MPSoC verification using a unified random program approach

Methodology, tool and case-study

Jayram Moorkanikara Nageswaran*
Philips Research, Eindhoven

Ronald Bos
Philips Research, Eindhoven

ABSTRACT

This paper discusses a simulation-based verification approach for Multiprocessor Systems-on-Chip using a unified random program generator. Similar to design abstration, we first explain the concept of verification abstraction. Then we analyse the typical bugs encountered in a MPSoC design and levels of verification abstraction at which they can be found. Based on this analysis, we derive an unified approach for generating random MPSoC test program with a single architecture specification, and test constraint specification. The main idea is to use an unified random program generation approach which covers different levels of verification abstraction and allows to trigger bugs mainly related to interconnection network and cache-coherence logic. The approach explained in this paper was applied in the verification of cache-coherence logic in Wasabi media MPSoC at Philips Research, and helped us to discover many bugs not discovered by other approaches. The tool has been used for the verification of different MPSoC RTL models, SystemC models because of its configuration and abstraction features.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

Keywords

Hardware Verification, Random Program Generation, MPSoC

1. INTRODUCTION

Verification of a System-on-Chip (SoC) is considered by the industry as one of the most important and a critical phase in an SoC development project. Verification ensures that the SoC complies with its specification, both in terms of functionality and performance. Many surveys of industrial projects [3] showed that more that 60-70 % of the development time is spent on functional verification, debugging and validation. Also more than 50 % of the total human resource is spent on the verification phase alone. Thus new advances in verification methodology are crucial for surviving the SoC revolution [6].

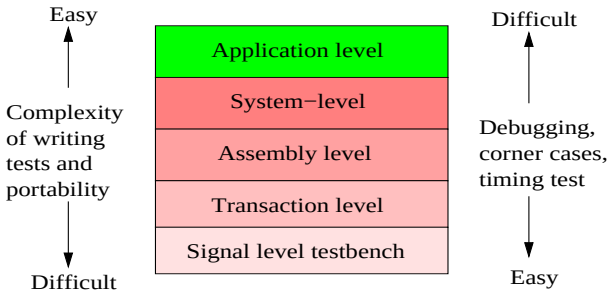

Figure 1: Various levels of test abstraction for verification

In simulation-based verification methodology, verification engineers create testbench which consists of a large body of tests (referred as test programs or test vectors). Each test is applied on the system-under-test (SUT) and its result is checked for compliance with the system specification. Similar to design abstraction, test programs can be specified at various levels of abstraction with respect to the underlying hardware. As shown in Fig. 1, a signal-level RTL testbench written in Verilog or VHDL language operates at the lowest level of abstraction (actually, no abstraction), directly manipulating the hardware signals, and checking for the correctness of the response. The transaction-based verification(TBV) [8, 12] methodology increases the level of verification abstraction by grouping low-level test sequences into logical system-level 'transactions' of a particular type. The system-level testbenches written in C/C++ tests the SUT at even higher levels of abstraction and are usually employed to verify higher level properties of the system (cache coherency, cache consistency etc).

As shown in Fig. 1, when the level of verification abstraction is increased, it becomes easier to write, generate and port the test programs to different systems. At the same time, it becomes harder to create a particular timely intricate test pattern (usually called a corner case) to trigger difficult bugs, and also harder to debug the design when the verification abstraction level is increased. Therefore when the verification abstraction is increased, larger number of

*Currently working for his PhD at University of California, Irvine

MTV '06 Austin, Texas USA
.

simulation cycles need to be spent in order to increase the probabilty of triggering the corner cases bugs. Similar result was reported in [2].

At each level of abstraction, test could be either a directed self-test or a constrained random test. Directed self-test are usually hand written to test a particular functionality of the design based on the design specification. However, as the complexity of the SUT increases it becomes hard to cover all the intricate cases using directed approach alone. In most cases constrained randomization is the key technique to automate the test process and cover large state-space of complex MPSoC. We remark that since verification at high level of abstraction requires a larger number of tests, randomization becomes particularly suitable as it allows to generate them automatically.

