

Transaction-Based Debugging of System-on-Chips with Patterns

Amir Masoud Gharehbaghi^{1,2}, Masahiro Fujita^{1,2}

¹VLSI Design and Education Center, University of Tokyo, Tokyo, Japan

²CREST, Japan Science and Technology (JST), Tokyo, Japan

amir@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp

Abstract— This paper presents a debug method for system communications in post-silicon verification. First, we extract transaction sequences at run-time using on-chip circuits and store them in a trace buffer. Then, we read the stored transactions and analyze them with software. The analysis software tries to find certain patterns in the extracted transactions that are defined by our *transaction debug pattern specification language* (TDPSL). We have also defined a number of standard patterns for common communication problems such as race and deadlock in TDPSL. To show the feasibility of the method, it is applied to a number of on chip buses. It is shown that the area overhead of the method is very low. Also we have implemented the analysis software and shown that it is memory efficient, scalable and effective to find bugs. The proposed method can also be applied to fault analysis including transient faults.

I. INTRODUCTION

Nowadays, modern System-on-Chips (SoCs) have many processors, IP cores and other functional units. The complexity of systems has increased both in functionality of each core and also communication among cores. As a result, complete verification of whole systems before implementation is becoming infeasible; hence there may be some errors in prototypes of systems.

Post-silicon debug is the process of finding the cause of errors in the prototype or initial implemented system. Since modern SoCs have a number of cores, interaction among cores are usually very complex and can be erroneous.

Traditional post-silicon debug methods [1] concentrate on functional parts of systems; as a result they cannot efficiently handle errors related to complicated communications in modern SoCs.

Recently the idea of communication centric debug [2] has been introduced to overcome the complexity of debug of communications in SoCs that are based on Network-on-Chips (NoCs).

Nowadays, most of the large designs begin from higher levels of abstraction than RTL. Transaction level modeling (TLM) [11] concept has been introduced to help the designers to cope with the complexity of systems consisting of both hardware and software parts with complicated communications. Using the TLM methodology, systems are partitioned into computation parts and communication parts. During the system refinement and synthesis, computation parts become functional units, cores or programs running on processors. Communication parts become on-chip

communication lines, buses or networks.

In this paper we introduce a method for debugging communications in SoCs at transaction level. Our method is based on analyzing transactions among IPs/cores in a system to find potential erroneous communication behavior of the system. For example, if before ending a write transaction, a read begins from the same place; or if there are two consecutive write transactions (from different sources) to one place, it may be a bug because changing the order of writes can change the behavior of the system.

We have employed the concepts of requests and responses in TLM [11] to extract the basic transaction elements from control signals following the on-chip buses communication protocols using on-chip monitor circuits. The extracted transaction elements are stored in a trace buffer and are ready to be scanned out for further analysis.

In this paper we present a method to automatically analyze the extracted transactions for potential erroneous transaction sequences. Our method is based on finding special patterns; known as debug patterns; in transaction sequences that may lead to a bug. We have also defined a language for transaction debug pattern specification: TDPSL. The syntax of the language is similar to PSL [15] although it has different semantics mainly due to its parameterized expressions.

To show the effectiveness of the method, we have implemented the software in C++ language. We have specified a number of patterns for common communication problems such as race and deadlock in TDPSL language. We have also analyzed a number of traces with the specified assertions and we could find bugs. We have shown that our analysis method is memory efficient and also scalable. For example we have analyzed more than 100,000 transactions in less than a second with only 96 KB memory usage to find race conditions. Also we have shown that increasing the trace size does not affect the memory usage while the run-time increases linearly.

We have selected OPB [12] and PLB [13] on-chip buses that are part of IBM CoreConnect bus system to study the hardware overhead of the method. We have shown that area overhead of our method is very low, just 1966 gates for OPB bus and 2206 gates for PLB bus (assuming 4 masters and 4 slaves) excluding the trace buffer. Also we have shown that with a small trace buffers, such as 16 KB, it is possible to store more than 13000 transactions.

The main contributions of the paper are as follows.

- Introducing the link between the transactions and post-silicon debug for transaction level designs.
- Defining a language for specification of debug patterns based on transactions.
- Presenting a novel analysis method to find potential erroneous transaction sequences based on our transaction debug pattern specification language.
- Introducing an infrastructure with low hardware overheads to enable on-chip transaction level debug. Also experimental results show the effectiveness of the proposed method.

