Utilizing Formal Assertions for System Design of Network Processors

Xi Chen, Yan Luo, Harry Hsieh, Laxmi Bhuyan
University of California, Riverside, CA
{xichen, yluo, harry, bhuyan}@cs.ucr.edu

Felice Balarin
Cadence Berkeley Laboratories, Berkeley, CA
felice@cadence.com

Abstract

System level modeling with executable languages such as C/C++ has been crucial in the development of large electronic systems from general processors to application specific designs. To make sure that the executable models behave as they should, the designers often have to "eye-ball" the simulation traces and at best, apply simple "assert" statements or write simple trace checkers in some scripting languages. The problem is the lack of a concise and formal method to specify and check desired properties, whether they be functional or performance in nature. In this paper, we apply assertion checking methodology to the system design of network processors. Functional and performance assertions, based on Linear Temporal Logic and Logic of Constraints, are written during the design process. Trace checkers and simulation monitors are automatically generated to validate particular simulation runs or to analyze their performance characteristics. Several categories of assertions are checked throughout the design process, such as equivalence, functionality, transaction, and performance. We demonstrate that the assertion-based methodology is very useful for both system level verification and design exploration.

1 Introduction

The increasing complexity of electronic systems today demands more sophisticated design and test methodologies. Designing only at the Register Transfer Level (RTL) is no longer sufficient. Most electronic systems, from general purpose processors to application specific designs start out with a description at the higher system level. System level modeling with executable languages such as C/C++ or other modeling frameworks have been crucial in the development of large electronic systems. The executability serves at least three very important purposes. First, it allows the designers to communicate with other designers with concrete traces and scenarios. Second, it allows early verification of the design functionalities so the designers can check whether what they wrote is indeed what they want. Lastly, high level performance parameters can be gathered to enable system level design exploration so that better system level trade-off decisions can be made.

Unfortunately, it is not always easy to gather information from the behavior of the executable model. The designers often have to "eye-ball" simulation traces and at best, write simple checkers in some scripting language (such as Perl) to scan through the traces and analyze for specific characteristics. Embedded software assertion statements have been used by software designers for decades, but they are only suitable for simple error-flagging and are not amenable for analyzing more complex behavior and catching more complex bugs that can exist at the system level. Automatic and semi-automatic formal verification tools [8, 9] can prove the properties of a design, but their need for extensive manual proof-writing or high computational complexity, respectively, make them unsuitable to be integrated into design flow for exploration. The central problem lies in the lack of a concise and formal way to specify the desired functional and performance properties, and an efficient way to check the properties.

RTL assertion languages such as Sugar2.0 [7] and OpenVera [2] have been gaining in popularity. Functional assertions, based mostly on Linear Temporal Logic (LTL) [10], can be written and simulation monitors can be automatically generated to efficiently verify a particular run of the executable model. Every LTL formula can be represented by an equivalent finite state automaton, hence LTL cannot be used to express higher, transaction level, functional properties which may not be representable by finite-state automata [6]. Furthermore, neither LTL, nor its associated assertion languages (Sugar2.0 and OpenVera) have convenient ways to express quantitative performance properties, which is crucial in performing system level design exploration, as well as analyzing the performance capability of a system level model.

To apply the assertion methodology to the system level requires the ability to specify functional and performance properties at the RTL, at the transaction level, and at the system level. We propose to utilize LTL-based assertion languages (e.g. Sugar2.0) and Logic Of Constraints (LOC) [4], a logic that is more suited for quantitative performance and transaction level assertions, in the functional and performance verification of system level designs.

This paper presents two main contributions. The first is to demonstrate that the assertion-based verification methodology is useful for catching errors in the design of a network processor at the system level. To show this, we integrate our assertion verification methodology into the design flow for high performance network processors. Based on Intel IXP1200 [1] network processor model, in-house designers have been putting together a new architecture which is capable of higher throughput, lower latency, and lower cost. The processor model is parameterized, so that a whole range of different architectures can be explored. Using our assertion verification methodology, designers were able to write assertions and automatically generate trace checkers or simulation monitors throughout the design process to check functionality and performance characteristics. Bugs were subsequently found and corrected.

