A First-Order Logic based Framework for Verifying Simulations

Hui Meen Nyew, Nilufer Onder, Soner Onder and **Zhenlin Wang**
Dept. of Computer Science
Michigan Technological University
Houghton, MI 49931
{hnyew,nilufer,soner,zlwang}@mtu.edu

Abstract

Modern science relies on simulation techniques for understanding phenomenon, exploring design options, or evaluating models. Assuring the correctness of simulators is a key problem where a multitude of solutions ranging from manual inspection to formal verification are applicable. Formal verification incorporates the rigor necessary but not all simulators are generated from formal specifications. Manual inspection is readily available but lacks the rigor and is prone to errors. In this paper, we describe an automated verification system (AVS) where the contraints that the system must adhere to are specified by the user in general purpose first-order logic. AVS translates these constraints into a verification program that scans the simulator trace and verifies that no constraints are violated. The advantage is the ability to verify any simulator trace using a formal specification of domain facts. Computer microarchitecture simulations were used to demonstrate the proposed approach. The system was implemented successfully to yield preliminary results.

Introduction

Contemporary computer processor design inherently relies on simulating new processors before they are built. The design of a new architecture typically starts with instruction set design. Instruction set development and system software development usually go hand-in-hand by using a *functional simulator* which implements the semantics of instructions and allows running programs in a simulated environment. The design and development of the processor architecture is then carried out using much more sophisticated and detailed simulators that can provide accurate information about how many processor cycles it will take to execute a given program under the new design. These simulators are called *cycle accurate* simulators and their implementation typically takes tens of thousands of lines of high-level program code, such as C. Once satisfactory results are obtained, the rest of the design is carried out with the help of *gate-level* and *circuit-level* simulators, which can provide detailed information about attainable clock speeds as well as the estimated power consumption of the target processor before it is built.

Formal techniques are increasingly being used at various levels of the design process. At the cycle-accurate level, domain specific architecture description languages allow efficient automatic generation of cycle-accurate processors, at the same time making it easier to apply formal validation techniques. Examples of such languages include Mimola, nML, Lisa,Expression, ASIP Meister, TIE, Madl, ADL++, GNR, among others (Mishra and Dutt 2008). However, because of the enormous complexity involved, application of such formal techniques are limited. Furthermore, hand-coded simulators are still widely used as companies rely on their developed code base to improve future versions of existing processors. In this domain, verification of simulators is still a difficult task and remains an area dominated by ad-hoc techniques, except for simpler embedded processors where a formal specification language is used to describe the architectural details.

Our motivation therefore in developing AVS has been to provide a formal means of verification outside the developed simulator. In this paper, we describe a general purpose automated verification system (AVS) which can be widely applied both to traditional hand-written simulators as well as to those generated from a formal specification. AVS has been implemented and tested on microarchitecture simulations.

System overview

The AVS system verifies a set of user specified constraints in a *trace file* generated by a simulator. The trace file contains a sequence of *events*, ξ, represented as n-tuples: $\xi = < e_1, \cdots, e_n >$ where, e_i refers to an attribute of an event, each $e_i \in E_i$, and E_i is the domain of e_i. For example, $\xi = < a, c, s, t >$ is an an event generated by a processor simulator where a is the address of an instruction, c is the instance number of the instruction (each instruction can execute multiple times), s is the pipeline stage, and t is the cycle time of the event. A *constraint* is a quantified statement that includes arithmetic and Boolean expressions and contains the domain facts specified by the user. For example, the following constraint specifies that each instruction that goes through the instruction decode (ID) stage should go through the instruction issue (II) stage unless a rollback that flushes the pipeline occurs.

```
forall z in T exists y in T,
 z.stage==ID
```

```
iff y.addr==z.addr and y.count==z.count
 and (y.stage==II
  or (y.stage==ROLLBACK and y.time>=z.time));
```

We used Flex and Bison to implement a compiler for AVS. The compiler takes the specification of first-order logic statements and the constraints as input and creates one or more independent C++ programs that perform the actual simulation verification. Fig. 1 depicts a sample program that consists of nested loops to check the `forall` and `exists` conditions for the constraint "`forall z in T exists y in T` statements".