As SoC design progressesfrom uni-core towards multi-core systems, the verification complexity increases and imposes new challenges. Some of the major issues which need to be addressed in the verification of programmable MPSoC systems are verification of the cache-coherency and memory consistency properties, and guaranteed absense of race-conditions resulting in deadlocks, live-locks and starvation[9].

.

In this paper we present a tool and approach that mainly targets verification of cache-coherency properties, deadlock and race-conditions of MPSoC. The tool is called Reconfigurable Offline Multi-processor Random Program Generator (or, ROMRPG). The tool can be configured to generate self-checking MPSoC test programs which can be run on different platforms at different stages of verification.

The paper is structured as follows. We first discuss related works in Section 2. Section 3 provides the motivation for the development of the ROMRPG tool with a short description of a generic MPSoC architecture, and summarize some typical kinds of error scenarios (or bugs) which are encountered in practical design projects. Section 4 describes the overall structure of the ROMRPG. Section 5 presents a case study showing howthe tool has been applied for MPSoC verification. Section 6 presents the algorithm used in the tool and some results obtained with the tool. Finally, Section 7 discuss the conclusions, and future works related to this paper.

2. RELATED WORKS

Verification of uniprocessor architecture by means of random program generation (RPG) at processor instruction level was presented by [11]. The approach presented in [11] is very useful for verifying the processor pipeline, bus interface, and L1 cache interface. An attempt to extend this approach for MPSoC has been presented in [16]. This extended approach can be applied only on specific architectures and is not a generic approach. Another tool called ARCHTEST [7] could be used to verify the memory consistency properties of multiprocessor or multicore system. This comes under the category of system-level test-bench (Fig. 1) and is useful only in the latter stages of verification to check cache-consistency properties[1]. Wood et. al [20] also present a random test generation approach for verifying multiprocessor cache controller. This approach is suitable for assembly level verification of multiprocessor, and does not provide ways to configure the MPSoC architecture and constrain the test generation. Transaction-level verification approaches [5, 8] are mainly suitable for block-level verification, and cannot be easily applied for MPSoC cache-controller verification (which is one of the main goals of this paper). A Trace-driven validation methodology for MPSoCs is presented in [4] which can be used along with our approach to validate programs with true data-sharing. To the best of our knowledge none of previous approaches address the problem of holistic verification scheme for targeting different levels of abstraction.

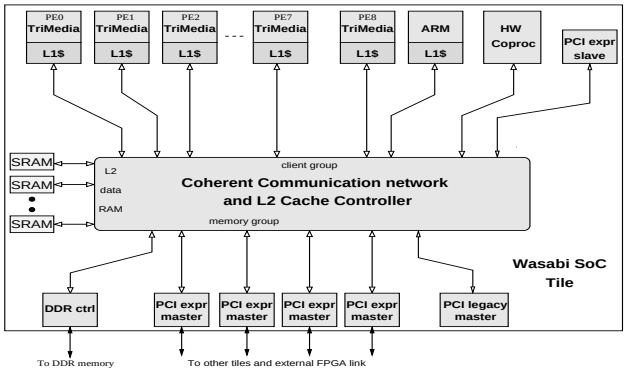

Figure 2: Philips Wasabi media CMP: An example of an MPSoC architecture

3. MOTIVATION

Different verification tools are applied in the real-world projects to verify designs at each level of abstraction (like block, processor or system level etc) In our case, during the verification of the Wasabi CMP project [18], three verification teams were implementing different verification tools each addressing different level of abstraction. One team was applying the concept of e-verification components[5], another team implemented a low-level traffic generator controlled by a script, and another team was doing system-level verification by incorporating the complete system (including cache-coherent processors). We observed that the underlying mechanism used by different teams were almost similar (i.e. trying to check for coherency violations, or checking various protocol issues). The only difference was in the input language, and some specific features of the applied tool. It would make more sense to use a single flow which can generate test for each abstraction level by generating the test program in the required format from a generic trace format. Also in all previous approaches[20, 16] customizations and configurations of the tool for the intended SoC architecture were always embedded within the test code, which made understanding and re-using the approach for verifying other MPSoC quite cumbersome.

