Accelerating Online Model Checking

Mona Qanadilo
Computer Engineering Department
An Najah National University
Nablus, Palestine

Sufyan Samara
Computer Engineering Department
An Najah National University
Nablus, Palestine

Yuhong Zhao
Heinz Nixdorf Institute
Paderborn University
Paderborn, Germany

***Abstract*—Online model checking is a lightweight verification technique to ensure at runtime the safety of the current execution trace of the system application under test. Doing model checking online suffers from the limited execution time allocated to each checking cycle. In this paper, we focus on accelerating online model checking so that as large the model space as possible can be explored in time. For this purpose, we introduce offline backward exploration so as to reduce the workload of online forward exploration. As a result, online model checking becomes online reachability checking. SAT solver is used as verification engine for online model checking. We improve the performance of the SAT solver zChaff by optimizing and customizing zChaff with respect to the online model checking specific features. Moreover, we take advantage of the parallel feature and the multi-port memory available on FPGA chips. We present a new underlying architecture using 2 SAT solvers as verification engine for online model checking. We implement a quick prototype of the new underlying architecture for online model checking. Several experiments are done to test the performance of our online model checking.**

***Index Terms*—Online model checking, Bounded Model Checking, SAT Solver, FPGA.**

I. Introduction

Driven by the demand for more functionality, the complexity of software and hardware design continues to increase. Clearly, the need for reliable software systems make formal verification techniques paramount important in order to ensure the overall product quality. For instance, the vehicles' electronics systems are usually controlled by software with millions of lines. For such kind of complex software in safety-critical systems subtle errors are extremely difficult to reproduce in a laboratory environment. David Cummings described one such subtle error they encountered in the flight software for NASA's Mars Pathfinder spacecraft [1]:

"Because of Pathfinder's high reliability requirements and the probability of unpredictable hardware errors due to the increased radiation effects in space, we adopted a highly 'defensive' programming style. This included performing extensive error checks in the software to detect the possible side effects of radiation-induced hardware glitches and certain software bugs. One member of our team, Steve Stolper, had a simple arithmetic computation in his software that was guaranteed to produce an even result (2, 4, 6 and so on) if the computer was working correctly. Many programmers would not bother to check the result of such a simple computation. Stolper, however, put in an explicit test to see if the result was even. We referred to this test as his"two-plus-two-equals-five check." We never expected to see it fail. Lo and behold, during software testing we saw Stolper's error message indicating the check had failed. We saw it just once. We were never able to reproduce the failure, despite repeated attempts over many thousands if not millions of iterations."

For large complex systems, it becomes difficult for traditional model checking [2] to store or explore the state space of the system model with reasonable time and memory consumption. This is the so called state space explosion problem. To overcome this problem, many efficient techniques have been presented in the literature so far: partial order reduction [3], compositional reasoning [4], abstraction technique [5], bounded model checking [6], to name just a few. These improvements enable model checking to verify more complex systems at the cost of making the checking process to some extent complicated.

Traditional testing [7] provides a partial proof of correctness at system implementation level. For untested inputs, undiscovered errors in deep corner might show up during system execution.

Runtime Verification ([8], [9], [10], [11], [12], [13], [14]) works also at system implementation level. It aims to check the correctness of the sequence of states monitored or derived from the current execution trace. Runtime verification can proceed further only after a new state is observed. Therefore, it is hard to detect errors before they have already been occurred.

Online Model Checking [15] works at the system implementation level, too, but it checks the correctness of the corresponding partial system model. Errors at the model level might indicate potential errors at the implementation level. Simply speaking, online model checking is a lightweight verification technique to ensure at runtime the correctness of the current execution trace of the system application under test by checking a sequence of the partial model space covering the execution trace. For this purpose, we need to monitor the system execution trace from time to time. The observed runtime state information is used to locate the partial state space being explored. In this way, we can avoid the state space explosion problem. Online model checking aims to "look" into a near future in the state space of the system model to see whether there exist potential errors or not. As side effect, the conformance of the implementation to the corresponding model can also be checked in the meantime. The counterexample provided by online model checking is a clue to locate the error(s), which might be in a deep corner of

978-0-7695-4962-0/13 $26.00 © 2013 IEEE
DOI 10.1109/LADC.2013.20

IEEE computer society

the model space and thus hard to reproduce.