The second main contribution is to extend the use of assertions to design exploration and performance analysis. To this purpose, we use assertions to express performance-related statements that should always hold, and those that should hold as often as possible. For the latter, we generate assertion checkers that can report not only if the statement holds, but also how often it holds. We gather these performance numbers, and use them to make better design decisions.

In the next section, we discuss the essential aspects of our network processor architecture model NePSim, which is based on Intel IXP1200. We introduce our assertion verification methodology in Section 3. Section 4 presents a detail design study. We discuss the kinds of assertions we apply, the bugs that we were able to find, and how the performance assertions and goals can be used in design exploration. We conclude in Section 5 and give some future directions.

2 Network Processor Models

The Intel IXP1200 [1] is chosen as the reference model due to its overwhelming popularity in the network processing applications. Given normal-size IP packets, it can achieve up to 2.2 Gbps routing bandwidth. Being sold commercially, the processor model has been made available to the public in order to help the designers build systems based on IXP1200. The basic architecture of the processor is shown in Figure 1. IXP1200 consists of a StrongARM core, 6 multi-threaded processing units, which are called microengines, and controllers of peripheral units. The StrongARM core initializes the program store of the microengines and loads necessary data into memory before enabling the microengines. Each of the six microengines runs up to four threads concurrently. Thus, a total of up to 24 threads can be programmed to receive, process and transmit IP packets. The controllers of SRAM, SDRAM and IX Bus units serve the processor as interfaces to off-chip SRAM, typically used to store forwarding table, SDRAM, typically used to store IP packets, and network devices through the IX Bus.

The threads in a microengine share common ALU, pipelining, and scheduling units. Inside a microengine, each thread has an independent set of registers including general purpose registers, local control registers, SRAM transfer registers and

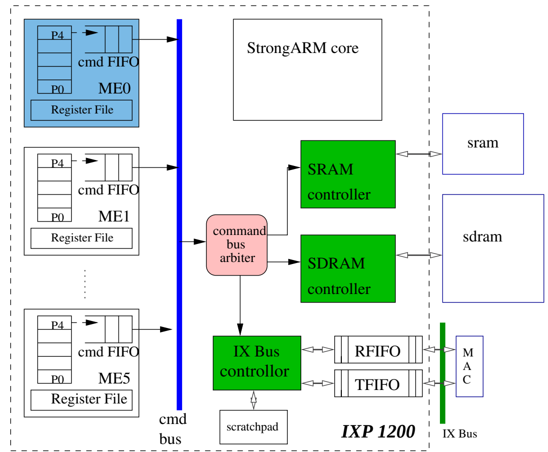

Figure 1. IXP1200 architecture.

SDRAM transfer registers. The microengine's instruction set architecture contains 33 categories of instructions. Because each instruction may have a number of different operations, the total number of op-codes implemented is around 120. Each microengine has a 5-stage pipeline (P0 through P4): Instruction Fetch, Instruction Decoding, Operand Fetch, Instruction Execution and Write-Back.

In each microengine, memory references, which are called commands, are issued to a two-entry command FIFO. The commands are then sent to the command bus and scheduled by the command bus arbiter. Based on the priority of commands, the command bus arbiter selects one or more reference commands among the six command FIFOs and move them to the corresponding memory controller. The SRAM controller handles all SRAM reference commands issued from the microengines. Each SRAM reference command is enqueued, dequeued, committed and finally done. SDRAM reference commands are handled similarly by the SDRAM controller.

Our NePSim network processor architecture is based on the basic IXP1200 architecture. When the architectural parameters (e.g. number of microengines, number of threads in each microengine, number and length of FIFO queues, size of the caches, scheduling policies) are set to be that of the IXP1200, the behaviors of the two processor models are expected to be very similar. When we vary the parameters, the functional "correctness" is expected to be maintained while the performance attributes are expected to change, trading off one metric against another. Ultimately, we can arrive at a design that is most suitable to the applications at hand.