The approach which used in this paper is motivated by our experiences obtained during the verification of the Philips Wasabi media processor [18]. Wasabi is a shared-memory multiprocessor (shown in Fig. 2) and consists of a number of processing elements (PEs), which are typically programmable media-processors (TriMedia media CPUs [17]) and ARMs and few hardware coprocessors (for graphics acceleration, image scaling etc), all connected with each other, and to the system's L2 cache via a high performance interconnection network (HPIN). The chip supports hardware cache-coherency or memory-coherency protocol[9], which ensures that every processor receives the latest content of the

memory location even if the content resides in another processor's private L1 cache. In general, any memory operation performed by a (co-) processor PE, which cannot be served by the PEs L1 cache results in a transaction (usually a series of transaction) serviced by the HPIN.At any cycle eachPEcan initiate a new transaction, and each transaction takes multiple cycles for completion. Furthermore, since PEs are able to issue multiple outstanding transactions, the number of transactions being concurrently serviced by HPIN is quite large (up to 20-30). Under many situations the newly issued transactions can be dependent on the behaviour of those already under execution. These situations will be referred to as "corner cases", "transition cases" or "race conditions". Such cases should be extensively tested during the verification process. Typical class of verification targets are summarized below.

- Deadlocks: Though the actual underlying protocol used by the interconnection network for communication and cache-coherence can be guaranteed to be deadlock free, deadlock can still arise due to various implementation decisions. This is a recurrent problem that needs to be specifically addressed in the verification approach.
- Memory coherency violations: If we look into the implementation of a HPIN for many SoCs[18], large number of bufiers or queues are employed at different parts to increase the concurrency, and thus the throughput of the chip. If the order of the transaction response or the selection of the right buffer is altered then memory coherency breaks, resulting in incorrect execution of the program.

We remark that in some cases formal verification have been applied by other authors to identify potenetial deadlocks. But for a large design such as HPIN with many complex interacting FSMs, it is hardly feasible due to state explosion of the composite systems. Therefore we have chosen for simulation-based verification. Due to complexity of the Wasabi SoC, it is not feasible to explicitly trigger all the corner cases by means of directed tests. These observations have motivated us to develop a tool for simulation-based verification which allows to trigger many corner cases implicitly and automatically. Also nowadays MPSoC comes in various flavour with different cache configuration, cache size, coherency mechanism, interconnection type etc. Hence we implemented the idea of single, separate architecture specification and a separate user-defined constraints to drive test at different level of abstraction and targeting various MPSoC architectures. This systematic flow and approach for addressing the verification of MPSoC by using random program generation is the main contribution of this paper.

4. ROMRPG: ORGANIZATION AND COMPONENTS

In the previous section we gave a brief description of the CMP architecture, and typical error scenarios which could happen in the implementation of such SoC architecture. Error conditions arise when the HPIN or the *PE*s are unable to correctly handle the incoming transactions. These error conditions are triggered by the ROMRPG tool. ROMRPG is designed to be a customizable, general purpose tool to automate the process of verification of a broad classes of MPSoCs.

The ROMRPG tool (represented in Fig. 3) takes a memory

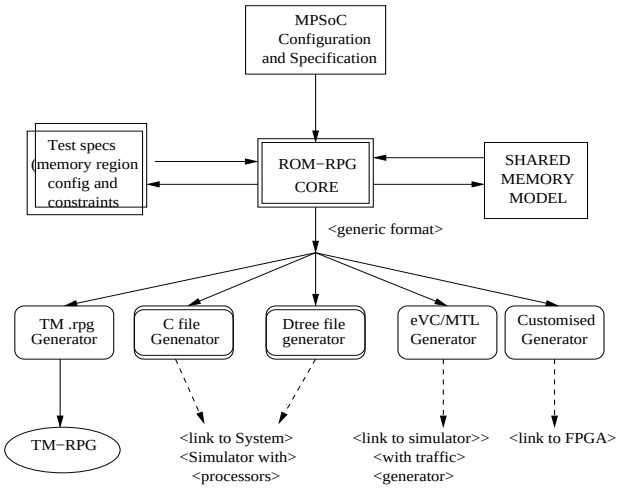

Figure 3: ROMRPG tool components

region specification, user-defined constraints, a MPSoC architecture specification, and generates generic traffic trace. The traces are used by the *PE*s to initiate appropriate transactions. Based on the settings of the tools it is possible to generate self-checking programs, or rely on the run-time protocol checker [14] to detect the protocol violations. Self-testing programs can only be generated for deterministic memory transaction, where it is possible to predict the correct outcome of a memory transaction during program generation phase. Self-checking programs are better than run-time HW checkers for simulation or validation scheme, where implementing a run-time protocol checker is infeasible (like in emulators or prototype silicon). A very simple example to illustrate the kind of output generated for a three CPU system is shown in Fig. 4. Each of the *PE*s or traffic generators

