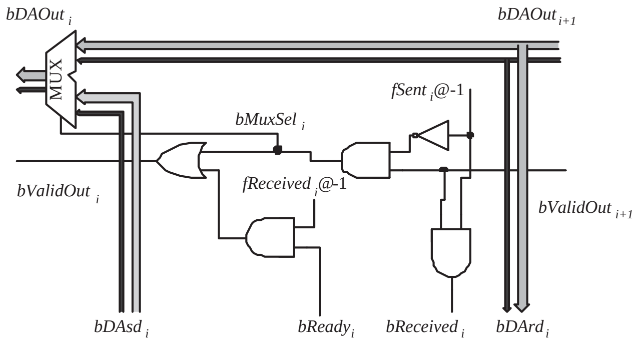

Figure 5: Logic operation in response phase of a backward access controller.

(identified as $fDAOut_{i-1}$) received from the previous interface unit $i-1$ is for a valid bus transaction. The output of the address decoder is asserted if the received address belongs to this unit. Signal $fReceived_i$ informs the attached module whether a bus transaction for this unit is accepted; the signal is asserted when both $fValidOut_{i-1}$ and the output of address decoder are asserted. The data/address received from the previous unit will then propagate to the next unit through $fDAOut_i$ if and only if $fValidOut_{i-1}$ is asserted and the output of address decoder is negated. Signal $fReady_i$ is an internal signal generated by the interface unit. It is asserted in the request phase if (a) the data/address on $fDAsd_i$ is the valid data/address of a pending communication, and (b) the unit is the arbitration winner, a unit after the arbitration winner, or a unit whose transaction destination is not after the arbitration winner. Signal $fValidOut_i$ informs the next interface unit whether the data/address on $fDAOut_i$ is valid, and it is asserted if either $fReady_i$ or $\overline{fMuxSel_i}$ is asserted. Signal $Sent_i$, which is equal to $fReady_i \wedge \overline{fMuxSel_i}$, informs the interface unit whether its pending communication on $fDAsd_i$ is sent.

Figure 5 shows the logic operation of an backward access controller in the response phase. Since the response data of a bus transaction accepted in one sub-bus is always carried in the other one, the operation status of the corresponding forward access controller in the last request phase is used to determine the operation of this access controller. $fRecevied_i$@-1 and $fSent_i$@-1 refer to $fReceived_i$ and $fSent_i$ in the last request phase of the forward access controller, respectively. $bReceived_i$ indicates whether a response for this unit is received, which is equal to $fSent_i$@-1 $\wedge$ $bValidOut_{i+1}$. Here, the assertion of $bValidOut_{i+1}$ means that the corresponding $bDAOut_{i+1}$ is a valid response from an interface unit after unit *i*. The valid response is further propagated to other interface units if $fSent_i$@-1 is negated, which means that interface unit *i* has not initiated any bus transaction on the forward sub-bus in the last request phase. $bReady_i$ is asserted if $fReceived_i$@-1 is negated or $bDAsd_i$ has the response for the accepted transactions. $bValidOut_i$ is asserted if either unit *i* sends out its response or propagates the response from another unit after unit *i*. In the response phase, the logic AND of all *bReady* signals indicates whether all bus responses are generated, and this can be used to extend the response phase if not all bus responses are provided in time.

We have presented the operations of a forward access controller in the request phase and a backward access controller in the response phase. In fact, any access controller is required to work in both request and response phases, because bus transactions are initiated on both sub-buses in a request phase, and they are all responded on both sub-buses in the following response phase. Therefore, an access controller should perform either kind of operations depending on the bus access phase.

The bus clock period is determined by the data/address propagation delay from one bus end to the other end and the arbitration delay. With properly pipelined arbitration, the bus clock period is dominated only by the signal propagation delay on the bus. What an interface unit shown in Figure 4 introduces to the critical path delay of the entire bus is the delay of the path from the address decoder to the multiplexer control, and then to the multiplexer output. Note that the delay introduced by interface units for slave-only and master-only modules are even smaller. The bus clock period should not be affected significantly if the inter-unit bus interconnect delay is much larger than the access controller logic delay. This is true for most current and future SoC designs due to the large gap between global interconnect delay and logic delay. Consequently, the bus clock period of SAMBA-bus should be similar to that of traditional buses.

