2014

MINING AND VERIFICATION OF TEMPORAL EVENTS WITH APPLICATIONS IN COMPUTER MICRO-ARCHITECTURE RESEARCH

Hui Meen Nyew
Michigan Technological University

MINING AND VERIFICATION OF TEMPORAL EVENTS WITH APPLICATIONS IN COMPUTER MICRO-ARCHITECTURE RESEARCH

By

Hui Meen Nyew

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY

2014

This dissertation has been approved in partial fulfillment of the requirements for the Degree of DOCTOR OF PHILOSOPHY in Computer Science.

Department of Computer Science

Dissertation Co-Advisor: *Dr. Nilufer Onder*

Dissertation Co-Advisor: *Dr. Soner Onder*

Committee Member: *Dr. Zhenlin Wang*

Committee Member: *Dr. Ossama Abdelkhalik*

Department Chair: *Dr. Charles Wallace*

To my parents,
my family,
friends, and teachers.

Contents

[3] ©2014 ACM. Portions reprinted with permission, from **Hui Meen Nyew**, Nilufer Onder, Soner Onder, and Zhenlin Wang, "*Verifying Micro-architecture Simulators using Event Traces,*" in Proceedings of the 2014 International Conference on Supercomputing (ICS'14).

List of Figures

List of Tables

Preface

This dissertation contains my research work done during my PhD years at the Department of Computer Science at Michigan Technological University. The main contributions of this work are in twofold, namely, mining simulation data and verifying of simulation properties. Two previously published articles are included in Chapter 4 and Chapter 5.

Chapter 4 contains material previously published in AAAI conference in 2013. My contributions to this work were the design and implementation of the entire framework including the FOLSCL language, translator, sliding window algorithm, and event instrumentation. In addition, I designed and conducted the experiments. This work could not have been done without the help of my research committee members: Dr. Nilufer Onder (main advisor), Dr. Soner Onder (co-advisor), and Dr. Zhenlin Wang, who gave me continual guidance and help. They participated in the initial research idea, improved the language and its implementation, and pin-pointed the events needed for instrumentation. They gave me a clear idea about the experimental setup and participated fully in writing the article.

Chapter 5 contains portions of articles previously published in AAAI 2013 and ACM ICS 2014 conferences. My contributions to this work were to design and develop the verification framework and the SFTAG structure. In addition, I applied various data mining techniques to extract the properties of the simulation from the raw data stream and designed various distance measurements. I designed and conducted the experiments. The success of this work stems from the collaborative work of my research committee members Dr. Nilufer Onder, Dr. Soner Onder and Dr. Zhelin Wang, who provided enormous help. They brainstormed ideas, provided valuable insights in designing the SFTAG internal representation and the visualization. Furthermore, they suggested a lot interesting ideas such as clustering, measurement metrics, and a lot important simulation events for additional instrumentation. They spent a lot time writing and perfecting the article with me.

In both of the published articles, Dr. Nilufer Onder's expertise in artificial intelligence and suggestions to use temporal networks and reasoning has led to the development of SFTAG and the expansion of the initial work. Dr. Zhenlin Wang, a professor and researcher who is very knowledgeable in system area, helped me a lot in designing the FOLSCL language and implementing the translator. Dr. Wang's idea of using MinneSPEC instead of full inputs for the SPEC2000 benchmark led to significantly shorter experimental turnaround times. He also suggested various distance measurements which widened my knowledge in this area and led to the design of the "least least squares" distance metric and better clustering results. Dr. Soner Onder, an excellent and prominent researcher in computer architecture,

guided me in the instrumentation process. He spent a lot time helping me analyzing and understanding the data. He also kindly provided the computing resources (research cluster named *Istanbul*) needed for this work. The personal contributions described above are just some of the highlights. All of the work was done collectively and the contributions are gratefully acknowledged.

Abstract

Computer simulation programs are essential tools for scientists and engineers to understand a particular system of interest. As expected, the complexity of the software increases with the depth of the model used. In addition to the exigent demands of software engineering, verification of simulation programs is especially challenging because the models represented are complex and ridden with unknowns that will be discovered by developers in an iterative process. To manage such complexity, advanced verification techniques for continually matching the intended model to the implemented model are necessary. Therefore, the main goal of this research work is to design a useful verification and validation framework that is able to identify model representation errors and is applicable to generic simulators.

The framework that was developed and implemented consists of two parts. The first part is First-Order Logic Constraint Specification Language (FOLCSL) that enables users to specify the invariants of a model under consideration. From the first-order logic specification, the FOLCSL translator automatically synthesizes a verification program that reads the event trace generated by a simulator and signals whether all invariants are respected. The second part consists of mining the temporal flow of events using a newly developed representation called State Flow Temporal Analysis Graph (SFTAG). While the first part seeks an assurance of implementation correctness by checking that the model invariants hold, the second part derives an extended model of the implementation and hence enables a deeper understanding of what was implemented. The main application studied in this work is the validation of the timing behavior of micro-architecture simulators. The study includes SFTAGs generated for a wide set of benchmark programs and their analysis using several artificial intelligence algorithms. This work improves the computer architecture research and verification processes as shown by the case studies and experiments that have been conducted.

Chapter 1

Introduction

"In this, the idea is that we interpret the input from our senses in terms of a model we make of the world. One can not ask whether the model represents reality, only whether it works. A model is a good model, if first, it interprets a wide range of observations in terms of a simple and elegant model. And second, if the model makes definite predictions that can be tested, and possibly falsified, by observation."

— Stephen Hawking, *Origin of the Universe, Berkeley, March 13, 2007*

Computer simulation programs are essential tools for scientists and engineers to understand a particular system of interest. As expected, the complexity of the software increases with the depth of the model used. In addition to the exigent demands of software engineering, verification of simulation programs is especially challenging because the models represented are complex and ridden with unknowns that will be discovered by developers in an iterative process. To manage such complexity, advanced verification techniques for continually matching the intended model to the implemented model are necessary. Therefore, the main goal of this research work is to design a useful verification and validation framework that is able to identify model representation errors and is applicable to generic simulators. To achieve this, we defined five pillars of our design philosophy.

1. *Formalism* - The methodology is based on formal foundations.
2. *User friendliness* - The verification system is easy to use. Simulation properties can be described in a clear and concise manner.
3. *Independence* - The verification process is decoupled from the actual simulator. The modifications needed on existing simulators are minimal.

4. *Reuse value and flexibility* - A property is defined once and used in subsequent verifications. Defined properties can easily be transferred to the variants of the model being studied.

5. *Scalability* - It is possible to process very large amounts of data, and data streams with unknown length. The execution overhead of the verifier is minimal.

Our techniques rely on generating event traces from an execution of the target simulator and using the trace in two complementary processes. Figure 1.1 shows the general framework. For the first process, we developed a first-order logic based language, which we call, *First-Order Logic Constraint Specification Language (FOLCSL)*. Using the language, *invariants* of the model under consideration are specified. Examples of such invariants in the computer architecture domain include, every fetched instruction must be decoded (instruction pipeline constraint), no more than two load instructions can simultaneously access the cache (resource constraint), and the execution step of an integer type instruction takes a single cycle (temporal constraint). From the first-order logic specification, we automatically synthesize a verification program as shown in Figure 1.1(a). This verification program reads the event trace generated by running the simulator using a particular input set as shown in Figures 1.1(b) and 1.1(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 for the set of inputs tested, the simulator has faithfully followed the model. Unfortunately, the domain of invariants is large and even domain experts might omit necessary 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. To represent the simulated model we developed a representation, which we call *State Flow Temporal Analysis Graph (SFTAG)*. An SFTAG presents a temporal, probabilistic view of the states encountered during simulation. For example, in the computer micro-architecture domain, the first nodes of an SFTAG may specify that 47% of the instructions start at the "instruction fetch" stage, whereas 53% start at the "reorder buffer full" stage. Presenting the constraints in the form of temporal graphs enables the 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 process. These two processes are used iteratively with different inputs as shown in Figure 1.1, 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 and visually describing what the simulator implements. Furthermore, this is done in a very practical and general manner. Event traces can be easily

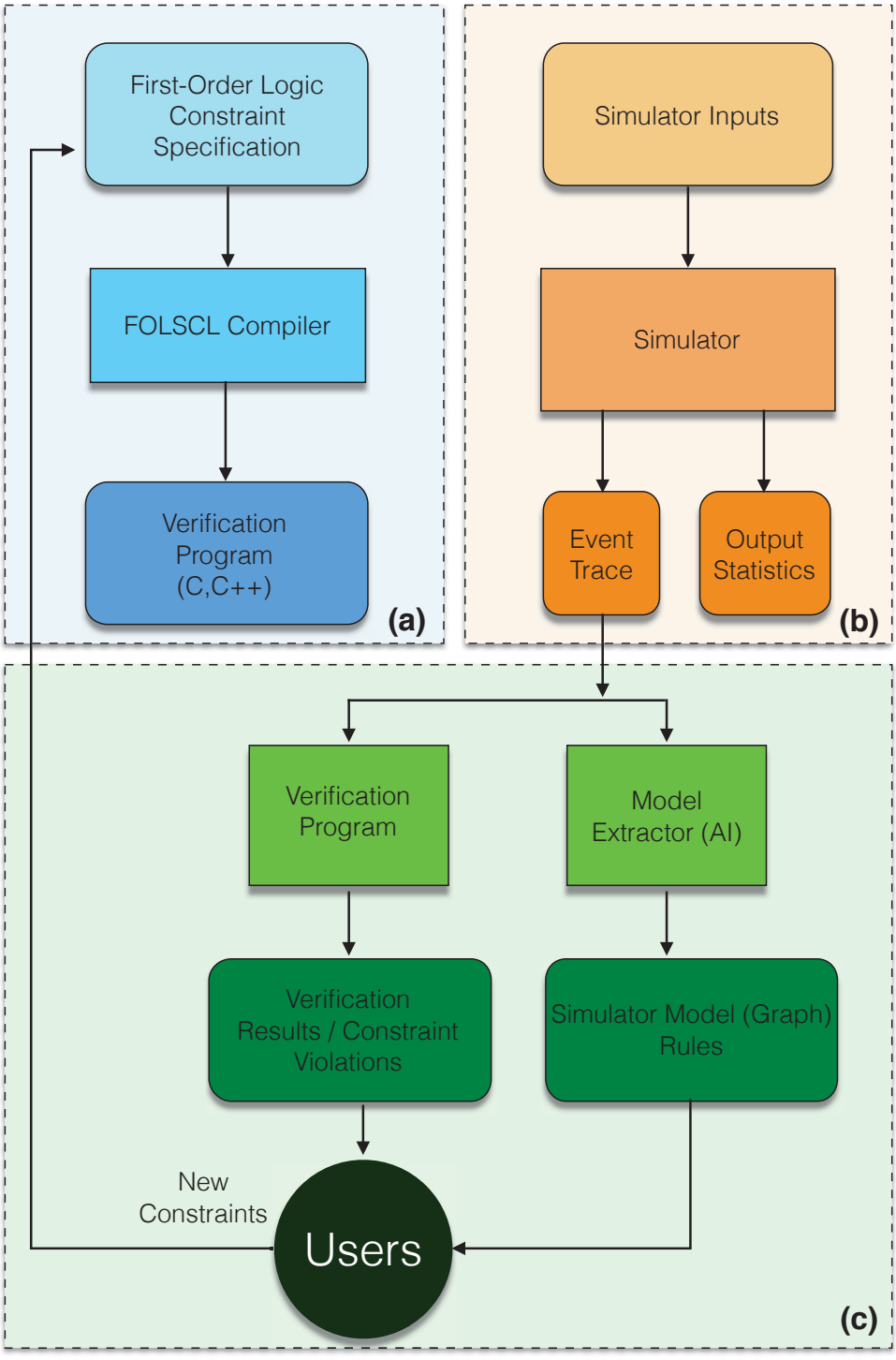

Figure 1.1: Simulator verification using event traces.

generated from any type of simulator by inserting output statements at specific points. Most existing simulators already provide mechanisms to generate an event trace. For example, SimpleScalar [1] and Gem5 [2] simulators used by computer architects produce event traces by simply setting an option. It also is easy to generate an event trace in automatically synthesized simulators such as FAST ADL [3] by augmenting appropriate points in the description.

Our experiments and case studies 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.

The main contributions of this research work are twofold. First, we designed a formal verification language and implemented the software that allows users to describe simulation properties and check the properties against the output events. Second, we developed and implemented an algorithm that is capable of processing large data sets and representing the temporal information in a graphical form that is amenable to both visual inspection and automatic analysis. Our framework improves the computer architecture research and verification processes as shown by the case studies and experiments we have conducted.

The organization of this dissertation is as follows. In Chapter 2, previous work related to this research is discussed. Techniques and algorithms that this research is built upon are reviewed in the same chapter. Chapter 3 gives an overview of the framework. The constraint language, First-Order Logic Constraint Specification Language (FOLCSL), is presented in Chapter 4. In Chapter 5 we describe State Flow Temporal Analysis Graphs (SFTAGs) and the algorithms to construct them. We illustrate the application of SFTAGs through case studies and experiments in Chapter 6. The summary of our research, conclusion and future work are given in Chapter 7.

Chapter 2

Background and Related Work

"Perfection is not attainable. But if we chase perfection, we can catch excellence."

— Vince Lombardi

"Never perfect. Perfection goal that changes. Never stop moving. Can chase, cannot catch."

— Abathur

The main topics in this section include verification, validation, and rule induction. We provide a brief review of verification and validation in a general project management context, within software engineering, and in the context of simulation software. We present a summary of rule induction techniques along with metrics of rule quality.

2.1 Verification and Validation

Verification and validation are the foundations of software testing. Without referring to the exact definitions, some of us might think they mean the same thing. But they do not. The definition of these words depend on the context.

The Project Management Body of Knowledge (PMBOK) Guide [4] definitions cover a wide scope providing the following definitions:

Verification - The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process.

Validation - The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers.

One of the earliest definitions of software verification and validation comes from Barry Boehm [5] who defined both terms as follows:

Verification - to establish the truth of the correspondence between a software product and its specification (derived from Latin word *veritas*).

Validation - to establish the fitness or worth of a software product for its operational mission (derived from Latin word *valere*).

In layman's terms, verification can be expressed as "Am I building the software right?" and validation can be expressed as "Am I building the right software?" [5]. The distinction between these two terms is important because it defines the focus, scope, and the life-cycle for each process.

On the other hand, Sargent's [6] definitions are more specific toward simulation software. His definitions are as follows:

Model verification - ensuring computer program of the computerized model and its implementation are correct.

Model validation - substantiation that a computerized model within its domain of applicability possesses a satisfactory range of accuracy consistent with the intended application of the model.

We adopt Sargent's definitions because of their specificality and relevance to our work.

Other closely related concepts are *model credibility* and *model accreditation* [6]. Model credibility is concerned with developing the users' confidence and trust they require in order to use a model and in the information derived from that model. Model accreditation determines if a model satisfies specified model accreditation criteria according to a

specified process. The definition of the term model accreditation also debatable. Balci [7] mentioned that the definition given by the Department of Defense differs from the ISO definition. By Department of defense standard, accreditation is the certification that a model or simulation is acceptable for a specific purpose. On the other hand, ISO defines accreditation as the formal recognition of a body or person that is competent to carry out specific tasks and certification as third party written assurance that a product, process or service conforms to specific characteristics. Balci used the ISO definitions because they are more widely used in engineering disciplines and educational sectors. According to the ISO standard, first party refers to the application sponsor, second party refers to the application developer and the third party is the independent certification agent.

In computer architecture research, the term *validation* is used in a manner that is close to Sargent's definition. To validate a model, the *accuracy* of the implemented model is compared with an actual implementation of a specific processor. For example, the SimpleScalar Sim-alpha implementation, which faithfully implements the specific structure of the Alpha 21264 processor, has been validated by comparing the estimated cycle counts produced by the simulator with the cycle counts obtained by running the same benchmark program on the physical machine [8, 9]. The results indicated that the simulated cycles are within 2% of the actual implementation [10].

2.2 Simulation Verification and Validation

A specific category of verification is the verification of simulations. Recently, the accuracy of simulation models have attracted interest, especially in scientifically and politically sensitive areas such as global climate models [11, 12] and medical decision making [13]. Due to the large amount of simulation data that needs to be interpreted and verified by domain experts, a range of visualization techniques have been developed.

Chen et al. describe visualization as a search process where the users start with a set of data and visualization tools, and search for the best parameters and configurations to visualize the set of data [14]. Using the same concept of search, Ahrens et al. developed an iterative verification method that is used for comparing simulations that run different algorithms [15]. The method involves four steps, namely, identifying the features to compare, making an hypothesis about the identified feature, visualizing the identified features, and finally creating quantitative plots or charts that reveal the differences between the simulations. The process is repeated until the simulators used are verified. Han et al. developed a three step methodology visualizing the assembly line of modular buildings. In the first step, the proposed production line is developed using *Value Stream Mapping*. In the second step, the simulation of the proposed design is built for verification. Finally, 3D visualizations

are automatically developed for validation based on the outputs of lean production and simulation.

Sargent proposed another simulation verification and validation process where a simulation model is separated into a conceptual model and a computerized model [6]. The conceptual model is developed first and is followed by conceptual model validation. This process is repeated until the model is satisfactory. Next, the conceptual model is turned into a computerized model and is followed by computerized model verification. Similarly, the process is repeated until the model is satisfactory. This verification process was adopted by Huang et. al.'s agent based simulation where the conceptual model was evaluated by six domain experts and the computerized model verification was done by code walkthrough, trace analysis, input/output testing, and boundary testing [16].

Verification can also be done by analyzing the simulation output using statistical techniques (e.g., simulations in computational fluid dynamics [17], agent-based modeling [18]). Sanchez described the important issues that researchers should be aware of while analyzing simulation outputs [19]. One of the issues is the *initialization bias* which means that the outputs that include the warm-up period may cause the overestimation or underestimation of the steady state performance. The initialization bias is also one of the well researched problems in computer architecture domain. SimPoint [20] and SMARTS [21] reduce the initialization bias by warming up the simulator before collecting the simulation data.

Kleijnen discussed the suitability of specific statistical tests based on the availability of data [22, 23]. He identified three situations, (i) no real data is available, (ii) real data and simulated data are available but input data is not available, and (iii) both input and output data are available. For the first case he suggested using sensitivity analysis, for the second case, student *t*-test is appropriate, and regression test or bootstrapping can be applied for the last case.

2.3 Runtime Verification

Runtime verification is a process that verifies a program's dynamic execution behavior against formally specified behavioral properties [24] and has its roots in model checking [25]. Leucker and Schallhart define runtime verification as "the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property" [26].

Broadly speaking, runtime verification involves four major research areas [27]:

1. logics for monitoring
2. online checking algorithms
3. extraction of observations necessary for checking
4. reduction of checking overhead

The logics used for monitoring provide a means to specify the behavioral properties of a running program and are mainly based on linear temporal logic (LTL) [28]. LTL allows reasoning about states using four operators, namely, `next` (property holds in the next state), `final` (property will hold at a state in the future), `global` (property holds at every state on the path), `until` (property hold until finally another property holds), and `release` (a second property holds along the path up to and including the first state where the first property holds). Recently published LTLs include AspectJ, which provides a runtime verification framework for Java programs [29], and EAGLE, which provides support for recursive parameterized equations [24]. Comparison of LTLs with different levels of expressivity is provided by Bauer, Leucker and Schallhart [26, 30].

The monitoring algorithms for runtime verification are built on model checking algorithms, which are updated to work online and incrementally [27, 31]. Generation of traces, also commonly known as *instrumentation*, deals with the question “How are observations made and recorded?". Inefficient implementation of runtime verification can degrade a system’s performance significantly. Major sources of overhead reside in observation extraction and checking algorithms. This issue falls into overhead management [27, 32, 33]. Last but not least, feedback and runtime enforcement addresses the question of what to do when a violation is discovered.

