Verifying Micro-Architecture Simulators using Event Traces

Hui Meen Nyew Nilufer Onder Soner Onder Zhenlin Wang

Department of Computer Science
Michigan Technological University
Houghton, MI 49931
{hnyew,nilufer,soner,zlwang}@mtu.edu

ABSTRACT

Contemporary micro-architecture research inherently relies on cycle-accurate simulators to test new ideas. Typical simulator implementations involve tens of thousands of lines of high-level code. Although general software engineering verification and validation techniques can be applied, the mere complexity of simulators makes using formal techniques difficult and calls for domain-specific knowledge to be a part of the verification process. This domain-specific information includes modeling the pipeline stages and the timing behavior of instructions with respect to these stages.

We present an approach to simulator verification that uses domain-specific information to effectively capture a potential mismatch between the assumed architecture model and its simulator. We first discuss how a simulator-generated event trace can be fed into an automatically generated verification program from a first-order logic specification to verify that the simulator obeys the invariants. We then show techniques that extract simulator behavior from traces and present the results to the user in the form of graphs and rules. While the former seeks an assurance of implementation correctness by checking that the model invariants hold, the latter attempts to derive an extended model of the implementation and hence enables a deeper understanding of what was implemented.

Our techniques are applicable to any micro-architecture simulator. We present the application of our techniques to hand-written simulators as well as to those generated from an architecture specification language.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: Modeling of computer architecture; C.1 [**Processor Architecture**]:

General Terms

Verification

Keywords

Architecture simulation, verification, first order logic

ICS'14, June 10–13 2014, Munich, Germany.

http://dx.doi.org/10.1145/2597652.2597680 .

1. INTRODUCTION

State of the art micro-architecture research inherently relies on cycle-accurate simulators to develop and test new ideas. Cycle-accurate simulators need to correctly model the processor behavior in sufficient detail so that accurate information about how a given program will execute under the new design can be quantitatively estimated. Cycle-accurate simulators are rather complex pieces of software as their implementation typically takes tens of thousands of lines of high-level program code, such as C. Cycle-accurate simulators also serve a crucial role in actual processor development and their use is essential to finalize the micro-architecture design. Currently, hand-coded cycle-accurate simulators such as SimpleScalar [3, 16], RSIM [13], M5 [2], GEM5 [1] as well as those generated from domain-specific architecture description languages are used widely both by the industry and academia. Examples of architecture description languages include Mimola, nML, Lisa, Expression, ASIP Meister, TIE, Madl, ADL++, GNR, among others [11].

While generation from an architecture description language can facilitate the application of formal validation techniques, using an architecture description language in itself will not prevent model representation errors. Furthermore, hand-coded simulators are still widely used as companies rely on their developed code base to improve the future versions of existing processors. As a result, 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 can be used to describe the architectural details. Our motivation therefore is to develop techniques which can identify model representation errors and do so in a simulator independent manner.

Our techniques rely on event traces generated from an execution of the target simulator by using the trace in two complementary processes. Figure 1 shows the general framework. First, we develop a first-order logic based language, which we call *First-Order Logic Constraint Specification Language* (FOLCSL). Using the language, the invariants of the model under consideration are specified. Examples of such invariants include that every fetched instruction must be decoded, no more than two load instructions can simultaneously access the cache, or, the execution step of an integer type instruction takes a single cycle. We then automatically synthesize a verification program from the first-order logic program as shown in Figure 1(a). This verification program reads the event trace generated by running the simulator using a particular benchmark (Figure 1(b) and (c)) and signals whether all invariants are respected. In this approach, if the constraint specification is complete and the verification program returns no errors, it can be stated that the simulator has faithfully followed the model for the set of benchmark programs tested. Unfortunately, the domain of invariants is large and even domain experts may omit the necessary

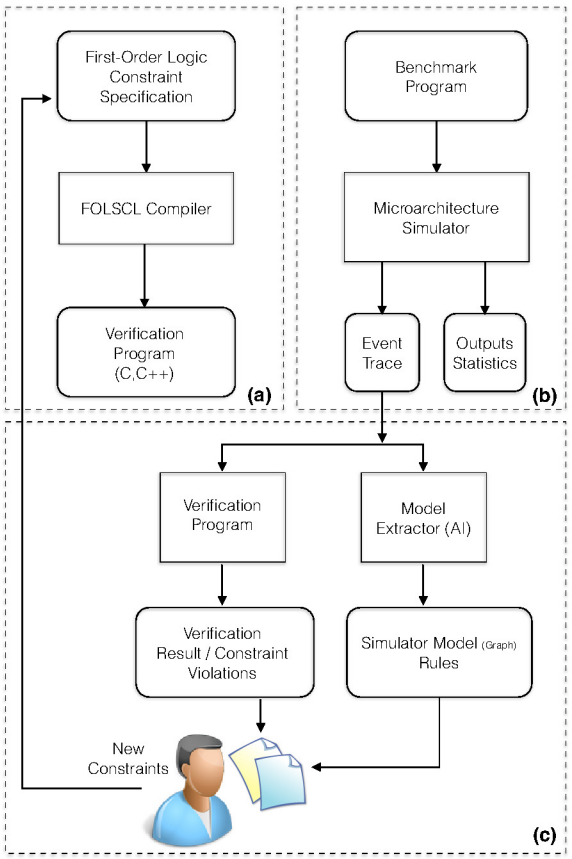


Figure 1: Simulator verification using event traces.

constraints to catch all the errors in the simulator. We therefore developed a second approach complementing the first.

In our second approach, we process the event trace using several artificial intelligence algorithms, and attempt to derive the simulated model from the event trace. This model, presented in the form of temporal graphs and rules, can be inspected by the human user to gain a deeper understanding of what the simulator actually implements. Using the derived model, further invariant constraints can be formulated and added to the initial set of constraints, in essence complementing the former approach. These two processes are used iteratively by repeating the process with different benchmark programs as shown in Figure 1, each time improving the set of constraints.

The strength of the outlined approach lies in its power to verify the implementation by both performing a check of the invariants as well as visually describing what the simulator implements. Furthermore, this is done in a very practical and general manner. Event traces can be easily generated from any type of simulator by inserting output statements, i.e., instruments, at specific points in the simulator. Most existing simulators already provide mechanisms to generate an event trace. For example, SimpleScalar can produce an event trace by simply setting an option. It also is easy to generate an event trace in automatically synthesized simulators by augmenting appropriate points in the description. We consider every type of activity within the simulator to be an *event*, and broadly classify events into two main groups, namely, those events which affect a single instruction and those which globally affect the processor's operation and hence all the instructions. For example, fetching, decoding, and executing an instruction are all considered to be events which affect a single instruction. In contrast, events such as a branch misprediction initiated rollback in a superscalar processor simulator is considered to be *global* as it affects every instruction in the processor. One of our contributions is to show that such traces contain sufficient information to verify that the simulator faithfully implements its execution model when these traces are processed with the appropriate algorithms. Although the data sets are very large and these algorithms have bad asymptotic complexity, by applying advanced filtering and windowing techniques, we are able to keep the running times within reasonable limits.