3.3. Practical considerations

Interface unit addresses: An important basis of the proposed architecture is that the addresses of interface units are in-order. For a completely new system design, the assignment of in-order addresses to interface units can be easily implemented. However, for some designs with fixed or partly fixed bus addresses, their addresses may be difficult to be ordered along the bus. To overcome that, one approach is to introduce in each interface unit a look-up table that maps the out-of-order addresses to their relative positions on the bus. Note that the relative position of interface units are used only in the selection of sub-bus to be used, and in the generation of *Ready* signals of bus access controllers in the request phase. The former introduces delay to the arbitration process, which can be pipelined; the latter is not in the critical bus delay path because it is not dependent on the bus operations of other units, and can be computed as long as the arbitration winner address is available.

Slow response: In the previous sections, we assume that all slave modules can provide the response immediately in the next response phase. However, some systems may have slow modules, which require more time to obtain the response data. Extending the response phase can be used in such a case. Essentially, the response phase takes more clock cycles for the slow slaves to obtain their data. A response phase can be extended until $Ready_i$ for every slave unit is asserted, which means that either the slave unit does not obtain any communication request or it has already provided the response. This method may decrease the bus performance due to very slow modules. The loss of bus performance due to slow slaves can be overcome by using split transactions. Through that, a slow slave may terminate current transaction when it is determined that it will take a long time to get the response data; a new transaction will be initiated to transfer the response data after the slow slave acquires it.

4. EXPERIMENTS AND RESULTS

In this section, we present our experimental results. A simulator for both the SAMBA-bus architecture and a traditional bus architecture is implemented using C++. All modules in the simulated buses can either initiate bus transactions or respond to bus transactions initiated by other modules. We also assume that the response of all bus transactions can be provided in the next bus clock cycle. Two-level TDMA arbitration [3] is used to decide the arbitration winner for both traditional buses and SAMBA-buses.

Bus communication traffic is generated by a method similar to those in [7] and [6]. Several parameters are used to direct the generation of communication traffic. The frequency of bus access requests is controlled by the *inter-communication intervals*. For an interface unit, the inter-communication interval is defined as the number of bus cycles after which a new bus transaction is generated since the previous bus access has been granted. The inter-communication interval is randomly generated following Poisson distribution. The probability that a unit is the destination of bus transactions is assigned based on the distribution of *communication distance*. Here, the communication distance of a bus transaction refers to the number of interface units between the transaction initiator and the destination. Three different communication distance distributions are used: uniform, Poisson, and exponential. In the uniform distribution, a unit has equal probabilities of initiating transactions to all other units. In the other two distributions, an average communication distance of $(Number\_of\_units)/4$ is also used to direct the generation of bus traffic, where $Number\_of\_units$ refers to the number of units attached to the bus.

Figure 6 shows the effective bandwidth and average latency of SAMBA-buses and traditional buses with different number of interface units. Here, the effective bandwidth is the number of finished bus transactions over the total number of bus cycles used. The latency refers to the number of bus cycles an interface unit spends to obtain bus access. In these experiments, we assume that the arbitration latency is 1, which means that an interface unit can be granted with bus access by the arbiter as early as in the next bus cycle after requesting the bus access. The average inter-communication interval is equal to 3. Note that for the traditional buses, the communication distance distribution does not affect the latency or bandwidth; therefore, we provide only one set of results for traditional buses.

In order to perform a fair comparison, we assume that the traditional buses have two sub-buses, each of which can be used independently. Therefore, when the communication traffic on those traditional buses is high enough, the effective bandwidth is always 2.

From Figure 6, we can see that both the effective bandwidth and the average latency are greatly improved for all configurations. Test cases with the exponential communication distance distribution have the highest improvement over both bandwidth and latency. The bandwidth improvement can be as high as 3.5 times, while the latency can be reduced up to by 15 times. The reason is that for exponential distribution, there are more short distance communications, leading to higher possibilities of compat-