Online model checking is in essence a Bounded Model Checking (BMC) problem [6] but applied at runtime. Model checking with SAT solver is currently enjoying a substantial success in several industrial fields and opens up new research directions [16]. Using SAT solver as verification engine has proved to be efficient and scalable for checking both hardware and software systems. Therefore, we use SAT solver as verification engine for online model checking.

The performance of online model checking depends on the checking problem, the searching method and the underlying hardware architecture. In this paper, we focus on accelerating online model checking from this 3 aspects so that as large the model space as possible can be explored in time. To do this, we introduce offline backward exploration so as to reduce the workload of online forward exploration. As a result, online model checking becomes online reachability checking, a simple form of bounded model checking. We exploit the online model checking specific features to improve the performance of the SAT solver zChaff [17]. Nowadays FPGA is commonly found in realtime and critical embedded systems. We take advantage of the parallel feature and the multi-port memory available on FPGA chips. We present a new underlying architecture using 2 SAT solvers working in parallel on two soft-core processors implemented in FPGA. A quick prototype is implemented to test the performance of our online model checking.

The remainder of the paper is organized as follows: Section II introduces the basic idea of online model checking; Section III explains the speed-up techniques for online model checking from three aspects: reducing workload, speeding up SAT solver, and using parallel computing; Section IV reports the experimental results done on the quick prototype to test the performance of our online model checking; Section V points out some related work; Finally, we conclude our work in Section VI.

II. Online Model Checking

To make online model checking available, we need to know of course the model of the system application under test as well as the property to be checked. The model can be obtained at design time or extracted from the implementation of the system application. The property can be any invariant or linear temporal logic formula, from which we can always derive in advance an unsafe region, i.e., a set of error states. In addition, we suppose that the execution trace of the system application can be monitored in some way from time to time.

Fig. 1 illustrates the basic idea [15] of our online model checking mechanism. Simply speaking, at each checking cycle, whenever a new current state s_i is monitored during the system execution, we can use the corresponding abstract state $\widehat{s_i} = \alpha(s_i)$ to reduce the state space to be explored by online model checking, where function $\alpha(\cdot)$ maps each concrete state s_i at the implementation level to the corresponding abstract state $\widehat{s_i}$ at the model level. At runtime, we can explore a partial state space of the system model starting from the abstract state $\widehat{s_i}$. Our goal is to figure out whether there exists a(n) (error) path starting from some state $\widehat{s_i}$ to the unsafe region, i.e. a set of error states. It is worth mentioning, if no abstract state is consistent with $\alpha(s_i)$, it means that the implementation of the application does not conform to its model. This consistency checking is a by-product of online model checking. Because of the limited checking time allocated to online model checking, at each checking cycle only finite transition steps, say the next k steps, starting from the observed state might be explored. Therefore, online model checking is in essence Bounded Model Checking (BMC) [6] applied at runtime.

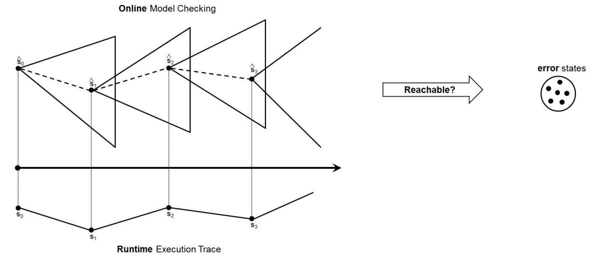

Fig. 1: Online Model Checking

SAT solver can be used as verification engine for online model checking. In case of a relatively small k, [6] concludes that "SAT based BMC is typically faster in finding bugs compared to BDDs". Unfortunately, [6] also concludes that "The deeper the bug is, the less advantage BMC has." However, by doing BMC at runtime it is quite possible to find deep corner bugs (if any) in the state space of a large complex system. If no error is detected, then the execution trace is safe at least within the next k steps. Once an error is detected, online model checking will inform the underlying (real-time) operating system in time. It is up to the operating system to decide how to deal with this case.