3 Assertion-Based Verification Methodology

We use LTL-based assertion language Sugar2.0 [7] and LOC [4] to formally specify the functional and performance properties of the network processor model. LTL and LOC have different domains of expressiveness and indeed complement each other quite well [6]. According to our experience,

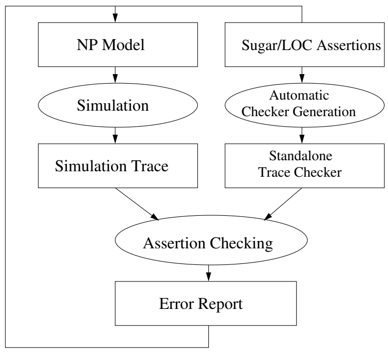

Figure 2. Assertion-Based Simulation Verification Methodology.

most functional properties, such as mutual exclusion, non-starvation and safety, can be easily expressed with LTL. On the other hand, LOC is more suitable for expressing quantitative performance properties such as rate and latency, and transaction level functional properties such as I/O data consistency. Figure 2 illustrates our methodology for assertion-based simulation verification with automatic trace checker generation.

Linear Temporal Logic [10] is defined over *executions* of a system, i.e. linear sequences of *state transitions*. We use Sugar2.0 [7] as the specification language for LTL formulas. In our verification methodology for system level models, the state transitions are modeled as event occurrences. This is consistent with transaction abstraction since only the events ordering are considered, not their tick by tick, cycle level behavior. We leverage the existing tool, FoCs [3], to generate the checker core and then use our tool to automatically generate wrappers that are necessary for the simulation of the processor models and for stand-along trace checkers. Since the simulation sessions are finite, we interpret the temporal operators over the finite traces by checking the conditions only up to the end of the traces.

Logic Of Constraints [4] is a formalism designed to reason about execution traces with quantitative performance values (annotations). It is also very well-suited for analyzing traces from the execution of higher, transaction level system model. LOC consists of all the terms and operators allowed in sentential logic, with additions that make it possible to specify quantitative assertions without compromising the ease of analysis. The basic components of an LOC formula are: event names (e.g. *pipeline* and *sram_enq*), instances of events (e.g. *pipeline*[*4*]), indices of event instances (e.g. 0, 1, ..., etc), the index variable i and annotations (e.g. *cycle*, *pc* and *addr*). LOC can be used to specify many important system level performance properties that are inconvenient, and sometimes impossible, to express with LTL. For example, the rate property:

$$cycle(pipeline[i+1]) - cycle(pipeline[i]) = 10 \qquad (1)$$

requires that the difference between the values of annotation *cycle* for any two consecutive instances of *pipeline* event should equal to 10. There exist automatic tools [5] that can generate trace checkers or simulation monitors for any given LOC formula. A checker or monitor evaluates the formula instance by instance, where a formula instance is a formula with i evaluated to some fixed positive integer value. For example, $cycle(pipeline[30]) - cycle(pipeline[29]) = 10$ is the 29th instance of the formula (1).

4 Verification Studies

We perform three categories of assertion verification throughout the design process. First, we would like to know whether NePSim, with parameters equal to that of IXP1200, would achieve the same functionality and “similar” performance. We then vary the design parameters (i.e. number of microengines, number of threads, configuration of the microengines, amount of caches, ..., etc,) and in each case, check functional correctness of the new design with functional assertions. Varying the design parameters obviously affect the performance. We use performance assertion checking to determine the performance of a particular design given a particular simulation input. We effectively explore the design space with functional and performance assertions.

Table 1. List of Event Types.