Recent work related to checking and monitoring is the Temporal Rover [34]. It uses temporal logic to describe assertions. The assertion statements are written as source code comments (C, C++, Verilog, VHDL) which are then embedded into the original source code via the provided parser. In the mechatronics field, runtime verification is utilized on a self-optimizing system. Zhao et al. [35] developed a service on top the real time operating system (RTOS) to dynamically monitor and check the consistency and safety of a system after performing component replacements. The authors first model the system using real-time UML state charts. Following this, a series of translations are applied to process the model into *Kripke* structure and *Büchi* automata which will be fetched during the verification process. Further applications of runtime verification are in multi-agent simulations of natural domains such as biological, cognitive and social domains. Bosse and

colleagues show that the widely used differential equations in multi-agent system modeling are inadequate [36]. They address it by designing logic based Temporal Trace Language (TTL) to check and analyze multi-agent systems dynamically. Runtime verification can also be used to ensure C memory safety. Rosu et al. proved that strong termination and strong memory safety are undecidable in general, but strong memory safety of strong terminating programs is decidable, thus it is runtime verifiable [37].

2.4 Invariant Extraction

Verification can also be done by observing a simulator's invariant properties such as every instruction must terminate at some point. If an observed invariant property doesn't make sense, it might indicate that the simulator has some flaws. For example, an invariant of a processor simulator stating that there exists an instruction that stays in the pipeline for a million cycles raises a red flag because it is unlikely to happen in a real implementation.

Some work related to this area has been published in recent years. One example is IODINE [38] that automatically extracts low-level dynamic invariants such as state machine protocols, request-acknowledge pairs, and mutual exclusion. Another example is GoldMine [39]. It performs static analysis on a register-transfer level (RTL) design and constructs a decision tree using a supervised learning algorithm on a simulation trace. The decision tree represents the assertions of the design. Another recent work by Mandouh and Wassal [40] utilizes frequent and sequential patterns mining and known templates to extract RTL design properties.

Extracting invariant properties can be done in variety of ways. Data mining techniques are very efficient for this purpose. For example GoldMine utilize decision tree algorithms and Mandouh and Wassal use frequent and sequential pattern mining algorithms to automatically generate hardware design properties. In the next section, we describe some data mining techniques we used in our work.

2.5 Rule Induction

Rule induction techniques are effective in extracting properties of a simulation that follows certain patterns. We used rule induction to extract bus arbiter properties which are best described in the form of *if* and *then* rules (Section 5.3.3). Two widely used rule formats are association rules and decision rules. We begin by describing the basic concepts and follow

by listing some of the mining algorithms.

2.5.1 Association Rules

The concept of *association rules* is formally defined in Rakesh Agrawal et. al.'s 1993 paper [41] where they use it to study a large database of customer transactions in a supermarket environment. The following definition is based on Agrawal et al. Let $\mathscr{I} = I_1, \cdots, I_m$ be a set of binary attributes with size m, called *items*. Let T be a database of transactions. Each transaction $t \in T$ is represented as a binary vector, with $t[k] = 1$ if t contains item I_k and $t[k] = 0$ otherwise. Let X be a subset of $\mathscr{I}$. A transaction t *satisfies* X if for all items I_k in X, $t[k] = 1$.

An association rule is an implication of the form $X \implies I_j$, where $X \neq \emptyset$, $X \subset \mathscr{I}$, $I_j \in \mathscr{I}$ and $X \cap I_j = \emptyset$. The rule $X \implies I_j$ is satisfied in T with the confidence factor c where $0 \leq c \leq 1$ if and only if at least $c\%$ transactions in T that satisfy X also satisfy I_j. In some text [42] the I_j is replaced with a set Y, i.e., $X \implies Y$, where $X \neq \emptyset$, $Y \neq \emptyset$ $X, Y \subset \mathscr{I}$, and $X \cap Y = \emptyset$.

2.5.2 Decision Rules

Besides association rules, there is another class of rules called *decision rules*. If we regard a decision tree as graphical representation then its text equivalent representation is a set of decision rules. Decision rules are also known as classification rules [43]. A rule can be written in disjunctive form (Horn clause) or in implication form. In disjunctive form it follows Horn form which is a clause with at most one positive literal (unnegated literal).

Disjunction form:

$$\neg p \vee \neg q \vee \cdots \vee \neg t \vee u \tag{2.1}$$

If a rule is in Horn form, it can easily be rearranged into an equivalent implication form.

$$p \wedge q \wedge \cdots \wedge t \implies u \tag{2.2}$$

Nevertheless, the implication form is more commonly used. Sometimes it is written in the

form of an *if-then* statement.

$$\text{if } p \text{ and } q \text{ and } \cdots \text{ and } t \text{ then } u \tag{2.3}$$

The *if* part (left side) of a rule is known as the rule *antecedent* or *precondition*. The *then* part (right side) of a rule is known as the rule *consequent* and it contains a class prediction. If the condition in a rule antecedent holds true for a given tuple, we say that the rule *covers* the tuple.

Several rule metrics has been developed to measure the quality of the rules. These metrics are used in filtering, pruning and the measurement of confidence during or after the rule extraction process. In the next two sections, we describe some of the well known rule measurement metrics.

2.5.3 Association Rule Metrics

The *support* of a rule $X \implies Y$ that holds in a set of transactions T is defined as:

$$supp(X \implies Y) = P(X \cup Y) \tag{2.4}$$

The notation $P(X \cup Y)$ indicates the probability that a transaction contains the union of sets A and B. This should not be confused with $P(X \text{ or } Y)$.

The *confidence* of a rule $X \implies Y$ that holds in a set of transactions T is defined as:

$$\begin{aligned} conf(X \implies Y) &= P(Y|X) \\ &= \frac{supp(X \cup Y)}{supp(X)} \\ &= \frac{freq(X \cup Y)}{freq(X)} \end{aligned} \tag{2.5}$$

where $freq(A)$ is number of transactions that contain the itemset A. A set of items is known as *itemset*.

Lift is a correlation measure of two itemsets. The lift between itemset X and itemset Y can be measured as follows:

$$lift(X,Y) = \frac{P(X \cup Y)}{P(X)P(Y)} \tag{2.6}$$

If the lift value less than 1, it means the occurrence of X is *negatively correlated* with the occurrence of Y. If the lift value is greater than 1, it means the occurrence of X is *positively correlated* with the occurrence of Y. If the lift value is zero, it means X and Y are independent and there is no correlation between them.

Support and confidence are usually used to measure rule interestingness. For example one might be interested on rules with 100% confidence. Lift, on the other hand, serves as an extra measurement metric, that is rule correlation measurement.

2.5.4 Decision Rule Metrics

The *coverage* of a rule is the ratio between the number of records covered by a rule and the number of records in a data set.

$$coverage(R) = \frac{n_{covers}}{|D|} \tag{2.7}$$

where R is a rule, n_{covers} is the number of records covered by R and $|D|$ is number of records in D.

The *accuracy* of a rule is percentage of the rule can correctly classify, defined as:

$$accuracy(R) = \frac{n_{correct}}{n_{covers}} \tag{2.8}$$

where $n_{correct}$ is number of records that are correctly classified by R.

2.5.5 Rule Mining Algorithms

Decision rules can be extracted by first constructing a decision tree using algorithms such as Quinlan's ID3 [44], ID3's successor [45], and CART by Breiman et al. [46]. Then, the paths from the root node to each leaf node in the tree are traced to form the rules. Alternatively, we can use sequential covering algorithms to learn the rules directly from a data set. Widely used sequential covering algorithms include AQ [47] and CN2 [48]. The general strategy is shown in algorithm 1:

Algorithm 1 Sequential Covering Algorithm

Input:
D data set
C set of class values
V set of all attributes and their possible values
Output:
R set of rules
1: $R \leftarrow \emptyset$
2: **for all** $c \in C$ **do**
3: **repeat**
4: $r \leftarrow learn1Rule(D, V, c)$
5: remove records covered by r from D
6: $R = R \cup r$
7: **until** termination condition(s)
8: **end for**
9: **return** R

Algorithm 1 works by generating a rule that correctly classifies some instances in D that belong to a class c, removing the instances that are covered by the generated rule, and repeating the process for the remaining instances. The termination condition usually is "when all the instances in D that belong to class c are correctly classified."

The construction of rule r in algorithm 1 can be done in many different ways. One method is shown in algorithm 2. Since the consequent is fixed to a class c, only the antecedent needs to be constructed. The process starts with an empty antecedent and on each iteration it adds the most promising attribute-value pair (i.e. $a^* = v^*$) to the rule's antecedent. The process repeats until the rule r correctly classifies all the class c instances in D.

Algorithm 2 learn1Rule

Input:

D data set
V set of all attributes and their possible values
c a class value

Output:

r a rule

1: $\alpha \leftarrow \emptyset$ (antecedent)
2: **repeat**
3: **for all** attribute-value pair, $(a = v) \in V$ **do**
4: $r' \leftarrow$ *if* $\alpha \wedge (a = v)$ *then* c
5: computeAccuracy(r')
6: **end for**
7: let $(a^* = v^*)$ be the attribute-value pair of the maximum accuracy over D
8: $\alpha \leftarrow \alpha \wedge (a^* = v^*)$
9: $V = V - \{a^*\}$
10: $r \leftarrow$ *if* α *then* c
11: **until** r correctly classifies all c instances of D
12: **return** r

In the next section, we describe artificial intelligence techniques for representing, reasoning with, and mining temporal information.

2.6 Reasoning with Temporal Data

Temporal reasoning is an important field of artificial intelligence as evidenced by continual developments and growing number of applications [49]. Temporal reasoning operates on a formal representation of time and provides a means to reason about temporal aspects of knowledge [50, 51]. There are two main ways of representing temporal information. *Qualitative models* represent relations between events such as "A occurs during B" or "A is before B" [52, 53]. *Quantitative models* represent numeric information using points of time [54] or using intervals of time [55]. The reasoning problems solved using temporal representations can be broadly classified into three as follows:

1. *Consistency checking*: Finding whether a collection containing temporal data is free of contradictions.

2. *Inference*: Answering queries based on temporal data.

3. *Optimization*: Minimizing temporal networks or finding a minimal set of temporal constraints.

Simple temporal networks (STNs) introduced by Dechter, Meiri, and Pearl [55] are widely used as a representation of quantitative intervals. STNs have been extended in a range of directions, including disjunctive temporal networks [56, 57, 58], temporal networks with alternatives [59], preferences and uncertainty [60, 61, 62, 63], fuzzy preferences [64], and time dependent temporal constraints [65].

Temporal data mining is the application of artificial intelligence and statistical techniques to extract information from static or longitudinal temporal data [66]. Widely studied domains of temporal data mining include finding *temporal association rules* [67], discovery of *frequent sequences* [68, 69, 70], and describing and discovering common trends in *time series* [71, 72, 73, 74]. Examples of recent work in these areas are finding calendar-based [75] or relative [76] temporal association rules, finding frequent sequences in longitudinal electronic patient records [77] or spatiotemporal human activity data [78], and finding patterns of temporal variation in online media [79]. *Higher order mining* refers to mining results of temporal discovery for further discoveries such as finding trends or changes in association rules [80, 81].

In our work, we generate temporal flow information in the form of probabilistic state flow graphs called SFTAGs as explained in Chapter 5. We analyze SFTAGs by clustering with respect to edges and benchmark programs. We also process these graphs for higher order relationships by clustering the clusters as discussed in Chapter 6. In the next chapter, we present an overview of our framework's design and its application domain.

Chapter 3

Application Domain and Framework

"If it is not useful or necessary, free yourself from imagining that you need to make it. If it is useful and necessary, free yourself from imagining that you need to enhance it by adding what is not an integral part of its usefulness or necessity. And finally: If it is both useful and necessary and you can recognize and eliminate what is not essential, then go ahead and make it as beautifully as you can."

— Paul Rochleau & June Sprigg, *Shaker Built: The Form and Function of Shaker Architecture*

While the techniques developed in this dissertation can be applied to any simulator, the main emphasis of our work is the validation of the timing behavior of micro-architecture simulators. Like many other fields, state of the art computer architecture research inherently relies on software simulations to develop new ideas and to improve existing well-established designs. We can broadly classify these simulators into three main groups:

1. Functional Simulators: Functional simulators implement an interpreter for the simulated architecture's instruction set. No hardware details are modeled. By using a functional simulator, researchers can start developing the system software for a new architecture before the architecture is built, as the functional simulators enable simulated execution of the programs compiled for the new architecture. Since they do not model any of the specifics of the architecture, they are mainly used to develop and debug system software. They are also useful for collecting statistics such as the number and the type of instructions executed by a benchmark program, or generate instruction and data address traces which can be used to study the hit/miss behavior of caches and the memory subsystem.

2. Cycle-Accurate Simulators: Cycle-accurate simulators model a processor architecture in sufficient detail so that accurate information about how a given program would execute under a new design can be quantitatively estimated. The simulator typically simulates the behavior of individual hardware structures within a processor, such as registers, register files, pipeline stages, buffers, arbiters as well as the details of the datapath, including on-processor busses. While it is impossible to predict the attainable clock-speed for the processor (i.e., the cycle time), the number of cycles it would take to execute the given program is accurate and will match the actual processor when it is built, if the simulator has been correctly implemented.

   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 [82, 1], RSIM [83], M5 [84], GEM5 [2] 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 [85].

3. Full System Simulators: Typically, whether it is a functional or cycle accurate simulator, operating system calls are intercepted and executed on behalf of the simulated program. Components of a computer system other than the CPU, such as various I/O devices, are not modeled. Alternatively, the simulator may also implement the behavior of these components, allowing the simulation of a complete computer system. In such a case, it is possible to "boot" an operating system within the simulator framework. Such a framework then becomes usable for developing the device drivers and studying the behavior of the operating system as well. Such simulators are called *full-system simulators*. Due to significant processing overhead, the CPU is modeled only at the functional level and not at the cycle-accurate level. Since the boundary between functional and cycle-accurate simulation is not rigid, it is also possible to simulate certain components at the cycle-level and the rest in a functional manner.

   Full-system simulators are also useful to create "boot and run" environment for machines that no longer exist, allowing us to preserve the history of computing.

This dissertation specifically targets cycle-accurate simulators. Having a formal verification framework is important in this context due to two reasons. First, while generation from an architecture description language can facilitate the application of formal validation techniques for cycle-accurate simulators, using an architecture description language in itself will not prevent model representation errors. Second, 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 cycle-accurate 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.

In order to facilitate a better understanding the issue of timing and how our framework fits into the cycle-accurate simulation domain, we briefly review the general structure of cycle-accurate simulators. Mauer et al. [86] give a taxonomy of simulators and classify them into *Integrated*, *Functional-First*, *Timing-Directed* and *Timing-First*. In this work, we used ADL [3] generated simulators used in this study are all *integrated* simulators whereas SimpleScalar [1] simulators can be considered *Functional-First*. Irrespective of this classification, almost all cycle-accurate simulators embody a main loop of simulation such that each iteration of the loop corresponds to one clock cycle of the architecture. Within each iteration, procedures (or methods) which implement the functionality of individual hardware components are called. For example, a cycle-accurate simulator which implements a two pipeline stage micro-architecture of *instruction fetch* and *execute* will first calls the instruction fetch and then the execute procedures. Since the modeling is performed at a *register level*, (i.e., the changes in the values of processor registers are accurately reflected in the corresponding program variables) calling of the instruction fetch might result in loading the instruction word into the simulated instruction register. The execute procedure then can perform the desired operation, and the loop is iterated again. Typically, machine registers, buffers, etc. become variables in the simulator, including the memory which can be represented as an array.

It is important to observe that in this modeling approach, the simulator generates a series of *events* which result from the execution of procedures that model the behavior of various hardware components. While a functional-first simulator would act like a functional simulator in terms of actual interpretation of instructions, to qualify as a cycle-accurate simulator, it has to model and find the actual clock cycle a particular events happens, such as the execution or writing back the result of an instruction. Therefore, cycle-accurate simulators can easily be annotated to print out a trace file which contains the events that take place and the clock cycle at which each event happens. We explain the structure of the trace file in Section 4.1.

In addition to performing simulator verification, our framework can be used as a debugging tool while developing a simulator. We present our working framework through a use-case scenario example. Alma, a micro-architecture researcher developed a novel processor prototype in an existing simulator. She wants to make sure that the modified simulator does not break the unmodified portion of the simulator and the modified portion of the simulator correctly represents her conceptual model.

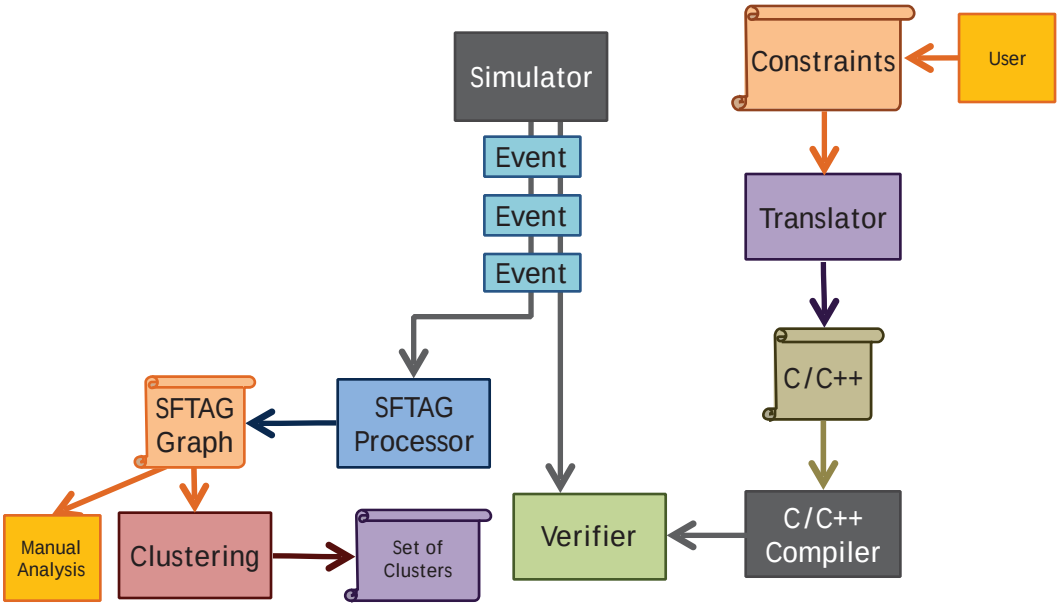

Figure 3.1: Framework overview.

Using our framework, all Alma needs is to instrument the events that she is interested in and write the constraint specifications in FOLCSL based on her conceptual model. The written set of constraint specifications are translated to C/C++ code via FOLCSL's translator and then compiled into an executable verifier. The right section of Figure 3.1 depicts the process. To check whether the simulator adheres to the constraint specifications, Alma runs the verifier in parallel with the simulator. The events output from the simulator are streamed into the verifier and the verifier checks the events against the constraint specifications. The verifier signals Alma if it finds any violations. In addition, Alma can utilize the output events from the simulator to generate SFTAG graphs. Instead of running the verifier, Alma runs the SFTAG processor which takes the events generated by the simulator and processes them into SFTAG graphs. The SFTAG graphs show the temporal relationship of the events. Furthermore, if Alma can produce the SFTAG graphs of the original simulator and compare them with the current graphs, she can learn the changes between the graphs and reason about the modifications she made. Alma can further study the SFTAG graphs as a whole using data mining techniques such as clustering to reveal difficult to see patterns. The left section of Figure 3.1 depicts this process.

In the next two chapters, we explain the components of our framework in detail. In Chapter 4, we present the structure of the trace files and events, the first-order logic constraint specification language (FOLCSL) we developed, and the verification algorithms that use FOLCSL constraints. In Chapter 5, we present the SFTAG structure, the algorithms to generate SFTAGs, and the case studies we conducted.

Chapter 4

First Order Logic Constraint Specification (FOLCSL) [1] [2]

"A mind all logic is like a knife all blade. It makes the hand bleed that uses it."

— Rabindranath Tagore

"Logic will get you from A to B. Imagination will take you everywhere."

— Albert Einstein

First-order logic constraint specification language (FOLCSL) is designed to specify the invariants which must hold during the execution of the simulator. The language allows constraint specification using a subset of 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 and then follow with the instrumentation methods.

[1] ©2013 AAAI. Portions reprinted with permission, from **Hui Meen Nyew**, Nilufer Onder, Soner Onder and Zhenlin Wang, "*A First-Order Logic Based Framework for Verifying Simulations*" , in Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence (AAAI 2013), Pre-PhD student Abstracts.

[2] ©2014 ACM. Portions reprinted with permission, from **Hui Meen Nyew**, Nilufer Onder, Soner Onder, and Zhenlin Wang, "*Verifying Micro-architecture Simulators using Event Traces,*" in Proceedings of the 2014 International Conference on Supercomputing (ICS'14).