III. Speed-up Techniques for Online Model Checking

It is easy to see that online model checking just uses the current state observed from the system execution to locate its starting point at model level (Fig. 1), afterwards, it goes to explore a partial state space of the model from this starting point. This is independent of the system execution, which means that it might run ahead of or fall behind the system execution. In the former case it is possible to predict subtle errors before they've really happened, which is the goal of our online model checking. For this purpose, we'd like to speed up online model checking by reducing workload, speeding up SAT solver and using parallel computing.

A. Reduce Online Model Checking to Reachability Checking

We can reduce the workload of online model checking by introducing offline backward exploration as shown in Fig. 2. To do this, we need to deduce in advance a set of unsafe states, called (initial) unsafe region, given the model of the system application and the property to be checked.

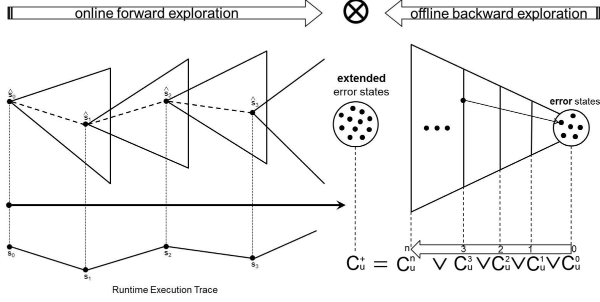

Fig. 2: Speed up Online Model Checking

Without loss of generality, let $M = (V_M, D_M, R_M, I_M)$ be a system model to be checked, where

- $V_M = \{v_1 : D_1, v_2 : D_2, \cdots, v_n : D_n\}$ is the set of system variables $v_1, v_2, \cdots, v_n$ associated with domains $D_1, D_2, \cdots, D_n$ respectively;
- $D_M = D_1 \times D_2 \times \cdots \times D_n$ is the state space of M;
- $R_M \subseteq D_M \times D_M$ is the transition relation of M;
- $I_M \subseteq D_M$ is the initial condition of M.

A state s of M is just a valuation for the variables in V_M, i.e., $s : V_M \to D_M$. In the state space D_M, there is an edge between two states s and s', if $R_M(s, s')$ holds. For the sake of convenience, we suppose the transition relation R_M to be *total*, for every state $s \in D_M$ there exists a state $s' \in D_M$ such that $R_M(s, s')$ holds. In fact, we can always make R_M total by adding an auxiliary edge to each state s without successors so that $R_M(s, s)$ holds. A path ρ of M from a state s is an infinite sequence of states $\rho = s_0, s_1, s_2, \cdots$ such that $s_0 = s$ and $R_M(s_t, s_{t+1})$ holds for all $t \geq 0$. Let S_0 be the set of initial states of M that satisfy the initial condition I_M, i.e., $S_0 = \{s \in D_M \mid s \models I_M\}$. A state s_n is reachable, if there exist a path $s_0, s_1, \cdots, s_n$ in M such that $s_0 \in S_0$. Notice that not all of the states in D_M are reachable from the given initial states. Let S be the set of reachable states of M. It is worth mentioning that for large complex systems, especially parallel systems, it is usually difficult to identify all the reachable states of M. Online model checking can avoid searching the whole reachable states of M.

Let's first consider a simple case, i.e., online invariant checking. Let f be an invariant to be checked. Then $\neg f$ is an *unsafe* condition of M. That is, a state $s \in D_M$ is *unsafe* state if $s \models \neg f$. Let $S_u = \{s \in D_M \mid s \models \neg f\}$ be the set of unsafe states in M with respect to f. Offline we can calculate a backward reachable set S_u^+ of those states starting from the unsafe states in S_u up to n steps. We say S_u^+ is an extended set of unsafe states in M with respect to f. We do not need to calculate S_u^+ explicitly. Instead, we can deduce an extended unsafe condition C_u^+ from $C_u^0 = \neg f$ such that $S_u^+ = \{s \in D_M \mid s \models C_u^+\}$ in the following way:

$$C_u^+ = \bigvee_{t=0}^{n} C_u^t \text{ where } C_u^t = \exists s'.s' \models C_u^{t-1} \wedge R_M(s, s') \text{ and } C_u^0 = \neg f.$$

