Improving Accuracy and Performance Through Automatic Model Generation for Gate-Level Circuit PDES with Reverse Computation

Elsa Gonsiorowski, Justin M. LaPre, Christopher D. Carothers
Rensselaer Polytechnic Institute
Troy, NY 12180
{gonsie,laprej,chrisc}@cs.rpi.edu

ABSTRACT

Gate-level circuit simulation is an important step in the design and validation of complex circuits. This step of the process relies on existing libraries for gate specifications. We start with a generic gate model for Rensselaer's Optimistic Simulation System (ROSS), a parallel discrete-event simulation framework. This generic model encompasses all functionality needed by optimistic simulation using reverse computation. We then describe a parser system which uses a standardized gate library to create a specific model for simulation. The generated model is comprised of several functions including those needed for an accurate model of timing behavior.

To quantify the improvements that an automatically generated model can have over a hand written model we compare two gate library models: an automatically generated LSI-10K library model and a previously investigated, handwritten, simplified GTECH library model [19]. We conclude that the automatically generated model is a more accurate model of actual hardware. The generated model also represents the timing behavior with an approximately 50 times higher degree of fidelity. In comparison to previous results, we find that the automatically generated model is able to achieve better optimistic simulation performance when measured against conservative simulation. We identify peak optimistic performance when using 128 MPI-Ranks on eight nodes of an IBM Blue Gene/Q machine.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Design, Performance

Keywords

Parallel discrete-event simulation; digital logic simulation; automatic logic generation; design automation

SIGSIM-PADS'15, June 10–12, 2015, London, United Kingdom.

ACM 978-1-4503-3583-6/15/06 ...$15.00.
DOI: http://dx.doi.org/10.1145/2769458.2769463.

1. INTRODUCTION

In the world of simulation, model creation can be a complex act of creativity and ingenuity. Model developers must create a detailed representation of a complex real-world system. In addition, this representation must be built within a specific simulation framework. When working with a discrete-event simulation tool, developers must account for model memory and understand the details of how a model's state may change over time. By creating a generic model for a common simulation use case, we increase model flexibility to allow for any set of specific details. A user can then make use of an existing parser to transform a domain-specific description into code used by the generic model.

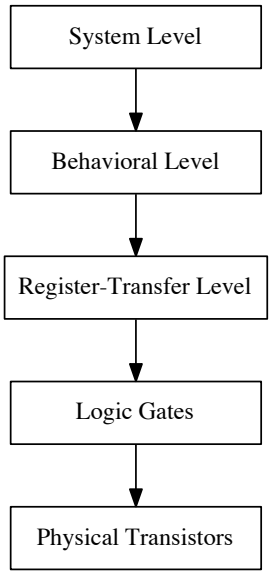

Figure 1: Levels of abstraction within the circuit design process. Automatic tools perform synthesis on a given design to generate the design at the next (lower) level. This design is then validated through one or more tests.

The process for designing large integrated circuits (such as processors) involves simulation at many levels of abstraction (see Figure 1). Gate-level simulation usually begins with circuit designs at the Register-Transfer Level (RTL), written in the Verilog Hardware Description Language [23, 24]. To synthesize the RTL description, one must select a gate library. This library contains descriptions of the logic gates available. While this can be as simple as the set of *and*, *or*, and *not* boolean logic gates, more extensive gate libraries

are often used. Commercial gate libraries will describe the electrical properties of a gate, such as its power draw and resistance.

At the logic gate level of circuit abstraction there are many functional validation tests that are performed. Here, simulation is a common tool. Simulation is used to test expected output signals (given a known set of input signals). Simulation can also be used as a virtual oscilloscope to verify electrical signals at various places within the circuit.

Integrated circuits have become so complex that sequential simulation tools are no longer adequate. Many commercial gate-level simulation tools have limited parallel capability, thus limiting simulation to sub-components of the larger circuit. Large circuits have a wealth of inherent parallelism that can be fully exploited with parallel discrete-event simulation.

This paper describes a generic model for logic-level (gate-level) circuit simulation. This generic model provides the core capability upon which any particular logic gate library can be modeled and simulated. Here we demonstrate that by using a generic model with automatically generated library details the overall simulation is more accurate and has better performance for optimistic simulation, when compared to conservative simulation. To our knowledge, our contributions include the first ever results of gate-level circuit simulation on a modern supercomputing platform, namely the IBM Blue Gene/Q.

We begin with a review of simulation and gate-level circuit modeling, including essential terminology, in Section 2. Next, details of the generic model are described in Section 3. We then describe the process of parsing an existing gate description written to domain-specific Liberty language in Section 4. In Section 5, we illustrate improved accuracy and performance by comparing a handwritten model with the generic model with automatically generated gate library details. Section 6 presents related research in the area of automatic model generation and highlights the importance of logic-level simulation. Finally, we present a summary of the value and importance of tools for automatic model generation in Section 7.

2. BACKGROUND

Computer simulations model real world behavior in order to understand a complex system over time. Within discrete-event simulations the state of the modeled system can only be modified through a sequence of time-stamped events [39]. Thus these simulations are from constructed of two entities:

- **Events**
  Each event occurs at a particular instance in time and contains a specific piece of information. An event will directly affect only one logical process, but a series of events can be said to be causally linked.

- **Logical Processes or LPs**
  These entities encapsulate the particular state of a portion of the modeled system. LPs are directly affected by events, and can cause change to other parts of the system by "sending" or "scheduling" events in the future.

When a simulation engine exists within a parallel or distributed environment it falls in the research area of *Parallel Discrete-Event Simulation* (PDES). The primary challenge that a legitimate PDES engine must address is that of unique event serializability [17]. That is, the ordering of events within a simulation must be deterministic across any underlying configuration of execution hardware (whether executed in a single processor environment or on any number of parallel processor configurations). There are two approaches used by parallel simulation engines to achieve the strict and reproducible order of events: conservative and optimistic synchronization [17].