Algorithm 1 Translation for "forall z in T exists y in T"

```
Input: Trace $T$
Output: $status$
 1: for all windows $w$ in T do
 2:     for all $z \in w$ do                          ↪ forall $z$ in $w$
 3:         $zStatus \leftarrow$ FALSE
 4:         for all $y \in w$ do                      ↪ exists $y$ in $w$
 5:             $yStatus \leftarrow$ FALSE
 6:             if statements = TRUE then
 7:                 $zStatus \leftarrow$ TRUE
 8:                 $yStatus \leftarrow$ TRUE
 9:             end if
10:
11:             if $yStatus =$ TRUE then                  ↪ exists
12:                 break                              ↪ quantifier
13:             end if
14:         end for
15:
16:         if $zStatus =$ FALSE then       ↪ forall quantifier
17:             $status \leftarrow$ FALSE
18:             return
19:         end if
20:     end for
21: end for
22:
23: $status \leftarrow$ TRUE
24: return
```

The current AVS implementation uses a sliding window (Mannila, Toivonen, and Inkeri Verkamo 1997) to check the constraints using a window size specified by the user. The advantage of using sliding windows is to allow the algorithm to process very large input or infinite streams. The time and memory requirements are also significantly reduced. Our next step will be to analyze the temporal relationships in the constraints and automatically compute the window size by using the maximum distance. Due to space restrictions, we show only the highlights of the algorithm. Further details can be found in the longer version of the paper (Nyew et al. 2013).

In addition to modeling the pipeline, we coded resource and dependency constraints. Resource contraints ensure that only the available number of resources are used. For example only as many memory instructions as the number of memory ports can complete simultaneously. An example of a dependency constraint is shown below. It specifies that two dependent instructions must be ordered.

```
forall z in REG_T forall y in REG_T
   exists x in STAGE_T exists w in STAGE_T,
   (z.iter>y.iter and z.dir==SRC and
    y.dir==DEST and z.reg==y.reg) implies
   (x.addr==z.addr and x.count==z.count
    and x.stage==EX and w.addr==y.addr
    and w.count==y.count
    and w.stage==EX and x.time>w.time);
```

Currently, we leave it to the constraint programmer to feed multiple parallel constraints separately as different inputs or merge them as one input. In the short term, the former approach will help generate multiple verifiers to enforce different types of constraints. For instance, we can generate one verifier for time constraints and one for resources. Multiple verifiers can run in parallel to take advantage of the computing power provided by modern machines.

Conclusion

We described a verification system for microarchitecture simulations. The system uses domain facts written by the user in first-order logic to scan the trace generated by a simulator and shows if any constraints are violated. Our implementation and preliminary experiments show that this approach is feasible. In addition to being able to verify basic facts, we noticed that the framework helps the user to iteratively improve the constraints. For instance, we had initially coded the example constraint to require each instruction's ID stage to be followed by an II stage. When the trace file failed the verification process, we coded the second part of the constraint which tells that a processor "rollback" causes the pipeline to be flushed and instructions are discarded before fully executing. Our future work involves improving the performance of AVS in two dimensions. First, microarchitecture simulators typically generate gigabytes of data. We plan to apply stream-mining techniques to address this issue. Second, the user needs to specify a window size for the verifier to execute efficiently. For domains where a window size cannot be specified or the window size is too large to bring efficiency gains, it will be helpful to further restrict the language to a precondition-effect based temporal language such as Planning Domain Definition Language (PDDL) (Fox and Long 2003).

References

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)* 20:61–124.

Mannila, H.; Toivonen, H.; and Inkeri Verkamo, A. 1997. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery* 1(3):259–289.

Mishra, P., and Dutt, N. 2008. *Processor Description Languages*. San Francisco, CA, USA: Morgan Kaufmann.

Nyew, H. M.; Onder, N.; Onder, S.; and Wang, Z. 2013. A first-order logic based framework for verifying microarchitecture simulations. Technical Report CS-TR-13-01, Department of Computer Science, Michigan Technological University. `http://www.cs.mtu.edu/~hnyew/cs-tr-13-01.pdf`.