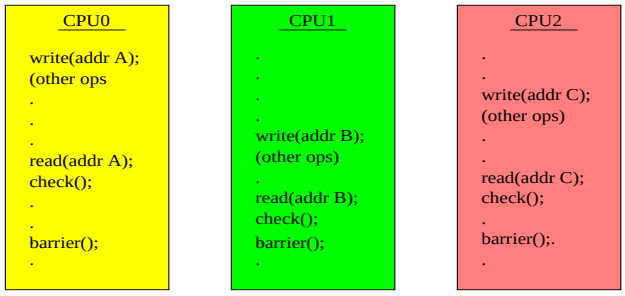

Figure 4: Ilustration of kind of output generated by ROMRPG

essentially performs various kinds of memory operations. If we abstract these memory operations from the type of traffic generators, we can classify these operations into one of the following types:

1. *simple write* (`attributes`)
2. *read and check* (`attributes`)
3. *read without check* (`attributes`)
4. *atomic write* (`attributes`)
5. *atomic read* (`attributes`)
6. *barrier sync* (num_cpu)

7. *user defined operations* (...)
The ROMRPG core generates memory operations in the generic format (represented in Fig. 5), and the tool's back-end converts them from the generic format to an appropriate format needed by the traffic generators or PEs.
The `attributes` (labed above as arguments in the generic memory operations) are usually addresses, length of transaction, transaction type, array of data, data size, iteration times etc. The constraints for each of the `attributes` are specified using SystemC verification (SCV) constrained randomization facilities[15]. The attribute 'transaction type' could be cached, un-cached, and other types depending upon what is supported by the underlying platform. By means of the attribute 'iteration times' we specify how many times the given transaction need to be repeated. 'Iteration times' attribute is an important idea to automatically generate back to back transaction of various types which are hard to generate using just simple randomization technique.
We briefly describe in this paragragh some non-intutive operations classified above. In *atomic write* operation, only one PE can update a memory location, and thus deterministic results can be ensured for more than one updates. In *read and check* operation, along with the read transaction, the ROMRPG supplies the expected value which need to be checked against the result of this transaction. In case of any mismatch between the expected data and the returned data, the test code jumps to an error sub-routine with relevant information for debugging. In *barrier sync*, a very common scheme for inter-process synchronization used in parallel programming, all the PEs waits until all the other PEs reach and execute the *barrier synch* operation. Barrier-based synchronization is very useful to generate deterministic self-checking parallel programs involving true-sharing of data[16]. Other than the given standard operations, *user defined operations* are also supported to allow customization, and generation of new sequence of transactions, which are difficult to automatically generate using simple randomization approach. An example for *user-defined operation* is a sequence of back-to-back store/load operations on same address. This kind of customization allows the verification engineer to come out with interesting sequence patterns which are almost impossible to generate by randomization.
The addresses within the data memory region or section where various operations are performed can be configured by user-defined constraints. The tool consists of certain standard configured constraints or modes like aligned addressing, un-aligned addressing, interleaved addressing, L1 cache-boundary crossing addressing, un-restricted addressing etc. Alignment of the addressing is done with respect to the L1 cache-line boundary (specified by the architecture description). In 'interleaved addressing', the first word of an address is given to PE^0, the next word to PE^1, so on until PE^x and the assignment pattern repeats thereafter. This scheme of addressing is useful for false-sharing based MP programs [16]. In 'un-restricted mode' no self-checking is done as we do not impose any restriction or constraints in addressing various data section of the memory region by different PEs. The system should rely on the run-time coherency checking mechanism [14] to do the necessary memory coherency checking or use trace-driven checkers [4].