It must be mentioned that although TDPSL and the analysis method are used for the purpose of post-silicon debug in this paper, they can be also used for verification of TLM designs; since we have employed the concepts and definitions of TLM and abstracted the on-chip signal values to high-level transactions. This issue is not in the scope of this paper and will be addressed in other papers.

Also, according to our analysis method there is no difference between the errors related to transient faults or logical (or design) bugs. Therefore, our method can reveal both kinds of bugs.

The paper is organized as follows. Section II introduces the related work. Section III presents the transaction-based debug method. In Section IV transaction debug pattern specification language is introduced. In Section V implementation of the method is presented. Finally, Section VI concludes the paper.

II. RELATED WORK

Most of the previous works on SoC debug have been focused on adapting the traditional debug techniques to SoCs. [1] and [4] have good reviews of those works. Also it is necessary to change the scan structure to become suitable for SoC designs [5]. All of these works focus on debugging the functionality of SoC without any emphasis on communication and they are different from our method.

Modern SoCs consist of a number of processors/cores with very complicated communication among them; hence, they require new debug methods. [2] proposes such method with emphasis on network-on-chip (NoC) communications and presents the complete NoC debug system in [6]. Also in [7] a probe structure suitable for NoC debug is proposed.

[3] reviews the benefits of diagnosing transaction level models without presenting the relation between transaction level and lower levels. Our work uses the concept of communication debug that is presented in [3] and establishes a relationship between transaction level communications and chip traces for SoC debug.

The idea of using patterns to assist debug, has been previously proposed in [8] and [9] for debugging parallel programs and have been used for SystemC debugging in [10]. We have used the debug pattern concept to introduce a new language that is suitable for specification of debug patterns for communications. Our language is different from assertion languages like PSL [15] and SVA [17] because we

have employed transaction level concepts like [16]. Unlike [16] we have used parameterized expression; hence, it can be seen as a generalization to [16].

In addition, we have used our debug language in transaction level using the trace information that is extracted from the run-time. Therefore, it is different from works like [19] that are debugging transaction level designs in TLM virtual platform. However, our debug method and the proposed language can be used at that level as well.

III. DEBUG METHOD

Transaction level debug requires both on-chip hardware and off-chip software support. As shown in Fig. 1, the system consists of k modules (Module 1 to Module k) communicating with each other through the channel (e.g. on-chip bus). The on-chip part mainly deals with extraction of basic transaction elements and also storing them on an on-chip trace buffer. The software is responsible to read the trace buffer, build the transactions and provide transaction analysis. In this section the details of hardware and software elements that are shown in Fig. 1 are explained.

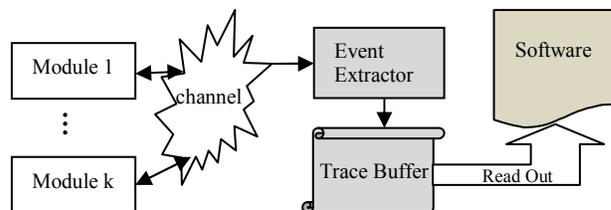


Fig. 1. Debug Infrastructure

A. Transaction Elements

According to TLM [11] terminology, initiator (or master) requests and target (or slave) responds. In this paper, those terms are used interchangeably. In TLM, each transaction consists of a request and a response. Therefore, the following four basic phases exist in each transaction:

1) *Start of Request (SoRq)*: in TLM corresponds to putting the request in the channel by the initiator. In RTL it is usually when the master has taken the bus ownership and begins the transaction, but may be different depending on the on-chip bus protocol.

2) *End of Request (EoRq)*: in TLM corresponds to getting the request by target. In RTL it is the time when slave is selected to start the operation. It may be after the address decoding or by explicit signal activation from bus, depending on the on-chip bus protocol.

3) *Start of Response (SoRp)*: in TLM corresponds to putting the response in the channel by the target. In RTL it is usually when the slave acknowledges end of transfer, but may be different depending on the on-chip bus protocol.

4) *End of Response (EoRp)*: in TLM corresponds to getting the response from the channel by the initiator. In RTL it is usually when the master is acknowledged, but may be different depending on the on-chip bus protocol.

In addition to the above basic phases, two additional

elements are defined in order to handle error conditions:

- 5) *Request Error (RqErr)*: corresponds to any kind of error that causes a request to be ended unsuccessfully after *SoRq*.
- 6) *Response Error (RpErr)*: corresponds to any kind of error that causes a response to be ended unsuccessfully after *SoRp*.