4.1 Trace Files and Events

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, $\xi^i =< a^i, s^i, t^i >$ is an event generated by a processor simulator where a^i is the instruction sequence number, s^i is the pipeline stage or special events such as reorder buffer full, 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 a^i-th instruction is at state s^i. A sample trace generated by the simulator is:

1. 114, *IF*, 1008
2. 114, *ROB Full*, 1008
3. 109, *EX*, 1008

The first line of the trace states that instruction 114 is at instruction fetch stage (*IF*) and at machine cycle 1008. At the same time reorder buffer full (*ROB Full*) event occurs for the same instruction (second line of the trace). The last line of the trace indicates that instruction 109 is at execution stage (*EX*) at machine cycle 1008. FOLCSL does not require the declaration of text attributes such as *ID*, *ROB Full* or *EX* above. Text attributes have no domain specific meaning attached to them by the language and they are treated just like any other constant.

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 object and those which globally affect all or a subset of objects. For example, in a superscalar processor simulator, an object is an instruction. Fetching, decoding, and executing an instruction are all considered to be events which affect a single instruction. In contrast, events such as the initiation of a *rollback* due to a branch misprediction is considered to be *global* as it affects every instruction in the processor.

4.2 Instrumentation

Trace data is generated by inserting instrumentation statements into the simulator. An instrumentation statement outputs an event in the format described in Section 4.1. In our early implementation, instrumentation statements were simple `printf` statements in C

language. The `printf` statement outputs an event in comma-separated values (CSV) format. Each field in the CVS line is an attribute of the event. For example:

```
printf("%lld, %d, %lld\n", sequence, state, cycles);
```

The above example outputs an event with 3 attributes. The advantage of using this approach is that CSV is a widely used format. CSV APIs are available for many different programming languages. Parsing CSV data is as simple as calling the appropriate function. Furthermore, user can add additional event attributes by just changing number of CSV fields when invoking the verifier or the SFTAG processor without altering their event reader. However, this flexibility comes with a cost. The `printf` statement is expensive and each will be executed billions of times while the simulator is running. This overhead increases the simulator running time by twofold or more. For instance, 173.applu benchmark using large MinneSPEC [87] as input running on an uninstrumented simulator takes about 3 minutes to complete, but the same benchmark using the same input running on an instrumented simulator takes more than 6 minutes to finish. Because of the incurred overhead, we decided to trade some flexibility for efficiency. To achieve this, instead of outputting the events in CSV text format, we output events in fixed size binary format. This way we free a lot of computational power in parsing CSV lines and converting text strings to binary values and vice versa.

All of our tested simulators are written in C/C++ language. Here we demonstrate how instrumentation for a simulator in C can be done. Simulators written in other language can adapt similar coding structures. First we declare a C structure as follows:

```
typedef struct {
  signed long long  cycle;
  unsigned int      instruction_seq;
  int               state_id;
} Event;
```

Then we assign the proper value to each structure member before outputting the structure in binary as shown in the code below.

```
void emit_event(long long int machine_cycles,
  unsigned int sequence, int state_id, FILE* fd) {
  Event event;
  event.cycle           = machine_cycles;
  event.instruction_seq = sequence;
  event.state_id        = state_id;
  fwrite(&event, sizeof(Event), 1, fd);
}
```

Outputting the events in binary format dramatically reduces the incurred overhead. For example, the 173.applu benchmark using large MinneSPEC as input running on an unmodified simulator takes about 3 minutes to complete. Same benchmark with same inputs running on simulator that outputs binary events takes a little over 3 minutes.

Alternatively, if the simulator contains a built-in trace generator, one can avoid instrumentation by converting the original trace into proper format before feeding it into the verifier or the SFTAG processor. For example, the SimpleScalar simulator [82, 1] has a `-ptrace` option that outputs instruction events.

Although instrumentation is relatively straight-forward in this framework. Great care must be taken while placing the instrumentation statement in the simulator code. The rule of thumb is that every output event must correctly represent the instruction, the instruction location, and the time at which the instruction is at that state. Besides that, global events which affect a subset of instructions require attaching the event to all affected instructions individually. For example, *rollback* is a global event in a processor pipeline but it only 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 uncommitted instructions when a rollback event occurs.

4.3 The FOLSCL Grammar

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 4.1 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.

$$\forall z \in T \; \exists y \in T, \quad (s^z = IF) \;\Rightarrow\; (a^y = a^z) \wedge ((s^y = ID) \vee (s^y = RB)) \tag{4.1}$$