5. ROMRPG: CONFIGURATION AND CASE-STUDY

```
(* thread0 header *)
(* function0 header *)
/*step0:*/ operation function1 ( arguments );
/*step1:*/ operation function2 ( arguments );
....
/*step 20:*/ simple write (addrA, iter_times, ... );
/*step 20:*/ simple write (addrB, iter_times, ... );
....
/*step 29:*/ read and check (addrB, iter_times, ... );
....
/*step 49:*/ read and check (addrA, iter_times, ... );
(* jump to next function *)
(* function0 footer *)
....
barriers sync ( num_cpu ); /* synchronization */
....
(* functionN footer *)
```

Figure 5: Generic Program Format. All the statements between (*, *) and the operation functions (like *read and check*) are in-lined into appropriate code by the back-end of the tool. Statements within /* */ are comments.

Application of the tool to uncover bugs in the MPSoC implementation requires a systematic approach of going from a simple configuration to detailed configuration. A systematic verification process for a typical shared-memory MPSoC is explained in the paper [16]. It consists of four modes of verification namely: Non-sharing, false-share, deterministic true sharing and non-deterministic true sharing. Each of these modes correspond to different ways of sharing data between different processors. All these modes are easily configurable in ROMRPG by proper selection of different configuration parameters in the tool (explained below).
In order to simplify the allocation of memory region to different processor, and also to incorporate various modes of sharing we use a generic memory allocation model shown in Fig. 6(b). A sample memory configuration specification is shown in Fig. 7(a). Similarly a part of the architecture specification is shown in Fig. 7(b). The current specification scheme is simple and consist of basic parameters to understand the configuration of the MPSoC, and in future we intend to use Architecture Description Language (ADL) based description or XML for this purpose. We will briefly explain some of the parameters associated with the memory allocation model and specification. By controlling these parameters it is possible to apply different kinds of traffic to the MPSoC.
The distinct part of memory which is allocated to each initiator is called a *chunk*. Every initiator owns the *chunk* of memory on which it can perform different kinds of operations. The kind or type of traffic which can be generated on each *chunk* of memory is called an *operation*. Each *operation* (described in 4) has a number of attributes or arguments like address, data, number of data, and others. For each *write operation* a corresponding *read and check* is scheduled or delayed after some random number of steps, but less than the maximum value specified by the user (called *maximum step delay*). By controlling the *maximum step delay* parameter it is possible to generate a realistic traffic like back-to-back

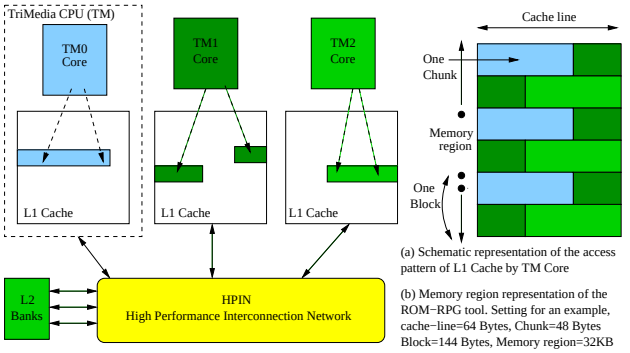

Figure 6: Example of ROMRPG tool customization (a) Representation (b) Memory region and its settings

traffic, sparse traffic etc.

memory region = A	num_cpus = 3
memory_region_size = 1 MB	L1_cache_size = 64 kbytes
chunk_size = 48 bytes	L2_cache_size = 1 MB
num_of_masters = 3	L1_line_size = 64 bytes
ops_per_function = 1000	L2_line_size = 256 bytes
enable_barrier = true	num_banks = 8
num_operations = 10 K	bus_width = 64 bits
maximum_step_delay = 100	burst_size = {1:16}
self_checking = true	HW_cache_coherence = true
(a)	(b)

Figure 7: (a) A sample memory configuration for test generation. (b) A sample MPSoC configuration specification

An example for the memory allocation, and the impact of it on the given SoC platform is shown in Fig. 6. In this example we have configured the ROMRPG to generate false-sharing traffic. False-sharing happens in cache-coherent MP-SoCs, if two processors access memory address which fall on the same L1-cache line, but at different memory location. The MPSoC in 6 consist of 3 Trimedia Processor with a L1-cache line-size of 64Bytes. The memory map is configured as follows:*memory region* = 32KB, *chunk size* = 48 Bytes. The given chunk size causes cache-line conflict between three processor as indicated in 6(a), causing false sharing between TM0 and TM1, TM1 and TM2. To generate a configuration without any sharing (non-sharing) we can set the *chunk size* as 64 Bytes, so processors can make accesses to its L1 cache content without causing any cache-coherence traffic.