Event Type	Comments
pipeline	an instruction enters the pipeline
sram_enq	an SRAM access request is enqueued
sram_deq	an SRAM access request is dequeued
sram_done	an SRAM access request is committed
ip_lookup_start	an IP address lookup starts
ip_lookup_done	an IP address lookup finishes
forward	an IP packet is forwarded

The simulation traces generated from processor architectural models contain a set of architectural execution events that occur during the simulation run. They include instructions entering or leaving the pipeline, memory reference commands being put into or removed from the command queues in memory controllers, signals being generated from or consumed by functional units and threads. Table 1 lists some events that we are interested in for our verification studies. To differentiate events generated by different microengine, different threads, and different architecture models, each event could be appropriately prefixed and suffixed. For example, *m2_t1_pipeline_IXP* represents the pipelining event from the microengine 2, thread 1 of the Intel IXP1200 model. An event is annotated with timing, identification, and other quantitative information. In our study, each event instance is associated with four annotations, *cycle*, *pc*, *addr* and *data*, where *cycle* is used to measure time in clock cycles, *pc* is the PC (Program Counter) value for the current instruction. Depending on the event, *addr* may represent memory address or next PC (NPC) address, and *data* may represent data read from memory or ALU operation result.

The trace from NePSim is taken "as is" and fed into our automatically generated trace checkers. The log traces from Intel IXP1200 are preprocessed in a straightforward manner before being fed into the trace checker so that each event has the above annotations on the same line, though there is no difficulty in writing a separate trace format for the IXP1200. A snapshot of the simulation trace from NePSim is shown in Figure 3.

In the following verification studies, we run a trie-based route lookup benchmark [11] on NePSim and Intel IXP1200 processor models. The benchmark program is an infinite loop which continuously look up the route for a list of IP addresses. The trie data structure is stored in SRAM and accessed through the SRAM instructions.

```
                      ⋮
1731  68  00120100  00000000  m2_t2_pipeline
1732  34  0000FFF8  00000000  m5_t1_receive
1733  19  00000300  00000000  m0_t0_sram_enq
1733  30  00000300  00000000  m1_t0_pipeline
1733  30  00000300  00000000  m2_t0_pipeline
1733  30  00000300  00000000  m3_t0_pipeline
1733  20  00000300  00000000  m0_t0_sram_deq
1734  69  0000003F  0000050C  m2_t2_pipeline
1735  34  00000300  00000518  m5_t2_pipeline
1736  35  0000001B  0000051C  m5_t2_forward
1736  36  00178000  00000520  m5_t2_pipeline
1737  72  00020100  00000524  m2_t2_pipeline
1737  20  00000300  00000000  m0_t0_sram_done
                      ⋮
```

Figure 3. NePSim Simulation Trace.

4.1 Checking with Reference Model

Using LOC, we can formally and accurately specify both functional equivalence and performance similarity of two designs. We run the same benchmark on NePSim and Intel IXP1200 models, and use the simulation trace from IXP1200 as the reference trace. First, we check if the NePSim model is functionally equivalent to the reference model. The primary function of the model resides in the forwarding table lookup for IP packets being processed, which involves correct reading from the SRAM. More specifically, we want to check the following property:

"For each SRAM access on NePSim and IXP1200, the memory address referenced and data read out should be the same, and all the SRAM references are executed with the same order."

This property can be expressed with an LOC formula:

$$\begin{aligned} &addr(sram\_enq[i]) = addr(sram\_enq\_IXP[i]) \wedge \\ &data(sram\_done[i]) = data(sram\_done\_IXP[i]) \qquad . \end{aligned} \tag{2}$$

We run the benchmark on NePSim and IXP1200 for one million cycles to obtain traces of about 3×10^5 lines. Both models are configured with a single working microengine and a single working thread so that the packet processing order is deterministic. With the automatically generated trace checker, we show that this formula pass with the given benchmark trace in under 6 seconds of CPU time (see Table 2.) All the trace checkings presented in this paper were run on our Athlon 1.5GHz Linux machine with 1GB memory, though the simulation sessions were run by the designers on their own machines. We report time and memory usage only for the trace checking operations.

Comparing the simulation trace from NePSim to the reference IXP1200, we also want to make sure that the instruction pipelining behavior of NePSim is "similar" in performance to that of IXP1200. More specifically, this property requires that

"On the two models, all the instructions in the benchmark are executed with the same order, and the execution time of every pipelining instruction by NePSim is no more than certain clock cycles away from the execution time of the corresponding instruction by Intel IXP1200."