2.1 Conservative Synchronization

Conservative synchronization within a parallel simulation engine ensures that there is both local and global in-order execution of events. That is, through a conservative synchronization algorithm, no LP is allowed to process an event unless it can be proven that all earlier events which affect the given LP have already been processed.

The YAWNS algorithm is most common conservative synchronization algorithm used in modern simulation engines [35,36]. This conservative algorithm uses global synchronization coupled with a local processing window (called the *global lookahead* window). The global lookahead is used to determine the minimum allowable timestamp increment for events. That is, upon processing an event at time t, the given LP can only schedule events at a time greater than $t + global\ lookahead$.

The relationship between lookahead window size and the average length of event delay within in a system greatly impacts the speedup that can be achieved by a conservative simulation [11]. If the lookahead window is relatively small in comparison the average event delay, there will negative impact on conservative performance. This is due to the fact that global synchronization must happen more frequently as the simulation progresses. If all event delays are of a uniform size, then the lookahead window can be quite large and conservative simulation can perform increasingly well.

2.2 Optimistic Synchronization

Within optimistically synchronized parallel simulations there are no guarantees of global in-order execution of events. This means that an LP may process a sequence of events out of serial order. The most prevalent algorithm in this area is Time Warp [26]. This method keeps track of inter-event causality. As such, when an out-of-order event is detected, the system is able to recover [31].

System recovery occurs at an LP level. One method of LP recovery is called *reverse computation* [12]. This method uses a function to "un-process" a given event. This allows LPs to *rollback* or *reverse* the effects of a series of events and begin forward event processing with a more correct ordering. Usually, the onus of writing the function for reversing the state of an LP lies on the model developer.

When an LP detects out-of-order event, the event is said to cause the LP to *rollback*. This rollback may require that certain messages be canceled. This cancellation process is done through *anti-messages*. Within Time Warp systems there are two categories of rollbacks [17]:

- **Primary Rollback**
  A rollback triggered by receiving a late message. For an LP at time t, a primary rollback occurs when it receives an event at a time less than t. This may cause some anti-messages to be sent.

- **Secondary Rollback**
  A rollback triggered by an anti-message corresponding to a message which has already processed by an LP.

2.3 ROSS

Rensselaer's Optimistic Simulation System, ROSS, is a prominent PDES engine [7, 10, 12, 21]. This ANSI C engine includes a reversible random number generator and is designed for fast and

efficient performance. ROSS performs conservative simulation using the YAWNS protocol and optimistic simulation using the Time Warp algorithm with reverse computation. Recently, ROSS has been shown to be remarkable scalable [6].

2.4 Circuit Design Process

Circuit design for modern processors is a complex activity. The design process involves several levels of abstraction and uses simulation for verification. When a circuit design is finally realized at the logic gate level, the size of the problem to be simulated has increased greatly. Traditionally, this digital logic simulation step has been a bottle neck in the design process [9].

Parallel simulation, particularly PDES, has been found to be quite appropriate for this problem [5,9]. Implementations of parallel digital logic simulation exist for many flavors of the viable PDES algorithms, as this was once an engaging area of research [9,34,40,41]. More recently, however, there has been a lack of activity in this field, with some believing that there is little room for improvement on existing techniques [22]. This limited view uses traditional metrics which only evaluate the simulation systems themselves [5, 13, 16], without perspective on the larger circuit design process.

The research presented here involves several concepts from electrical engineering, including the following terms:

- **Gate**
  A small component that performs a logic operation on a set of inputs to produce a set of outputs. A *nand* gate is an example of one boolean logic gate. Some gates may be able to store data and have an internal state (e.g., a *latch*).
- **Pin**
  A point of contact for an electrical component within a circuit. Electrical signals are carried in to and out of a logic gate along its pins. When a gate has an internal state, the data is said to reside on "internal pins."
- **Netlist**
  A list of all components (gates) and their connections for a given circuit. A particular netlist description of a circuit uses components specified in a provided gate library.

3. GENERIC MODEL

The generic gate model is the foundation for any gate instance LP. This generic model encompasses all nonspecific gate behavior, including: state initialization, message passing, event handling, and reverse event handling. The specific functions of a particular gate type are abstracted away with function pointers. By implementing the fundamentals of a gate model in a nonspecific way, any number of gates types can be modeled. In the subsequent section, we review the common functionality that the generic gate model provides.

3.1 Gate State

An LP's *state* in ROSS must be flexible enough to encapsulate the state of any logic gate. By condensing the core characteristics of any logic gate into a defined set, we created a generic gate model. This is relatively straightforward for logic gates and they can be defined by the following characteristics:

- **Type**
  This attribute simply identifies which type of gate (from the provided gate library) this instance represents.
- **Input Pins**
  This vector represents the pins where electrical signals are received. The number of incoming wires to a single input pin of a gate is called *fanin*. Fanin is usually limited as no more than one wire connects to an input pin.
- **Internal Pins**
  This vector represents the internal state of a logic gates and is only used by some more complicated gate types, such as flip-flops and latches.
- **Output Pins**
  This vector represents the outgoing electrical signals a gate can produce. The number of outgoing wires from any output pin is called *fanout*. Fanout is not limited, meaning several wires can connect from one output pin.

3.2 State Initialization

During LP initialization, a particular circuit model is instantiated. The generic gate model works with a description of a particular circuit that has been pre-processed for fast parallel instantiation. Each LP is provided with a predefined-size chunk of string data which it parses into its gate type and pin connections.