B. Debug Hardware

Debug hardware, as shown in Fig. 1, consists of two main parts: event extractor and trace buffer.

Event Extractor is the heart of the overall transaction debug method and is responsible for extracting the basic transaction elements from the signal events. As explained before, we only need to extract start and end of transactions to build the transaction sequence from the signal events.

The main advantage of extracting only start and end of transactions and error states are as follows. First, we do not need to implement full communication protocol, since the required information can be easily extracted by very simple circuits, as will be shown in Section VI. This way, the area overhead of the on-chip debug hardware will be very low.

Second, the implementation of modern protocols, such as PCI Express, are difficult and also time consuming. Our debug method does not need this process; hence does not impose high extra effort to design and verification engineers.

The disadvantage of not implementing the full protocol is that detecting some erroneous behaviors such as no response in splitted transactions can only be detected by software. In addition, since we rely on error detection mechanism of the bus for detecting errors in requests or responses, if the bus error detection mechanism is erroneous itself, we may miss some bugs.

Trace buffer is a buffer to store the basic transaction elements along with the information on masters and slaves (as shown in Fig. 2). In our methodology, we store the extracted events (*Command* field) with event type (read/write in *RdWr* field) and also the master and slave IDs. We have abstracted the actual target address and save it along with ID in *Slave Addr* field. This way, all the transactions among masters to slaves can be later extracted in software without losing the essential information. *Tag* is a sequence number that indicates the transaction number. It is only used for buses like OCP that allow non-blocking requests and out-of-order responses.

Master ID	Slave ID	Slave Addr	RdWr	Command	Tag
-----------	----------	------------	------	---------	-----

Fig. 2. Trace Buffer Fields

It must be noted that for memory mapped buses, there is no need to store all the address lines (for example 32 bits) which takes a lot of space in trace buffer. Instead we can only store an ID for each master or slave. That is because for most SoCs and embedded systems we know in advance the address mapping of devices, hence with simple address decoding circuits we can save a lot of space in trace buffer, as will be shown in Section VI.

Also we have abstracted the target address of transactions as follows. In addition to the slave ID which is an abstraction of the higher address bits, we store the abstracted information about the lower bits.

The lower bits of address are shown with 3 states that correspond to 2 bits of storage: SAME, SEQ, OTHER; although they may be expanded to have more different values in the future.

SAME specifies that in current transaction, slave address is the same as previous transaction address for this slave. It is mainly useful for detection of some bugs like race, as will be explained in Section IV.C.

SEQ specifies that in current transaction, slave address has one word difference with the previous transaction address for this slave. The difference may be negative or positive. It is mainly useful in the case of DMA transfers or finding sequential access to a slave. SEQ address is not used in this paper because we have not considered DMA transfers yet and it is for future extensions of the work.

OTHER specifies that in the current transaction, the slave address is neither SAME nor SEQ.

To access the trace buffer data it is necessary to read its content. It can be done with any method that is provided on-chip like standard scan-chain or special debug ports and facilities like ChipScope [14] in Xilinx FPGAs. We have not addressed this issue in the paper.

C. Debug Software

Debug software has three main responsibilities. The first responsibility is to merge the basic events that are stored in trace buffer to make full transactions. The second responsibility is to find any potential incomplete transactions. For example requests with no corresponding response or error.

The third responsibility is to verify the assertions that are specified by TDPSL language. This way, verification engineer can further analyze the trace to find and locate the cause of bugs that are related to communication and synchronization problems. The details of the TDPSL language are explained in the next section.

IV. TRANSACTION DEBUG PATTERN SPECIFICATION LANGUAGE

A. Language Syntax

TDPSL syntax is similar to PSL [15] property specification language. It has three layers: Boolean layer, temporal layer and verification layer (Fig. 3).

Boolean layer consists of *trans_exp* that represents the basic elements of transactions, as explained in Section III.A. As each transaction consists of a request and a response, TDPSL allows properties on requests and responses, as well as transactions. All transactions start with a request and end with a response, or error if unsuccessful. Therefore, the start of transaction (*SoTr*) and end of transaction (*EoTr*) are the same as *SoRq* and *EoRp* respectively. Also *ErrTr* represents an error in transaction and corresponds to *ErrRq* or *ErrRp*.