This property can be expressed with an LOC formula:

$$\begin{aligned} &(pc(pipeline[i]) = pc(pipeline\_IXP[i])) \wedge \\ &(|cycle(pipeline[i]) - cycle(pipeline\_IXP[i])| \leq A \cdot i + B). \end{aligned} \tag{3}$$

where A and B are constants. The second part of the formula holds if and only if, for a particular pipeline event, the difference in time of occurrence in NePSim and Intel IXP1200 is within $A \cdot i + B$. As simulation progress, the difference accumulates, which is reflected by $A \cdot i$. The difference in startup time is accounted for by the constant B. If the two designs are truly identical, both values should be zero. The designers decided that, to account for the differences in the two designs, the acceptable values of A and B should be 0.05, and 8, respectively. The formula failed almost immediately, and after going through the error report and debugging the NePSim design, it was found that the SRAM access latency was not modeled correctly. Once the error was fixed, the performance assertion passed (see Table 2). This performance margin is sufficient for designers to declare that NePSim and IXP1200 are similar in performance.

With the same formula (3) and substituting for the *pipeline* event, we can check the performance similarity of other critical events such as *sram_enq*, *sram_done*, and *sdram_enq*, with different acceptable values of A and B, determined by the designers.

4.2 Functional Verification

Due to the non-determinism in thread handling within the network processor models, it is difficult to perform deterministic functional equivalence trace checking when there are more than one thread enabled. For normal multi-thread operations, functional property verification, based on both LTL and LOC, can be very useful. Designers can write their functional assertions in LTL-based Sugar2.0 or LOC. For Sugar2.0 assertions, we use FoCs to generate the assertion checking code in C++, and our tool then generates the necessary wrappers for trace checking.

To verify the normal operation of the NePSim processor model, We configure it with 6 working microengines (4 of them (m0 - m3) used for forwarding table lookup and 2 of them (m4, m5) used for IP packet transmission) and 4 working threads for each microengine, and run the benchmark on NePSim for one million cycles. Using Sugar2.0, we specify a non-starvation property for the SRAM controller of NePSim: "Once an SRAM access request from a thread (e.g. thread 0) of a microengine (e.g. microengine 0) is enqueued, it must be eventually committed within the next 300 SRAM related event occurrences." This property can be expressed with the Sugar2.0 formula:

$$\texttt{always}(m0\_t0\_sram\_enq \rightarrow \texttt{next\_e}[1:300](m0\_t0\_sram\_done)) \quad (4)$$

To check this property, we only produce the events that are related to SRAM references to get a trace of 2.8×10^5 lines. The parameterized wrapper generator can easily generate this assertion for all threads in all microengines, and for SDRAM controller and IX bus controller.

Another important property of the memory access scheduler is the correct occurring order of the events *sram_enq*, *sram_deq* and *sram_done*, which requires that "after an SRAM request by a thread (e.g. thread 1) of a microengine (e.g. microengine 0) is issued and put into the scheduling FIFO, it cannot be done before it is dequeued". This property of occurring order can be expressed with the formula:

$$\texttt{always}(m0\_t1\_sram\_enq \rightarrow \\ !\, m0\_t1\_sram\_done \ \texttt{until}\ m0\_t1\_sram\_deq) \quad . \quad (5)$$

Note that if the simulation trace ends, the verification of the formula will be interpreted on a finite trace. For example, formula (5) will not be violated if *sram_enq* occurs and then neither *sram_done* nor *sram_deq* occurs when the trace ends.

Using LOC, we can express the data consistency properties for different functional units. For example, when an SRAM access request is put into to the scheduling FIFO by a thread (e.g. thread 2) of a microengine (e.g. microengine 1) and then eventually committed, the memory address it refers to should be the same. We express this property with the LOC formula:

$$addr(m1\_t2\_sram\_enq[i]) = addr(m1\_t2\_sram\_done[i]) \quad , \quad (6)$$

where the annotation *addr* is used to represent the referenced memory address. With the automatically generated trace checkers, formula (4) - (6) are checked with no error. The verification results are listed in Table 2.