In the remainder of the paper, first, in Section 2, we describe our first-order logic based constraint specification language and its use in the verification process. This section starts by formally describing the properties of the trace.

2. FOLCSL CONSTRAINT LANGUAGE

FOLCSL is designed to specify the invariants which must hold during the execution of the simulator. The language allows constraint specification using first-order logic. The constraints are specified by referencing a particular event and associating it with other events. Expressions refer to the names used in a given trace, therefore, we first formally describe the expected form of trace data.

A trace T is a sequence of *events* $T = \xi^1 \cdots \xi^l$, represented as n-tuples: $\xi^i =< e^i_1, \cdots, e^i_n >$ where, e^i_j refers to the j^{th} attribute of the i^{th} event. Each e^i_j is an integer. For example, $\xi^i =< a^i, c^i, s^i, t^i >$ is an event generated by a processor simulator where a^i is the address of an instruction, c^i is the instance number of the instruction (each instruction can execute multiple times), s^i is the pipeline stage, and t^i is the cycle time at which the i^{th} event has been observed. It can be read as follows: At time t^i, the c^i-th instance of the instruction with address a^i is in state s^i. A sample trace line generated by the simulator is:

```
0xaabbccdd, 2, ID, 1000
```

The above trace line states that the second instance of the instruction fetched from address 0xaabbccdd is at instruction decode stage (*ID*) and at machine cycle 1000. FOLCSL does not require the declaration of text attributes such as *ID* above. Text attributes have no domain specific meaning attached to them by the language and they are treated just like any other constant.

A *constraint* C is a quantified statement that includes arithmetic and Boolean expressions and contains domain facts specified by the user. For example, the following constraint specifies that each instruction that goes through the instruction fetch (IF) stage should go through the instruction decode (ID) stage unless a rollback (RB) that flushes the pipeline occurs.

$$\forall z \in T\ \exists y \in T,\ (s^z = IF) \Rightarrow (a^y = a^z) \wedge (c^y = c^z) \wedge ((s^y = ID) \vee (s^y = RB)) \quad \text{(C.1)}$$

Verbally, the above expression specifies that for every event z that has the s attribute equal to instruction fetch (*IF*) and the instance attribute c, the verifier needs to find another event y with the same address and instance attribute values such that the stage attribute of event y is equal to instruction decode (*ID*), or it needs to find another matching event whose stage attribute is rollback (*RB*).

FOLCSL constraints consists of fully quantified variables, arithmetic expressions and Boolean expressions. The language has the simple grammar shown in Figure 2. Note that a *Terminal* in the language is an integer or an event attribute and an *Identifier* is a variable name or a function. In our current implementation func-

```
    constraint → quantification, statement;
     statement → ¬statement
               → statement ∧ statement
               → statement ∨ statement
               → statement ⇔ statement
               → statement ⇒ statement
               → expression relation expression
               → (statement)
               → identifier
    expression → expression + expression
               → expression − expression
               → expression * expression
               → expression / expression
               → (expression)
               → terminal
               → identifier
      relation →> | ≥ | < | ≤ | = | ≠
quantification → ∀ | ∃
```

Figure 2: FOLCSL Grammar.

tions are restricted to built-in functions only, and they are implicitly declared.

2.1 Instrumentation

Trace data is generated by inserting instrumentation statements into the simulator. Each instrumentation statement outputs an event in comma-separated value (CSV) format. For example, *printf("%lld,%lld,%d,%lld", addr, instance, state, cycles)* outputs an event with 4 attributes. SimpleScalar simulator's *-ptrace* option outputs similar data although it is not in CSV format [3, 16]. We translate it to CSV using a postprocessing program instead of modifying the simulator.

Instrumentation statements must be injected into proper locations in the simulation code. Global events which affect a subset of instructions require attaching the event to individual instructions. For example, a rollback is a global event in a processor pipeline but it affects a subset of instructions, namely, all uncommitted instructions that are still in the pipeline. In order to properly handle these types of events, we attach the rollback state to all affected instructions whenever a rollback occurs.

2.2 Stream Processing and Sliding Windows

FOLCSL and the associated trace description treat an instruction as an object which moves through different stages at some time point. The language allows the user to command the full power of first-order logic in specifying the invariants which need to hold. A direct consequence of this flexibility is the enormous size of the trace data which needs to be processed. As an invariant can reference arbitrary events, it may be necessary to compare all events to each other. Given that the number of dynamic instructions for a benchmark program are in the order of billions and each instruction will have multiple events, an uncompressed full trace of a single benchmark program takes many terabytes of storage space. Therefore, instead of storing the trace and processing it afterwards, we process the data as a stream. In our approach, whenever all the *required events* are available they are immediately processed and all the *expired events* are discarded. As a result, a minimum amount of data is kept in memory during the verification process. In order to reduce the processing time, we employ *sliding windows*.

2.2.1 Sliding Window

The sliding window approach views the trace as a chronologically ordered stream of events. Let ξ^i be our pivot event. We can buffer all events from time $t^i - t_b$ to time $t^i + t_f$ to form a sliding window [9] that pivots at time t^i. If we assume that an instruction's maximum time to live (ttl) in the pipeline is t_{ttl}, then a given constraint can be verified by just checking events in the sliding window that pivots at time t^z with $t_b = t_f = t_{ttl}$. Note that, in the event of a context switch or a roll-back, the ttl values are reset, so the window is always bounded. The *required events* are all those events which reside in the sliding window and the *expired events* are all events such that their occurrence time is less than $t^z - t_b$. Figure 3 depicts the sliding window for constraint C.1, where t^z is the pivot.

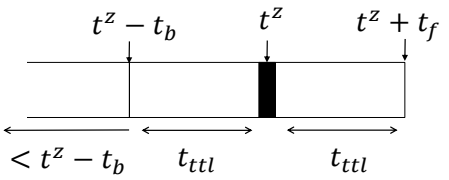


Figure 3: Sliding window.

The sliding window data structure provides three advantages. First, it requires minimal amount of storage space. Second, the verification process can begin even before the full trace is generated, allowing traces with an unknown length to be processed, such as a data stream from a network. Finally, for each pivot event, only the events residing in the sliding window need to be considered instead of all the events in the full trace. This significantly speeds up the verification process and processing very large traces becomes feasible.