One challenge of LP initialization is unconstrained fanout for a particular logic gate instance. A classic example of this is a poorly defined clock tree, where one logic gate defines the clock signal for an entire circuit. In the OpenSPARC T2 example (in Section 5), a single gate experiences a fanout of over 5,000 wires. The issue arises during pre-processing, when a logic gate instance must be defined in a pre-determined amount of space. This ensures fast startup, but cannot include all information about a gate instance's outgoing connections. Instead, we provide a size for the output pins vector, and perform a 2-phase initialization. First, each LP allocates memory based upon the gate instance information read from the initialization file. Second, each LP sends a message to the LPs connected on its input pins notifying them of a link.

3.3 Messages

In any discrete-event simulation, the messages represent small packets of communication between LPs. These messages trigger any and every change within the system. When using ROSS's optimistic mode with reverse computation, the messages gain additional importance. The same message processed during forward event handling is un-processed as an *anti-message* during reverse event handling. We take advantage of this fact by using the message to store pieces of LP state that are destroyed during forward event handling. These values are easily restored from the anti-message during rollback.

Messages store the following:

- **Sender Information**
  This includes the sender ID and which pin the value was sent from.
- **Receiver Information**
  This includes which pin the value is received upon.
- **Signal Information**
  This may be a high or low value (some gate libraries make use of more than two possible signal values).

Additional space in the message is allocated for latch-style gates with internal logic pins. These gates store information in internal state that must be state-saved to the message during forward event handling.

3.4 Forward Event Handling

Event handling during simulation is relatively straightforward for any generic logic gate model. Upon receiving an electrical signal (such as a digital 1 or 0) on an input pin, the gate calculates signals to send through its output pins. The model LPs then send messages with a specific delay. For a code snippet, see Figure 2. It is important to note that any LP state value that is overwritten is saved to the message before destruction. The message processed here becomes the anti-message which is processed during an optimistic simulation rollback.

The flexibility to model any given library of gates is done through type defined function pointers. For each gate type-defined in the Liberty library, several functions are generated. The most essential function is the logic function. Each gate type generates a logic function adhering to the type definition: `typedef void (*logicFunction) (int input[], int internal[], int output[])`. The values of the output pins are then sent through a message simulating an electrical signal. These messages have a delay associated with them, depend on whether the incoming input signal was rising or falling. We use the following delay function type: `typedef float (*delayFunction) (int inputPin, int outputPin, int risingFlag)`.

To make these specific functions easily accessible by the generic model, they are placed in an array. These arrays can be referenced by the integer defining an LP's particular gate type.

3.5 Reverse Event Handling

The ability of an LP to rollback, or revert to its state to a previous point in time, is a key property of any optimistic simulation model. Being able to achieve this in a programmatic manner is not always an easy task, particularly if a model has large state or encapsulates complex logic. This is decidedly not the case when it comes to simple logic gates. The crucial detail of a logic gate is that the output signals can be calculated from the given set of input (and possibly internal) pins. Thus, an event that stores information used by the forward event handler can easily store the same information needed to revert the LP state.

The reverse event handler for the generic gate model is surprising uncomplicated (see Figure 3). The ROSS engine does the hard work of tracking event causality. An individual LP needs to only "un-process" an anti-message. For the generic model, that involves using values stored the anti-message to revert the state. These are the same values that were saved to the message during the forward event handling.

For any given gate library, there is no need to undo any logic which took place within the gate. The output pins will be re-evaluated during the next forward execution of an event. This elementary aspect of logic gates makes the automatic generation of any model straightforward.

4. MODEL GENERATION

To model the specific behavior of gates in a specified library, the library description file must be parsed. The goal for this process is to transform the domain-specific gate descriptions into an equivalent model for ROSS written in C. Currently, our generic model (Section 3) uses only the timing and boolean-logic data for each gate. The simplicity of the generic model makes it easy to extend to other gate properties, such as power usage.

This parsing is straight forward for any library that adheres to the domain-specific Liberty format [30]. To parse the grammar defined by Liberty, we created a Python Lex-Yacc LR-parser [8].

This parsers builds a collection of Gate_Type objects, consisting of pins and "special" internal functions. Each pin is associated with a direction, and output pins are associated with a boolean expression representing a logic function. Once this object hierarchy is created from the input Liberty file, each Gate_Type preforms a local analysis to its rename pins with array references. Finally, the model C code is generated.

Figure 4 shows an example of this transformation from Liberty format to equivalent C code. This automatically generated model not only includes the logic function of each gate, but encompasses the timing information as well. For validation, the generated code can be directly compared to the provided gate library definition. With automatic model generation, an end-user can experiment with many different gate libraries. This allows the simulated model to be consistent with the gates used during fabrication.

By creating an automatic generation tool, we have remove the need for a human to hand write model code. For example, the LSI-10K library (discussed in Section 5) details 163 different logic gates. The model code describing this library is over 9,000 lines-of-code. The flexibility of our approach allows for any standardized library to be used within ROSS by simply running a script. In the remainder of this section we describe the important elements of this model generation process.

4.1 Pin Names

The pins associated with a particular logic gate are guaranteed to have unique names. These names can be easily alphabetized, creating a lexicographical ordering. It is this alphabetically order that maps pin names to array indices. This name mapping takes place when a boolean expression (i.e., the logic function for an output pin) is converted to the equivalent C representation.

For example, take a Liberty defined OR gate with two input pins, *A* and *B*, and one output pin *Z* (see Figure 4). This becomes an instantiation of a generic gate model with an input pin array of size two and an output pin array of size one. Any reference to pin *A* is transformed to a reference to `input[0]`, pin *B* becomes `input[1]`, and pin *Z* becomes `output[0]`. Thus, the logical OR function defined by the Liberty description (`"A+B"`) is rewritten using the boolean operations available in C: `(input[0] || input[1])`.