6. ALGORITHM AND RESULTS

Here we will sketch the algorithm used in the ROMRPG core (boxed in Fig. 3 as A) to generate the generic traffic format shown in Fig. 5. Pseudo-Code:1 describes the algorithm for generation of MP program with N threads. In case of hardware based traffic generator that does not have the notion of thread and context-switching, one thread is assigned to one initiator permanently. In simple terms, each MP program has a number of threads. Each thread is further split into number of functions having a function header and footer. Each function has M number of steps. In each step we generate one operation of a particular type. The various arguments of the operation should satisfy the constraints and adhere to the SoC architecture specified by the user. SytemC verification library returns appropriate attributes statisfying the given user-defined constraints. We now briefly discuss the important data structures used in Pseudo-code:1. We have two linked list data structure for each thread. One linked list corresponding to the scheduled operations (*stepQueue*, and arranged based on *step* number. Another linked list corresponding to the executed operations ($finishQueue$), and is used for checking dependency which is useful for generating Dtree[13] based code. We also have a global sparse memory data structure ($sparceMemory$), which maintains the value of different memory location at given time, and is used for generating checking operation to validate memory coherency.

The tool was applied for verification of different implementation of MPSoC like in simulator, various MPSoC RTL implementation. The generic interface allows to customize the tool for new simulation platforms or simulation accelerators. A detailed application-level coverage statistics is generated by the tool after each test-case generation. The

Pseudo-Code 1 ROMRPG Core Implementation

1: $stepQueue \Leftarrow NULL$
2: $sparseMemory \Leftarrow 0$
3: $finishQueue \Leftarrow NULL$
4: {generate thread header for all threads}
5: **while** $stepNo < MAX\_STEP$ **do**
6: **for** $i = 1$ to N **do**
7: $funcStepNo = stepNo\%STEPS\_PER\_FUNCTION$;
8: **if** $funcStepNo == 0$ **then**
9: {*generate* function header}
10: **end if**
11: {get operation type for this step from *stepQueue*}
12: **if** $opType == readType$ **then**
13: {get address from *stepQueue*}
14: {get recent data from the $sparseMemory$}
15: {get other atributes satisfying constraints}
16: {get additional constraints based on output format}
17: {generate appropriate *read operation* with attributes}
18: {put operation into $finishQueue$}
19: **else**
20: {select suitable write type *simple write, atomic write*}
21: {select suitable attributes statisfying constraints}
22: {check more constraints from $finishQueue$ based on output format}
23: {generate the *write operation* with attributes}
24: {Find a free step from *stepQueue* within *maximum step delay*}
25: {Schedule read operation by putting into *stepQueue*}
26: **end if**
27: **if** $funcStepNo == STEPS\_PER\_FUNCTION$ **then**
28: {if enabled generate *barrier* operation}
29: {*generate* function footer}
30: **end if**
31: **end for**
32: **end while**
33: {generate thread footer for all threads}

detailed coverage statistics consist of percentage of different operations generated for each thread and also address related statistics. These informations are useful in fine-tuning the various configuration parameters. We discovered various kinds of bugs related to deadlocks, and incorrect memory coherency in the simulator, and the MPSoC RTL implementation in the Wasabi project. Since many of the bugs which were discovered involves detailed description of the architecture we omit the details of the bug scenario description in this paper. With simple configurations we could achieve more than 90% code coverage without much effort. Further configurations and constraints are necessary to achieve 100 % line, and state machine coverage.