2.2.2 Checker

Within a sliding window, all permutations of events are verified against the constraints. A more efficient way would be to view the verification process as an assignment of values to event variables, similar to constraint satisfaction problems (CSP). Using that view, existing CSP algorithms can be used. Our main checker algorithm is a backtracking search algorithm which uses depth-first search by assigning values to each variable and backtracking when a given assertion fails. To further reduce processing time, we prune the search space by evaluating critical expressions of a constraint before all variables get assigned a value. In constraint C.2 shown below, if the evaluation of expression $s^z = IF$ is true and $a^y = a^z$ is false, we know that the constraint is guaranteed to be false regardless of the value x. As a result, we can immediately backtrack and assign another value for y.

$$\forall z \in T\ \forall y \in T\ \exists x \in T,$$
$$(s^z = IF) \Rightarrow (a^y = a^z) \wedge (t^x > t^y) \qquad \text{(C.2)}$$

More efficient CSP heuristics such as propagation, variable ordering and intelligent backtracking [14] can also be applied into the checker.

2.3 Constraint Examples

While the domain of constraints is fairly large, several classes of constraints are particularly interesting to look at as they are necessary to catch some of the most common modeling errors. A common error in simulator development is the violation of resource

constraints. For example, if an architecture provides only two memory ports, at no time we should have more than two memory operations performing an access. While such an error would immediately get caught in a real hardware implementation as the hardware would not run, a simulator may continue to execute and yield incorrect results. In this section, we give examples targeting several common modeling errors which occur while modeling the resources involved, the temporal behavior of instructions and modeling competing instructions such as arbitration. In order to easily specify such constraints, FOLCSL includes several built-in functions. One of these functions is car which computes the cardinality of a set. The following example specifies a constraint that indicates at most two instructions can simultaneously access the memory ports. Note how sets are utilized to enforce resource based invariants.

$$\forall q \in T, car(set(\forall z \in T, (s^z = MEMPORT) \wedge (t^z = t^q))) \leq 2 \quad \text{(C.3)}$$

Similar to resource constraints, temporal constraints can be violated without a visible indication that such a violation has occurred. Temporal constraint violations include omission of a simulation step (i.e., a corresponding hardware stage), as well as when a particular instruction does not respect the latency of a particular pipeline stage. Such violations are very difficult to catch using adhoc techniques, particularly when these violations occur only for a small subset of the executed instructions. The following example encodes the requirement that an instruction that leaves the instruction fetch stage (IF) must either enter the instruction decode (ID) stage or the rollback (RB) stage and in doing so, it should take at least a cycle, but no more than K cycles, where K is a constant:

$$\forall z \in T\ \exists y \in T,\ (s^z = IF) \Rightarrow (a^y = a^z) \wedge (c^y = c^z) \wedge (t^y - t^z > 0) \wedge (t^y - t^z \leq K) \wedge ((s^y = ID) \vee (s^y = RB)) \quad \text{(C.4)}$$

When multiple instructions compete for a particular resource a subset of those instructions are granted access. This process, which is typically carried out by an arbiter at the hardware level, is particularly difficult to verify as the combination of the set of instructions must be taken into account while writing the FOLCSL statements. In the following example, we specify through constraint C.5 that $LOAD$ instructions are given priority to move from EX to WB stage.

$$\forall z \in T\ \forall y \in T,\ \exists x \in T\ \exists w \in T\ (s^z = EX) \wedge (h^z = LOAD) \wedge (s^y = EX) \wedge (h^y \neq LOAD) \wedge (t^y = t^z) \Rightarrow (a^x = a^z) \wedge (c^x = c^z) \wedge (a^w = a^y) \wedge (c^w = c^y) \wedge [((s^x = WB) \wedge (s^w = WB) \wedge (t^x \leq t^w)) \vee (s^x = RB) \vee (s^w = RB)] \quad \text{(C.5)}$$

As it can be seen through our examples, FOLCSL provides a convenient and easy to use way of specifying the invariants which must hold during the execution of the simulator. The challenge is to produce a sound and complete set of constraints for a given simulator implementation so that the correctness of the simulator can be trusted with high confidence. We have developed a large number of constraints targeting these common errors in modeling and tested two simulators, one automatically synthesized from an Architecture Description Language (ADL) [12] specification and the other for SimpleScalar out-of-order simulator [16]. Both of these simulators model sophisticated superscalar processor architectures. We found that both simulators respect the timing and resource constraints they are believed to model. During this process, several "errors" we found turned out to be incomplete constraint specifications. Because this is an iterative process, each run yielded better constraint specifications which provided improved coverage. As both of these simulators are mature and have been verified multiple times using different means of verification in the past, the lack of errors is expected.

The fundamental value of our technique is the assurance it provides when these simulators are modified to model an architectural variation of the original design. The verifier's presence will provide confidence that after the modification the resulting simulator remains a trustworthy model of the architecture under consideration.

We tested various hand-written constraints in ADL and SimpleScalar simulators. The constraints that we tested include:

1. For each instruction type, the stages that must be visited are indeed visited.
2. All stage latencies such as integer operations, divide and multiply latencies, cache access latencies, as well as floating point calculation latencies are respected.
3. Global events, such as rollback are properly included.
4. Resource constraints, such as the number and type of available memory ports are respected.
5. The width of each stage, such as the number of instructions fetched, decoded, and retired match the architecture description.

While the invariant verification provides an assurance and a "yes" or "no" answer to simulator correctness, micro-architecture research can benefit immensely from better understanding the implemented model's behavior under various execution scenarios. We therefore extend the utilization of trace data to model extraction. Extracted models provide the user with the ability to develop further constraints and better understand the implications of newly developed techniques. This is the topic of the next section.

3. DERIVING TEMPORAL MODELS FROM THE TRACE

Cycle-accurate simulators typically model the flow of instructions from one pipeline stage to the next, and it is this timing that eventually provides estimates about how many cycles it would take to execute the given program under the modeled architecture. Depending on the modeled architecture, the number of stages and the latency through each stage will be different. In addition, a range of events will affect the flow of instructions through the stages. We directly derive the pipeline structure, stages simulated, how instructions flow from one stage to the next as well as various events taking place from the event trace and represent them on a temporal graph where the nodes of the graph represents the *state* (as opposed to the stage) an instruction is in. This graphical representation is called an SFTAG (State Flow Temporal Analysis Graph) and used to display the paths instructions follow through the pipeline as well as conditions and events under which such flow occurs.

