Fast Simulation and Equivalence Checking Using OAGear

Kai-hui Chang, David A. Papa, Igor L. Markov and Valeria Bertacco

CSE Division, University of Michigan, Ann Arbor, MI 48109-2121
{changkh, iamyou, imarkov, valeria}@umich.edu

ABSTRACT

The OpenAccess Gear package addresses several key tasks in synthesis, verification and layout of digital logic, and encourages synergies between such optimizations. Primitives for logic simulation and equivalence-checking are particularly useful in both verification and logic synthesis, as exemplified by the AIG algorithms implemented in OAGear.

We describe our re-design of simulation and equivalence-checking engines in OpenAccess Gear, and our empirical results show that our simulator runs 100 times faster on large netlists than the current implementation. To ensure a broad adoption of these core engines in the user community, we provided adequate GUI support using OAGear standard user interface.

1. INTRODUCTION

Combinational equivalence-checking is important in both formal verification and logic synthesis, where merging equivalent nodes can reduce circuit area. Equivalence proofs, usually performed by a SAT solver, are expensive. However, disproving equivalence can be much easier — counterexamples selected from logic simulation runs on random inputs often distinguish most pairs of candidate signals [5]; faster simulation allows one to avoid more SAT calls. Additionally, if assertions are falsified during sequential simulation, the need for expensive bounded model checking is reduced.

The current equivalence-checker in OAGear is based on And-Inverter Graphs (AIGs) [3]. While powerful and easy to use, that checker has limited flexibility because (1) building an AIG for a circuit may be costly, and (2) no counterexamples are returned by failed checks. To address these limitations, we implemented a lightweight equivalence checker based on random simulation and Satisfiability (CNF-SAT). We also improved the interface, allowing one to request counterexamples and obtain additional information.

After realizing that the OAGear simulator scaled poorly, we implemented our own simulator based on oblivious and event-driven algorithms, which sped up simulation by 100 times in some cases. Our Graphical User Interface (GUI) for equivalence-checking and simulation enhances the OAGear Bazaar package, allowing the user to (1) conduct random simulation, (2) specify input patterns, (3) view simulation results, (4) check equivalence of two given signals, and view a counterexample if the check fails, (5) check the equivalence of two circuits and view counterexamples. With this GUI, circuit debugging becomes much easier.

The rest of the paper is organized as follows. Section 2 describes our algorithms for equivalence checking and simulation. Section 3 illustrates the new enhancements to the user interface. Software engineering concerns are addressed in Section 4 and empirical results are shown in Section 5.

2. EQUIVALENCE CHECKER

Our equivalence checker first uses random simulation to quickly detect signals that are not equivalent. For the signals that cannot be distinguished by random simulation, SAT-based equivalence checking is used, and counterexamples found during SAT-solving are reused as additional simulation patterns to distinguish more signals [5]. Our implementation and its interface also support incremental verification, as explained below.

Simulation Algorithms: our simulator first extracts logic information (an AIG) for each cell used in the design from OAGear's Func package. Next, we simulate all possible input combinations of each cell to construct its truth-table. Using such look-up tables during simulation is far more efficient than traversing AIGs of individual cells. To further improve speed, our simulator employs bit-parallel simulation (32 or 64 patterns simulated at once depending on the definition of `SimulationVector`) and treats most common gate types as special cases. In order to efficiently simulate patterns with different event activity, we implemented an oblivious algorithm as well as an event-driven algorithm [4].