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

```
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 4.1: The FOLCSL grammar.

FOLCSL constraints consists of fully quantified variables, arithmetic expressions and Boolean expressions. The language has the simple grammar shown in Figure 4.1. 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, functions are restricted to built-in functions only and they are implicitly declared.

4.4 Stream Processing and Sliding Windows

FOLCSL and the associated trace description treat an instruction as an object which moves through different states 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 minimal amount of data is kept in memory during the verification process and the number of event comparisons is minimized. To achieve this, we employ an algorithm based on *sliding windows* [68] while checking the events against the constraint specifications.

4.4.1 Sliding Window

The sliding window approach views the trace as a chronologically ordered stream of events. Let ξ^z be our pivot event. We can buffer all events from time $t^z - t_b$ to time $t^z + t_f$ to form a sliding window 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 4.2 depicts the sliding window for constraint 4.1, where t^z is the pivot.

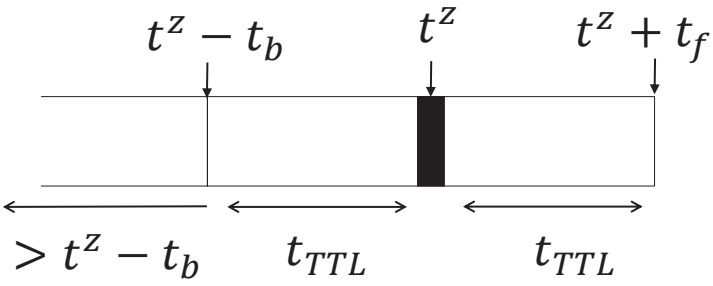

Figure 4.2: Sliding window.

The sliding window data structure provides three advantages. First, it requires minimal amount of memory space for data. Second, the verification process can begin 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 makes processing very large traces feasible.

4.4.2 Constraint Checker

Within a sliding window, all permutations of the 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 4.2 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. Note that, instead of evaluating the expression $a^y = a^z$, one can evaluate the expression $t^x > t^y$ first and if it evaluates to false we can still claim that the constraint is guaranteed to be false without knowing the result of expression $a^y = a^z$. But doing so will not eliminate any nodes from the search space because three of the variables z, y and x already had values assigned when the expression $t^x > t^y$ was evaluated.

$$\forall z \in T\ \forall y \in T\ \exists x \in T,\ \ (s^z = IF)\ \ \Rightarrow\ \ (a^y = a^z)\ \wedge\ (t^x > t^y) \tag{4.2}$$

More efficient CSP heuristics such as propagation, variable ordering and intelligent backtracking [88] can also be employed by the checker.

4.5 Constraint Specifications

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 that use sets to enforce resource based invariants. Two of these are the *set* function which collects events into a set, and the *car* function 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.

$$\forall q \in T,\ car(\,set(\,\forall z \in T, (s^z = MEMPORT)\ \wedge (t^z = t^q))) \leq 2 \tag{4.3}$$

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 cases such as the violation of the latency of a particular pipeline stage. Such violations are very difficult to catch using ad-hoc 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*) state and in doing so, it should take at least a cycle, but no more than K cycles, where K is a constant value that depends on the simulated model:

$$\begin{aligned}
&\forall z \in T\ \exists y \in T,\ (s^z = IF)\ \ \Rightarrow (a^y = a^z) \wedge \\
&\quad (t^y - t^z > 0) \wedge (t^y - t^z \leq K) \wedge \\
&\quad ((s^y = ID) \vee (s^y = RB))
\end{aligned} \tag{4.4}$$

When multiple instructions compete for a particular resource, a subset of those instructions might have higher priority over other instructions. 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 4.5 that *LOAD* instructions are given priority to move from *EX* to *WB* stage.

$$\begin{aligned}
&\forall z \in T\ \forall y \in T,\ \exists x \in T\ \exists w \in T \\
&\quad (s^z = EX) \wedge (h^z = LOAD) \wedge (s^y = EX) \wedge (h^y \neq LOAD) \wedge (t^y = t^z)\ \ \Rightarrow \\
&\quad (a^x = a^z) \wedge (a^w = a^y) \wedge \\
&\quad [((s^x = WB) \wedge (s^w = WB) \wedge (t^x \leq t^w)) \vee (s^x = RB) \vee (s^w = RB)]
\end{aligned} \tag{4.5}$$

As it can be seen through our examples, FOLCSL provides a convenient and easy way to specify 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) [3] specification and the other for SimpleScalar out-of-order simulator [1]. 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 techniques 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 matches 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 extended our framework to utilize trace data for model extraction. The 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.

Chapter 5

State Flow Temporal Analysis Graph (SFTAG) [1]

"The soul never thinks without an image."

— Aristotle*, On The Soul*

Cycle-accurate simulators typically model the flow of instructions from one pipeline stage to the next, and it is this timing which eventually provides estimates about how many cycles it will take to execute a given program. Depending on the modeled architecture, the number of stages and the latency through each stage will be different. As the flow of instructions through the stages is modeled, various events affect their flow. 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. This graphical representation is called an *SFTAG* (State Flow Temporal Analysis Graph) and used to display the paths the instructions follow through the pipeline as well as the conditions and events under which such flow occurs.

[1] ©2014 ACM. Portions reprinted with permission, from **Hui Meen Nyew**, Nilufer Onder, Soner Onder, and Zhenlin Wang, "*Verifying Micro-architecture Simulators using Event Traces,*" in Proceedings of the 2014 International Conference on Supercomputing (ICS'14).

5.1 SFTAG graph

An SFTAG is a labeled, directed graph $< N, E >$, where N represents the set of nodes and E represents the set of edges. The nodes of the graph represent the *state* an instruction is in. This is different from pipeline stages because each node includes one or more state (or stage) titles representing the states 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 states, or additional events took place simultaneously while the instruction is in that state. 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 (*W4O* stands for waiting for operands). 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 single *parallel 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*.

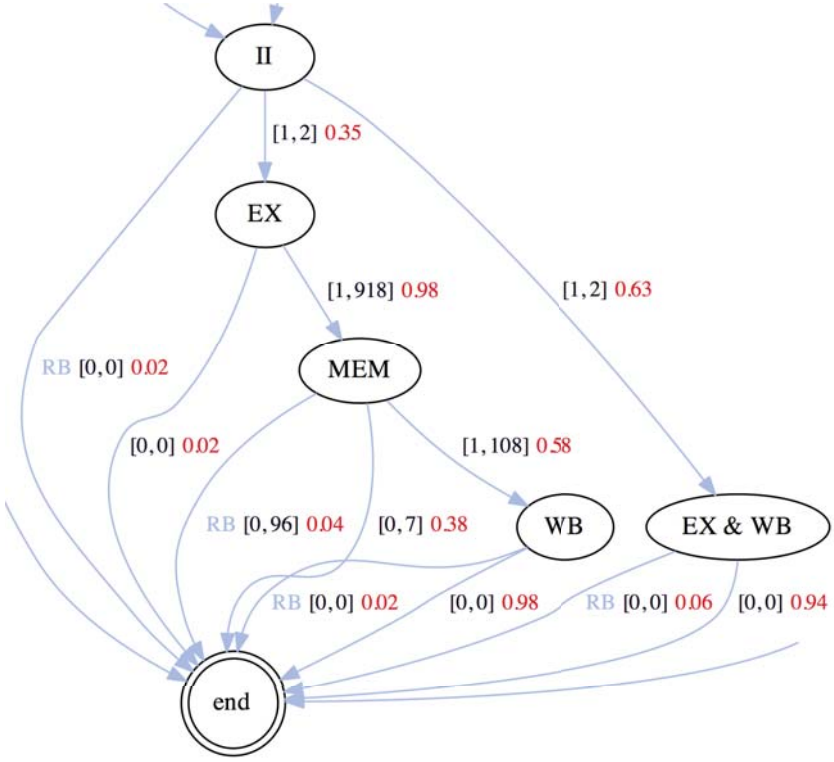

Figure 5.1: Portion of FAST pipeline temporal representation. EX&WB has two outgoing edges ended at END. One edge is due to rollback and the other is normal termination.

In a graph, an edge $e \in E$ is a quadruple $< n_s, h, r, n_d >$, where, $n_s \in N$ represents the source node, h is a 2-tuple representing the minimum and maximum time taken for the transition, r represents the ratio of instructions performing the state transition, and $n_d \in N$ represents the destination node. The titles for nodes n_s and n_d come from the set $\{\mathscr{S} \cup \texttt{start} \cup \texttt{end}\}$, where $\mathscr{S}$ 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. A tuple h is shown as a $[h_{min}, h_{max}]$ pair where h_{min} is the minimum time taken by the transition and h_{max} is the maximum time taken by the transition. For example, the three edges emanating from the node titled "*II*" in Figure 5.1 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.

5.2 Construction of SFTAGs

We use Algorithm 3 to create an SFTAG from a trace. The events in a trace used to generate an SFTAG minimally must contain three attributes, namely, instruction sequence number or more generally, an object sequence number, state, and time. The attributes are the same as the event attributes discussed in Section 4.1. The algorithm uses sliding windows as explained in Section 4.4.1. The window size is set such that all the events related to a particular instruction are within the window. Events emitted from the simulator flow to the SFTAG processor. It first constructs windows (line 8 of Algorithm 3) by buffering the events based on the specified window size. Figure 5.2 shows the process.

Algorithm 3 Analyze object transition.

Input:

T trace consisting of event triplets $id, state(s), time(t)$,
$maxsamples$ maximum number of samples
$prob$ probability
t_f forward time for sliding window
t_b backward time for sliding window

Output:

G graph

1: $G \leftarrow$ initialize to empty graph
2: $H \leftarrow$ initialize to empty histogram container
3: $B \leftarrow \emptyset$
4: $\xi \leftarrow firstEvent(T)$
5: **while** $length(BINS) < maxsamples$ **do**
6: $\quad e_{seq} \leftarrow getSequence(\xi)$
7: $\quad$ **if** $random() < prob$ **and** $e_{seq} \notin B$ **then**
8: $\quad\quad w \leftarrow window(T, \xi, t_f, t_b)$
9: $\quad\quad bin \leftarrow group(w, e_{seq})$
10: $\quad\quad bin \leftarrow sort(bin)$
11: $\quad\quad bin \leftarrow cse(bin)$
12: $\quad\quad bin \leftarrow cpe(bin)$
13: $\quad\quad H \leftarrow update(H, bin)$
14: $\quad\quad G \leftarrow merge(G, bin)$
15: $\quad\quad B \leftarrow B \cup e_{seq}$
16: $\quad$ **end if**
17: $\quad \xi \leftarrow nextEvent(T)$
18: **end while**
19: **return** G

Figure 5.2: Events emitted from the simulator are processed into windows.

For each window, all the events with a sequence number that matches the sequence number of the pivot event are grouped into a bin. Figure 5.3 shows a segment of trace (a window) with three instructions with distinct sequence numbers 1 (blue), 2 (green) and 3 (orange). The constructed bin shown on the right hand side contains all the events with sequence number of 1 (sequence number of the pivot event). Then the bin is sorted with respect to

time (line 10 of Algorithm 3). If the simulator outputs events in strict chronological order, the sorting process can be skipped.

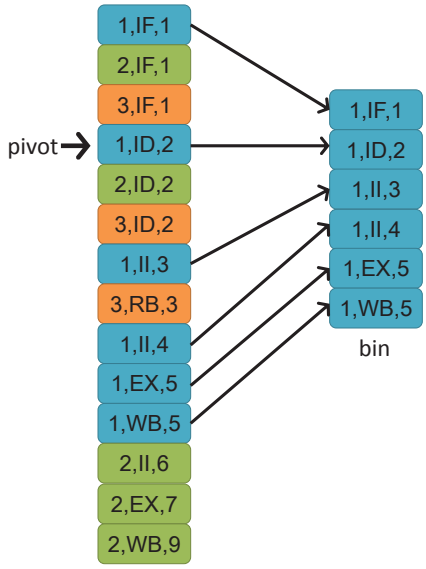

Figure 5.3: Group events with specific sequence number into a bin.

The bin is then represented as a graph. Every event is a node in the graph. The node label is the event state. All the nodes are ordered based on the bin ordering. This way, events that have the earliest timestamp will be placed at the top of the graph and events that have the latest timestamp will be placed at the end of the graph. For events that have the same timestamp, ordering does not matter because they will be combined in the subsequent step. Next, edges are added in between two nodes. The direction of the edges indicate the flow of time. Edge labels represent the transition time between two events. For example, in Figure 5.4, *ID* to *II* takes 1 time unit and *EX* to *WB* takes zero time unit which indicates that the events are parallel.

Next, adjacent nodes with same label are combined to form a single node with a unique label (line 11 of Algorithm 3). This is done in recursive manner. The edge transition time is recomputed by adding the transition times of the original nodes' outgoing edges. Figure 5.5 shows two adjacent nodes with label *II* are combined into single node. The dotted line depicts that their transition times are added together. The resulting graph has one fewer node and the transition time between node *II* and node *EX* changes from 1 to 2.

In the next step, parallel nodes are combined (line 12 of Algorithm 3). Two nodes are defined to be parallel with each other when the edge that connects them has zero transition time. The combined node has a new label which is the concatenation of the two original node labels. The edge transition time is same as the edge transition time of the lowest node

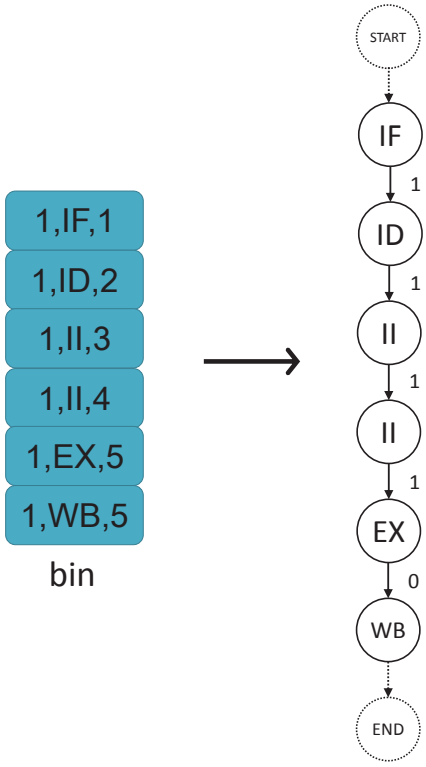

Figure 5.4: Constructed bin represented as a graph.

among the nodes that were combined. Figure 5.6 shows nodes *EX* and *WB* are combined into a single node with label *EX & WB*. The outgoing edge for the combined node does not have a transition time because it is connected to the `end` node. It is the same as the outgoing edge of node *WB*. If node *WB* had an outgoing edge with a transition time of 1, the newly combined node would have a transition time of 1 too.

While processing the trace events, the algorithm also keeps track of the computed transition time in the form of histograms (line 13 of Algorithm 3). The transition times for each distinct edge, that is, a pair consisting of a source node (n_s) and a destination node (n_d), is tracked in separate histograms. In the final output process, all the histograms will be generated along with the final SFTAG graph.

At this point, bin processing is complete. The last step is to merge the *bin graph* (graph representation of the bin) into the *total graph G* (line 14 of Algorithm 3). Initially the total graph *G* is empty, thus merging yields the bin graph itself. Figure 5.7 depicts the process of merging the empty total graph with first bin graph. Notice that the edge label changes from single values to three values expressed in the form $[h_{min}, h_{max}]$ r. As mentioned in Section 5.1, h_{min} is the minimum transition time, h_{max} is the maximum transition time and r is the ratio of instructions that moved along the edge. Since the bin is merged with an empty total

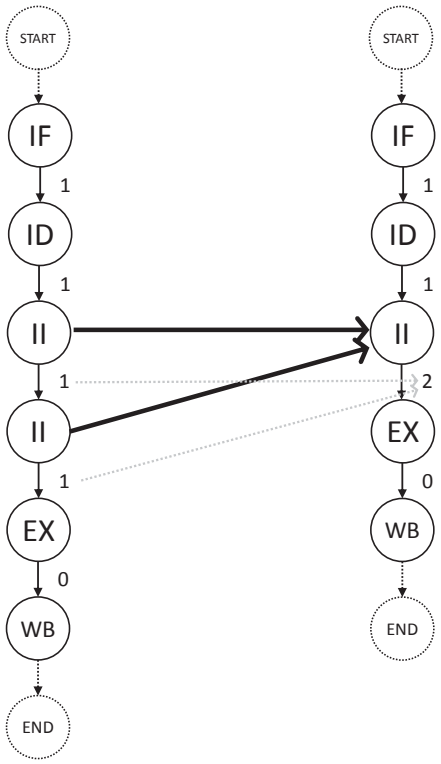

Figure 5.5: Combining adjacent nodes with the same label.

graph, the minimum transition time is same as the maximum transition time and each node has only one outgoing edge with a ratio of, 1.0 or 100%. If the total graph is not empty, all the edges and the nodes in the bin graph that are not already in the total graph are added to the total graph. This operation can be expressed as follows: Let $G = (N_G, E_G)$ be the total graph, N_G is all the nodes in G and E_G is all the edges in G. Also, let $G_{bin} = (N_{G_{bin}}, E_{G_{bin}})$ be the bin graph. $N_{G_{bin}}$ is all the nodes in G_{bin} and $E_{G_{bin}}$ is all the edges in G_{bin}. The total graph nodes and edges are updated as follows:

$$N_G = N_G \cup N_{G_{bin}} \tag{5.1}$$

$$E_G = E_G \cup E_{G_{bin}} \tag{5.2}$$

The minimum and maximum transition times are compared and updated accordingly. The ratio also needs to be recomputed so that it reflects the changes. Figure 5.8 shows the merging process of the total graph (after being merged with the first bin as shown in Figure 5.7) with the second bin graph. The maximum transition time of the edge from node *ID* to node *II* is updated to 4 and the ratios on the two outgoing edges from node of node *II*

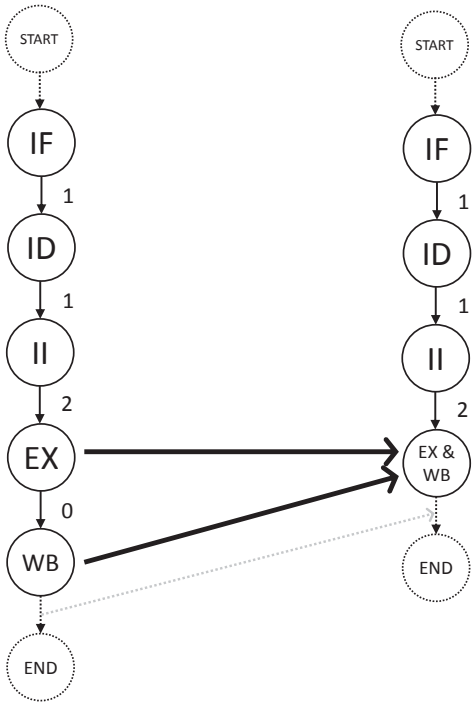

Figure 5.6: Combine parallel nodes.

is updated to 0.5. This indicates that 50% of the instructions moved from node *II* to node *EX* and the remaining 50% of instructions moved from node *II* to parallel node *EX & WB*. The third bin graph is shorter than the other bin graphs. This is because it indicates that the instruction encountered a rollback sequence during execution. As a result, the instruction only passed through *IF* and *ID* stages.

5.3 Case Studies

In this section we present three case studies we conducted using empirical data. The data traces were obtained from FAST ADL [3] and SimpleScalar out of order [1] simulators. We manually instrumented various events in the 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 accesses. 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 from two variants of SimpleScalar. The first simulator faithfully implements a

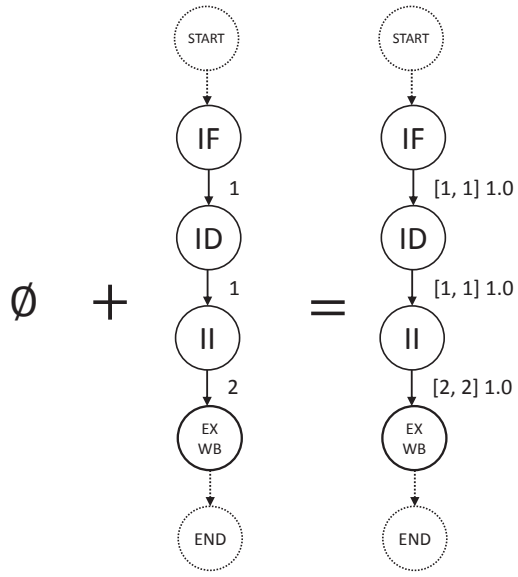