Table 2. Verification Results for Formulas(2-6)

Formula	Formula Instances	Trace Lines	Mem	Time
(2)	10267	3×10^5	40KB	6s
(3)	295582	3×10^5	64.8 KB	7s
(4)	5690	2.8×10^5	0.4KB	77s
(5)	5739	7.0×10^6	50 Bytes	24s
(6)	5708	7.0×10^6	12 Bytes	59s

4.3 Performance Assertions

The goal of design exploration for network processor is to find an architecture which would perform "better" than the existing model. It is therefore very important to be able to analyze quantitative properties of a design. With LOC, we can express the performance requirements or expected quantitative features. In this section, we continue with the parameter setting of 4 microengine for IP address lookup and 2 microengines for IP packet transmission. For each microengine doing IP address lookup, we experiment with either running 2 threads or 4 threads. As a consequence, we compare the performance metrics for an 8-thread processor model against the one with 16 threads. We run our benchmark on both configurations for one million cycles, and get traces of about 3 million lines.

One primary function of the network processor is to perform IP address lookup, which requires very frequent access to SRAM. Therefore, we want to check the latency between an SRAM access request enqueued and when it is committed. We first check the SRAM access latency from a thread (e.g. thread 0) of a microengine (e.g. microengine 2) for the two configurations. We consider the maximum latency constraint, which can be expressed with the following LOC formula:

$$cycle(m2\_t0\_sram\_done[i]) - cycle(m2\_t0\_sram\_enq[i]) \leq l1 \quad . \quad (7)$$

We iteratively search for the smallest $l1$ that will allow the traces to pass the performance assertion (e.g. with a simple bi-partition approach on the range). For the 8-thread configuration, we were able to set $l1 = 50$, and the assertion can pass the trace checking without any error. For the 16-thread configuration, in order to make the assertion pass, we have to increase the $l1$ to 100. More threads can cause more memory access contention, and degrade the latency for individual memory accesses. See Table 3 for a summary of the result.

The total number of running threads can actually affect the latency for individual IP address lookups. The maximum latency of IP address lookups in a thread (e.g. thread 1) of a microengine (e.g. microengine 0) can be expressed in LOC using the formula:

$$cycle(m0\_t1\_ip\_lookup\_start[i]) - \\ cycle(m0\_t1\_ip\_lookup\_done[i]) \leq l2 \quad . \quad (8)$$

For the 8-thread configuration, we set $l2$ to be 900 for the assertion to pass. For the 16-thread configuration, $l2$ needs to be 1200. Using the formula (8), we have explicitly shown that the 8-thread configuration has lower latency for individual IP address lookups than the 16-thread configuration.

Of course, latency does not tell the whole story. Throughput is an equally important design characteristic for network processors. More threads should achieve better overall throughput. At the instruction level, we can check the throughput of pipelining instructions for the processor using the LOC formula:

$$cycle(pipeline[i+10000]) - cycle(pipeline[i]) \leq t1 \quad , \qquad (9)$$

which requires that within $t1$ cycles, at least 10000 instructions need to be issued to the pipeline of the processor. For the 8-thread configuration, we need to set $t1 = 4200$ for the assertion to pass. This corresponds to a minimum throughput of 2.3 instructions per cycle. For the 16-thread configuration, $t1$ need to be set to 3500, which corresponds to a minimum throughput of 2.8 instructions per cycle. The 16-thread configuration has better instruction throughput according to the analysis using the performance assertion (9).