SAT-based Equivalence Checking: our equivalence checker first generates the CNF of every cell in the cell library. This is accomplished by traversing the AIG of each cell and converting the ANDs and INVERTERs to their corresponding circuit-CNFs. Next, we build a miter for the signals to be checked for equivalence and convert it to CNF. A miter is a circuit consisting of an XOR gate combining the signals and their fanin cones with depth such that the inputs to each cone are the same. We set the output of the miter to 1 and use MiniSAT [2] to determine satisfiability. If the CNF is not satisfiable, the signals are equivalent, alternatively, a counterexample is returned by the SAT solver. We employ a simple interface to a SAT-solver so that MiniSAT can be easily replaced and CNF conversion can be improved. This interface allows one to (1) add a clause, (2) add multiple clauses, (3) solve the CNF, (4) check whether the CNF is satisfiable, and (5) obtain the value of any literal in a satisfying solution. The user can adjust the number of initial patterns used by random simulation. Setting that number to 0 turns off random simulation and resorts to SAT-based equivalence checking.

Incremental Verification: our equivalence checker is suitable for incremental verification in that we provide an interface to perform equivalence checking on a portion of the design. The user can specify two sets of gates, connect their corresponding inputs and outputs, and perform equivalence checking between them. This is especially useful when small changes to the netlist must be verified.

We define an estimate of the similarity between two netlists, ckt_1 and ckt_2, that utilizes fast simulation, called the *similarity factor*. This metric is based on simulation signatures of individual signals, i.e. the k-bit sequences holding signal values computed by simulation on each of k input patterns (e.g., k=1024). Let N be the total number of signals (wires) used by the two circuits. Out of those N signals, we distinguish M *matching* signals — a signal is considered matching if and only if both circuits include signals with an identical signature. The similarity factor between ckt_1 and ckt_2 is then M/N. This metric is computed very quickly using our fast simulation tool and returned to the user.

Intuitively, the similarity factor of two identical circuits should

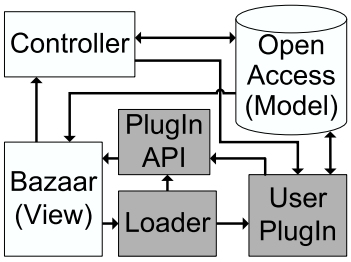

Figure 1: Design of plug-in interface incorporated into Bazaar.

be 1. If a circuit is changed slightly but is still equivalent to the original version, then its similarity factor should drop only slightly. However, if the change breaks the equivalence, the similarity factor can drop significantly, depending on the number of signals affected by the change. The new similarity metric relies on fast simulation but not on SAT solvers, and therefore can be computed very quickly. However, two equivalent circuits may be dissimilar, e.g., a Carry-Look-Ahead adder and a Kogge-Stone adder.

3. IMPROVED USER INTERFACE

The OAGear graphical user interface, Bazaar, now supports the following use-cases, (1) conducting random simulation, (2) specifying input patterns, (3) viewing simulation results, (4) checking equivalence of two given signals, and viewing a counterexample if the check fails and (5) checking the equivalence of two circuits and viewing counterexamples.

To minimize the impact of integrating our tools into Bazaar, we created a plug-in that can be loaded optionally by the user at runtime. Previously, Bazaar had no interface for loading plug-ins, therefore we developed such an interface. Figure 1 shows the overall design of Bazaar with its new plug-in interface shaded.

Bazaar adheres to the Model-View-Controller design pattern [1], where an integrated circuit is represented in an underlying data model (OpenAccess) and can be viewed and modified using a graphical user interface (Bazaar). To facilitate dynamic loading of plug-ins, we added two additional components to the design above, a plug-in API object (piAPI) and a loader object.

When the user loads a plug-in into Bazaar, the loader creates a piAPI object and supplies it with a pointer to Bazaar. A user-provided function `MyPlugIn::load(auto_ptr<piAPI>)` is then called by the loader. This function takes possession of the piAPI object and creates menus, toolbars, windows, and commands in Bazaar. Thus, we decouple Bazaar from plug-ins since the loader prevents Bazaar from accessing the piAPI object, and the plug-in can only access Bazaar through the provided API. This allows Bazaar to safely load various configurations of plug-ins on demand.