To specify the *master* and the *slave* that are involved in a transaction, we can explicitly specify their *id*, represent them symbolically with a *var*, or leave them as don't care with "-" symbol. *Type* and *address* of a transaction can be also explicitly specified or left as don't care. *Tag* specification is also similar to *master* or *slave* specification.

For example in *SoTr(m1, 1, Rd, -, -)* transaction, “*SoTr*”, “*m1*”, “*1*”, “*Rd*”, “-” and “-” correspond to *trans_type*, *master*, *slave*, *type*, *address* and *tag* correspondingly. Therefore, it represents start of a read transaction from a master (named m1) to slave number 1 with any address and any tag.

Temporal layer is used to define the properties in terms of transaction sequences. The basic element of the temporal layer is *sere_exp* that is similar to SERE (sequential extended regular expression) in PSL. Simple SERE is one transaction element. SEREs can be concatenated with *sere_ops*.

Concatenation operator ";" informally means that the sequence on the right hand side of the concatenation begins the next time after the end of the sequence on the left hand side.

Fusion operator ":" informally means that the begin of the right hand side sequence has overlap with the end of the left hand side sequence.

"|" operator means that at least one of the left hand side or right hand side SEREs must hold at each time.

"&" operator means that both the left hand side and right hand side SEREs must hold all the times.

Repeated SEREs can be defined with repetitions operators. Consecutive repetition operator "*" defines that after end of a SERE, another repetition of the SERE begins without any other transaction between them.

Non-consecutive repetition operator "=" defines that after end of a SERE, another repetition of the SERE begins, but there may be other transactions between them.

Repetitions can be exactly a number of times, or within a range. If no *count_exp* is defined, "*" and "=" mean zero or more times, and "+" means one or more times.

Three simple foundation language (FL) operators are defined to express a behavior for a *temporal_exp*.

The *always* operator means that the temporal expression must hold at all the times starting from the present time. The *never* operator means that the temporal expression must never hold. The *eventually* operator means that the temporal expression must hold at the current time or some future time.

In the verification layer of the language, only *assert* statement is defined. The optional *filter_exp* specifies a filter over the execution path for the evaluation of the assertion statement. Filter expression is explained in details in the next section.

```

/***** Boolean Layer *****/
trans_exp ::=
  trans_type "(" master " " slave " " type " " address " " tag ")";
trans_type ::=
  "SoRq" | "EoRq" | "SoRp" | "EoRp" | "ErrRq" | "ErrRp" |
  "SoTr" | "EoTr" | "ErrTr";
master ::= var | id | "-";
slave ::= var | id | "-";
tag ::= var | id | "-";
type ::= "Rd" | "Wr" | "-";
address ::= "SAME" | "SEQ" | "OTHER" | "-";
var ::= IDENTIFIER;
id ::= NUMBER;

/***** Temporal Layer *****/
fl_exp ::= ("always" | "never" | "eventually") temporal_exp;
temporal_exp ::= seq_exp | temporal_exp t_op temporal_exp;
t_op ::= ">" | "&&" | "||";
seq_exp ::= sere_exp | "{" sere_exp "}" | seq_exp repeat_exp;
repeat_exp ::= repeat_op [ count_exp ];
repeat_op ::= "*" | "+" | "=";
count_exp ::= NUMBER [ "." (NUMBER | "inf") ];
sere_exp ::= trans_exp | sere_exp sere_op sere_exp;
sere_op ::= ";" | ":" | "|" | "&";

/***** Verification Layer *****/
property_exp ::= "assert" fl_exp [ filter_exp ];
filter_exp ::= "filter" "(" masters " " slaves " " types ")";
masters ::= "*" | "-" | id_list;
slaves ::= "*" | "-" | id_list;
types ::= "*" | type;
id_list ::= id | id_list "&" id;

```

Fig. 3. The syntax of TDPSL

B. Language Semantics

The semantics of the TDPSL language is based on the definition of time as the order of occurrence of transactions and also pLTL logic that is LTL logic with parameters [18].

In transaction-level modeling, usually there is no explicit synchronization signal; that is known as clock in RTL. In addition, the concept of time is somehow abstracted to the order of occurrence of transactions; because, the order of transactions is enough for high-level functional verification and there is no need to keep track of the real time except for performance or detailed timing verification purposes.