4.2 Predefined LPs Types

Circuit design netlists are made up of a few components: gates, wires, and input and output connections (external to the current circuit). Each gate or "net" in a netlist becomes an LP during simulation (defined by the generic gate model). Straight wires (wires connecting two pins) do not become LPs and instead the direct connection information is stored within a gate's state. In contrast, wires which represent a fanout (where one output pin connects to two or more input pins) do become LPs. These *fanout* LPs, along with special *input* and *output* LPs are treated as generic gate types that are not defined by the Liberty gate library. Instead, they created by default during automatic model generation.

4.2.1 Input and Output LPs

Input LPs represent signals coming from other circuit components. In the example discussed in Section 5, these LPs randomly feed high and low signals into the circuit during simulation.

Output LPs represent signals exiting the current circuit module. Currently these LPs act as a sink for any incoming signals.

4.2.2 Fanout LPs

In most circuit designs, there is no fanin. That is, a single input pin on any gate is connected to only one wire. This is not the case for fanout as one output pin may connect many wires, each leading to separate input pins. These fanout wires are represented as LPs

```
void gates_event(gate_state *s, message *in_msg){

    if (in_msg->type == LOGIC_MSG) {
        int in_pin = in_msg->id;

        if (s->inputs[in_pin] == in_msg->value) return; // no change for input pins

        int rising = (s->inputs[in_pin] < in_msg->value);

        // save current state of input and internal pins in case of reverse event
        // we store old values in the current message, which is used during reverse
        SWAP(&(s->inputs[in_pin]), &(in_msg->value));
        if (gate_internal_size[s->gate_type] > 0) {
            in_msg->internal_pin0 = s->internals[0];
            in_msg->internal_pin1 = s->internals[1];
        }

        // perform logic operation
        logic_function_array[s->gate_type](s->inputs, s->internals, s->output_val);

        // send messages to my output pins
        for (int i = 0; i < gate_output_size[s->gate_type]; i++){
            // calculate the delay for outgoing signals
            float delay = delay_function_array[s->gate_type](in_pin, i, rising);
            // ...
}
```

Figure 2: Snippet of generic gate forward event handler.

```
void gates_event_rc(gate_state *s, message *in_msg){

    if (in_msg->type == LOGIC_MSG) {

        if (s->inputs[in_msg->id] == in_msg->value) return;

        // restore the state of input and internal pins
        SWAP(&(s->inputs[in_msg->id]), &(in_msg->value));
        if (gate_internal_size[s->gate_type] > 0) {
            s->internals[0] = in_msg->internal_pin0;
            s->internals[1] = in_msg->internal_pin1;
        }

        // No need to undo changes to output pins
        // ROSS handles event causality
    }
}
```

Figure 3: Snippet of generic gate reverse event handler.

which simply forward an incoming signal to all connected outputs. These LPs use an event delay which is smaller than the smallest gate delay in the current library.

5. ILLUSTRATIVE EXAMPLE

To understand the value of using automatic model generation, we compare an automatically generated model to a handwritten one. A comparison of the details of these two models can be seen in Table 1. Both models were used to simulate the crossbar switch (CCX) from the OpenSPARC T2 processor [38]. The simulations of the CCX models lasted for 3,000 simulation time units which is equivalent to 3,000 clock cycles for the handwritten model and 3000 ns for the automatically generated model. Both models use a similar round-robin style partitioning scheme. This scheme makes no attempt to group the LPs by communication patterns and leads to a very high percentage of messages set to an LP within another MPI rank.

Table 1: Detail comparison between a handwritten simplification of a GTECH model and an automatically generated timing-accurate LSI-10K model. These details are from the simulation of the crossbar switch module from an OpenSPARC T2 processor.

	Handwritten GTECH Simplified Model	Auto-Generated LSI-10K Model
Gate Count	211,001	200,981
Net Events	1.4 Billion	0.2 Billion
Time Unit	Clock Cycle	Nanosecond
Lookahead	0.4	0.009

The handwritten model used as the basis for comparison has been previously presented in [19]. This model was a dramatic simplification of the GTECH standard cell library provided by Synopsis [37]. While the GTECH library contains over 100 gates, the

```
cell(OR2) {
  pin(A) {
    direction : input;
  }
  pin(B) {
    direction : input;
  }
  pin(Z) {
    direction : output;
    function : "A+B";
    timing() {
      intrinsic_rise : 0.38;
      intrinsic_fall : 0.85;
      related_pin : "A";
    }
    timing() {
      intrinsic_rise : 0.38;
      intrinsic_fall : 0.85;
      related_pin : "B";
    }
  }
}
```

(a) LSI-10k Liberty code

```
void OR2_func
(int input[], int internal[], int output[]) {
  //Z : A+B
  output[0] = (input[0] || input[1]);
}

float OR2_delay_func
(int in_pin, int out_pin, int rising) {
  //['Z']
  if (out_pin == 0) {
    //['B']
    if ( in_pin == 1 ) {
      if (rising) return 0.38;
      if (!rising) return 0.85;
    }
    //['A']
    if ( in_pin == 0 ) {
      if (rising) return 0.38;
      if (!rising) return 0.85;
    }
  }
}
```

(b) The corresponding, automatically generated C code

Figure 4: Example of an OR logic gate defined in the LSI-10k Liberty library and the automatically generated functions used by the ROSS generic gate model. Note that one can do a line-by-line comparison to validate both the logic and timing information.

handwritten model consisted of an over decomposition of each logic gate into one of eight basic gates: *repeater*, *not*, *and*, *nand*, *or*, *nor*, *xor*, and *xnor*. This decomposition transformation was accomplished by a parser provided by Li et al. [29] and is written in Lex and Yacc [28].