An SFTAG is a labeled, directed graph $< N, E >$, where N represents the set of nodes and E represents the set of edges. Each node includes one or more state titles representing the stage(s) an instruction is in, and the associated conditions. For example, a node

titled "IF" means that the instruction is in the "Instruction Fetch" stage. Having multiple titles shows that the instruction is either in many stages, or additional events took place simultaneously while the instruction is in that stage. For example, a node titled "II & $W4O$" means that the instruction is in the "Instruction Issue" stage and is waiting for its operands to be ready. Similarly, in the simulated architecture if two sub-operations are performed in the same clock cycle and the trace contains a separate event data for each sub-operation, they will be combined into a *state* which represents both. For example, if the modeled architecture performs execution (EX) and register-file write (WB) in the same cycle, the corresponding state will be $EX\&WB$.

In a graph, an edge is a quadruple $< n_s, I, r, n_d >$, where, n_s represents the source node, I is an interval representing the time taken for the transition, r represents the ratio of instructions performing the state transition, and n_d represents the destination node. The titles for n_s and n_d come from the set $SS \cup \{\texttt{Start}\} \cup \{\texttt{End}\}$, where SS is the set of states shown in the trace file, `Start` is the special start state showing the entrance of the instructions to the pipeline, and `End` is the special end state showing the exit of the instructions from the pipeline. An interval I is shown as a [lower bound, upper bound] pair. For example, the three edges emanating from the node titled "II" in Figure 5 show that 2% of the instructions end at the II stage, 63% of the instructions transition from II to EX&WB taking between 1 to 2 cycles, and 35% of the instructions transition to the EX stage taking between 1 to 2 cycles. Note that the instructions that end at the II stage end due to a rollback.

Algorithm 1 Analyze object transition.

Input: Trace T consisting of event triplets $id, state(s), time(t)$, Max. Samples $maxsamples$, Probability $prob$, Window forward time t_f, Window backward time t_b
Output: Graph g

```
 1: g ← ∅
 2: BINS ← ∅
 3: ξ ← first event
 4: while length(BINS) < maxsamples do
 5:     if random() < prob AND ξ^id ∉ BINS then
 6:         w ← window(T, ξ, t_f, t_b)
 7:         bin ← group(w, o, ξ^id)
 8:         bin ← sort(bin, t)
 9:         bin ← cse(bin, s, t)
10:         bin ← cpe(bin, s, t)
11:         g ← merge(g, bin, s)
12:         BINS(ξ^o) ← bin
13:     end if
14:     ξ ← next event
15: end while
16: return
```

We use Algorithm 1 to create an SFTAG from a trace. A trace used to generate an SFTAG contains three attributes, namely, id, state, and time. The id of an instruction consists of its address and instance. In Figure 4(a) we show a trace segment with three instructions. The algorithm uses a sliding window as explained in Section 2.2 setting the window size such that all events related to a particular instruction are within the window. The process starts by grouping instructions into bins based on their unique address and instance (*group* on line 7). Each bin is sorted with respect to time (*sort* on line 8) and duplicate state names are combined (*combine serial events (cse)* on line 9) into one as shown in Figure 4(b). For example, in the original trace, Instruction 1 is in "Instruction Issue" (II) state at cycles 3 and 4 before transitioning to "Execute" (EX) and "Write Back" (WB) states at cycle 5. The bin contains events $< 1, II, 3 >$, $< 1, EX, 5 >$ and $< 1, WB, 5 >$ to reflect this flow.

We generate a temporary flow graph for each bin as shown in Figure 4(c). In this example, Instruction 1 is in "Execute" and "Write-back" stages at the same time (cycle 5). Therefore, a new state representing parallel states EX and WB is created (*combine parallel events (cpe)* on line 10). Instruction 2 enters the IF state at cycle 1 and the ID state at cycle 5. Therefore, the link from IF to ID is labeled as 4, the time it takes to move from the IF state to the ID state.

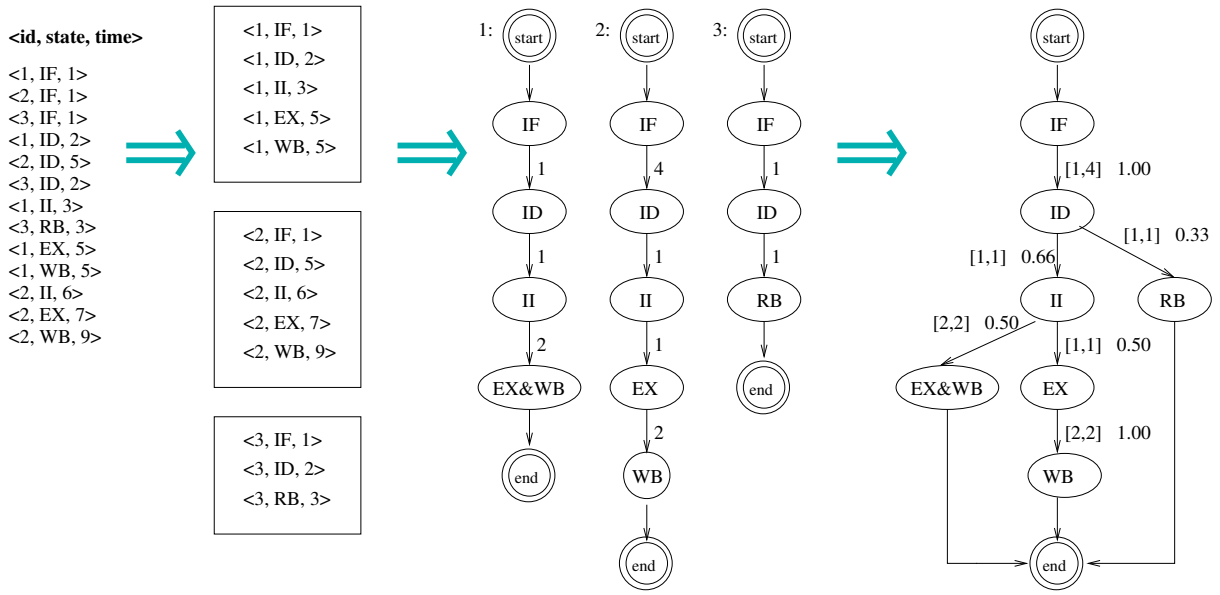
Finally, the temporary flow graphs are combined into a single SFTAG as shown in Figure 4(d) and line 11 of the algorithm. For each transition in the SFTAG, the minimum and maximum cycles needed for the transition are computed and presented next to the edge in $[min, max]$ format. For example, the transition times from IF to ID are $\{1, 4, 1\}$, hence the label in the SFTAG is $[1, 4]$. The decimal value on each edge is the ratio of the instructions that move to the destination state among the instructions in the source state. For example, in Figure 4(d), node II has 2 outgoing edges, one to $EX\&WB$ state and one to EX state. Both edges have the ratio of .50 which means that 50% of instructions in II move to $EX\&WB$ and the remaining 50% move to EX.