Therefore, we have abstracted the timing information of transactions to their orders of occurrences. Consequently, two transactions may begin (or end) at the same time, or one of them after the other one. Also, there may be other transactions between them, or they may occur consecutively.

This is one of the main differences between our language and languages like PSL or SVA which consider a triggering event (clock) for evaluation of their temporal expressions.

One of the most important aspects of TDPSL is that it allows parameters in transactions. This way, a TDPSL formula can be parameterized; hence it can be seen as a generalized form of its equivalent PSL formula. The parameters in TDPSL expressions are assigned to the actual values when they are being evaluated by software. The evaluation and parameter assignment procedures are explained later in this section.

The formalism behind the parameterized TDPSL expressions are taken from pLTL logic [18] which extends

LTL logic with variables and quantification. The pLTL logic is superset of the LTL logic. In the special case when all the variables are assigned, parameterized LTL (pLTL) becomes the classic LTL logic.

The evaluation of a TDPSL assertion is similar to its syntactically equivalent PSL assertions; except for the variable assignment. If we consider that the assignment of the variables are done and the order of transactions is used as a virtual clock for PSL assertions, it is possible to verify the assertion with an automaton [20].

For evaluation of TDPSL assertions, a *parameterized automaton* [18] is created. A parameterized automaton is similar to an ordinary automaton, but it additionally passes the assignment information for the variables to the next states through the transition function. This way, assignment of variables can be done at run-time. For simplicity, a parameterized automaton can be seen as an automaton that replicates its states according to the domain of variables at run-time.

One of the issues that are not discussed yet is the definition of *filter_exp* in properties. The filter expression can be used to filter some of the unwanted transactions in a trace. This way, a partial order trace can be created from the total order trace that is more suitable for verification.

Definition of the filter also helps the verification engineers to specify the required assertions easier, because they do not need to worry about the transactions between other devices that may occur in parallel.

Filters can be defined over masters, slaves and transaction types. Filters can be defined by explicitly naming the masters, slaves or transaction types. "-" means no filter and "*" means that only consider related masters, slaves or transaction types. "*" is the default filtering mode when it is not mentioned. By "related" we mean only the items that are currently being evaluated for this assertion.

For example consider the following simple assertion:
`assert never SoTr(m1, s1, -, -, -) ; SoTr(m1, s2, -, -, -)`

Also, consider that the trace that the above assertion is being evaluated is as follows:

`SoTr(1, 1, ...);SoTr(1, 2, ...);SoTr(2, 1, ...);SoTr(1, 3, ...)`

Under the default filter or *filter(*, *, *)*, the above assertion fails 2 times, whereas without filtering, or *filter(-, -, -)*, the above assertion fails 1 time. The difference is due to filtering of *SoTr(2, 1, ...)* transaction with default filtering mode. The default filtering mode eliminates the transaction from master number 2 because the above assertion uses only one master (m1); therefore, when the assertion is being evaluated for m1=1, other masters such as number 2 are unrelated and should not be considered.

C. Example Debug Patterns

In this section we specify some of the common synchronization problems; namely race, deadlock and livelock; in TDPSL.

One of the race conditions is that two masters are writing to the same place with overlapped or consecutive transactions. That is because changing the order of

transactions may change the result. As shown in Fig. 4, one the following situations may happen:

- 1) *One write transaction to the same place occurs during the previous write*
- 2) *One write transaction to the same place occurs just after the end of the previous write*

```

assert never
  SoTr(m1, s1, Wr, -, t1) ; SoTr(m2, s1, Wr, SAME, t2) ;
  EoTr(m1, s1, Wr, SAME, t1)
  | EoTr(m1, s1, Wr, -, -) ; SoTr(m2, s1, Wr, SAME, -)
  filter (*, *, *)

```

Fig. 4. Specification of Race Condition in TDPSL

Another situation that may occur in concurrent systems is deadlock. Deadlock happens when two or more processes are waiting for the other processes to release shared resources. In this example we consider the case of two processes, but it can be easily generalized to more processes. Usually there is a semaphore that both processes access. Therefore, accessing a semaphore is equivalent to accessing the same address by different masters. The simple scenario for deadlock between two masters and two semaphores are as follows. It is also shown in Fig. 5.