Due to a lack of timing information for the GTECH library, a very simplistic timing model was used, with each logic gate having a uniform, single clock-cycle delay. This delay was achieved through the use of self-scheduled wakeup messages. These messages carried no logic, but contributed to a large event population for the simulation as a whole. This simplistic timing model also had an effect on the lookahead value that could be used for conservative simulation. With such a regularly timed model, lookaheads of 0.4 time units were used. This is quite large in comparison to the average event "in-flight" time of 0.5 time units.

For our automatically generated model, we use the LSI-10K library written in Liberty format [30]. This generated model is approximately 10,000 lines of code. This code includes the logic function for each gate type, as well as a function to calculate a gate's internal delay (see Figure 4). The generated C code can be compared on a line-by-line basis with the Liberty library description for verification of both the logic and timing information.

For any automatically generated model, the lookahead value used during conservative simulation must be derived from the timing values in the provided gate library. Within the LSI-10K model, the smallest defined timing delay is 0.01 ns. Thus we safely set the lookahead window at 0.009 ns for this model. The smaller lookahead window represents an increase of fidelity of the timing behavior model. In this respect, the automatically generated model is nearly 50 times more accurate than the handwritten model.

We have shown that the automatically generated model leads to an accurate representation of the gates being modeled and simulated. The next task is to determine any effects the automatically generated model on simulation performance. We address this by recreating and comparing results with an experiment described in [19], where a handwritten gate model was used.

5.1 Experimental Setup

The circuit design chosen was the CCX component of the OpenSPARC T2 processor. We start with the open source RTL description. This source was then synthesized into a gate-level netlist using the Synopsys Design Compiler and the selected gate library (either GTECH or LSI-10K).

The experiments with the handwritten model were performed on an IBM Blue Gene/L with cores operating at 700 MHz [19]. This supercomputer debuted in 2004 as the first of its class. It was designed around a relatively slow processor speed, allowing for a lower overall power consumption [18]. It also introduced a system-on-a-chip design, with all node components (including two processor cores) embedded in one chip. These nodes were connected to several global communication networks, including a separate network for global MPI communications (e.g., *all reduce*).

To create a similar experimental setup, the automatically generated model was run on an IBM Blue Gene/Q machine, with cores running at 1.6 GHz. This is the latest edition of the IBM supercomputer line. The Blue Gene/Q features 18 processor cores per chip [20]. 16 of the 18 cores are devoted to application use; one core is dedicated to operating system functionality; the final core is a spare, used when another core within the chip fails. While the Blue Gene/Q is capable of running up to four hardware threads per core, this work only uses one hardware thread per core (i.e., one MPI rank per core).

On both Blue Gene systems, ROSS benefits from the high-speed communication networks [7,21]. For these results and the previous results pertaining to the handwritten model, we use the terms *MPI rank* and *core* interchangeably.

5.2 Results

To understand the impact on overall simulation performance of an automatically generated model, we examine the relationship between the performance of optimistic and conservative simulation. Figure 5 shows the optimistic/conservation comparison during strong scaling for the handwritten model. Here, conservative performance continues to improve as parallelism increases. In contrast, as par-

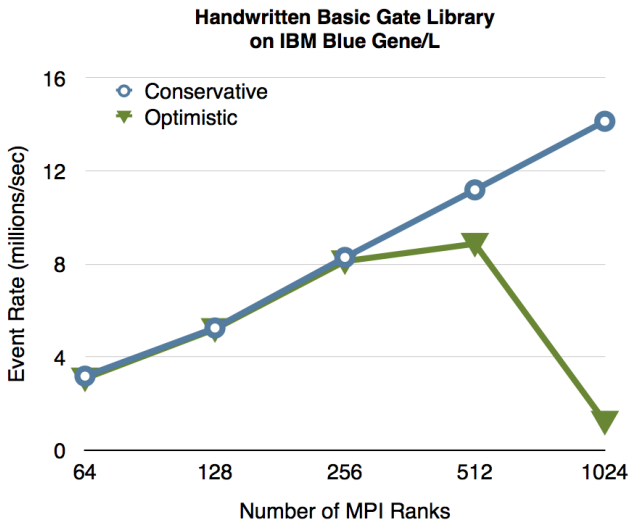

Figure 5: Single crossbar experiments on 700 MHz Blue Gene/L cores. Inputs are generated randomly every two simulation time units. This experiment was originally published in [19].

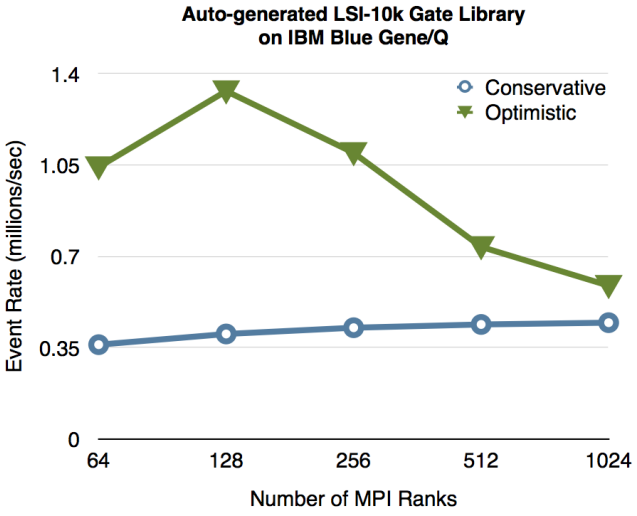

Figure 6: Single crossbar experiments on 1.6 GHz Blue Gene/Q cores. Inputs are generated randomly every two simulation time units. Note that the lower event rate (when compared to Figure 5) is due to a decrease in overall event population as well as an increase in event complexity.

allelism increases optimistic simulation never outperforms conservative and in fact becomes under decomposed and experiences a decrease in performance.