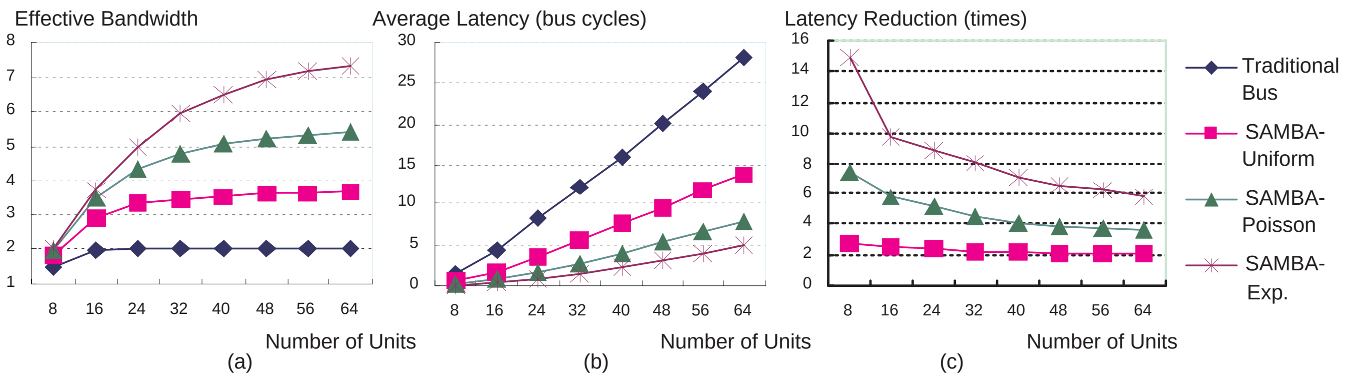

Figure 6: Effective bandwidth and average latency for buses with different numbers of units.

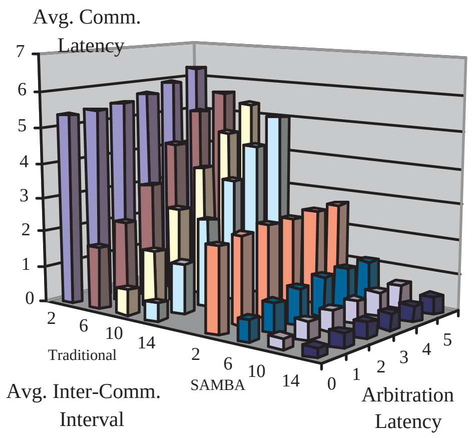

Figure 7: Effect on communication latency of the arbitration latency.

ible multiple transactions. For this set of experiments, the inter-communication interval distribution for every interface unit is the same. As a result, there are more bus traffic and contentions for test cases with more interface units. Another observation is that SAMBA-buses have higher effective bandwidth improvement when the bus access contention is higher. SAMBA-buses still have significant latency reduction, which is as high as 15 times, when bus access contention level is low and bandwidth is not the bottleneck for communication.

Another set of experiments is performed to study the effect on communication latency of bus access contentions and arbitration latency. All the buses under test have 16 interface units, and the communication distance distribution is uniform. We adjust the bus contention level by setting different average inter-communication intervals. The results are shown in Figure 7. The results clearly show that high bus contention level will lead to high communication latency, but the SAMBA-buses always have significantly lower communication latency than the traditional buses with same average inter-communication interval and arbitration latency. Note that these results are acquired on buses with uniform communication distance distribution, and the improvement is even larger under other communication distance distributions. More important, the results also show that the arbitration latency affects the communication latency for traditional buses significantly when the bus contention level is not very high. However, the average communication latency for SAMBA-buses is affected only slightly by the arbitration latency. The reason is that bus accesses can be performed through compatible transaction detection without waiting, typically with long latency, for the bus access grants when the bus access contention level is low. When the bus contention level is high, the latency is largely dominated by bus transaction delivery ability, or bus bandwidth. Therefore, SAMBA-bus architecture can tolerate longer arbitration latency with minor penalty on bus performance. This can alleviate the task of arbiter design.

5. CONCLUSION

A new high performance SAMBA-bus architecture is proposed in this paper. Multiple compatible bus transactions can be performed simultaneously without introducing additional arbitration complexity. Experimental results show that SAMBA-bus architecture can have up to 3.5 times improvement in the effective bandwidth, and up to 15 times reduction in the average communication latency. In addition, the performance of SAMBA-bus architecture is affected only slightly by long arbitration latency. This feature is desirable in SoC designs with large number of modules and long communication delay between modules and the bus arbiter.

6. REFERENCES

[1] IBM. *CoreConnect Bus Architecture*, 1999.

[2] ARM, Limited. *AMBA Specification*, 1999.

[3] Sonics, Inc. *Open Core Protocol Specification*, 1999.

[4] M. Drinic, D. Kirovski, S. Meguerdichian, and M. Potkonjak. Latency-guided on-chip bus network design. In *Proc. Int. Conf. on Computer Aided Design*, 2000.

[5] K. Lahiri, A. Raghunathan, and S. Dey. Efficient exploration of the SoC communication architecture design space. In *Proc. Int. Conf. on Computer Aided Design*, 2000.

[6] C. Hsieh and M. Pedram. Architectural energy optimization by bus splitting. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 21:408–414, April 2002.

[7] K. Lahiri, A. Raghunathan, and G. Lakshminarayana. LOTTERYBUS: A new high-performance communication architecture for System-on-Chip designs. In *Proc. Design Automation Conf*, 2001.