- 1) *Master1 locks first semaphore.*
- 2) *Master2 locks second semaphore*
- 3) *Master1 waits for second semaphore*
- 4) *Master2 waits for first semaphore*
- 5) *Steps 3 and 4 are repeated*

```

assert never
  EoTr(m1, s1, Rd, -, -) ; EoTr(m1, s1, Wr, SAME, -) ;
  EoTr(m2, s2, Rd, -, -) ; EoTr(m2, s2, Wr, SAME, -) ;
  { EoTr(m1, s2, Rd, SAME, -) ; EoTr(m2, s1, Rd, SAME, -)
  | EoTr(m2, s1, Rd, SAME, -) ; EoTr(m1, s2, Rd, SAME, -)
  } [+]
  filter (*, *, *)

```

Fig. 5. Specification of Deadlock in TDPSL

Livelock is some kind of deadlock that two or more processes want to access shared resources that are locked by the other processes, but unlike deadlock, they release the locked resources to allow the other processes to proceed. The simple scenario for livelock between two masters and two semaphores are as follows. This case is more complicated than deadlock because more transactions should be considered to detect it. It is shown in Fig. 5.

- 1) *Master1 locks first semaphore.*
- 2) *Master2 locks second semaphore*
- 3) *Master1 waits for second semaphore*
- 4) *Master2 waits for first semaphore*
- 5) *Master1 unlocks first semaphore*
- 6) *Master2 unlocks second semaphore*
- 7) *Steps 1 to 6 are repeated*

```

assert never
{ EoTr(m1, s1, Rd, -, -); EoTr(m1, s1, Wr, SAME, -);
  EoTr(m2, s2, Rd, -, -); EoTr(m2, s2, Wr, SAME, -);
  { EoTr(m1, s2, Rd, SAME, -); EoTr(m2, s1, Rd, SAME, -)
  | EoTr(m2, s1, Rd, SAME, -); EoTr(m1, s2, Rd, SAME, -)
  }
  EoTr(m1, s1, Wr, SAME, -); EoTr(m2, s2, Wr, SAME, -)
} [+]
filter (*,*)

```

Fig. 6. Specification of Livelock in TDPSL

V. IMPLEMENTATION

A. Debug Hardware

We have selected OPB [12] and PLB [13] on-chip buses that are parts of IBM CoreConnect bus system for hardware implementation.

The on-chip peripheral bus (OPB) is a fully synchronous bus. It supports single transfer as well as DMA transfer. The transfer protocol is simple and does not allow splitting.

The processor local bus (PLB) is a high-performance bus. It supports address pipelining, split-bus transaction and overlapped read and write transactions to increase the bus bandwidth.

The on-chip extractors for OPB and PLB buses are coded in VHDL hardware description language, and they are 140 and 158 lines of code respectively. The code is written using generic parameters for number of masters and slaves, and it works for any arbitrary number of masters and slaves. After reading and understanding the bus specifications, it took just a few hours to write the on-chip extractors for each bus.

The on-chip extractors are synthesized with Xilinx ISE 10.1 assuming 4 masters and 4 slaves are attached to buses. It takes only 1966 gates for OPB and 2206 gates for PLB bus. Assuming a medium size design with 100,000 gates, the area overhead becomes about 2% which is very low. Obviously, for larger designs with millions of gates the area overhead becomes negligible.

It must be mentioned that about 1700 gates are for abstracting the slave address to 2 bits of SlaveAddr field; assuming that address bus is 32 bits and slave address size is 24 bits. Therefore, most of the area overhead (about 86% for OPB and 77% for PLB) is for SlaveAddr field that is used for analysis purpose. Event extractor takes only a very small number of gates (253 gates for OPB and 502 gates for PLB).

Because the extraction of transaction elements is done only for some phases of bus and that information can be usually obtained by checking a few signals, changing the bus does not affect the on-chip extractor overhead so much and it is very low compared to the overall design.

Assuming that there are 4 masters and 4 slaves, each entry of the trace buffer becomes 10 bits (2 bits for master ID, 2 bits for slave ID, 2 bits for slave address, 1 bit for rd/wr, 3 bits for command and 0 bits for tag). Therefore, by allocating a 16 KB memory block, we can save more than 13000 transaction elements in a trace buffer.

The trace buffer entry fields are independent of bus type. Therefore, changing the bus does not change its format;

although the number of bits for each field may change.