The overall performance of the network processor is measured by the throughput of IP packet forwarding, which can be expressed with the following LOC formula:

$$cycle(forward[i+1000]) - cycle(forward[i]) \leq t2 \quad . \qquad (10)$$

In order for the performance assertion to pass, We need to set $t2 = 3.7 \times 10^5$ for the 8-thread configuration, and set $t2 = 3 \times 10^5$ for the 16-thread configuration. If we assume the NePSim processor is running at 200MHz, we get the throughput for IP packet forwarding of 5.4×10^5 packets/sec and 6.6×10^5 packets/sec for the 8-thread and 16-thread configuration, respectively. Given the average packet size of 64 bytes, the routing throughput will be 2.8 Gbps and 3.3 Gbps respectively for the two configurations. Indeed, the designers need to trade off latency and throughput for any given application to achieve the best design. LOC assertion checking allows them to quantitatively analyze the performance of a system level specification. During the design process, the designers can also experiment with increasing the number of microengines, changing the size of scheduling FIFOs, or putting more caches between storage hierarchies. All these design space explorations may bring various performance trade-offs, which can be easily specified and analyzed by the formal performance assertions.

The verification results of these LOC performance assertions are listed in Table 3. Since a typical simulation session can take half an hour or longer, the CPU time and memory usage for the trace checkers are trivial by comparison. Without them, however, it becomes very difficult for the designers to conclude anything about the design except in very vague terms (e.g. "looks good"). Our assertion-based verification and design exploration methodology is indeed efficient for dealing with large designs.

5 Conclusions

In this paper, we have presented a verification methodology for functional and performance properties of system-level designs utilizing formal assertions. We have applied our methodology on the system design of network processors. We use LTL-based assertions to express and verify functional properties such as non-starvation and execution ordering. We also use LOC assertions to express quantitative performance and functional properties such as latency, throughput and data consistency. All these assertions can be checked with automatically generated trace checkers on the simulation traces using small amounts of CPU time and memory. Through a set of verification studies, we show that formal assertions, based on LTL and LOC, are very useful for concisely specifying and automatically verifying both functional and performance properties of system level designs. The ability to carry out performance evaluation at the system level also opens up design exploration avenue uncharted before.

We are planning to extend the Sugar2.0 language to support our LOC formalism so that designers can have a unified frontend to specify both functional and performance assertions easily. We are also considering adding automatic design exploration capability to NePSim.

Table 3. Results for Performance Assertions

Formula	Conf.	Param.	Time	Mem
(7)	8-thread	$l1 = 50$	18sec	12Bytes
	16-thread	$l1 = 100$	23sec	16Bytes
(8)	8-thread	$l2 = 900$	46sec	8Bytes
	16-thread	$l2 = 1200$	44sec	8Bytes
(9)	8-thread	$t1 = 4200$	20sec	40KB
	16-thread	$t1 = 3500$	26sec	40KB
(10)	8-thread	$t2 = 3.7 \times 10^5$	44sec	4KB
	16-thread	$t2 = 3 \times 10^5$	44sec	4KB

References

[1] Intel IXP1200 Network Processor Family: Hardware Reference Manual, Dec. 2001.

[2] OpenVera assertions white paper. *Synopsys, Inc*, 2002.

[3] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal. FoCs - automatic generation of simulation checkers from formal specifications. *Technical Report, IBM Haifa Research Laboratory, Israel*, 2003.

[4] F. Balarin, Y. Watanabe, J. Burch, L. Lavagno, R. Passerone, and A. Sangiovanni-Vincentelli. Constraints specification at higher levels of abstraction. In *Proceedings of International Workshop on High Level Design Validation and Test*, Nov. 2001.

[5] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe. Automatic trace analysis for logic of constraints. In *Proceedings of the* 40^{th} *Design Automation Conference*, June 2003.

[6] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe. Verifying LOC based functional and performance constraints. In *Proceedings of International Workshop on High Level Design Validation and Test*, Nov. 2003.

[7] C. Eisner and D. Fisman. Sugar 2.0 proposal presented to the accellera formal verification technical committee. Mar. 2002.

[8] M. Gordon and T. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1992.

[9] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–258, May 1997.

[10] A. Pnueli. The temporal logic of programs. In *Proceedings of the* 18^{th} *IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.

[11] M. Sanchez, E. Biersack, and W. Dabbous. Survey and taxonomy of IP address lookup algorithms. *IEEE Network Magazine*, 15(2):8–23, 2001.