Figure 5.7: Merge bin graph into the total graph *G*.

Rambus DRAM model while the second models the original SimpleScalar simple DRAM model. Through the generated histograms, we conclude that the observed behavior matches to expected behavior for these two models. Finally, we present an analysis of a bus arbiter implementation which uses the same algorithms as the ones 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.

5.3.1 Pipeline Temporal Information

When the simulator event traces are fed through the algorithms discussed in the previous section, two graphs shown in Figure 5.10 and Figure 5.11 result. These temporal graphs are obtained directly from trace data, without human intervention.

Figure 5.10 can be read as follows: Every instruction starts at IF state or $IF\&ROBFull$ state. The $IF\&ROBFull$ state means that the instruction is in IF state and at the same time

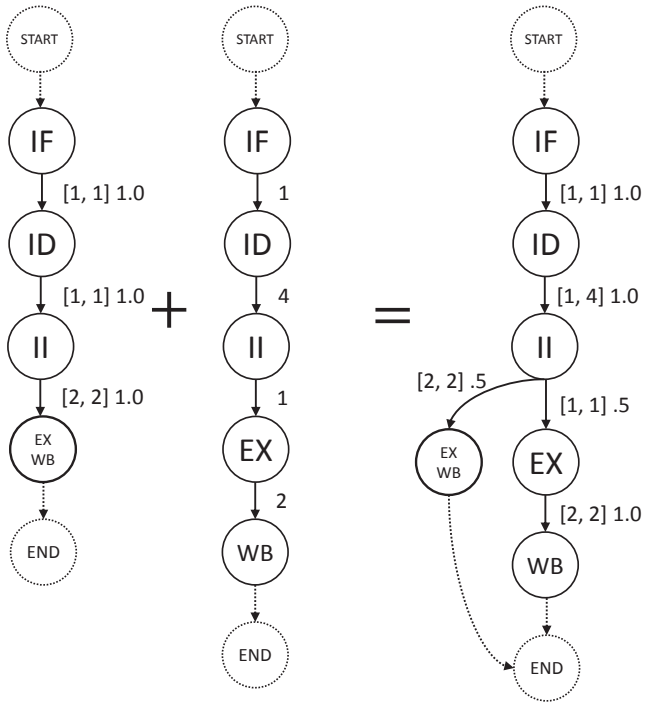

Figure 5.8: Merge second bin graph into the current total graph *G*.

reorder buffer (ROB) is full. 47% of the instructions will start at *IF* and the rest will begin at the *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 a minimum transition time of 3 and a maximum transition time of 113 cycles. In other words, a full *ROB* takes from 3 to 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 5.11 is similar to Figure 5.10 except that it represents the 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 the 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

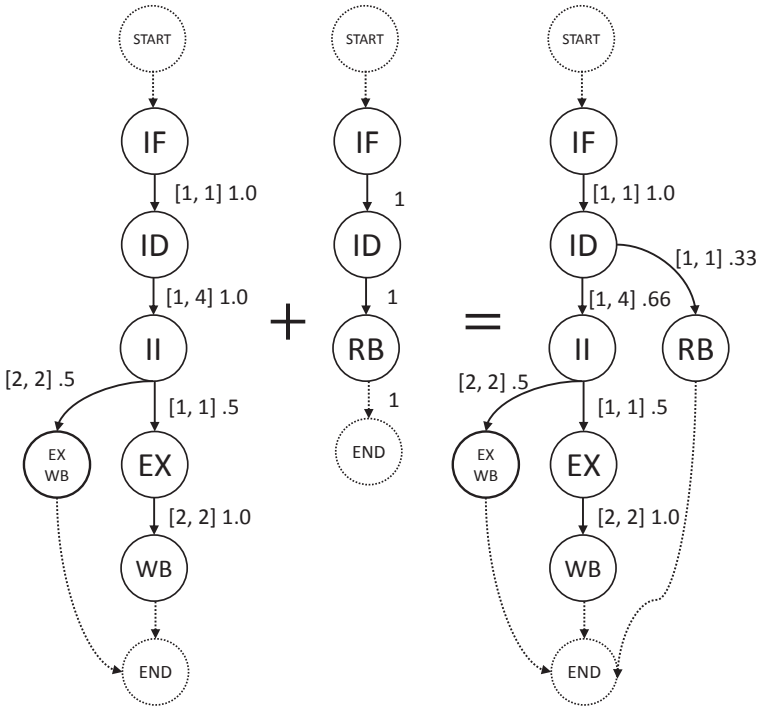

Figure 5.9: Merge third bin graph into the current total graph G.

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 constraint 4.4. For example, consider the outgoing edges from ID in Figure 5.10. 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 :

$$\forall z \in T\ \exists y \in T,\ (s^z = ID)\ \Rightarrow$$
$$(a^y = a^z) \wedge \Big[(s^y = II) \vee (s^y = RB)\Big] \tag{5.3}$$
$$\forall z \in T\ \forall y \in T\ \exists x \in T,\ (s^z = ID) \wedge (s^y = W4O) \wedge (a^y = a^z)\ \Rightarrow$$
$$(a^x = a^z) \wedge \Big[\Big((s^x = II) \wedge (t^x = t^y)\Big) \vee (s^x = RB)\Big] \tag{5.4}$$

Similarly, temporal information can be added as follows:

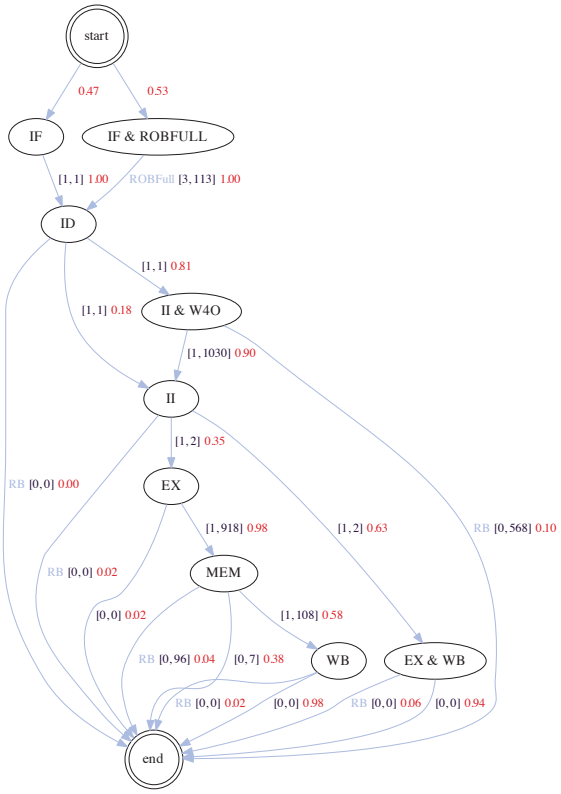

Figure 5.10: FAST pipeline temporal representation.

$$\forall z \in T\ \exists y \in T,\ (s^z = ID)\ \Rightarrow$$
$$(a^y = a^z) \wedge \Big[\Big((s^y = II) \wedge (t^y - t^z = 1)\Big) \vee \Big((s^y = RB) \wedge (t^y = t^z)\Big)\Big] \tag{5.5}$$
$$\forall z \in T\ \forall y \in T\ \exists x \in T,\ (s^z = ID) \wedge (s^y = W4O) \wedge (a^y = a^z)\ \Rightarrow$$
$$(a^x = a^z) \wedge \Big[\Big((s^x = II) \wedge (t^x = t^y) \wedge (t^x - t^z = 1)\Big) \vee \Big((s^x = RB) \wedge (t^x = t^z)\Big)\Big] \tag{5.6}$$

Figures 5.10 and 5.11 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 will expose the specific path taken by the selected instruction types.

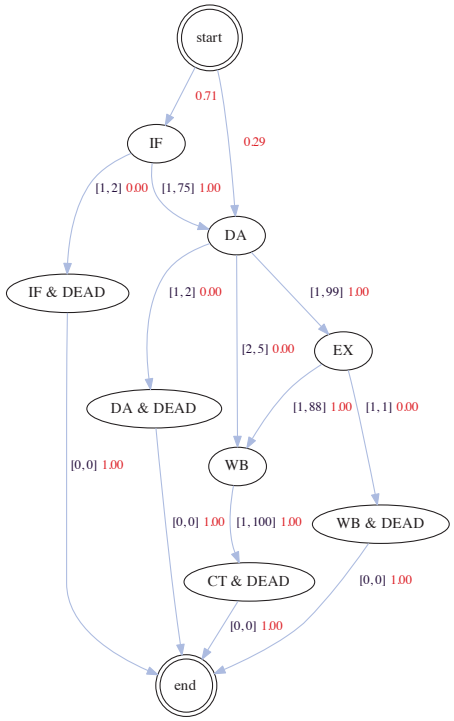

Figure 5.11: SimpleScalar pipeline temporal representation.

5.3.2 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 5.11 shows the SFTAG for a default superscalar machine with a simple DRAM model as used in SimpleScalar 3.0. The SFTAG shown in Figure 5.12 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 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).

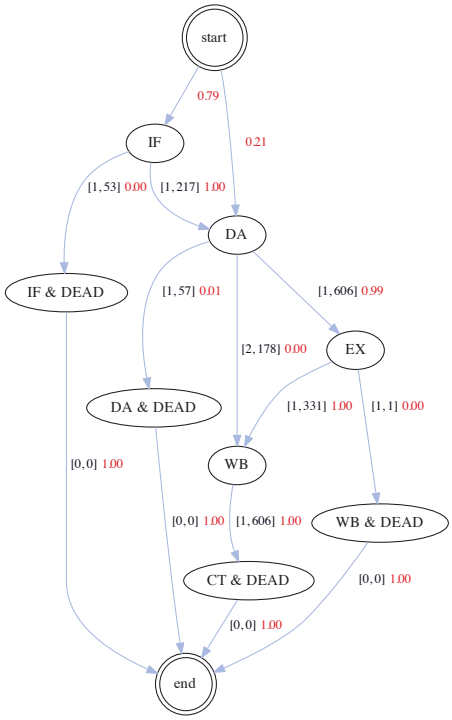

Figure 5.12: SimpleScalar/Rambus pipeline temporal representation.

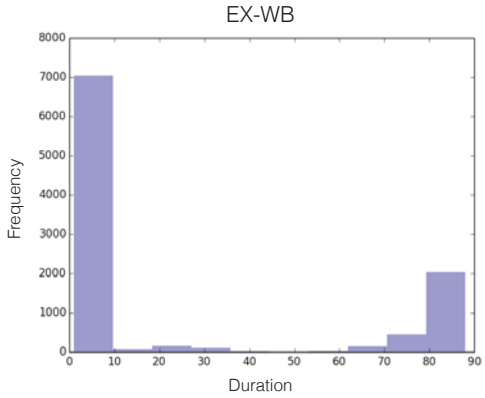

Figure 5.13: SimpleScalar EX to WB distribution.

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 5.13 and Figure 5.14 show the histograms of the transition from EX to WB for the original SimpleScalar 3.0 and its Rambus extension, respectively. Figure 5.13 demonstrates that a large number of load instructions indeed cause L2 misses and need to access the main memory. The range of the

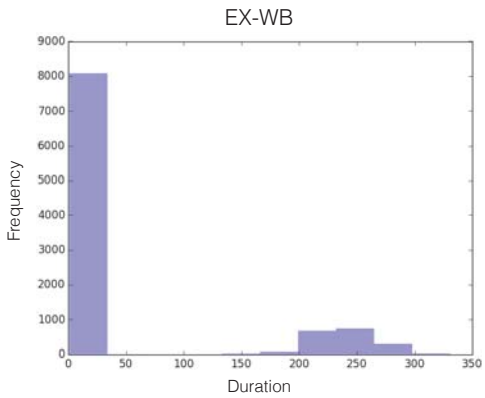

Figure 5.14: SimpleScalar/Rambus EX to WB distribution.

latencies follows the memory configuration and thus supports the correctness of memory system simulation. The Rambus DRAM (RDRAM) is much more complicated. Figure 5.14 suggests that an access to the RDRAM may have 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.

5.3.3 Bus Arbiter

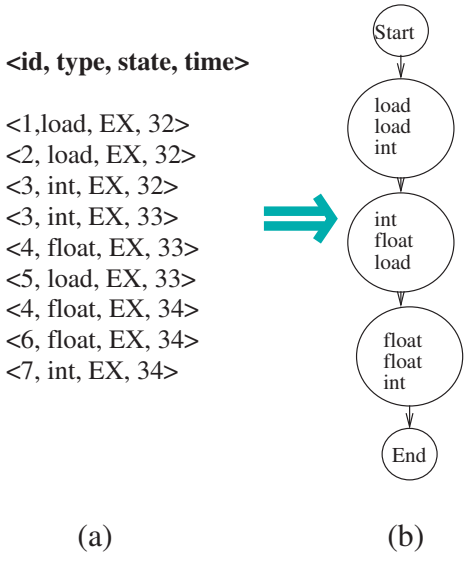

Figure 5.15: Finding the temporal information about the instructions leaving the EX stage.

Table 5.1
Type of instructions entering and leaving *EX* stage.

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

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 5.15(a). Next, we combine parallel events into single nodes (Figure 5.15(b)). We convert the graph into a tabular representation showing which type of instructions leave the EX stage as shown Table 5.1. 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 [48, 89] 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.

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. [90]. 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.

5.4 Performance

Through 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 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.

Our algorithm's performance is directly correlated with the number of events that need to be processed. Figure 5.16 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 as shown in Figure 5.17. This rate is close to the performance of an annotated simulator. Hence, on a

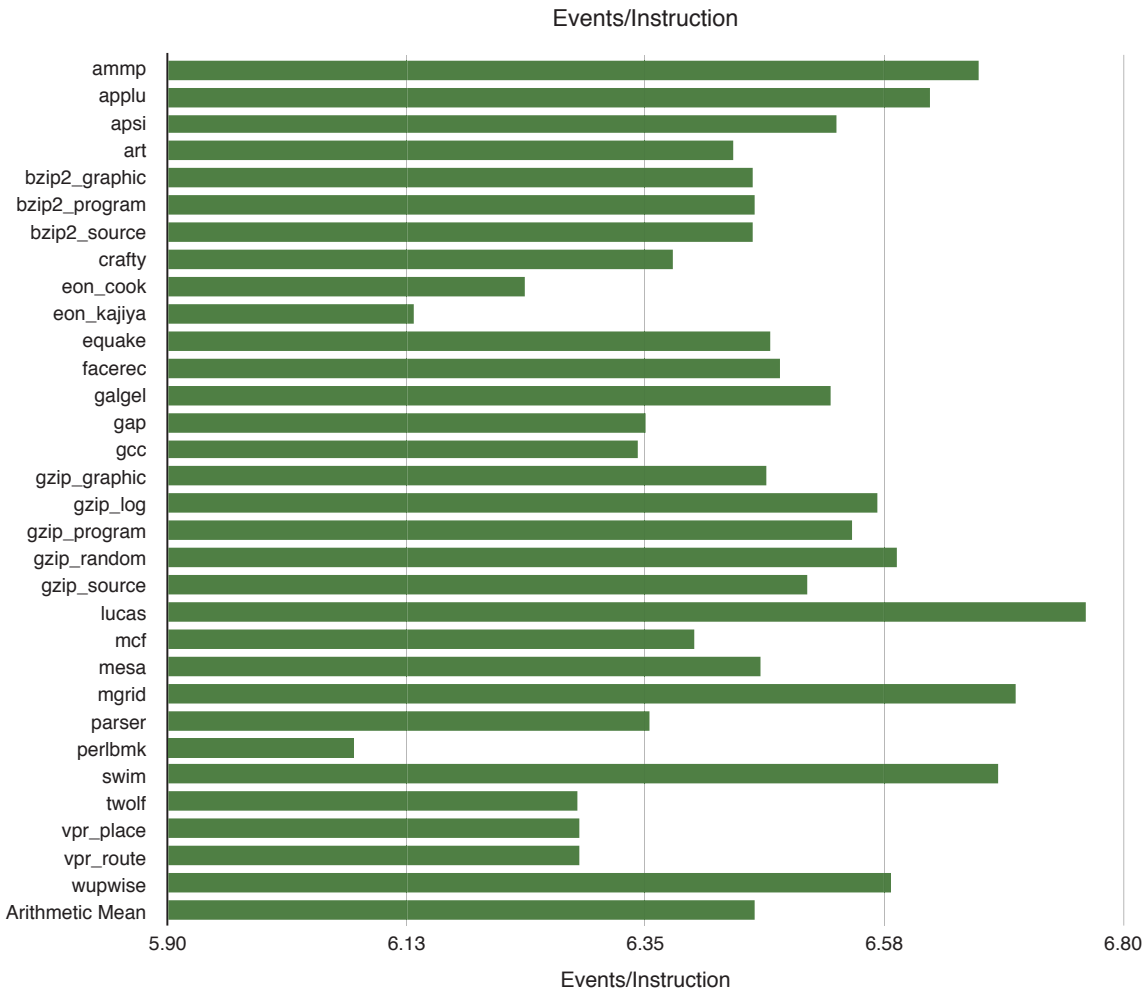

Figure 5.16: Events per instruction for benchmark programs.

dual-core system, it is possible to generate SFTAGs in parallel with the simulation as the data becomes available and no extra time will be added on top of the simulation time.

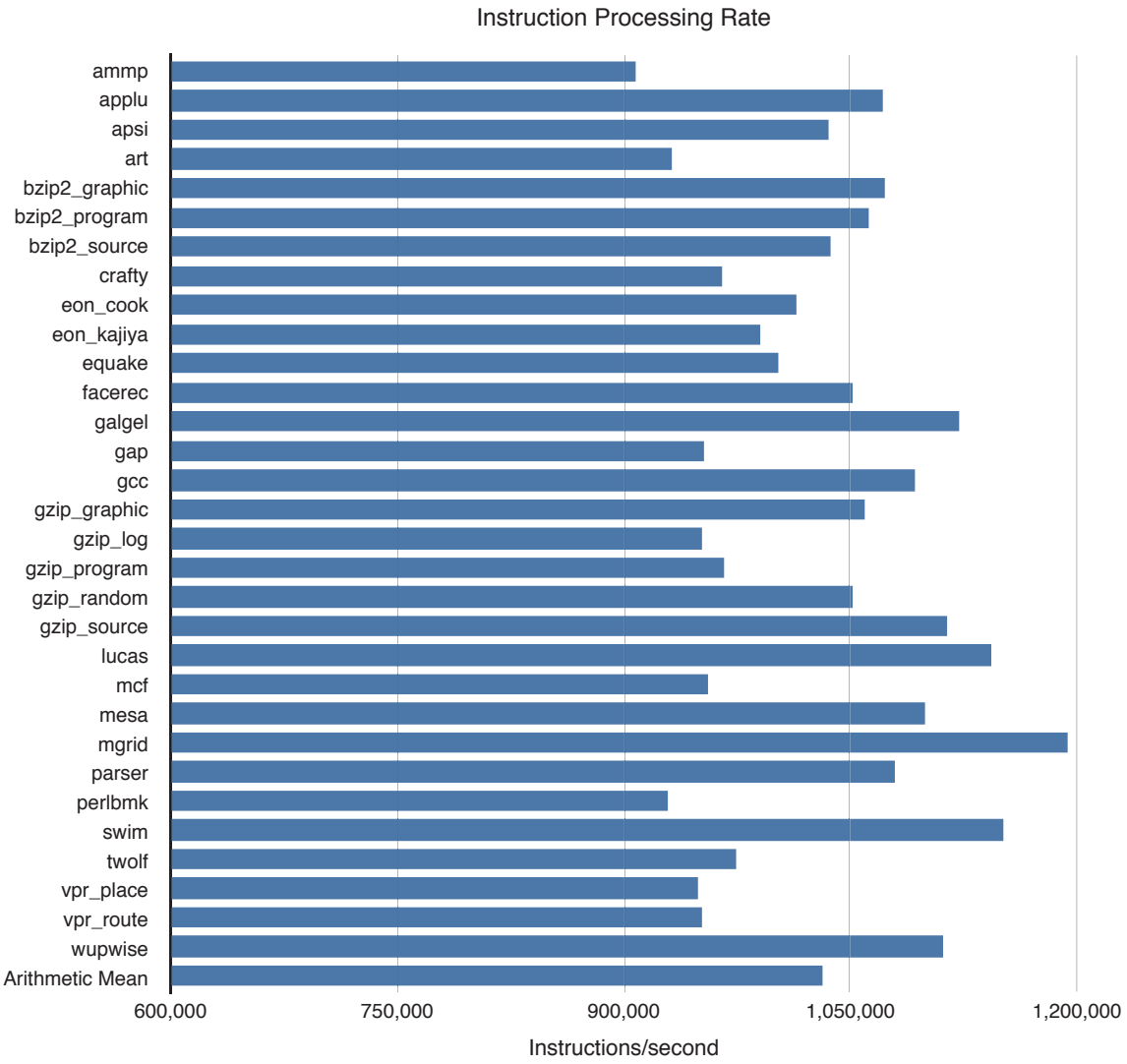

Figure 5.17: SFTAG processed instructions per second.

Chapter 6

Analysis

"You see, one thing is, I can live with doubt, and uncertainty, and not knowing. I think it's much more interesting to live not knowing than to have answers which might be wrong. I have approximate answers and possible beliefs and different degrees of certainty about different things. But I'm not absolutely sure of anything, and there are many things I don't know anything about, such as whether it means anything to ask why we're here, and what the question might mean. I might think about it a little bit. If I can't figure it out, then I go onto something else. But I don't have to know an answer. I don't feel frightened by not knowing things, by being lost in the mysterious universe without having any purpose, which is the way it really is, as far as I can tell – possibly. It doesn't frighten me."

— Richard Feynman, *Interview with BBC Horizon, 1981*

The SFTAG described in Chapter 5 is rich in data. The data can be used to cross-validate expected or known results. It can also be used as an evidence to infer a conclusion. The case studies in Section 5.3 analyze each benchmark program individually. Alternatively, we can analyze the benchmark suite as a whole to reveal their properties from a different perspective.

We processed about [1.2PB] of binary data from the FAST ADL simulator. From these data, we constructed 18 SFTAG graphs. Each graph represents a benchmark program in SPEC2000 benchmark suite [91] and on average there are 50 edges in a single SFTAG graph. The edges in an SFTAG graph represent the transition from a source state to the destination state. Furthermore, the distribution of the transition times on each edge is recorded in the form of a histogram. Analyzing this massive data all at once is difficult and very time consuming. To make the analysis process smoother, we applied clustering

techniques to group data with similar properties into clusters. By doing this, we are able focus our analysis on each cluster locally rather than looking all the data at once.

6.1 Clustering

The clustering method we use is hierarchical agglomerative clustering (HAC) [92]. HAC works as following. Initially, each data point is in a cluster by itself. At each iteration, two clusters that have the shortest distance are grouped together to form a new cluster. The iterations continue until there is only one cluster left in the cluster set. The cluster tree is usually presented as a *dendrogram*. Algorithm 4 depicts the clustering process. One of the most important component in clustering is the distance measurement. In HAC there are two types of distance measurements – the distances between individual data points and the distances between clusters (*icdist* in line 12 Algorithm 4). The next two sections describe these distance measurements.

Algorithm 4 Hierarchical Agglomerative Clustering (HAC).

Input:
D set of data
Output:
$G =< N, E >$ cluster tree

1: $G \leftarrow$ initialize to empty cluster tree ↪ *Graph (tree)*
2: $C \leftarrow \emptyset$ ↪ *set of clusters*
3: $m \leftarrow 0$ ↪ *number of clusters*
4: **for all** $d \in D$ **do**
5: $c_m = \{d\}$ ↪ *Data point is a cluster*
6: $N = N \cup \{c_m\} \cup \{d\}$ ↪ *Nodes*
7: $E = E \cup (c_m, d)$ ↪ *Edge*
8: $C = C \cup \{c_m\}$ ↪ *Clusters*
9: $m = m + 1$ ↪ *Increment number of clusters*
10: **end for**
11: **while** *length*$(C) > 1$ **do** ↪ *Repeat until only one cluster left*
12: $(c'_i, c'_j) = min\{icdist(c_i, c_j), \mid c_i \in C, c_j \in C, c_i \neq c_j\}$ ↪ *Intercluster distance*
13: $c_m = c_m \cup c'_i \cup c'_j$ ↪ *Create new cluster*
14: $N = N \cup \{c_m\}$ ↪ *Add new node*
15: $E = E \cup (c_m, c_i) \cup (c_m, c_j)$ ↪ *Add new edges*
16: $C = C \cup \{c_m\} \setminus \{c_i, c_j\}$ ↪ *Remove processed clusters*
17: $m = m + 1$ ↪ *Increment cluster count*
18: **end while**
19: **return** G

6.2 Intercluster Distance Measurement

Intercluster distance measures the distance or dissimilarity between two clusters. There are many different types of intercluster distance functions. The two well known distance functions are *nearest neighbor* (*single link*) and *furthest neighbor* (*complete link*). The intercluster distance we use in the clustering process is nearest neighbor. It is defined as follows:

$$icdist(c_i, c_j) = \arg\min_{x,y}\{dist(x,y) \mid x \in c_i, y \in c_j\} \tag{6.1}$$

where arg min refers to the x and y values that minimize the formula value.

We chose this measurement for two reasons. First, nearest neighbor method is simple and easy to understand. Second, it has a property that no other intercluster measurement possesses, which is if two pairs of clusters have the same distance, the overall results will be the same regardless of the order of the merger [93]. The simplicity of this distance measure comes with a price. It suffers from the “chaining" effect where a series of data points are merged into the same cluster. Chaining occurs because nearest neighbor merging criterion is strictly local regardless of the overall clustering. Besides that, it doesn’t account for the distribution in its local cluster [92, p. 382]. Having said that, nearest neighbor measurement still serves our objectives well. The “chaining" effect is desired for our purposes because we focus more on the distances between clusters rather than the overall shape of the clusters.

6.3 Data Distance Measurement

Data distance measurement or intra-cluster distance measurement is the measurement of the similarity of two data points. This is the most important distance measurement in the clustering process and it varies from data set to data set. The data point is represented as a feature vector, i.e., a vector of values corresponding to the features of each data item.

In SFTAG, each data point is a histogram which corresponds to an edge in the graph. The feature vector of each data point consists of a set of frequency values. Each frequency value corresponds to a bin’s frequency in the histogram and the width of the bins in the histogram

is 1 unit time or 1 cycle. HAC operates using pairwise distances between data points. In other words, each data point needs to be compared with every other data point in the data set. For the distance measurements to work, the number of bins in every histogram must be the same. We assume that the number of bins in all the histograms is the same and is some constant. If there are any unobserved bins in a histogram, their frequencies are assigned to be 0.

Clustering SFTAG data can be done from three different perspectives. First, we can cluster all the edges for a given benchmark program. Having groups of similar edges enables the domain expert to look at fewer transitions on a benchmark's SFTAG. Second, we can cluster all the benchmark programs for a given edge. This shows which benchmarks exhibit similar transition behaviors between states. Third, we can do a second level clustering on the first level of clusters. Using higher order clustering on the clusters from the first method, we will be able to learn the the similarity of benchmark programs. On the other hand, if we cluster the clusters from the second method, we will be able to learn the similarity of edges.

Clustering on the properties of individual benchmark programs is straight-forward but clustering on the properties across benchmark programs requires proper scaling because each benchmark program has a different number of executed instructions. To make all the histograms independent from the number of executed instructions, we convert their absolute frequencies to percentages by dividing each frequency value by the corresponding number of executed instructions.

We first experimented with the commonly used Euclidean distance as shown in the next section. Using these results, we then designed a distance measurement that is more tailored to the data set (Section 6.3.2). Finally, we incorporated domain knowledge into the distance measurement so that it captures the semantics of the data representation (6.3.3).

6.3.1 Euclidean Distance Metric

Euclidean distance is a true distance metric which by definition satisfies four *metric* criteria [92]:

1. $d(x,y) \geq 0$
2. $d(x,y) = 0 \iff x = y$
3. $d(x,y) = d(y,x)$

4. $d(x,z) \leq d(x,y) + d(y,z)$

It is also known as a special case of Minkowski's distance where the order p is equal to two. Euclidean distance metric works in any number of dimensions which makes comparing two histograms relatively straight-forward. Figure 6.1 shows the APSI dendrogram computed using the Euclidean distance metric. Here the figure shows that the edges that have similar histograms are grouped close to each other. The x-axis is the distance between the two histograms. The Euclidean distance measures the histogram dissimilarity directly without scaling thus the magnitude of the sum of frequencies of the histograms under consideration plays a very important role. Because of that, edges that are visited frequently have larger distances compared with edges that are visited less frequently even though their distance is small percentage-wise.

6.3.2 Least Squares Distance Metric

Alternatively, we can compute the distance between two histograms using the least squares approximation method. The intuition behind this measure is that the least squares approximates the overall differences all the bins between the two histograms. First, we compute the errors as shown in Equation 6.2. Next, we compute the *maximum histogram* as defined in Equation 6.3. We can write the error $\vec{E}$ as a fraction of the maximum histogram $\vec{H}$ as in Equation 6.4. The changes in d in relation to $\vec{E}$ is suitable for use as a distance measure. From the equation we can see that if there is no error or $|\vec{E}| = 0$ then $d = 0$ and if $\vec{E} = \vec{H}$ (maximum error) then $d = 1$. Since $\vec{E}$ and $\vec{H}$ contain multiple elements and d is a scalar, there are more equations than variables. Therefore, Equation 6.4 is unsolvable. However, we can approximate it by solving Equation 6.5. The final distance $\hat{d}$ is given in Equation 6.6 where $\overline{EH}$ and $\overline{H^2}$ are given in Equations 6.7 and 6.8, respectively. Equation 6.6 is in fact a least squares approximation without the intercept term.