B. Debug Software

Debug software is implemented in C++ language and Microsoft Visual C++ environment. It is about 1500 lines of code. The software accepts the assertions that are specified in TDPSL language and verifies them against a trace file. In case of a bug, software prompts the user with the current variable assignments and also the path (transaction sequences) to the buggy state.

In the current implementation, Boolean layer and verification layer are fully supported. From temporal layer, concatenation (;) and disjunction (||) SERE operators, all kinds of repeat expressions including consecutive and non-consecutive, and all FL operators including *always*, *never* and *eventually* are implemented. That is quite a reasonable subset and large enough for our experiments. The remaining temporal layer operators including implication (->), conjunction (&&) and disjunction (||) temporal operations, and also SERE fusion (:) and conjunction (&) operators are not implemented yet; but they can be easily added.

For our experiments, we have implemented a sample design with communication behavior like dining philosophers. The design has 5 masters (philosophers) and 5 slaves (chopsticks). The masters and slaves are divided into 2 groups. One group has 2 masters and 2 slaves, and the other group has 3 masters and 3 slaves. Both groups communicate in parallel. We have generated the trace file during the simulation.

We have verified the assertions that are shown in Section IV.C and the results for run-time and memory usage, on a PC with Microsoft Windows XP operating system with Intel Core2Duo 2.4 GHz processor and 2 GB memory, are shown in Table 1.

As shown in Table 1, the memory usage of the assertion checker is very low and it is independent of the trace size. The memory usage of checker depends on the number of transactions and concatenation (or fusion) SERE operators that correspond to the number of states and transitions in the resulting automaton, respectively. According to our implementation, all repetition operators are translated to some counters. Also other temporal operators become evaluating some logical expressions; hence do not add additional states or transitions in the resulting automaton.

As shown in Table 1, the run-time has increased linearly with the trace size. To evaluate the scalability of the method, we created a system with 10 masters and 10 slaves and created a trace file with 1,000,000 transactions. The assertion evaluation time was about 13 seconds for assertion in Fig. 6. Therefore, the method is scalable and can be used for systems with very large number of transactions.

For each transaction, we evaluate all the automata states in parallel to determine which transitions are enabled. Also, according to the communication behavior of system, binding of variables may differ; hence different number of transitions may become active at each time. Therefore, the run-time depends on both the assertion and also the

communication behavior of system.

To show the effectiveness of the method in finding bugs, we have implemented two versions of dining philosophers: one deadlock free version, and the other with potential deadlock. Our method found deadlock in the erroneous version, while no deadlock was reported in the correct version.

Table 1. TDPSL Assertions Analysis Results

Trace File Size	100,000 transactions	1000,000 transactions
Assertion in Fig. 4	Run-time: .81 s	Run-time: 8.08 s
	Memory: 96 KB	Memory: 96 KB
Assertion in Fig. 5	Run-time: 1.32 s	Run-time: 16.52 s
	Memory: 112 KB	Memory: 112 KB
Assertion in Fig. 6	Run-time: 1.38 s	Run-time: 13.77 s
	Memory: 124KB	Memory: 124KB

VI. CONCLUSIONS

With the increasing complexity of systems, the number of bugs in implemented systems has increased. This is because none of the current static or dynamic verification techniques can handle complexity of modern systems. As a result, post-silicon verification and debug techniques have become more important than before.

This paper addresses the post-silicon debug problem for complex SoCs. We have introduced a method to raise the debug abstraction level for communication parts of systems to transaction level. We have focused on communications in transaction level because in modern SoCs there are complex communications that may be erroneous and also currently many complex designs begin from transaction level that is higher than RTL.

We have introduced the basic elements for transaction level debug and their correspondence with actual signals in design. Furthermore we have presented an infrastructure and also the required hardware and software support for our debug method. Also we have presented TDPSL; a transaction debug pattern specification language; and shown its usefulness for finding bugs related to common communication and synchronization problems such as race, deadlock and livelock.

We have shown the full implementation of the proposed method for OPB and PLB buses that are part of IBM CoreConnect bus system. Also the methodology can be simply extended to more complex buses.

Our future work includes extending the work for DMA transfers, automating extraction of bus protocols, enhancing the method for performance analysis, and also using TDPSL for verification of general TLM designs at transaction level.

REFERENCES

[1] A.B.T. Hopkins and K.D. McDonald-Maier, "Debug Support for Complex Systems on-Chip: A Review", IEE Proceedings Computers and Digital Techniques, Vol. 153, No. 4, July 2006, pp. 197-207.