4. CASE STUDIES

In this section we present three case studies we conducted using empirical data. The data traces were obtained from FAST ADL [12] and SimpleScalar out of order [16] simulators. We manually instrumented various events in FAST ADL simulator. Instrumented events included major pipeline stages, various stall events and various global events. For SimpleScalar, we used its built-in trace generation and manually added extra events such as memory port access. The first of these studies shows how our technique can extract both the pipeline structure and the temporal behavior of the simulated model. We also illustrate how a human interpreter can write new constraints in FOLCSL by examining the temporal graphs. In the second case study, we compare the temporal graphs obtained for two variants of SimpleScalar. The first simulator faithfully implements a Rambus DRAM model while the second models the original SimpleScalar simple DRAM model. Through the generated histograms we conclude the observed behavior matches to expected behavior for these two models. Finally, we present an analysis of a bus arbiter implementation which makes use of the same algorithms which were used for pipeline temporal models but transposes the data so that instead of modeling instruction flow through the states, *flow of states through instructions* is performed. This transposition exposes resource arbitration by combining all those instructions which are simultaneously in the same stage. This is a powerful concept which can also be used to identify the forwarding requirements of a given architecture by allowing instructions to get their data as if full-forwarding is implemented, obtaining the trace data, analyzing it and implementing a realistic forwarding implementation back in the simulator. We believe each of these case studies are representative of common, time-consuming analysis efforts spent by the micro-architecture community.

4.1 Pipeline Temporal Information

When the simulator event traces are fed through the algorithms discussed in the previous section, two graphs shown in Figure 5 and Figure 6 result. These temporal graphs are obtained directly from trace data, without human intervention.



(a) Trace file (b) The bins (c) Instruction temporary flow graphs (d) Resulting SFTAG

Figure 4: Generating a State Flow Temporal Analysis Graph (SFTAG).

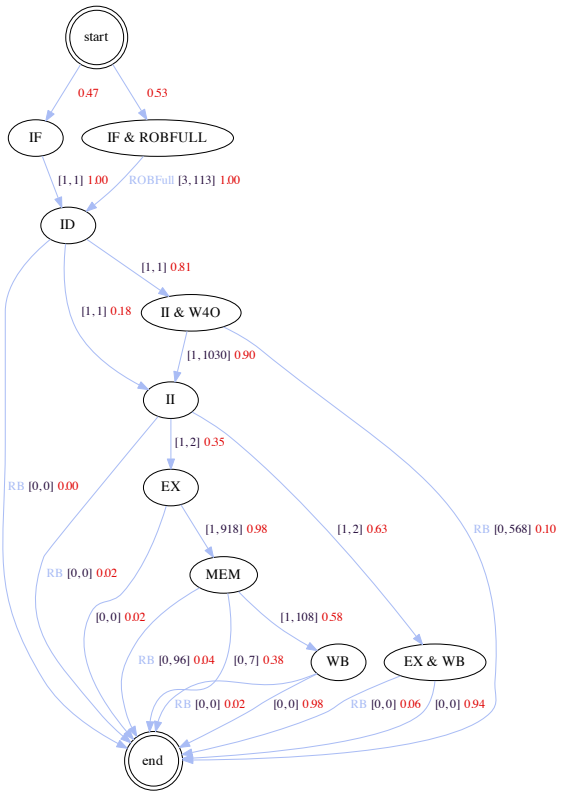


Figure 5: FAST pipeline temporal representation.

Figure 5 can be read as follows: Every instruction starts at IF state or $IF\&ROBFull$ state. The $IF\&ROBFull$ state means that the instruction in IF state and at the same time reorder buffer (ROB) is full. 47% of them will start at IF and the rest will begin with $IF\&ROBFull$ state. Instructions from both states then move to ID. Instructions which move from ID have a transition time of 1 cycle and instructions which originate from $ID\&ROBFull$ have minimum transition time of 3 and maximum transition time of 113. In other words, a full ROB takes minimum 3 cycles and maximum 113 cycles to make itself available again. From ID, instructions can move to II (instruction issue), $II\&W4O$ (instruction issue and waiting for operands) or RB (terminate due to rollback) state. The rest of the graph can be read in a similar fashion.

Figure 6 is similar to Figure 5 except that it represents SimpleScalar out of order architecture. One major difference depicted in both graphs is an instruction's starting state. In FAST, all instructions start at IF but in SimpleScalar an instruction can either start at IF or DA. Looking at the code revealed that SimpleScalar architecture splits *load* or *store* instructions into two instructions in dispatch stage. The trace treats these instructions as generic instructions and since their starting state is in DA (dispatch) and they never visit IF, they appear as if they fork out from the DA state. Alternatively, one can tag those instructions as special instructions and represent them differently but we preferred not to distinguish them. Our approach is to not modify the simulator at all with the exception of adding the necessary instrumentation code and thus keeping the modifications at a minimum. Nevertheless, this is a clear example of how our approach can provide information about what the simulator actually implements. Whether the simulator performed any instruction splitting and if so at which stage were not known to us at the beginning of the case study. This is an example of how the perception of the user and what is actually implemented may differ, which our approach has successfully identified.

Besides showing the user the pipeline temporal information, the graph can also serve as a guide to construct pipeline constraints such as C.4. For example, consider the outgoing edges from ID in

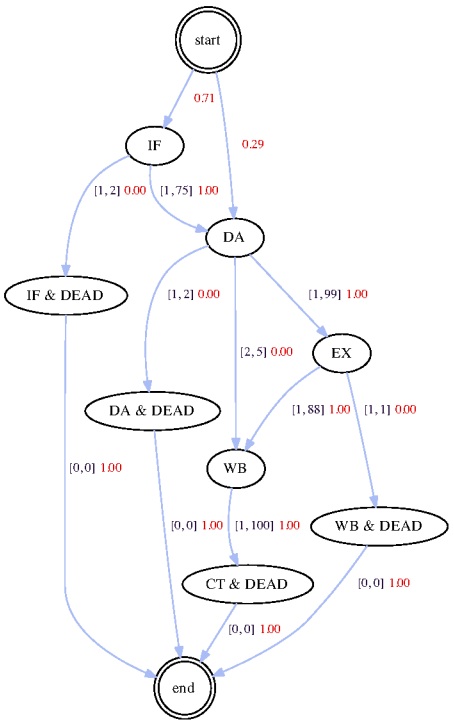


Figure 6: SimpleScalar pipeline temporal representation.