Figure 6.2 shows the APSI dendrogram computed using the least squares distance metric. Comparing Figure 6.1 with Figure 6.2, we prefer Figure 6.2 which used the least squares distance because it scales better. The reason for that is the least squares distance measures distance that is independent of the sum of the frequency values in a histogram whereas in Euclidean distance, the sum of the frequency values in a histogram affects the distance value. Of course, we can achieve the same scaled distance with Euclidean distance by explicitly scaling the histograms properly before computing their distances but we prefer least squares distance for its simplicity and elegance. Note that, even though the least squares distance metric does not satisfy last metric criteria, i.e. triangle inequality, the first three metric criteria it satisfies are sufficient for our measuring purposes. This is because our measuring goals do not rely on triangle inequality properties which are important for

Figure 6.1: APSI dendrogram computed using Euclidean distance.

problems such as optimization.

$$\vec{E} = \vec{H}_1 - \vec{H}_2 \tag{6.2}$$

$$\vec{H} = \{h_i \mid \max\{h_i^1, h_i^2\}, h_i^1 \in \vec{H}_1, h_i^2 \in \vec{H}_2, 1 \leq i \leq |\vec{H}_1|\} \tag{6.3}$$

$$\vec{E} = d\vec{H} \tag{6.4}$$

$$\vec{H}^T\vec{E} = \hat{d}\vec{H}^T\vec{H} \tag{6.5}$$

$$\hat{d} = \frac{\overline{EH}}{\overline{H^2}} \tag{6.6}$$

$$\overline{EH} = \frac{1}{n}\sum_{i=1}^{n} e_i h_i \mid e_i \in \vec{E}, h_i \in \vec{H} \tag{6.7}$$

$$\overline{H^2} = \frac{1}{n}\sum_{i=1}^{n} h_i^2 \mid h_i \in \vec{H} \tag{6.8}$$

6.3.3 Least Least Square Distance Metric

Often times a distance measurement design that includes domain knowledge is more accurate compared to a generic distance measurement. Consider the following empirical observation. A transition T from state A to state B has a minimum transition time of T_{min} and a maximum transition time of T_{max}. Suppose we have a transition T_1 that moves from state A to state B and another transition T_2 that moves from state B to state C. Further assume that $T_{1_{min}} = T_{1_{max}} = c_1$ and $T_{2_{min}} = T_{2_{max}} = c_2$, where c_1 and c_2 are scalar values. Because the transition times at T_1 and T_2 have no variability, the histograms of T_1 and T_2 contain only a single bin with values of c_1 and c_2, respectively. If a state B has only one incoming edge (T_1) and one outgoing edge (T_2), then the total frequencies in histograms T_1 and T_2 must be the same. If $c_1 \neq c_2$, the least squares distance will yield maximum errors even though their magnitudes are exactly the same. We call this *dislocation*.

To minimize the effects of dislocation, we rearrange the bins in the second histogram so that the absolute sum of errors is minimum (Equation 6.9). In Equation 6.10, P is the set of all permutation rules for histogram H_2, p is a single permutation rule and H_2^p is histogram H_2 where all the bins are arranged based on the permutation rule p. p^* is the permutation rule that yield the minimum sum of errors. Once the p^* permutation rule is determined, $\vec{E}$ and $\vec{H}$ can be computed as usual by substituting H_2 with permuted $H_2^{p^*}$. Equation 6.11 and 6.12 show the changes. Note that, this distance measurement is heuristic and is based on

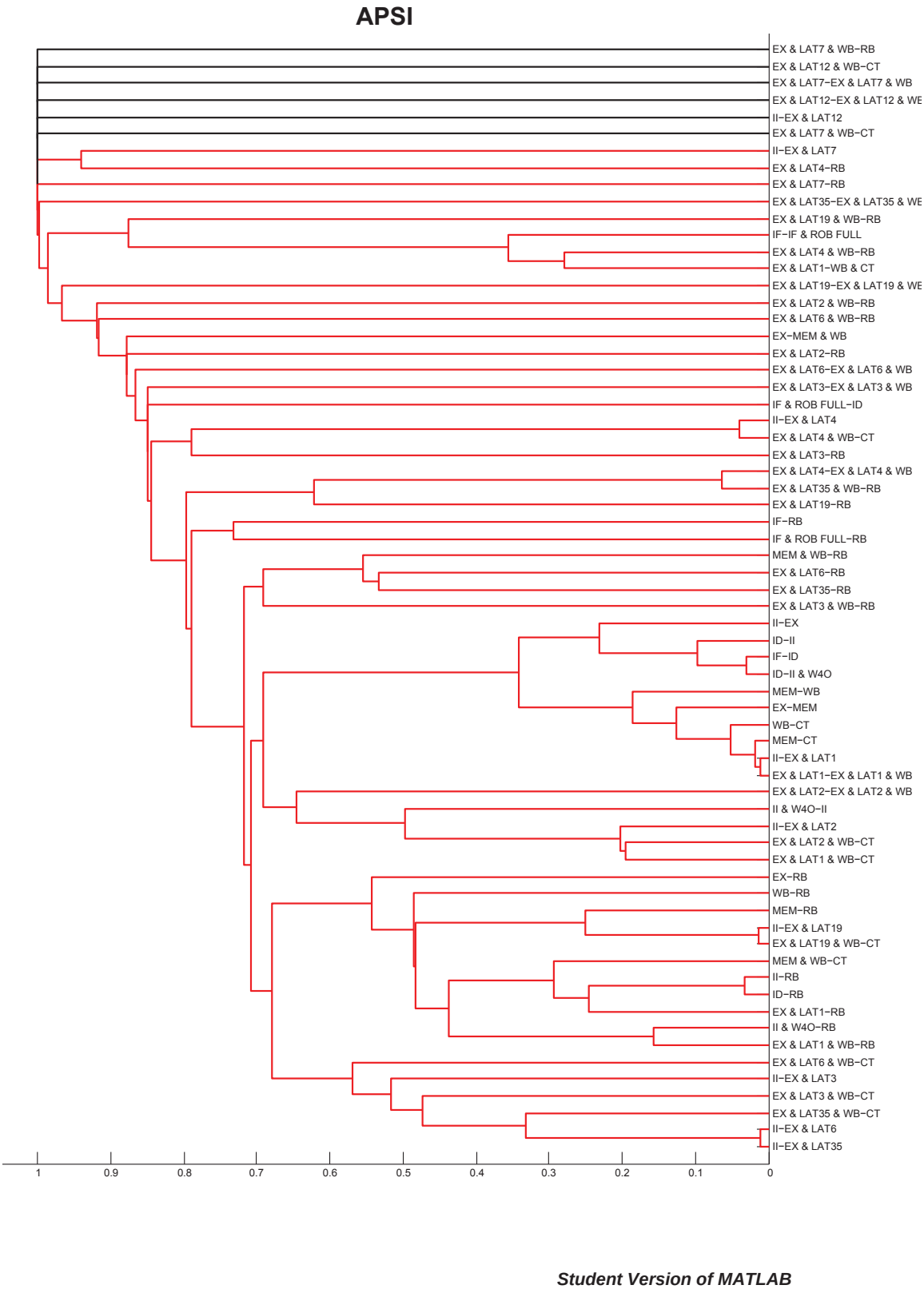

Figure 6.2: APSI dendrogram computed using least squares distance.

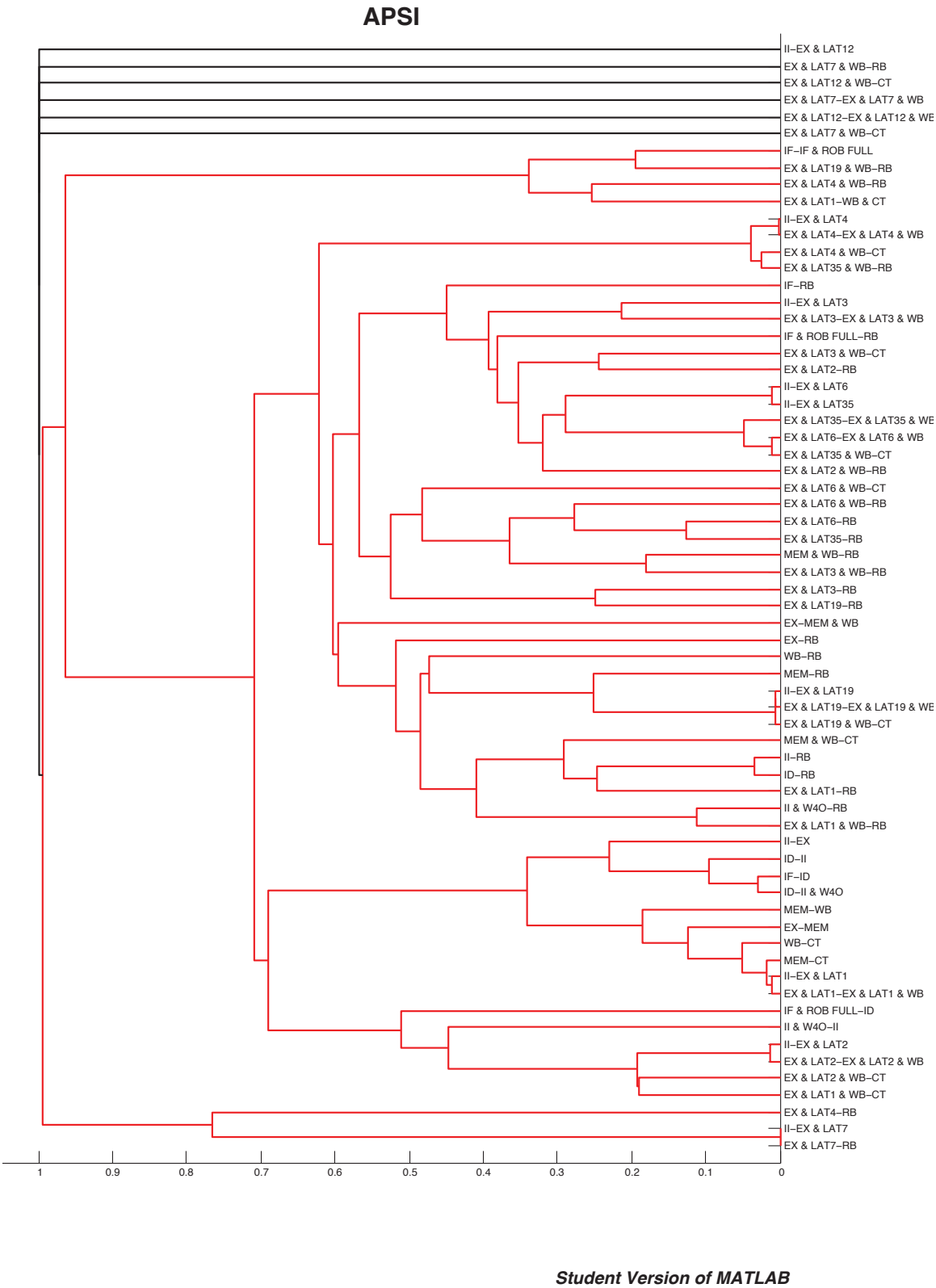

Figure 6.3: APSI dendrogram computed using least least squares distance.

the histogram patterns we observed from experiments. The reason this distance measure works is that the transition time have the tendency to cluster in a single bin or in very tight range. If two histograms have similar magnitudes across all the bins but differ in bin values, least least squares distance is able to match them. Figure 6.3 shows APSI the dendrogram computed with least least squares distance. Comparing Figures 6.1, 6.2 and 6.3, we can see that Figure 6.2 the distance between each clusters are more distinct than Figure 6.1 and Figure 6.3 the distance between each clusters are more distinct than Figure 6.2.

$$e_{H_1,H_2} = \sum_{i=1}^{|H_1|} |h_{1_i} - h_{2_i}| \big| h_{1_i} \in H_1, h_{2_i} \in H_2 \tag{6.9}$$

$$p^* = p \mid \min\{e_{H_1,H_2^p}\}, p \in P \tag{6.10}$$

$$\vec{E} = \vec{H_1} - \vec{H_2^{p^*}} \tag{6.11}$$

$$\vec{H} = \{h_i \mid \max\{h_i^1, h_i^2\}, h_i^1 \in \vec{H_1}, h_i^2 \in \vec{H_2^{p^*}}, 1 \leq i \leq |\vec{H_1}|\} \tag{6.12}$$

6.4 Evaluation

For our empirical work, we generated a total of 18 SFTAG graphs and hundreds of histograms using the FAST ADL simulator on a set of input programs from the benchmark set in SPEC2000 [91]. We then used HAC and least least squares as the distance measure to cluster the generated data. In this section, we present the analysis results of selected SFTAGs.

6.4.1 186.CRAFTY Dendrogram

Figure 6.4 shows a section of 186.CRAFTY dendrogram. This dendrogram is constructed by clustering all the edges of 186.CRAFTY. There are two clusters in the figure. The first cluster groups the II-EX&LAT4 edge and the EX&LAT4-RB edge together. II-EX&LAT4 indicates transition from state II (instruction issue) to state EX&LAT4 (execution and latency 4 instruction) and EX&LAT4-RB indicates transition from state EX&LAT4 to state RB (rollback). This grouping implies that their histograms are the same or they are very similar. The second cluster groups the EX&LAT4-EX&LAT4&WB edge and the EX&LAT4&WB-RB edge together. The first and second clusters are very similar; they both end in RB state. From this grouping results we can infer that, no matter which path

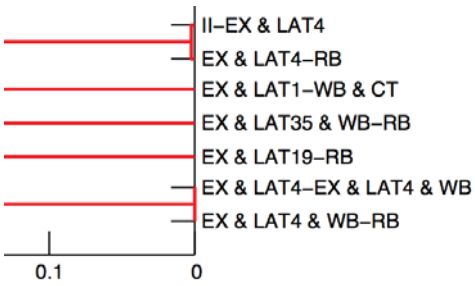

Figure 6.4: A Section of the 186.CRAFTY dendrogram

instructions with latency 4 go, they will end up in the RB state. Figure 6.5 confirms our observation. It shows that all the states that are involved with latency 4 instructions end up in the RB state.

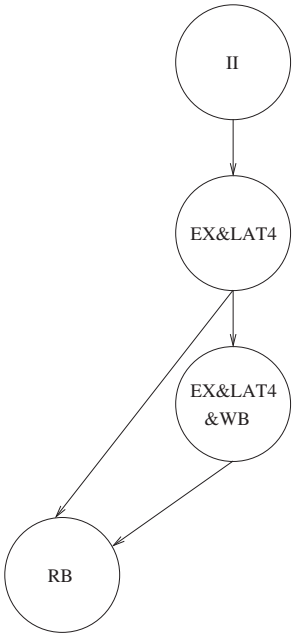

Figure 6.5: A section of the 186.CRAFTY SFTAG.

This dendogram and its analysis shows that the distance metrics and the clustering techniques effectively capture similarities of SFTAG edges. This observation indicates that the input does not cover this particular code path.

6.4.2 Floating Point Division Instruction In 256.BZIP2

Figures 6.6, 6.7 and 6.8 are generated by clustering all the benchmark programs for the edges that are related to latency 19 instructions (floating point division). 256.BZIP2 and its variations appear in Figure 6.6 but not in Figure 6.7 which means that latency 19 instructions are executed but never move to CT (commit) state. The reason that these instructions never move to CT state is shown in Figure 6.8. 256.BIZP2 and its variations reappear in Figure 6.8 which indicate that latency 19 instructions in 256.BZIP2 and its variations encounter rollback before process to CT state. To confirm our reasoning, we ran 256.BZIP2-PROGRAM on a perfect branch predictor architecture and Figure 6.9 shows the generated SFTAG graph. A perfect branch predictor architecture will never pick the wrong path thus no RB events will be generated. In Figure 6.9 not a single RB state appears and most importantly, no latency 19 states. This result matches our reasoning.

6.4.3 Reorder Buffer Full Ratio

Besides clustering based on an edge's histograms, we can also cluster benchmark programs based on an edge's ratio values. Table 6.1 lists the ratio values for START-IF&ROBFULL edge for every benchmark programs. From the table we can clearly see that it is divided into two parts. The upper part of the table contains low ratio value and is occupied by integer benchmark programs whereas the lower part of the table contains relatively high ratio and is occupied by floating point benchmark programs. The reason is that floating point programs have higher cache misses thus instructions stay in the pipeline longer and clog the reorder buffer. Therefore instructions have a higher chance to encounter the reorder buffer full event.

6.4.4 Cluster of Clusters

In addition to clustering the original data points, we can also apply clustering in a recursive manner, i.e. cluster the clusters to expose different relationships among the data. Figure 6.10 is generated by clustering the edge clusters for every benchmark program. Edge clusters are generated in a similar fashion to 186.CRAFT and 301.APSI. From the figure we can see that 164.GZIP and its variations are grouped together which means that they exhibit similar clustering patterns. This is the same for 256.BZIP and its variations. 176.GCC,

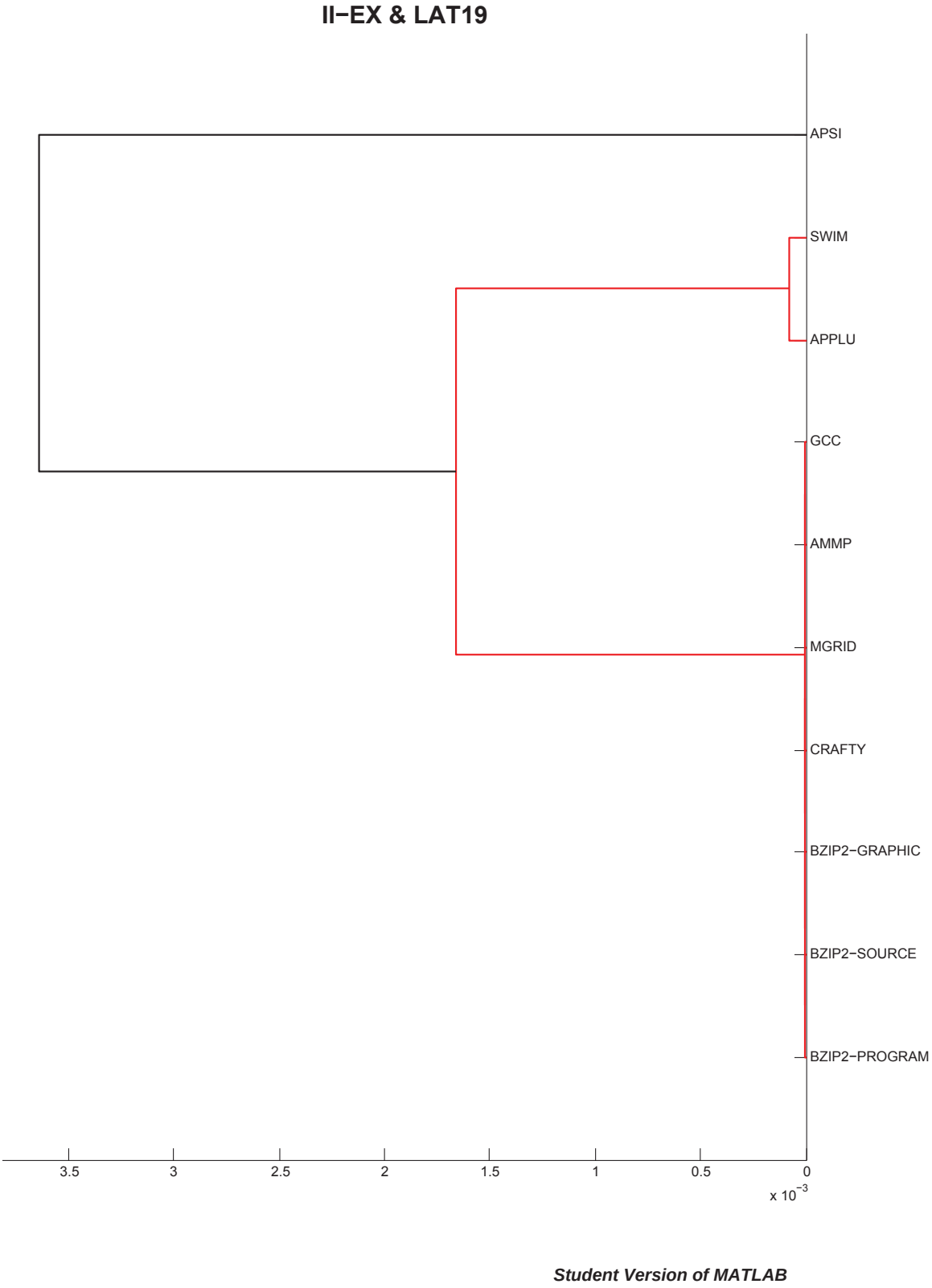

Figure 6.6: Dendrogram of II-EX&LAT19.

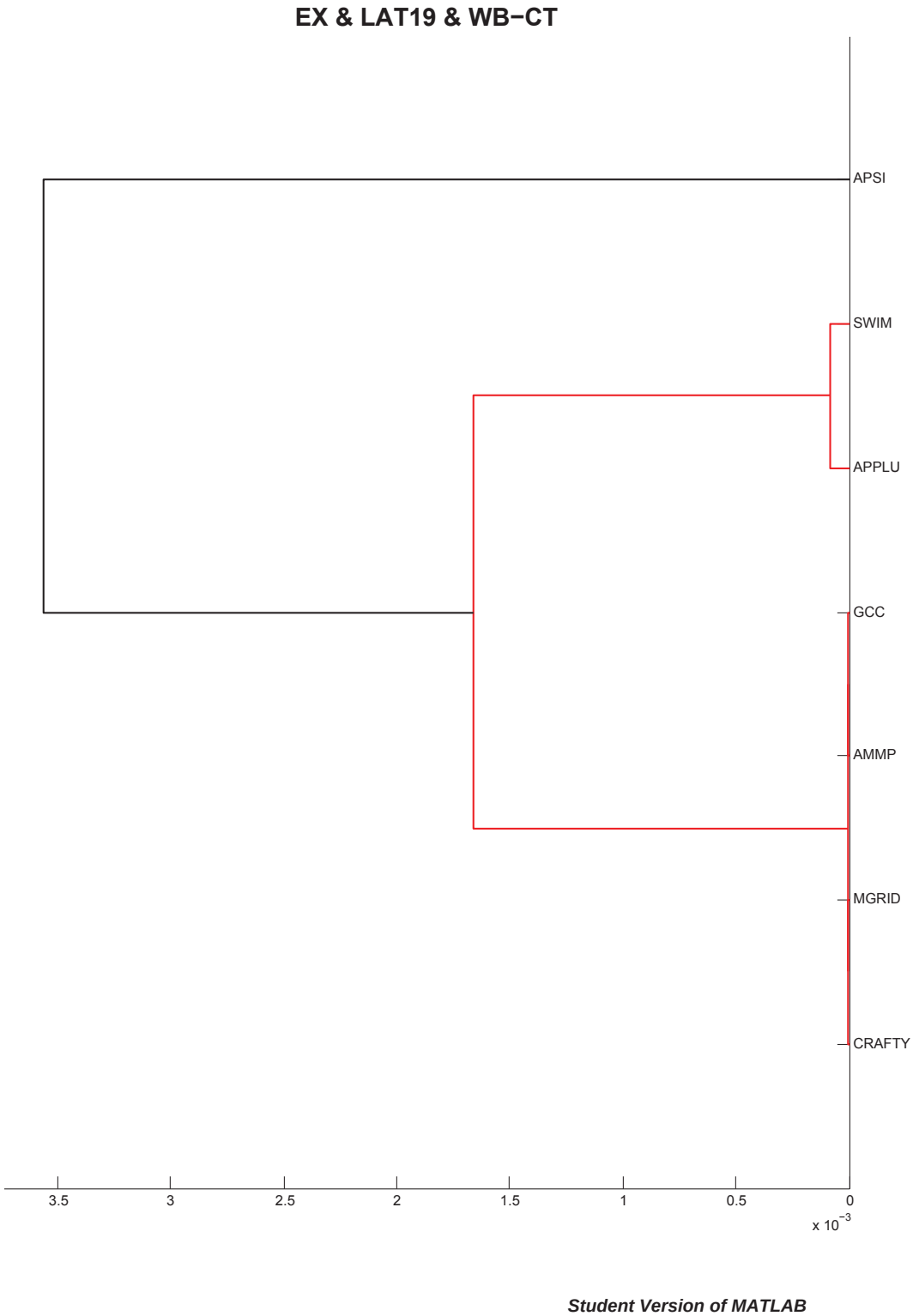

Figure 6.7: Dendrogram of EX&LAT19&WB-CT.

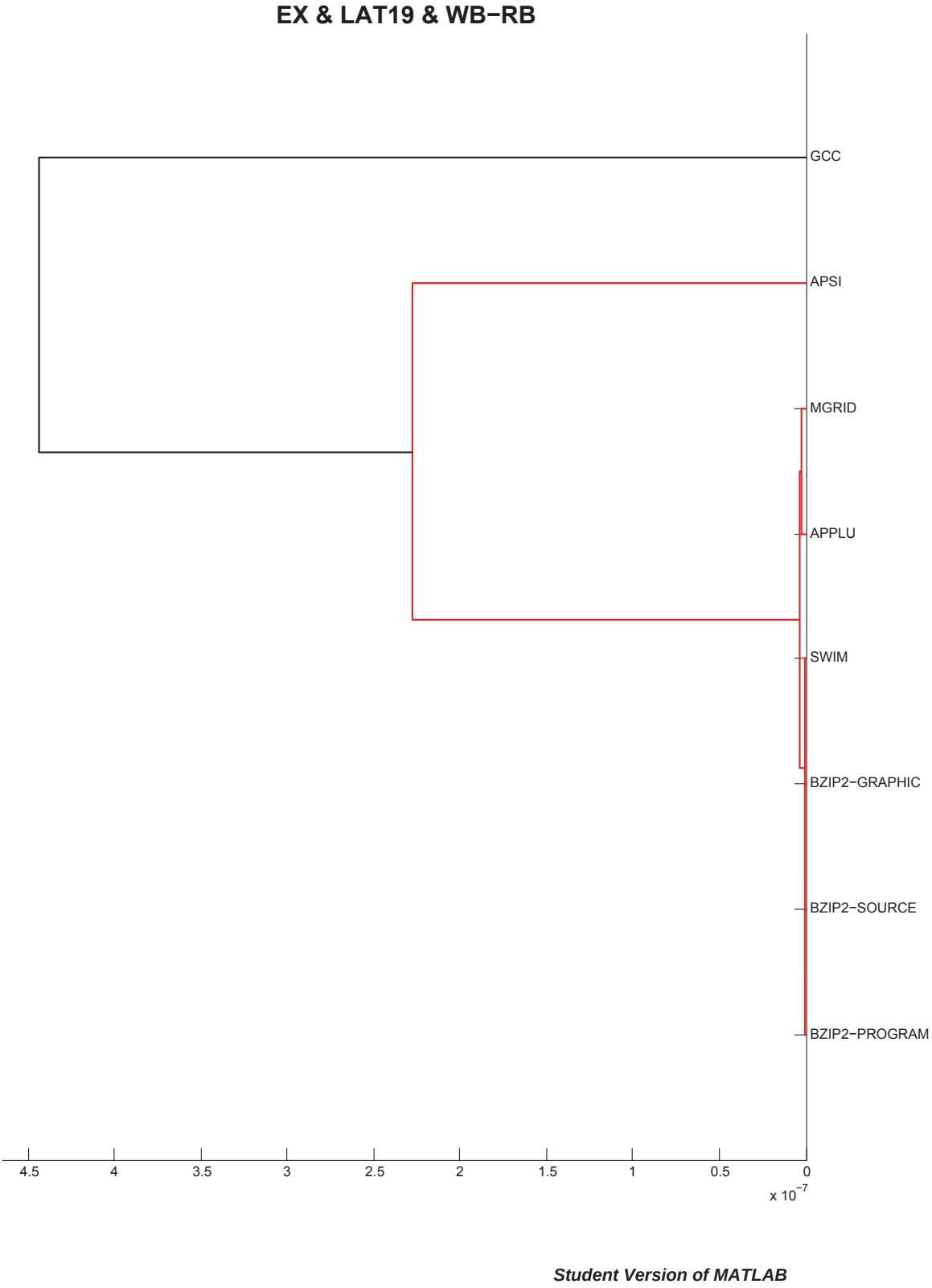

Figure 6.8: Dendrogram of EX&LAT19&WB-RB.

Table 6.1
List of ratio values for START-IF&ROBFULL edges.

Benchmark	Ratio	Type
CRAFTY	0.04	Integer
GAP	0.03	Integer
GCC	0.04	Integer
BZIP2-GRAPHIC	0.11	Integer
BZIP2-PROGRAM	0.08	Integer
BZIP2-SOURCE	0.05	Integer
GZIP-GRAPHIC	0.08	Integer
GZIP-LOG	0.14	Integer
GZIP-RANDOM	0.10	Integer
GZIP-SOURCE	0.08	Integer
PARSER	0.01	Integer
MCF	0.05	Integer
SWIM	0.86	Float
APPLU	0.73	Float
MGRID	0.96	Float
AMMP	0.30	Float
APSI	0.48	Float

197.PARSER and 254.GAP are grouped next to each other.

Clearly, using different input sets for a given benchmark does not make the benchmark behave in an entirely different manner. This grouping also shows that the algorithms implemented by a given benchmark profoundly affects how it behaves. For example, while both GZIP and BZIP2 are compression programs, they implement different algorithms, hence they are in different clusters. In addition, PARSER and GCC are grouped together. GCC spends a significant amount of time parsing the input streams using a similar parsing algorithm.

Figure 6.10 also shows that several floating point programs such as SWIM, MGRID, and APSI are grouped together. While each of these programs implement a different algorithm, they all possess the algorithmic structure of scientific computations such as linear algebra, and numerical function and differential computation algorithms. These results indicate that clustering SFTAGs at a higher level can provide insight into the algorithmic behavior of programs as well.

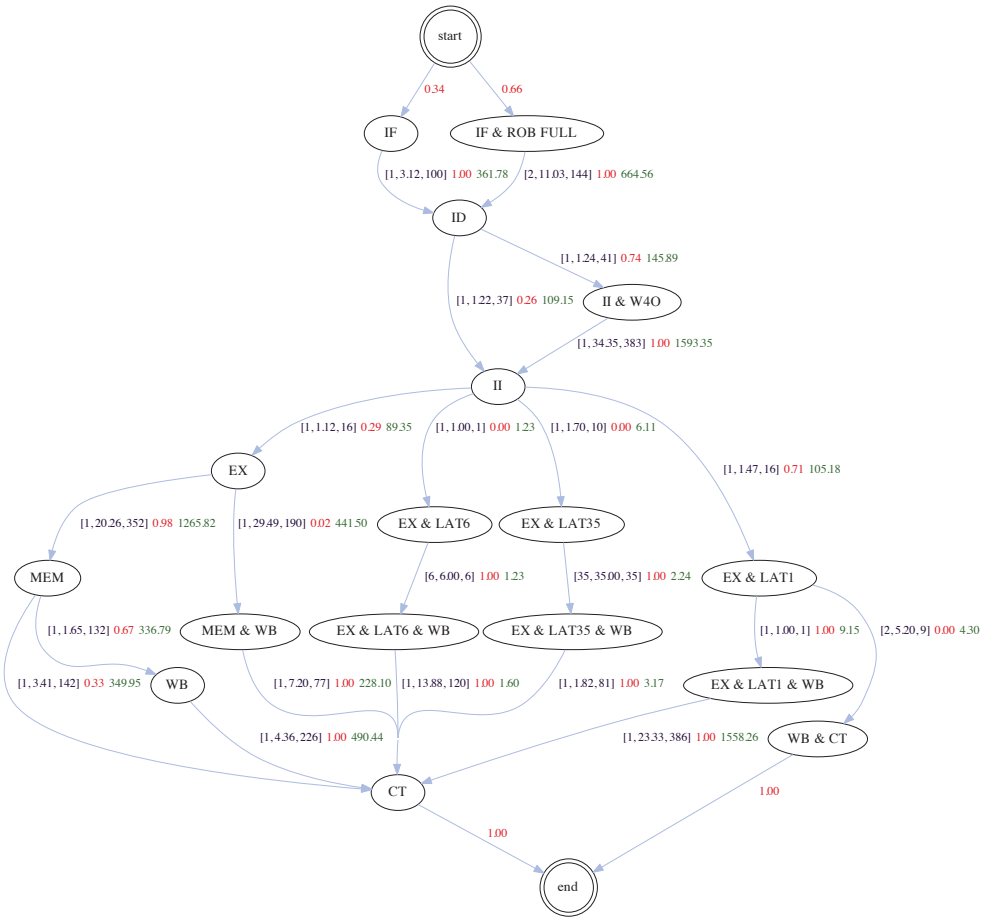

Figure 6.9: SFTAG of 256.BZIP-PROGRAM executed with a perfect branch predictor.

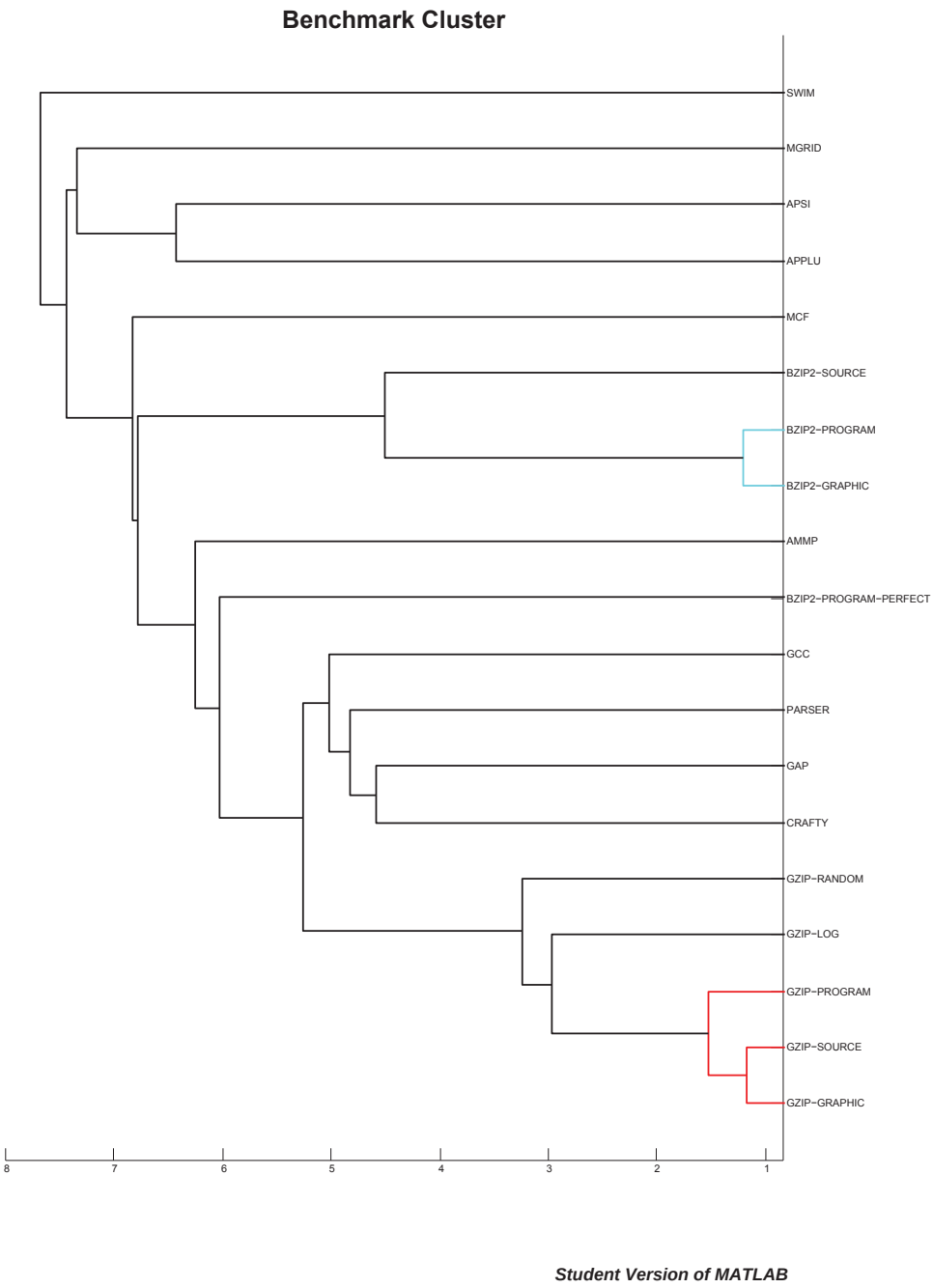

Figure 6.10: Benchmark cluster.

Chapter 7

Conclusion

In this dissertation, we presented a micro-architecture simulator verification framework that relies on visual and analytical discovery of patterns, as well as invariant checking. A considerable amount of time is spent in the micro-architecture research community to develop simulators for new techniques and make sure that they faithfully implement the desired models. While simulator validation (as opposed to verification) has been carried out for most widely used simulators, a robust verification framework is still needed because the validation of a simulator does not guarantee that the modified versions still correctly simulate the desired model. Our developed framework is a step to increase researchers' confidence in simulator implementations using an online process.

The framework we developed and implemented consists of two parts. The first part is First-Order Logic Constraint Specification Language (FOLCSL) that enables users to specify the invariants of the model under consideration. From the first-order logic specification, we automatically synthesize a verification program that reads the event trace generated by a simulator and signals whether all invariants are respected. The second part consists of mining the temporal flow of events using a newly developed representation called State Flow Temporal Analysis Graph (SFTAG). The study includes SFTAGs generated for a wide set of benchmark programs and their analysis using several artificial intelligence algorithms. Our framework improves the computer architecture research and verification processes as shown by the case studies and experiments we have conducted.

7.1 Contributions

The main contributions of this research work are in twofold. First, we designed a formal verification language and implemented the software that allows users to describe simulation properties and check the properties against the output events. Second, we developed and implemented an algorithm that is capable of processing large data sets and representing the temporal information in a graphical form that is amenable to both visual inspection and automatic analysis.

Our techniques serve both verification and debugging purposes and close an important gap in simulation-based research. They complement current approaches by providing temporal information in the form of state flow graphs. Our methods provide an efficient, scalable and practical framework with formal foundations. Our software is publicly available to allow widespread use and to lay the foundation for further improvements. We showed that the data in SFTAGs can be processed again using data mining techniques such as clustering and rule-base algorithms to uncover new patterns.

7.2 Future work

This dissertation work can be extended by developing new techniques for automatic and semi-automatic derivation of invariant rules from event-traces. In addition to the visual and analytical techniques we have outlined, artificial intelligence techniques can be employed to extract additional information from event streams. For example, the causal relationships between changes in performance and the related events that occur can be found using Bayesian learning techniques.

New algorithms that use SFTAGs can be developed to make comparisons to aid during the development of new micro-architectures. In this regards, one can compare different phases of a program within an architecture, different programs on the same architecture, and different architectures on the same program. Both visual and automated techniques can be used to make comparisons.

Finally, the visualization can be improved by creating interactive SFTAGs which display certain properties based on mousing over edges or nodes and by providing comparative views of SFTAG sections.

References

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

[2] 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 Computer Architecture News*, vol. 39, pp. 1–7, August 2011.

[3] S. Onder and R. Gupta, "Automatic generation of microarchitecture simulators," in *Proceedings of the International Conference on Computer Languages (ICCL)*, pp. 80–89, may 1998.

[4] Project Management Institute, *A Guide to the Project Management Body of Knowledge (PMBOK Guide)–Fifth Edition*. Newtown Square, PA, USA: Project Management Institute, Inc., 2013.

[5] B. Boehm, "Guidelines for verifying and validating software requirements and design specifications," in *Euro IFIP 79*, (North Holland), pp. 711–719, 1979.