Optimistic simulation is usually expected to outperform conservative. This result was attributed to the over-simplification of the handwritten model. As concluded in [19]:

"The lookahead of 0.4 time units (in relation to the timestamp increment of 0.5 time units) is in the medium to large range, based on the performance study by Carothers and Perumalla [11]. From this study, we note that conservative synchronization outperforms optimistic synchronization for medium and large lookahead up to 8,192 cores. We observe the same phenomena here. Should the lookahead become smaller due to changes in the model or core count increase, we expect optimistic performance to improve and potentially overtake conservative performance."

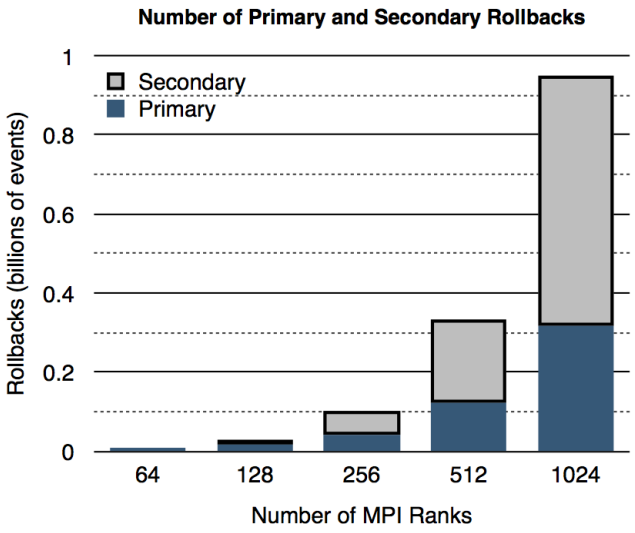

Figure 7: This graph shows the total number rollback events as a sum of the primary and secondary rollbacks. Note that there are total of 200 million net events in the simulation.

We can now demonstrate the observation that a smaller lookahead value will positively affect optimistic performance.

Figure 6 compares optimistic and conservative simulation performance for the automatically generated model. Here, optimistic synchronization outperforms conservative performance. Again, there is the same increase, then decrease in optimistic performance as number of MPI ranks grows. We can again contribute this to the fact that the overall model is small, on the order of 200,000 total LPs. With more than 128 MPI ranks, the model is under decomposed and the communication overhead of synchronization outweighs the total work done on any individual core.

A greater understanding of the under decomposition which occurred during the optimistic simulations can be gained through analyzing the rollbacks. Figure 7 shows that the total number of rollbacks as a sum of the primary and secondary rollbacks. It is important to remember that the number of net events in a full run of the simulation is 200 million. At 256 MPI ranks, optimistic simulation experiences approximately 100 million rollback events, on the order of one half the total number events which need to be processed. Put another way, for event two events which are processed, one of them is eventually rolled back. It is at this point (256 MPI ranks) that we begin to see a decrease in overall optimistic simulation performance.

We can further explore the relationship between primary and secondary rollbacks by analyzing them as a percentage of total rollbacks (seen in Figure 8). For the two experiments where optimistic simulation improves with parallelism (64 and 128 MPI ranks), less than half of the total rollbacks are secondary rollbacks. This means that most anti-messages were received before their counterpart forward message was processed by the receiving LP. The large portion of secondary rollbacks for the large experiments indicates that some MPI ranks within simulation are much farther ahead in time than others.

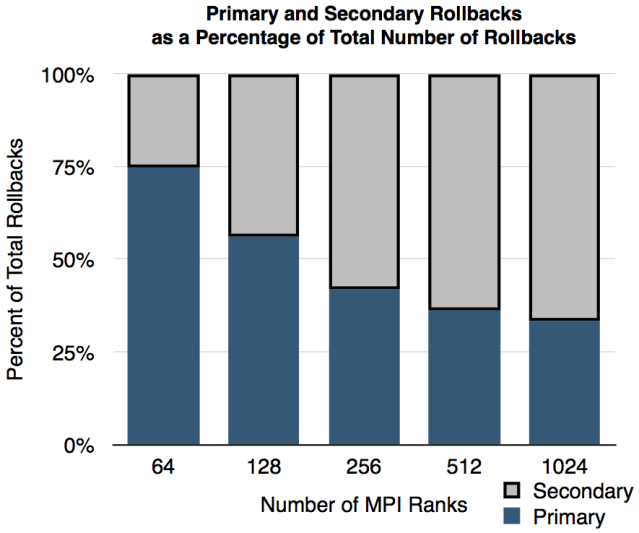

Figure 8: This graph shows the percentage breakdown of the total number of rollback events in terms of primary and secondary rollbacks. The large percentage of secondary rollbacks indicate a large disparity of virtual time across different MPI Ranks.

6. RELATED WORK

Automatic model generation is not a new area of study. The idea of predefining a conceptual (or generic) model shows up in many fields where simulation is an important part of a larger workflow. One such example comes from supply chain management [42]. This work identifies the key elements of the supply chain and how they relate to each other. With the generic definitions in place, it is possible to simulate the specifics of any number of configurations.

For decades, automatic model processing has been a part of the circuit design process (refer to Figure 1). This includes everything from generating behavior models within proprietary systems [25] to generating the schematic diagrams [1] to extracting a gate model from an existing transistor design [27]. Automatically generating gate level models for sequential circuit simulation has not been extensively studied academically, but is a highly patentable area (most recently in [15]).

Parallel digital logic simulation is a highly investigated area of research. This includes several investigations of the merits of both conservative and optimistic synchronization algorithms [3, 4, 14, 32]. The findings are usually mixed as both methods have their merits and either one may be more suitable for a particular circuit model. More recently, research has focused on improving the performance of optimistic simulation through a bottom-up implementation of a digital logic-specific simulation framework [2, 29, 33, 43].