Figure 5. We observe that every instruction that is in ID transitions to one of II state, RB state, or $W4O$ state. This can be encoded in a straight-forward manner as :

$$\begin{aligned}
&\forall z \in T\ \exists y \in T \exists x \in T, \\
&\quad (s^z = ID) \Rightarrow (a^y = a^z) \wedge (c^y = c^z) \\
&\quad \wedge \Big[(s^y = II) \\
&\quad \vee [(s^y = II) \wedge ((s^x = W4O) \\
&\quad\quad \wedge (a^x = a^z) \wedge (c^x = c^z))] \\
&\quad \vee (s^y = RB)\Big]
\end{aligned} \tag{C.6}$$

Similarly, temporal information can be added as follows:

$$\begin{aligned}
&\forall z \in T\ \exists y \in T \exists x \in T, \\
&\quad (s^z = ID) \Rightarrow (a^y = a^z) \wedge (c^y = c^z) \\
&\quad \wedge \Big[(s^y = II) \wedge (t^y - t^z = 1) \\
&\quad \vee [(s^y = II) \wedge (s^x = W4O) \\
&\quad\quad \wedge (a^x = a^z) \wedge (c^x = c^z) \\
&\quad\quad \wedge (t^y - t^z = 1) \wedge (t^x - t^z = 1)] \\
&\quad \vee (s^y = RB) \wedge (t^y - t^z = 0)\Big]
\end{aligned} \tag{C.7}$$

Figures 5 and 6 represent all the possible transitions for instructions. If the behavior of specific types of instructions is of interest, filtering the event trace for an instruction type exposes the specific path taken by the selected instruction type.

4.2 DRAM

One common use of architectural simulators is to verify and test new architectural designs. SFTAGs can help the designer to reason about the behavior of the new design both in its correctness and efficiency. We take SimpleScalar as an example to show how the main memory architecture can affect the processor pipeline.

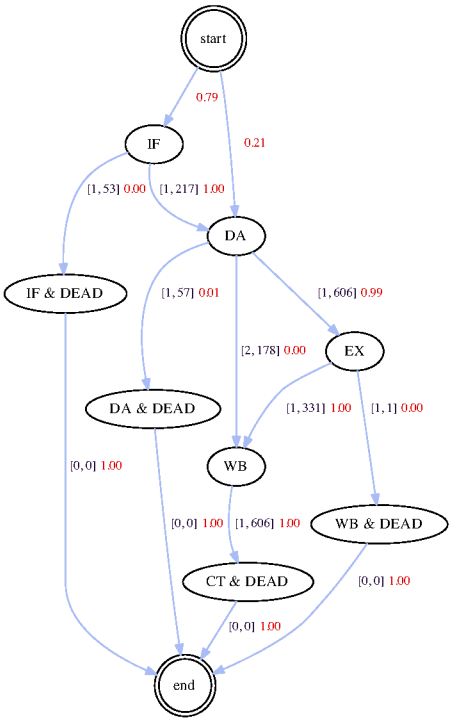


Figure 7: SimpleScalar/Rambus pipeline temporal representation.

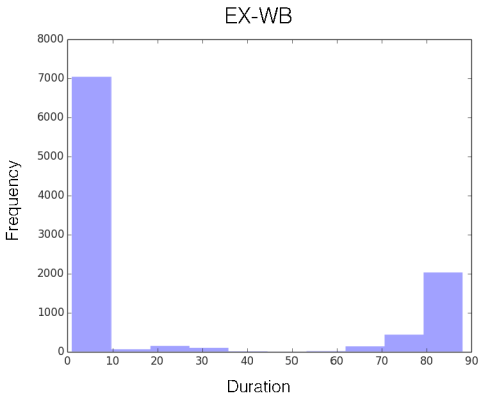


Figure 8: SimpleScalar EX to WB distribution.

Figure 6 shows the SFTAG for a default superscalar machine with a simple DRAM model as used in SimpleScalar 3.0. The SFTAG shown in Figure 7 is from an extension to SimpleScalar where Rambus DRAM is modeled. The two simulators are configured the same, otherwise. We configured SimpleScalar 3.0 with a memory latency of 72 to 88 cycles. Both simulators execute 171.swim from

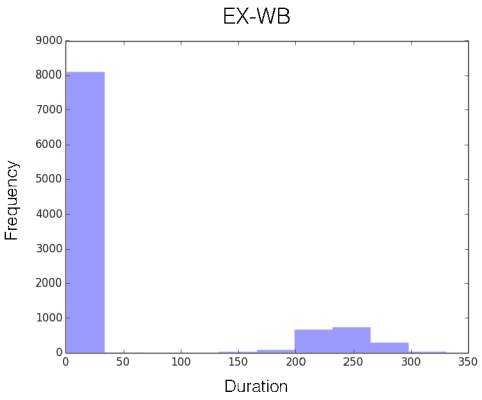


Figure 9: SimpleScalar/Rambus EX to WB distribution.

SPEC CPU2000. The Rambus DRAM can yield a latency of 200 to 300 cycles depending on the memory access pattern. As can be observed from the SFTAGs the increased DRAM latency causes longer transition times between instruction fetch (IF) and dispatch (DA), execute (EX) and write-back (WB), and WB and commit (CT).

We can further generate a histogram for a transition edge of interest in an SFTAG to show the distribution of transition times. The distribution is helpful for us to gain more insight into the simulated architecture and infer its behavior. Figure 8 and Figure 9 show the histograms of the transition from EX to WB for the original SimpleScalar 3.0 and its Rambus extension, respectively. Figure 8 demonstrates that a large number of load instructions indeed cause L2 misses and need to access the main memory. The range of the latencies follows the memory configuration and thus verifies the correctness of memory system simulation. The Rambus DRAM (RDRAM) is much more complicated. Figure 9 suggests that an access to the RDRAM can cause a latency of 200 to 300 cycles. This range fits our configuration of Rambus and thus increases our confidence in the correctness of the implementation.

4.3 Bus Arbiter

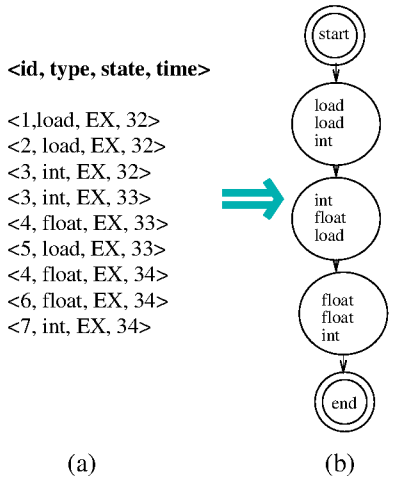


Figure 10: Finding the temporal information about the instructions leaving the EX stage.