[6] R. G. Sargent, "Verification and validation of simulation models," in *Proceedings of the Winter Simulation Conference (WSC)*, pp. 183–198, Winter Simulation Conference, 2011.

[7] O. Balci, "Verification, validation, and certification of modeling and simulation applications," in *Proceedings of the Winter Simulation Conference (WSC)*, vol. 1, pp. 150–158, December 2003.

[8] R. Desikan, D. Burger, and S. W. Keckler, "Measuring experimental error in microprocessor simulation," in *Proceedings of the 28th International Symposium on Computer Architecture (ISCA-28)*, pp. 266–277, ACM Press, New York, 2001.

[9] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *IEEE Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[10] R. Desikan, D. Burger, S. W. Keckler, and T. Austin, "Sim-alpha: a validated, execution-driven Alpha 21264 simulator," Tech. Rep. TR-01-23, Department of Computer Sciences, The University of Texas at Austin, 2011.

[11] P. M. Cox, R. A. Betts, C. D. Jones, S. A. Spall, and I. J. Totterdel, "Acceleration of global warming due to carbon-cycle feedbacks in a coupled climate model," *Nature*, vol. 408, pp. 184–187, 2000.

[12] P. Milly, K. A. Dunne1, and A. V. Vecchia, "Global pattern of trends in streamflow and water availability in a changing climate," *Nature*, vol. 438, pp. 347–350, 2005.

[13] J. Karnon, J. Stahl, A. Brennan, J. J. Caro, J. Mar, and J. Möller, "Modeling using discrete event simulation: A report of the ISPOR-SMDM modeling good research practices task force–4," *Medical Decision Making*, vol. 32, pp. 701–711, 2012.

[14] M. Chen, D. Ebert, H. Hagen, R. S. Laramee, R. van Liere, K.-L. Ma, W. Ribarsky, G. Scheuermann, and D. Silver, "Data, information, and knowledge in visualization," *IEEE Comput. Graph. Appl.*, vol. 29, pp. 12–19, Jan. 2009.

[15] J. Ahrens, K. Heitmann, M. Petersen, J. Woodring, S. Williams, P. Fasel, C. Ahrens, C.-H. Hsu, and B. Geveci, "Verifying scientific simulations via comparative and quantitative visualization," *IEEE Computer Graphics and Applications*, vol. 30, no. 6, pp. 16–28, 2010.

[16] Y. Huang, X. Xiang, G. Madey, and S. E. Cabaniss, "Agent-based scientific simulation," *Computing in Science and Engineering*, vol. 7, no. 1, pp. 22–29, 2005.

[17] W. L. Oberkampf and T. G. Trucan, "Verification and validation in computational fluid dynamics," *Progress in Aerospace Sciences*, vol. 38, pp. 209–272, 2002.

[18] E. Bonabeau, "Agent-based modeling: Methods and techniques for simulating human systems," *Proceedings of the National Academy of Science*, vol. 99, no. 3, pp. 7280–7287, 2002.

[19] S. M. Sanchez, "ABC's of output analysis," in *Proceedings of the Winter Simulation Conference WSC-2001*, pp. 30–38, 2001.

[20] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, (New York, NY, USA), pp. 318–319, ACM, 2003.

[21] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, (New York, NY, USA), pp. 84–97, ACM, 2003.

[22] J. P. C. Kleijnen, "Verification and validation of simulation models," *European Journal of Operational Research*, vol. 82, no. 1, pp. 145–162, 1995.

[23] J. P. C. Kleijnen, "Validation of models: statistical techniques and data availability," in *Proceedings of the Winter Simulation Conference (WSC)*, pp. 647–654, 1999.

[24] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-based runtime verification," in *Proceedings of the Fifth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI-2004)*, pp. 44–57, 2004.

[25] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[26] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, pp. 293–303, 2009.

[27] O. Sokolsky, K. Havelund, and I. Lee, "Introduction to the special section on runtime verification," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 3, pp. 243–247, 2012.

[28] A. Pnueli, "The temporal logic of programs," in *Proceedings of the Eighteenth IEEE Symposium on Foundations of Computer Science*, pp. 46–77, 1977.

[29] V. Stolz and E. Bodden, "Temporal assertions using AspectJ," *Electronic Notes in Theoretical Computer Science*, vol. 144, no. 4, pp. 109–124, 2006.

[30] A. Bauer, M. Leucker, and C. Schallhart, "Comparing semantics for runtime verification," *Journal of Logic and Computation*, vol. 20, no. 3, pp. 651–674, 2010.

[31] L. Pike, S. Niller, and N. Wegmann, "Runtime verification for ultra-critical systems," in *Proceedings of the Second International Conference on Runtime Verification (RV 2011)*, pp. 310–324, 2011.

[32] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok, "Software monitoring with controllable overhead," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 3, pp. 327–347, 2012.

[33] B. Bonakdarpour, S. Navabpour, and S. Fischmeister, "Sampling-based runtime verification," in *Proceedings of the Seventeenth International Symposium on Formal Methods (FM 2011), LNCS 6664*, pp. 88–102, 2011.

[34] D. Drusinsky, "The temporal rover and the ATG rover," in *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, (London, UK), pp. 323–330, Springer-Verlag, 2000.

[35] Y. Zhao, S. OberthÃijr, M. Kardos, and F. J. Rammig, "Model-based runtime verification framework for selfoptimizing systems," in *Proceedings of the Fifth Workshop on Runtime Verification*, pp. 125–145, 2006.

[36] T. Bosse, C. M. Jonker, L. V. D. Meij, A. Sharpanskykh, and J. Treur, "Specification and verification of dynamics in cognitive agent models," in *Proceedings of the Sixth International Conference on Intelligent Agent Technology (IAT'06)*, pp. 247–254, Society Press, 2006.

[37] G. Rosu, W. Schulte, and T.-F. Serbanuta, "Runtime verification of c memory safety.," in *RV* (S. Bensalem and D. Peled, eds.), vol. 5779 of *Lecture Notes in Computer Science*, pp. 132–151, Springer, 2009.

[38] 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 Design Automation Conference (DAC)*, pp. 775–778, 2005.

[39] 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 Design, Automation and Test in Europe (DATE) Conference*, pp. 626–629, 2010.

[40] E. El Mandouh and A. Wassal, "Automatic generation of hardware design properties from simulation traces," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 2317–2320, 2012.

[41] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*, (New York, NY, USA), pp. 207–216, ACM, 1993.

[42] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 3rd ed., 2011.

[43] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[44] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, pp. 81–106, 1986.

[45] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[46] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks, 1984.

[47] R. S. Michalski, I. Mozetič, J. Hong, and N. Lavrač, "The multi–purpose incremental learning system AQ15 and its testing application on three medical domains," in *Proc. Fifth National Conference on Artificial Intelligence*, (San Mateo, CA), pp. 1041–1045, Morgan Kaufmann, 1986.

[48] P. Clark and T. Niblett, "The CN2 induction algorithm," *Machine Learning*, vol. 3, pp. 261–283, March 1989.

[49] R. Barták, R. A. Morris, and K. B. Venable, *An Introduction to Constraint-Based Temporal Reasoning, Synthesis Lectures on Artificial Intelligence and Machine Learning*. San Rafael, CA: Morgan and Claypool Publishers, February 2014.

[50] L. Vila, "A survey on temporal reasoning in artificial intelligence," *AI Communications*, vol. 7, pp. 4–28, 1994.

[51] E. Schwalb and L. Vila, "Temporal constraints: A survey," *Constraints: An International Journal*, vol. 2, pp. 129–149, 1998.

[52] J. Allen, "Maintaining knowledge about temporal intervals," *Communications of the ACM*, vol. 26, pp. 832–843, 1983.

[53] J. Allen, "Towards a general theory of action and time," *Artificial Intelligence*, vol. 23, pp. 123–154, 1984.

[54] M. B. Vilain and H. A. Kautz, "Constraint propagation algorithms for temporal reasoning," in *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, pp. 377–382, AAAI Press, CA, 1986.

[55] R. Dechter, I. Meiri, and J. Pearl, "Temporal constraint networks," *Artificial Intelligence*, vol. 49, pp. 61–95, 1991.

[56] T. Vidal and H. Fargier, "Handling contingency in temporal constraint networks: from consistency to controllabilities," *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 11, pp. 23–45, 1999.

[57] K. Stergiou and M. Koubarakis, "Backtracking algorithms for disjunctions of temporal constraints," *Artificial Intelligence*, vol. 120, pp. 81–117, 2000.

[58] I. Tsamardinos and M. E. Pollack, "Efficient solution techniques for disjunctive temporal reasoning problems," *Artificial Intelligence*, vol. 151, pp. 43–89, 2003.

[59] R. Barták and O. Čepek, "Temporal networks with alternatives: Complexity and model," in *Proceedings of the Twentieth International Florida AI Research Society Conference (FLAIRS 2007)*, pp. 641–646, AAAI Press, CA, 2007.

[60] L. Khatib, P. Morris, R. Morris, and F. Rossi, "Temporal constraint reasoning with preferences," in *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-2001)*, pp. 322–327, 2001.

[61] F. Rossi, K. B. Venable, and N. Yorke-Smith, "Controllability of soft temporal constraint problems," in *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP-2004)*, pp. 588–603, 2004.

[62] B. Peintner and M. E. Pollack, "Anytime, complete algorithm for finding utilitarian optimal solutions to stpps," in *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-2005)*, pp. 443–448, AAAI Press, CA, 2005.

[63] K. B. Venable and N. Yorke-Smith, "Disjunctive temporal planning with uncertainty," in *Proceedings of the Twentyfirst International Joint Conference on Artificial Intelligence (IJCAI-2005)*, pp. 1721–1722, 2005.

[64] S. Badaloni, M. Falda, and M. Giacomin, "Integrating quantitative and qualitative fuzzy temporal constraints," *Artificial Intelligence Communications*, vol. 17, no. 4, pp. 187–200, 2004.

[65] C. Pralet and G. Verfaillie, "Time-dependent simple temporal networks: Properties and algorithms," *RAIRO Operations Research*, vol. 47, no. 2, pp. 173–198, 2013.

[66] J. F. Roddick and M. Spiliopoulou, "A survey of temporal knowledge discovery paradigms and methods," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 4, pp. 750–767, 2002.

[67] C. P. Rainsford and J. F. Roddick, "Adding temporal semantics to association rules," in *Proceedings of the Third European Conference in Principles of Data Mining and Knowledge Discovery (PKDD 1999), LNCS 1704*, pp. 504–509, 1999.

[68] H. Mannila, H. Toivonen, and A. Inkeri Verkamo, "Discovery of frequent episodes in event sequences," *Data Mining and Knowledge Discovery*, vol. 1, pp. 259–289, Jan. 1997.

[69] M. J. Zaki, "SPADE: An efficient algorithm for mining frequent sequences," *Machine Learning*, vol. 42, no. 1–2, pp. 31–60, 2001.

[70] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: current status and future directions," *Data Mining and Knowledge Discovery*, vol. 15, no. 1, pp. 55–86, 2007.

[71] D. Berndt and J. Clifford, "Finding patterns in time series: A dynamic programming approach," in *Advances in Knowledge Discovery and Data Mining* (U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, eds.), pp. 229–248, 1995.

[72] E. Keogh and M. Pazzani, "An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback," in *Proceedings of the Fourth ACM International Conference on Knowledge Discovery and Data Mining (KDD 1998)* (R. Agrawal, P. Stolorz, and G. Piatetsky-Shapiro, eds.), pp. 231–241, 1998.

[73] T. W. Liao, "Clustering of time series data–a survey," *Pattern Recognition*, vol. 38, no. 11, pp. 1857–1874, 2005.

[74] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh, "Querying and mining of time series data: experimental comparison of representations and distance measures," in *Proceedings of the Thirtyfourth International Conference Very large Databases (VLDB 2008)*, pp. 1542–1552, 2008.

[75] Y. Li, P. Ning, X. S. Wang, and S. Jajodia, "Discovering calendar-based temporal association rules," *Data and Knowledge Engineering*, vol. 44, no. 2, pp. 193–218, 2003.

[76] E. Winarko and J. F. Roddick, "ARMADA – an algorithm for discovering richer relative temporal association rules from interval-based data," *Data and Knowledge Engineering*, vol. 63, no. 1, pp. 76–90, 2007.

[77] G. N. Norén, J. Hopstadius, A. Bate, K. Star, and I. R. Edwards, "Temporal pattern discovery in longitudinal electronic patient records," *Data Mining and Knowledge Discovery*, vol. 20, no. 3, pp. 361–387, 2010.

[78] D. Lymberopoulos, A. Bamis, and A. Savvides, "Extracting spatiotemporal human activity patterns in assisted living using a home sensor network," *Universal Access in the Information Society*, vol. 10, no. 2, pp. 125–138, 2011.

[79] J. Yang and J. Leskovec, "Patterns of temporal variation in online media," in *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining (WSDM-2011)*, pp. 177–186, 2011.

[80] J. F. Roddick, M. Spiliopoulou, D. Lister, and A. Ceglar, "Higher order mining," *ACM SIGKDD Explorations Newsletter archive*, vol. 10, no. 1, pp. 5–17, 2008.

[81] D. Patnaik, M. Marwah, R. Sharma, and N. Ramakrishnan, "Sustainable operation and management of data center chillers using temporal data mining," in *Proceedings of the Fifteenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2009)*, pp. 1305–1314, 2009.

[82] D. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," *SIGARCH Computer Architecture News*, vol. 25, pp. 13–25, June 1997.

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

[84] 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*, vol. 26, pp. 52–60, 2006.

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

[86] C. J. Mauer, M. D. Hill, and D. A. Wood, "Full-system timing-first simulation," in *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2002)*, (New York, NY, USA), pp. 108–116, ACM, June 2002.

[87] A. J. KleinOsowski and D. J. Lilja, "MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research," *IEEE Computer Architecture Letters*, vol. 1, no. 1, p. 7, 2002.

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

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

[90] 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*, vol. 31, no. 4, pp. 19–24, 2004.

[91] Standard Performance Evaluation Corporation, "SPEC CPU2000 V1.3 documentation." `http://www.spec.org/cpu2000/docs/`. Accessed: 2014-07-10.

[92] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008. Available online: http://nlp.stanford.edu/IR-book/. Accessed: 2014-05-22.

[93] D. J. Hand, P. Smyth, and H. Mannila, *Principles of Data Mining*. Cambridge, MA, USA: MIT Press, 2001.

Appendix

Appendix A contains letter from ACM that gives permission to reprint Chapter 4 and Chapter 5. Appendix B contains letters from AAAI that give permission to reprint Chapter 4.

Appendix A
Copyright Permission from ACM for Chapter 4 and 5

ACM Copyright and Audio/Video Release

Title of the Work: Verifying Micro-architecture Simulators using Event Traces

Publication and/or Conference Name: ICS'14: Proceedings of the 2014 International Conference on Supercomputing Proceedings

Author/Presenter(s): Hui Meen Nyew; Nilufer Onder; Soner Onder; Zhenlin Wang

Auxiliary Materials (provide filenames and a description of auxiliary content, if any, for display in the ACM Digital Library. The description may be provided as a ReadMe file):

I. Copyright Transfer, Reserved Rights and Permitted Uses

* Your Copyright Transfer is conditional upon you agreeing to the terms set out below.