7. CONCLUSIONS

In a typical integrated circuit design workflow, gate-level simulation is done using proprietary, domain-specific simulation software. By creating a generic model and the tools required for parsing domain-specific descriptions, we have enabled substantial increases in modeler efficiency through the use of a high performing PDES implementation. The automatic model generation tool allows for any gate model written in Liberty to be realized in the ROSS framework. This flexibility will allow for an increase in usefulness of simulation within the circuit design and verification cycle.

In this paper we have described a generic model for gate-level circuit simulation within a conservative and reverse-computation-based optimistic simulation engine. This generic model is coupled with an automatic system which can generate model details from a Liberty-formated gate library. The accuracy of the automatically generated model can be validated through a side-by-side comparison of the original Liberty description and the resulting C code. This model also allows for greater precision in the model of timing behavior, which is evident in lookahead value used by conservative simulation.

To investigate the performance impact of the automatically generated model we compared the relationship of optimistic and conservative simulation performance to an existing set of results which used a handwritten model. The experiments conducted for this paper were preformed on an IBM Blue Gene/Q and represent some of the first PDES results on this machine. We found that the increased accuracy of model timing (and decrease in lookahead window size) contributed to improved performance of optimistic simulation over conservative simulation. We observed that when the high-fidelity model was executed on 128 MPI ranks, optimistic outperformed conservative simulation by a factor of 3.3. As the number of MPI ranks increased, to a maximum of 1024 ranks, optimistic did not drastically outperform conservative simulation due to a substantial increase in primary and secondary rollbacks.

8. ACKNOWLEDGMENTS

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-11-2-0065. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory or the U.S. Government.

9. REFERENCES

[1] A. Arya, V. Swaminathan, A. Misra, and A. Kumar. Automatic Generation Of Digital System Schematic Diagrams. In *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, DAC '85, pages 388–395, Las Vegas, NV, USA, 23–26 June 1985. IEEE Computer Society.

[2] H. Avril and C. Tropper. Clustered time warp and logic simulation. In *Proceedings of the Ninth Workshop on Parallel and Distributed Simulation*, PADS '95, pages 112–119, Lake Placid, New York, USA, 13–16 June 1995. IEEE Computer Society.

[3] R. Bagrodia, Y.-a. Chen, V. Jha, and N. Sonpar. Parallel Gate-level Circuit Simulation on Shared Memory Architectures. *ACM SIGSIM Simulation Digest*, 25(1):170–174, 1995.

[4] R. Bagrodia, Z. Li, V. Jha, Y. Chen, and J. Cong. Parallel Logic Level Simulation of VLSI Circuits. In *Proceedings of the 26th Conference on Winter Simulation*, WSC '94, pages 1354–1361, Orlando, FL, USA, 11–14 Dec. 1994. Society for Computer Simulation International.

[5] M. L. Bailey, J. V. Briner, Jr., and R. D. Chamberlain. Parallel Logic Simulation of VLSI Systems. *ACM Computing Surveys*, 26(3):255–294, Sept. 1994.

[6] P. D. Barnes, Jr., C. D. Carothers, D. R. Jefferson, and J. M. LaPre. Warp Speed: Executing Time Warp on 1,966,080

Cores. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS '13, pages 327–336, Montreal, Canada, 19–22 May 2013. ACM.

[7] D. Bauer, C. Carothers, and A. Holder. Scalable Time Warp on Blue Gene Supercomputers. In *Proceedings of the 23rd ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, PADS '09, pages 35–44, Lake Placid, NY, USA, 22–25 June 2009. IEEE Computer Society.

[8] D. Beazley. PLY: Python Lex-Yacc. *http://www.dabeaz.com/ply/index.html*, 2000–2011. Last access 11 April 2015.

[9] J. V. Briner, Jr., J. L. Ellis, and G. Kedem. Breaking the Barrier of Parallel Simulation of Digital Systems. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, DAC '91, pages 223–226, San Francisco, California, USA, 17–21 July 1991. ACM.

[10] C. Carothers, D. Bauer, and S. Pearce. ROSS: A High-Performance, Low Memory, Modular Time Warp System. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, PADS '00, pages 53–60, Bologna, Italy, 28–31 May 2000. IEEE Computer Society.

[11] C. Carothers and K. Perumalla. On Deciding Between Conservative and Optimistic Approaches on Massively Parallel Platforms. In *Proceedings of the 42nd Conference on Winter Simulation*, WSC '10, pages 678–687, Baltimore, MD, USA, 5–8 Dec. 2010.

[12] C. Carothers, K. Perumalla, and R. Fujimoto. Efficient Optimistic Parallel Simulations Using Reverse Computation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, PADS '99, pages 126–135, Atlanta, GA, USA, 1–4 May 1999. IEEE Computer Society.

[13] R. D. Chamberlain. Parallel Logic Simulation of VLSI Systems. In *Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference*, DAC '95, pages 139–143, San Francisco, California, USA, 12–16 June 1995. ACM.

[14] Y.-a. Chen, V. Jha, and R. Bagrodia. A Multidimensional Study on the Feasibility of Parallel Switch-level Circuit Simulation. *ACM SIGSIM Simulation Digest*, 27(1):46–54, June 1997.

[15] S. C. Cismas, K. J. Monsen, and H. K. So. *Automatic Code Generation for Integrated Circuit Design*, Feb. 7 2006. US Patent 6,996,799.

[16] A. Costa, A. de Gloria, P. Faraboschi, and M. Olivieri. An Evaluation System for Distributed-time VHDL Simulation. In *Proceedings of the Eighth Workshop on Parallel and Distributed Simulation*, PADS '94, pages 147–150, Edinburgh, Scotland, United Kingdom, 6–8 July 1994. ACM.

[17] R. M. Fujimoto. *Parallel and Distributed Simulation Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

[18] A. Gara, M. A. Blumrich, D. Chen, G.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, et al. Overview of the Blue Gene/L System Architecture. *IBM Journal of Research and Development*, 49(2.3):195–212, 2005.