We also noticed some short-coming in the methodology proposed in this paper. Firstly, detecting the error condition or bug is only one aspect of the design verification. From this point of mis-match between the expected result and the actual result, we need to detect the point in the RTL code where the actual bug occurs. This is a challenging and a tedious task which is not addressed in this methodology. Alternative approaches presented in [10, 19] can be used in combination this flow to address the problem. Secondly, adapting the constraint and traffic pattern after analysing the code-coverage and output statistics result is still a manual process. New approaches are required to complete the feedback loop and ensure efficient generation of traffic pattern so that better coverage can be achieved without much manual intervention.

7. CONCLUSION

A thorough verification plan should use a mix of different approaches, and an RPG approach is essential to identify weakly tested parts of the SoC design. It is almost impossible to generate all the corner-case, and detect bugs using truly or fully random programs. Constraining and targeting ROMRPG tool towards specific modules or functionality can only detect potential bugs. Hence it is essential for any RPG tool to support various configuration and constraints during program generation by which different part of the design can be stressed. Ideally an RPG tool should come during the later stage of the verification flow when all the directed and performance test-suites were successfully run on the RTL. Thus all the elementary or simple bugs are fixed before RPG, and RPG should mainly be used for continuous regression to reveal verification loopholes. Another aspect which was observed during various exercise is that the RPG tool should ideally match the module or design component verified. For example, if we are verifying an interconnection network, the RPG traffic should be the traffic directly driving the interconnection network and not coming as a side effect of other hardware component like processor etc. By this way we can directly stress the module and control the test pattern needed for verification.

8. ACKNOWLEDGMENTS

The authors thank the entire Wasabi design and verification team at Philips Research for their support and technical help.

9. REFERENCES

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] M. G. Bartley, D. Galpin, and T. Blackmore. A comparison of three verification techniques: Directed testing, pseudo-random testing and property checking. In *DAC*, pages 819 – 823, 2002.

[3] J. Bergeron. *Writing testbenches: Functional verification of HDL Models*. Kluwer, 2003.

[4] J. Bhadra, E. Trofimova, and et. al. A trace-driven validation methodology for multi-processor socs. In *IEEE SoC Conference*, 2006.

[5] Cadence/Verisity. E-verification component. http://www.verisity.com.

[6] H. Chang, L. Cooke, and et. al. *Surviving the SoC Revolution*. Kluwer, 1999.

[7] W. H. Collier. ARCHTEST by multiprocessor diagnostics. www.mpdiag.org.

[8] S. Cox, M. Glasser, W. Grundmann, and et. al. Creating a C++ library for transaction-based test bench authoring. In *FDL forum, France*, 2001.

[9] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture, a hardware/software approach*. Morgan Kaufmann Publishers, 1999.

[10] S. Fine, S. Ur, and A. Ziv. Probabilistic regression suites for functional verification. In *DAC*, pages 49–54, 2004.

[11] L. Fournier, Y. Arbetman, and M. Levinger. Functional verification methodology of microprocessors using the genesys test-program generator. In *DATE*, 1999.

[12] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Springer, 2002.

[13] J. Hoogerbrugge and L. Augusteijn. Instruction scheduling for TriMedia. *Journal of Instruction-level Parallelism*, 1999.

[14] R. Raghavan, J. Kreulen, and et. al. Mutiprocessor system verification through behavioral modeing and simulation. In *IEEE Phoenix Conf. on Computer and Communication*, 1995.

[15] J. Rose and S. Swan. Systemc verification randomization. http://www.openverificationfoundation.org/.

[16] M. Typaldos and B. Cavanaugh. Random test generation for multi-processor systems. White paper at www.obsidiansoftware.com.

[17] J. van de Waerdt and S. Vassiliadis. Instruction set architecture enhancements for video processing. In *Proc. of the 16th Int. Conf. on Application-specific Systems, Architectures and Processors*, July 2005.

[18] J. van Eijndhoven, J. Hoogerbrugge, M. N. Jayram, and et. al. Cache-coherent heterogeneous multiprocessing as basis for streaming applications. In *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, volume 3, pages 61–80. Springer, 2005.

[19] I. Wagner, V. Bertacco, and T. Austin. Stresstest: An automatic approach to test generation via activity monitors. In *DAC*, 2005.

[20] D. A. Wood, G. A. Gibson, and R. H. Katz. Verifying a multiprocessor cache controller using random test generation. *IEEE Design and Test of Computers*, August 1990.