We use the concept of state flow graph to find patterns of priority in a simulation. The case study we performed was to look at how instructions are prioritized by a bus arbiter. To achieve this, we first filter all the events where the instructions are in the "Execute" (EX) stage as shown in Figure 10(a). Next, we combine parallel events into single nodes (Figure 10(b)). We convert the graph into a tabular representation showing which type of instructions leave the EX stage as shown in the table below. In the table, the columns LOAD, STORE, INT and FLOAT indicate the number of that class of instructions which are simultaneously present at the EX stage, and columns E-LOAD, E-STORE, etc., indicate how many instructions of the given class leave the stage. We feed this table into the CN2 algorithm [4, 5] and find the rules regarding which instructions leave the execute stage. CN2 is a learning algorithm for rule induction. It takes a set of examples and induces rules in the form of IF-THEN statements. The algorithm uses information entropy as the search heuristic during the rule induction process, similar to decision tree induction algorithms.

LOAD	STORE	INT	FLOAT	E-LOAD	E-STORE	E-INT	E-FLOAT
2	0	1	0	2	0	0	0
1	0	1	1	1	0	1	0

This algorithm yields a set of rules which relate the given combination to the observed outcome. The simulated architecture permits up to 4 instructions at EX, and only 2 instructions exit EX at any given time. Below is a list of the rules generated by CN2.

1. **if** LOAD=1, STORE=1 **then** E-LOAD=1, E-STORE=1
   This rule reads as follows: if there is one LOAD and one STORE in EX stage then during the next transition, the LOAD and STORE will exit EX stage having priority over others.

2. **if** LOAD=0, STORE=0, INT=0, FLOAT=0 **then** E-LOAD=0, E-STORE=0, E-INT=0, E-FLOAT=0
   This rule is trivial. If EX stage contains no instructions then nothing will exit the stage.

3. **if** LOAD=2, STORE=0 **then** E-LOAD=2
   This rule states that if EX stage contains two loads, and no stores, they will leave (irrespective of presence of other types of instructions).

4. **if** STORE=1, INT=3 **then** E-STORE=1, E-INT=1
   STORE has precedence over INT instructions. STORE is given priority, remaining slots are filled by the rest.

5. **if** STORE=1, FLOAT=3 **then** E-STORE=1, E-FLOAT=1
   STORE has precedence over FLOAT instructions (same as above).

6. **if** STORE=2, LOAD=0 **then** E-STORE=2
   This rule states that if EX stage contains two stores, and no loads, they will leave (irrespective of presence of other types of instructions).

The above rules clearly match the implemented arbiter which gives precedence to memory instructions over others. Note that the technique can be used to learn additional information about the inner-workings of a given simulator. If the process yields unintended rules, this may point to significant problems in faithfully implementing the desired model.

Just like a simulator implementation may incorrectly implement an arbiter, it may inadvertently embody an "arbiter" when there is none. This problem originates from trying to map the simulation of

an inherently parallel implementation onto a sequential representation, a well studied problem by Vachharajani et al. [19]. For example, the polling order of the simulator may always give preference to a particular stage, in essence simulating an architecture which embodies an arbiter that always favors that stage. A concrete example is the utilization of ports. A hardware implementation may grant a particular port on a random basis. If the simulator polls a particular stage first, it always will get priority over others, different from the real implementation. In other words, observing a rule which should not be present is equally important as not observing a rule which should be present.

5. PERFORMANCE

Through the careful selection and implementation of our algorithms, we can process very large traces with reasonable running times. In this section, we give an evaluation of SFTAG generation performance for a large set of SPEC CPU2000 benchmarks. All experiments were performed on a machine which has a Quad Core Intel Core i7 processor running at 3.4 GHZ. The machine has 256 KB L2, 8 MB L3 cache and 24 GB of memory. The operating system is OS X 10.9.2 (13C64) with the kernel version Darwin 13.1.0.

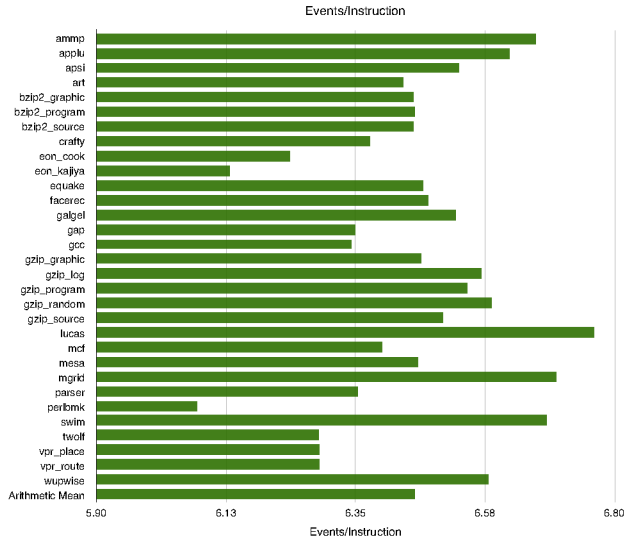


Figure 11: Events per instruction for benchmark programs.

Our algorithm's performance is directly correlated with the number of events that need to be processed. Figure 11 illustrates that the number of events per instruction is variable among different benchmarks, with a mean value of 6.5 events/instruction. SFTAG generation algorithm can process close to a million instructions per second, shown in Figure 12. This rate is close to the performance of an annotated simulator. Hence, on a dual-core system, it is possible to generate SFTAGs in parallel with the simulation as the data becomes available and hence add no extra time on top of the simulation time.

6. RELATED WORK

Mauer et al. [10] give a taxonomy of simulators and classify them into *Integrated*, *Functional-First*, *Timing-Directed* and *Timing-First*. ADL generated simulators used in this study are all *integrated* simulators whereas SimpleScalar simulators can be considered *Functional-First*. Mauer et al. develop the *Timing-First* simulator development approach which relies on a functional simu-

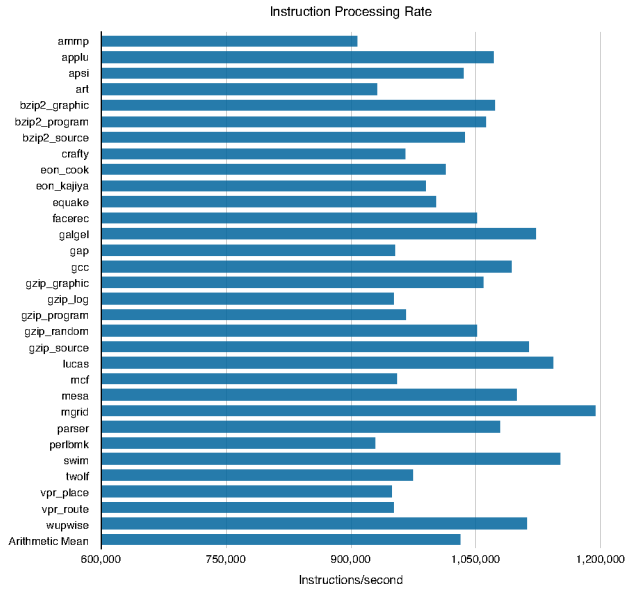


Figure 12: SFTAG processed instructions per second.

lator to verify the functional correctness of the timing simulator. A similar approach is taken by Tomič et al. [18] where dynamic run-time testing is used through a validated emulator. In both approaches, the correctness of the timing simulator needs to be established through manual techniques. Therefore, both techniques can benefit from the approaches presented in this paper. Similarly, one may choose to employ functional validation by relying on an already verified functional simulator and use our techniques to detect timing errors only. Such an approach would have the benefit of specifically concentrating on timing issues and hence reducing the amount of first-order logic invariants that need to be specified.