4. SOFTWARE ENGINEERING DETAILS

Our simulator and equivalence checker are both implemented natively in OpenAccess and follow the OAGear coding standards. Our new software includes documentation and is supplied with regression tests, just as other OAGear packages. The algorithms we employ scale at least as well as the existing OAGear algorithms.

Additionally, we provide a variety of convenient ways to use these tools including standalone binaries, point-and-click use-cases in Bazaar, and a library API. To encourage adoption of our new tools, their interfaces are designed to be compatible with the existing versions. Our package has some limitations, specifically (1) it only supports the datamodel of OA's block domain as opposed to the module or occurrence domains and (2) the netlist must be mapped to a cell library.

Benchmark	Gate count	Simulator runtime (sec)			EQcheck runtime (sec)
		OAGear	Our simulators		
		orig.	custom	native	
s27	19	9.8e0	0.4	0.4	0.3
s344	132	4.0e1	0.4	0.7	0.6
s1196	483	9.5e1	0.5	1.7	1.6
s15850	685	5.0e3	0.6	2.0	1.8
s9234_1	974	2.4e3	0.7	2.9	2.8
s13207	1218	1.7e4	0.8	3.3	3.4
s38417	8278	3.0e5	2.0	21.0	1.4e2
vga_lcd	124031	time-out	20.2	3.3e2	2.5e4

Table 1: A runtime comparison among the simulator included in OAGear, our simulator using custom data structures, and our native simulator. 3200 random patterns were considered for each benchmark, and time-out was set to 1 week. Runtimes of our equivalence checking tool are also reported. The comparisons show that our simulator outperforms the OAGear simulator by far on all benchmarks.

5. EXPERIMENTAL RESULTS

We compared the performance of our simulator with the one included in OAGear by simulating 3200 random vectors with each tool and measuring runtime. We also handcrafted an optimized simulator that runs on OpenAccess and employs the same algorithms but uses custom data structures, and present it for comparison. Empirical results in Table 1 show that both of our simulators perform asymptotically better than the one currently in OAGear; our runtime grows linearly with respect to gate count, whereas OAGear's current implementation appears to grow exponentially. Our handcrafted version optimizes netlist traversal by merging equivalent nets and avoids a hash look-up by storing simulation values in the wire object. In our OA-based code, we attempted to avoid this hash look-up by using `oaAppDef`, but that was too slow.

The existing simulator in OAGear would take several days to perform the aforementioned task on the modestly sized benchmark s38417, versus 20 seconds taken by our new simulator. Clearly the existing tool takes an impractical amount of runtime and simply does not scale to realistic circuit sizes. In addition to simulation, we report the runtime of our SAT-based equivalence checker in Table 1. We arranged for all equivalence checks in this experiment to be successful since completing such instances usually takes longer.

Conclusions: we have identified a major component of OAGear with poor scalability — the logic simulation engine. Our new implementation uses different algorithms and reduces runtime by up to 100 times in our experiments. We also evaluated five use-cases and extended OAGear's graphical user interface accordingly. Applying fast logic simulation, we defined a new metric of circuit similarity that can be useful in incremental verification and debugging.

6. REFERENCES

[1] S. Burbeck, "Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller", http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

[2] N. Eén and N. Sörensson, "An Extensible SAT-solver", *Theory and Applications of Satisfiability Testing, SAT*, 2003, pp. 502-518.

[3] A. Kuehlmann, V. Paruthi, F. Krohm and M. K. Ganai, "Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification," *IEEE Trans. CAD*, Vol. 21(12), 2002, pp. 1377-1394.

[4] D. M. Lewis, "A Hierarchical Compiled-Code Event-Driven Logic Simulator", *IEEE Transactions on Computer-Aided Design*, Jul. 1987, pp.601-617.

[5] J. Zhang, S. Sinha, A. Mishchenko, R. Brayton, and M. Chrzanowska-Jeske, "Simulation and Satisfiability in Logic Synthesis", *IWLS* 2005, pp. 161-168.