[19] E. Gonsiorowski, C. Carothers, and C. Tropper. Modeling Large Scale Circuits Using Massively Parallel Discrete-Event Simulation. In *Proceedings of the 20th IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems*, MASCOTS '12, pages 127–133, Arlington, VA, USA, 7–9 Aug. 2012.

[20] R. A. Haring, M. Ohmacht, T. W. Fox, M. K. Gschwind, D. L. Satterfield, K. Sugavanam, P. W. Coteus, P. Heidelberger, M. A. Blumrich, R. W. Wisniewski, et al. The IBM Blue Gene/Q Compute Chip. *Micro, IEEE*, 32(2):48–60, 2012.

[21] A. O. Holder and C. D. Carothers. Analysis of Time Warp on a 32,768 Processor IBM Blue Gene/L Supercomputer. In *Proceedings of the 20th European Modeling and Simulation Symposium*, EMSS '08, pages 284–292, Amantea, Italy, 17–19 Sept. 2008. CAL-TEK SRL.

[22] K. hui Chang and C. Browy. Parallel Logic Simulation: Myth or Reality? *Computer*, 45(4):67–73, April 2012.

[23] IEEE Standard for Verilog Hardware Description Language. *IEEE Standard 1364-2005 (Revision of IEEE Standard 1364-2001)*. *http://dx.doi.org/10.1109/IEEESTD.2006.99495*, 2006. Last access 12 May 2015.

[24] IEEE Standard for Verilog Register Transfer Level Synthesis. *IEEE Standard 1364.1-2002*. *http://dx.doi.org/10.1109/IEEESTD.2002.94220*, 2002. Last access 12 May 2015.

[25] R. G. Ingalls. Automatic Model Generation. In *Proceedings of the 18th conference on Winter simulation*, WSC '86, pages 677–685, Washington, DC, USA, 8–10 Dec. 1986. ACM.

[26] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[27] S. Kundu. Gatemaker: A Transistor to Gate Level Model Extractor for Simulation, Automatic Test Pattern Generation and Verification. In *Proceedings of the 1998 International Test Conference*, ITC '98, pages 372–381, Washington, DC, USA, 18–23 Oct. 1998. IEEE Computer Society.

[28] J. Levine, T. Mason, and D. Brown. *Lex & Yacc*. A Nutshell Handbook. O'Reilly & Associates, Sebastopol, CA, USA, 1992.

[29] L. Li, H. Huang, and C. Tropper. DVS: An Object-Oriented Framework for Distributed Verilog Simulation. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, PADS '03, pages 173–180, San Diego, CA, USA, 10–13 June 2003. IEEE Computer Society.

[30] Liberty, Version 2009.06. *IEEE-ISTO*. *http://www.opensourceliberty.org*, September 2009. Last access 12 May 2015.

[31] Y.-B. Lin and E. D. Lazowska. A Study of Time Warp Rollback Mechanisms. *ACM Transactions on Modeling and Computer Simulation*, 1(1):51–72, Jan. 1991.

[32] D. Lungeanu and C.-J. R. Shi. Distributed Simulation of VLSI Systems Via Lookahead-Free Self-Adaptive Optimistic and Conservative Synchronization. In *Proceedings of the 1999 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '99, pages 500–504, San Jose, CA, USA, 7–11 Nov. 1999. IEEE Computer Society.

[33] S. Meraji, W. Zhang, and C. Tropper. On the Scalability of Parallel Verilog Simulation. In *Proceedings of the 38th International Conference on Parallel Processing*, ICPP '09, pages 365–370, Vienna, Austria, 22–25 Sept. 2009. IEEE Computer Society.

[34] R. Mueller-Thuns, D. Saab, R. Damiano, and J. Abraham. VLSI Logic and Fault Simulation on General-Purpose Parallel Computers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(3):446–460, Mar 1993.

[35] D. Nicol. The Cost of Conservative Synchronization in Parallel Discrete Event Simulations. *Journal of the ACM*, 40(2):304–333, 1993.

[36] D. Nicol and P. Heidelberger. Parallel Execution for Serial Simulators. *ACM Transactions on Modeling and Computer Simulation*, 6(3):210–242, July 1996.

[37] S. Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesis*. Prentice Hall Press, Upper Saddle River, NJ, USA, second edition, 2003.

[38] I. Parulkar, A. Wood, S. Microsystems, and S. Mitra. OpenSPARC : An Open Platform for Hardware Reliability Experimentation. In *Proceedings of the Fourth Workshop on Silicon Errors in Logic System Effects*, SELSE '08, Austin, TX, USA, 26–27 Mar. 2008. IEEE Computer Society.

[39] S. Robinson. *Simulation: The Practice of Model Development and Use*. John Wiley & Sons, Hoboken, NY, USA, 2004.

[40] L. Soulé and A. Gupta. An Evaluation of the Chandy-Misra-Bryant Algorithm for Digital Logic Simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(4):308–347, Oct. 1991.

[41] W.-k. Su and C. L. Seitz. *Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm*. Technical report, California Institute of Technology, Pasadena, CA, USA, 1988.

[42] G. E. Vieira and O. C. Júnior. A Conceptual Model for the Creation of Supply Chain Simulation Models. In *Proceedings of the 37th conference on Winter Simulation*, WSC '05, pages 9–pp, Orlando, FL, USA, 4–7 Dec. 2005. IEEE Computer Society.

[43] Q. Xu and C. Tropper. XTW, A Parallel and Distributed Logic Simulator. In *Proceedings of the 19th ACM/IEEE/SCS Workshop on Parallel and Distributed Simulation*, PADS '05, pages 181–188, Kufstein, Austria, 1–3 June 2005. ACM.