[2] K. Goossens, B. Vermeulen, R. van Steeden and M. Bennebroek, "Transaction-Based Communication-Centric Debug", International Symposium on Networks on Chip, NOCS'07, Washington, DC, USA, May 2007, pp. 95-106.

[3] M. Abramovici, K. Goossens, B. Vermeulen, J. Greenbaum, N. Stollon and A. Donlin, "You Can Catch More Bugs with Transaction Level Honey", International Conference on Hardware/Software Codesign and System Synthesis, CODES/ISSS'08, Atlanta, GA, USA, Oct. 2008, pp. 121-124.

[4] A. Mayer, H. Siebert and K.D. McDonald-Maier, "Boosting Debugging Support for Complex Systems on Chip", IEEE Computer, Vol. 40, Issue 4, Apr. 2007, pp. 76-81.

[5] X. Liu and Q. Xu, "On Reusing Test Access Mechanisms for Debug Data Transfer in SoC Post-Silicon Validation", Asian Test Symposium, ATS'08, Sapporo, Japan, Nov. 2008, pp. 303-308.

[6] K. Goossens, B. Vermeulen and A. Beyranvand Nejad, "A High-Level Debug Environment for Communication-Centric Debug", Design, Automation and Test in Europe Conference and Exhibition, DATE'09, Nice, France, April 2009, pp.202-207.

[7] S. Tang and Q. Xu, "A Debug Probe for Concurrently Debugging Multiple Embedded Cores and Inter-core Transactions in NoC-Based Systems", Conference on Asia and South Pacific Design Automation, ASP-DAC'08, Seoul, Korea, Jan. 2008, pp. 416-421.

[8] D. Kranzlmuller, N. Stankovic, and J. Volkert, "Debugging Parallel Programs with Visual Patterns", IEEE Symposium on Visual Languages, Tokyo, Japan, Sep. 1999, pp. 180-181.

[9] S. Shende, J. Cuny, L. Hansen, J. Kundu, S. McLaughry and O. Wolf, "Event and State-Based Debugging in TAU: A Prototype", Symposium on Parallel and Distributed Tools, SPDT'96, Philadelphia, Pennsylvania, USA, May 1996, pp. 21-30.

[10] F. Rogin, E. Fehlauer, C. Haufe and S. Ohnewald, "Debug Patterns for Efficient High-level SystemC Debugging", Design and Diagnostics of Electronic Circuits and Systems, DDECS'07, Krakow, Poland, Apr. 2007, pp. 1-6.

[11] OSCI TLM 2.0 User Manual, OSCI, 2008, available online at: <http://www.systemc.org>

[12] On-Chip Peripheral Bus Architecture Specifications, IBM, 2001

[13] Processor Local Bus Architecture Specifications, IBM, 2007

[14] ChipScope Pro Software and Cores User Guide, Xilinx, available online at: <http://www.xilinx.com>

[15] IEEE Std 1850-2005, IEEE Standard for PSL: Property Specification Language, IEEE Press, USA, 2005

[16] W. Ecker, V. Esen, M. Hull, "Requirements and Concepts for Transaction Level Assertions", International Conference on Computer Design, ICCD'06, San Jose, California, USA, Oct. 2006, pp. 286-293.

[17] IEEE Std 1800-2005, IEEE Standard for System Verilog: Unified Hardware Design, Specification and Verification Language, IEEE Press, USA, 2005

[18] V. Stolz, "Temporal Assertions with Parametrised Propositions", Runtime Verification Workshop, RV 2007, Vancouver, Canada, Mar. 2007, in Lecture Notes in Computer Science (LNCS), Vol. 4839, pp. 176-187, 2007, Springer-Verlag.

[19] T. Kogel, M. Doerper, T. Kempf, A. Wiefierink, R. Leupers, H. Meyr, "Virtual Architecture Mapping: a SystemC Based Methodology for Architectural Exploration of System-on-Chips", International Journal of Embedded Systems, Vol. 3, No. 3, Sep. 2008, pp. 150-159.

[20] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal, "FoCs: Automatic Generation of Simulation Checkers from Formal Specifications", International Conference on Computer Aided Verification, CAV 2000, Chicago, Illinois, USA, Jul. 2000, in Lecture Notes in Computer Science (LNCS), Vol. 1855, pp. 538-542, 2000, Springer-Verlag.