The FOLCSL framework we developed is a general run-time verification technique, which is applied to the micro-architecture domain. Sargent [15] lists four approaches to decide model validity, namely, model verification, validation, accreditation and credibility [15]. He also discusses two paradigms in the verification and validation process. Various validation techniques such animation, historical data validation and extreme condition tests also mentioned. Sokolsky, et. al. [17] give an overview of runtime verification research, pointing out that there are four important directions. The first is the checking algorithm which includes the property specification language. The second is the generation of traces (instrumentation), i.e., how observations are made and recorded. Third is management of the overhead on system performance imposed by run-time checking. The last direction is feedback and run-time enforcement which address the question of what to do when a violation is discovered. Xiang, et. al. [21] apply various verification and validation in their agent-based simulation model. Agent-based modeling (ABM) is a method used to simulate multiple agents with some type of autonomous behavior. In their conclusion they mention that the suitability of a verification method might depend on the modeling technique used in the simulation.

The Temporal Rover [6] used temporal logic to describe assertions. The assertion statements are written as source code (C, C++, Verilog, VHDL) comments. The assertions are embedded into the original source code via the Temporal Rover parser. IODINE [8] automatically extracts low-level dynamic invariants such as state machine protocols, request-acknowledge pairs and mutual exclu-

sion. GoldMine [20] uses static analysis on RTL design and a decision tree based supervised learning algorithm on simulation traces to generate assertions. Mandouh and Wassal [7] propose a framework that utilizes frequent and sequential pattern mining and known templates to extract RTL design properties.

7. CONCLUSION AND FUTURE WORK

We presented a micro-architecture simulator verification framework which relies on analytical and visual discovery of patterns, as well as invariant checking.

We believe a considerable amount of time is spent in the micro-architecture research community to develop simulators for new techniques and making sure that they faithfully implement the desired models. Our developed framework is a step forward to increase the confidence of researchers in their results. While simulator validation (as opposed to verification) has been carried out for some of the most widely used simulators, such validation is no guarantee that a modified version of the same simulator will correctly simulate the desired model.

Our goal therefore is to develop complete FOLCSL programs for most commonly used simulators which can be used by the researchers together with a given simulator as they are modified. If the new modifications should not change the invariant behavior of the simulator, the FOLCSL specifications can be used as is, or modified accordingly. It is important to note that a limitation of our approach is its reliance on traces generated by a given set of benchmarks. As a result, modeling errors will only be caught if they are exercised by the set of benchmark programs used. While this will not affect the correctness of reported outcomes for the studied benchmark set, uncovered modeling errors may affect future studies when new benchmarks are added. Utilization of a large set of benchmarks upfront may improve the chances that modeling errors will be caught as early as possible.

Our future work involves automatic and semi-automatic derivation of invariant rules from event traces, in addition to the visual and analytical techniques we have outlined. We plan to use additional artificial intelligence techniques to extract further information from event streams. Our intention is to make the developed software available to the micro-architecture research community to contribute towards building simulation frameworks which can be trusted, even after extensive modifications.

8. REFERENCES

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[2] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26:52–60, 2006.

[3] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Computer Architecture News*, 25(3):13–25, June 1997.

[4] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3(4):261–283, March 1989.

[5] J. Demšar, T. Curk, and A. Erjavec. Orange: Data mining toolbox in Python. *Journal of Machine Learning Research*, 14:2349–2353, 2013.

[6] D. Drusinsky. The temporal rover and the ATG rover. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 323–330, London, UK, 2000. Springer-Verlag.

[7] E. El Mandouh and A. Wassal. Automatic generation of hardware design properties from simulation traces. In *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, pages 2317–2320, 2012.

[8] S. Hangal, S. Narayanan, N. Chandra, and S. Chakravorty. IODINE: A tool to automatically infer dynamic invariants for hardware designs. In *Proceedings of the 42nd Annual Design Automation Conference*, DAC '05, pages 775–778, New York, NY, USA, 2005. ACM.

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

[10] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. *SIGMETRICS Perform. Eval. Rev.*, 30(1):108–116, June 2002.

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

[12] S. Önder and R. Gupta. Automatic generation of microarchitecture simulators. In *Proceedings of the 1998 International Conference on Computer Languages*, ICCL '98, pages 80–, Washington, DC, USA, 1998. IEEE Computer Society.

[13] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM reference manual (version 1.0). Technical Report 9705, Rice University, Dept. of Electrical and Computer Engineering, Aug. 1997.

[14] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[15] R. G. Sargent. Verification and validation of simulation models. In *Proceedings of the Winter Simulation Conference*, WSC '11, pages 183–198. Winter Simulation Conference, 2011.

[16] SimpleScalar LLC. SimpleScalar toolset. `http://www.simplescalar.com`. Accessed: 2014-01-01.

[17] O. Sokolsky, K. Havelund, and I. Lee. Introduction to the special section on runtime verification. *International Journal on Software Tools for Technology Transfer*, 14(3):243–247, 2012.

[18] S. Tomič, A. Cristal, O. Unsal, and M. Valero. Rapid development of error-free architectural simulators using dynamic runtime testing. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2011 23rd International Symposium on*, pages 80–87, Oct 2011.

[19] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. The Liberty simulation environment, version 1.0. *Performance Evaluation Review: Special Issue on Tools for Architecture Research*, 31:2004, 2004.

[20] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson. Goldmine: Automatic assertion generation using data mining and static analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 626–629, 3001 Leuven, Belgium, 2010. European Design and Automation Association.

[21] X. Xiang, R. Kennedy, G. Madey, and S. Cabaniss. Verification and validation of agent-based scientific simulation models. In *Proceedings of the 2005 Agent-Directed Simulation Symposium*, volume 37, pages 47–55, 2005.