Copyright to the Work and to any supplemental files integral to the Work which are submitted with it for review and publication such as an extended proof, a PowerPoint outline, or appendices that may exceed a printed page limit, (including without limitation, the right to publish the Work in whole or in part in any and all forms of media, now or hereafter known) is hereby transferred to the ACM (for Government work, to the extent transferable) effective as of the date of this agreement, on the understanding that the Work has been accepted for publication by ACM.

Reserved Rights and Permitted Uses

(a) All rights and permissions the author has not granted to ACM are reserved to the Owner, including all other proprietary rights such as patent or trademark rights.

(b) Furthermore, notwithstanding the exclusive rights the Owner has granted to ACM, Owner shall have the right to do the following:

(i) Reuse any portion of the Work, without fee, in any future works written or edited by the Author, including books, lectures and presentations in any and all media.

(ii) Create a "Major Revision" which is wholly owned by the author

(iii) Post the Accepted Version of the Work on (1) the Author's home page, (2) the Owner's institutional repository, or (3) any repository legally mandated by an agency funding the research on which the Work is based.

(iv) Post an "Author-Izer" link enabling free downloads of the Version of Record in the ACM Digital Library on (1) the Author's home page or (2) the Owner's institutional repository;

(v) Prior to commencement of the ACM peer review process, post the version of the Work as submitted to ACM ("Submitted Version" or any earlier versions) to non-peer reviewed servers;

(vi) Make free distributions of the final published Version of Record internally to the Owner's employees, if applicable;

(vii) Make free distributions of the published Version of Record for Classroom and

Personal Use;

(viii) Bundle the Work in any of Owner's software distributions; and

(ix) Use any Auxiliary Material independent from the Work.

Authors should understand that consistent with ACMs policy of encouraging dissemination of information, each work published by ACM appears with the ACM copyright and the following notice:

When preparing your paper for submission using the ACM templates, you will need to include the rights management and bibstrip text blocks below to the lower left hand portion of the first page. As this text will provide rights information for your paper, please make sure that this text is displayed and positioned correctly when you submit your manuscript for publication.

"Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org."

☑ A. Assent to Assignment. I hereby represent and warrant that I am the sole owner (or authorized agent of the copyright owner(s)), with the exception of third party materials detailed in section III below. I have obtained permission for any third-party material included in the Work.

☐ B. Declaration for Government Work. I am an employee of the National Government of my country and my Government claims rights to this work, or it is not copyrightable (Government work is classified as Public Domain in U.S. only)

Are any of the co-authors, employees or contractors of a National Government?
○ Yes ◉ No
Country:

II. PERMISSION FOR CONFERENCE TAPING AND DISTRIBUTION (Check A and, if applicable, B)
A. Audio /Video Release

I hereby grant permission for ACM to include my name, likeness, presentation and comments in any and all forms, for the Conference and/or Publication.

I further grant permission for ACM to record and/or transcribe and reproduce my presentation as part of the ACM Digital Library, and to distribute the same for sale in complete or partial form as part of an ACM product on CD-ROM, DVD, webcast, USB device, streaming video or any other media format now or hereafter known.

I understand that my presentation will not be sold separately by itself as a stand-alone product without my direct consent. Accordingly, I give ACM the right to

use my image, voice, pronouncements, likeness, and my name, and any biographical material submitted by me, in connection with the Conference and/or Publication, whether used in excerpts or in full, for distribution described above and for any associated advertising or exhibition.

Do you agree to the above Audio/Video Release? ◉ Yes ○ No

B. Auxiliary Materials, not integral to the Work

I hereby grant ACM permission to serve files named below containing my Auxiliary Material from the ACM Digital Library. I hereby represent and warrant that my Auxiliary Material contains no malicious code, virus, trojan horse or other software routines or hardware components designed to permit unauthorized access or to disable, erase or otherwise harm any computer systems or software, and I hereby agree to indemnify and hold harmless ACM from all liability, losses, damages, penalties, claims, actions, costs and expenses (including reasonable legal expense) arising from the use of such files.

☑ I agree to the above Auxiliary Materials permission statement

III. Third Party Materials

In the event that any materials used in my presentation or Auxiliary Materials contain the work of third-party individuals or organizations (including copyrighted music or movie excerpts or anything not owned by me), I understand that it is my responsibility to secure any necessary permissions and/or licenses for print and/or digital publication, and cite or attach them below.

◉ We/I have not used third-party material.
○ We/I have used third-party materials and have necessary permissions.

IV. Artistic Images

If your paper includes images that were created for any purpose other than this paper and to which you or your employer claim copyright, you must complete Part IV and be sure to include a notice of copyright with each such image in the paper.
◉ We/I do not have any artistic images.
○ We/I have any artistic images.

V. Representations, Warranties and Covenants

The undersigned hereby represents, warrants and covenants as follows:

(a) Owner is the sole owner or authorized agent of Owner(s) of the Work;

(b) The undersigned is authorized to enter into this Agreement and grant the rights included in this license to ACM;

(c) The Work is original and does not infringe the rights of any third party; all permissions for use of third-party materials consistent in scope and duration with the rights granted to ACM have been obtained, copies of such permissions have been provided to ACM, and the Work as submitted to ACM clearly and accurately

indicates the credit to the proprietors of any such third-party materials (including any applicable copyright notice), or will be revised to indicate such credit;

(d) The Work has not been published except for informal postings on non-peer reviewed servers, and Owner covenants to use best efforts to place ACM DOI pointers on any such prior postings;

(e) The Auxiliary Materials, if any, contain no malicious code, virus, trojan horse or other software routines or hardware components designed to permit unauthorized access or to disable, erase or otherwise harm any computer systems or software; and

(f) The Artistic Images, if any, are clearly and accurately noted as such (including any applicable copyright notice) in the Submitted Version.

☑ I agree to the Representations, Warranties and Covenants

DATE: **03/23/2014** sent to hnyew@mtu.edu; nilufer@mtu.edu; soner@mtu.edu; zlwang@mtu.edu at **18:03:06**

Appendix B
Copyright Permission from AAAI for Chapter 4

Association for the Advancement of Artificial Intelligence
2275 East Bayshore Road, Suite 160
Palo Alto, California 94303 USA

AAAI COPYRIGHT FORM

Title of Article/Paper: A First-Order Logic Based Framework for Verifying Simulations.

Publication in Which Article/Paper Is to Appear: AAAI-13 Conference Proceedings

Author's Name(s): NYEIN HUI MEEN, NILUFER ONDER, SONER ONDER, ZHENLIN WANG

Please type or print your name(s) as you wish it (them) to appear in print

PART A – COPYRIGHT TRANSFER FORM

The undersigned, desiring to publish the above article/paper in a publication of the Association for the Advancement of Artificial Intelligence, (AAAI), hereby transfer their copyrights in the above article/paper to the Association for the Advancement of Artificial Intelligence (AAAI), in order to deal with future requests for reprints, translations, anthologies, reproductions, excerpts, and other publications.

This grant will include, without limitation, the entire copyright in the article/paper in all countries of the world, including all renewals, extensions, and reversions thereof, whether such rights current exist or hereafter come into effect, and also the exclusive right to create electronic versions of the article/paper, to the extent that such right is not subsumed under copyright.

The undersigned warrants that they are the sole author and owner of the copyright in the above article/paper, except for those portions shown to be in quotations; that the article/paper is original throughout; and that the undersigned right to make the grants set forth above is complete and unencumbered.

If anyone brings any claim or action alleging facts that, if true, constitute a breach of any of the foregoing warranties, the undersigned will hold harmless and indemnify AAAI, their grantees, their licensees, and their distributors against any liability, whether under judgment, decree, or compromise, and any legal fees and expenses arising out of that claim or actions, and the undersigned will cooperate fully in any defense AAAI may make to such claim or action. Moreover, the undersigned agrees to cooperate in any claim or other action seeking to protect or enforce any right the undersigned has granted to AAAI in the article/paper. If any such claim or action fails because of facts that constitute a breach of any of the foregoing warranties, the undersigned agrees to reimburse whomever brings such claim or action for expenses and attorneys' fees incurred therein.

Returned Rights

In return for these rights, AAAI hereby grants to the above author(s), and the employer(s) for whom the work was performed, royalty-free permission to:

1. Retain all proprietary rights other than copyright (such as patent rights).
2. Personal reuse of all or portions of the above article/paper in other works of their own authorship. This does not include granting third-party requests for reprinting, republishing, or other types of reuse. AAAI must handle all such third-party requests.
3. Reproduce, or have reproduced, the above article/paper for the author's personal use, or for company use provided that AAAI copyright and the source are indicated, and that the copies are not used in a way that implies AAAI endorsement of a product or service of an employer, and that the copies per se are not offered for sale. The foregoing right shall not permit the posting of the article/paper in electronic or digital form on any computer network, except by the author or the author's employer, and then only on the author's or the employer's own web page or ftp site. Such web page or ftp site, in addition to the aforementioned requirements of this Paragraph, shall not post other AAAI copyrighted materials not of the author's or the employer's creation (including tables of contents with links to other papers) without AAAI's written permission.
4. Make limited distribution of all or portions of the above article/paper prior to publication.
5. In the case of work performed under a U.S. Government contract or grant, AAAI recognized that the U.S. Government has royalty-free permission to reproduce all or portions of the above Work, and to authorize others to do so, for official U.S. Government purposes only, if the contract or grant so requires.

In the event the above article/paper is not accepted and published by AAAI, or is withdrawn by the author(s) before acceptance by AAAI, this agreement becomes null and void.

(1) [redacted] 4/9/2013

Author/Authorized Agent for Joint Author's Signature — Date

Employer for whom work was performed — Title (if not author)

(For jointly authored Works, all joint authors should sign unless one of the authors has been duly authorized to act as agent for the others.)

Association for the Advancement of Artificial Intelligence
2275 East Bayshore Road, Suite 160
Palo Alto, California 94303 USA

AAAI DISTRIBUTION LICENSE

Title of Article/Paper (Work): A First-Order Logic Based Framework for Verifying Simulations

Publication in Which Article Is to Appear: AAAI-13 Conference Proceeding.

Author's Name(s): Nyew Hui Meen, Nilufer Onder, Soner Onder, Zhenlin Wang

Please type or print your name(s) as you wish it to appear in print

Author's Grant

The undersigned, desiring to publish the above Work in a publication of the Association for the Advancement of Artificial Intelligence, (AAAI), hereby hereby grants to the Association for the Advancement of Artificial Intelligence the following nonexclusive rights:

(1) The perpetual, nonexclusive world rights to use the above-named Work as part of an AAAI publication, in all languages and for all editions.

(2) The right to use the Work, together with the author's name and pertinent biographical data, in advertising and promotion of it and the AAAI publication.

(3) The right to publish or cause to be published the Work in connection with any republication of the AAAI publication in any medium including electronic.

(4) The right to, and authorize others to, publish or cause to be published the Work in whole or in part, individually or in conjunction with other works, in any medium including electronic.

Author's Warranty

The undersigned warrants that they are the sole author and owner of the Work, except for those portions shown to be in quotations; that the Work is original throughout; that publication thereof will not violate or infringe any copyright or proprietary right; that the Work contains no scandalous, libelous, obscene, or otherwise unlawful matter, and that it does not invade the privacy or otherwise infringe upon the common-law or statutory rights of anyone; and that the undersigned's right to make the licenses set forth is complete and unencumbered.

Indemnifications; Enforcement of Rights

If anyone brings any claim or action alleging facts that, if true, constitute a breach of any of the foregoing warranties, the undersigned will hold harmless and indemnify AAAI, their grantees, their licensees, and their distributors against any liability, whether under judg- ment, decree, or compromise, and any legal fees and expenses arising out of that claim or actions, and the undersigned will cooperate fully in any defense AAAI may make to such claim or action. Moreover, the undersigned agrees to cooperate in any claim or other action seeking to protect or enforce any right the undersigned has granted to AAAI in the article/paper. If any such claim or action fails because of facts that constitute a breach of any of the foregoing warranties, the undersigned agrees to reimburse whomever brings such claim or action for expenses and attorneys' fees incurred therein.

If the foregoing correctly reflects the understanding between us, please sign this document in the place indicated below and return it to us. In the event the above article/paper is not accepted and published by AAAI, or is withdrawn by the author(s) before acceptance by AAAI, this agreement becomes null and void.

[redacted]
Author's Signature

4/9/2013
Date

Employer for whom work was performed

Title (if not author)