A set of states is *reachable* if at least one state in the set is reachable. Now we need to know whether the extended set S_u^+ of unsafe states is reachable or not? To put it in another way, whether the extended unsafe condition C_u^+ holds or not? We answer this question by online checking starting from each specific state $\widehat{s_i}$ obtained (while the system application is running) whether some unsafe state in S_u^+ is reachable within k transition steps or not? The whole checking process is illustrated in Fig. 2. That is, we can speed up online model checking by introducing offline backward exploration starting from the (initial) unsafe states. As a result, online model checking is reduced into online reachability analysis. Obviously, whenever the system application runs closely enough to the extended unsafe region, we have more chance to detect the errors before they have really happened.

In the general case, let f be a non-trivial LTL formula. We can always transform the negation of f into a normal *Büchi* automaton $B_{\neg f} = (V_B, D_B, R_B, I_B, F_B)$. Here we represent $B_{\neg f}$ in a way similar to the system model M, except that F_B is an acceptance condition. Notice that here the acceptance condition F_B means a final condition or a fairness condition. The former defines a set of unsafe states; the latter defines a set of states that appear infinitely often, i.e., there exist loops through these states. Then the composition of the system model M and the *Büchi* automaton $B_{\neg f}$ can be represented as $M \times B_{\neg f} = (V, D, R, I, F)$, where $V = V_M \cup V_B$, $D = D_M \wedge D_B$, $R = R_M \wedge R_B$, $I = I_M \wedge I_B$, and $F = F_B$. A (compound) state $s \in D$ is an *unsafe* state in $M \times B_{\neg f}$ with respect to f, if s satisfies the final condition; or s satisfies the fairness condition and there exists a loop through s. Similar to the invariant case, let S_u be the set of unsafe states in $M \times B_{\neg f}$ with respect to f. Offline we can also calculate a backward reachable set S_u^+ of those states starting from the unsafe states in S_u up to n steps. We say S_u^+ is an extended set of unsafe states in $M \times B_{\neg f}$ with respect to f. Again, we do not need to calculate S_u^+ explicitly. Instead, we can also deduce an extended unsafe condition C_u^+ from C_u^0 such that $S_u^+ = \{s \in D \mid s \models C_u^+\}$ in the similar way:

$$C_u^+ = \bigvee_{t=0}^{n} C_u^t \text{ where } C_u^t = \exists s'.s' \models C_u^{t-1} \wedge R(s, s').$$

Here the initial unsafe condition, $C_u^0 = F$ if F is a finial condition; or $C_u^0 = F \wedge C_l$ if F is a fairness condition, provided that C_l is the loop condition in $M \times B_{\neg f}$ such that for any state $s \models C_l$, there exists a loop of length w (for some $w \geq 1$) through s. As a result, given the extended unsafe condition C_u^+, we need to decide whether some state satisfying C_u^+ is reachable or not. Online model checking thus becomes online reachability problem, which can be solved efficiently.

It is worth mentioning that in the case of F being a fairness condition, we have to calculate the loop condition $C_l = \bigvee_{t=1}^{w} C_l^t$, where $C_l^1 = R(s_0, s_1) \wedge (s_1 = s_0)$ and $C_l^t = \exists s_1, s_2, \cdots, s_{t-1}.R(s_0, s_1) \wedge R(s_1, s_2) \wedge \cdots \wedge R(s_{t-1}, s_t) \wedge (s_t = s_0)$ for $t > 1$. For finite state systems, w is the depth of backward breadth first search from accepting states. If the

fixed point is hard to reach, we can stop at some proper bound. In this case, the unsafe state condition is not complete.

In effect, we speed up online model checking by introducing offline backward exploration. Many existing efficient solutions to model checking can be directly applied to offline backward exploration. As offline there are more time and memory, it is possible to explore backwards much deeper in the state space of the system model to be checked. In this way, the workload of online forward exploration will be much reduced.

B. SAT Solver Specific for Online Model Checking

Modern SAT solver can handle large SAT problems with hundreds of thousands of variables very quickly. Therefore, SAT solver can be used to do online model checking. By introducing offline backward exploration, we can set the upper bound k for online forward exploration to be a relatively small constant. [6] quotes some of the experiments conducted in IBM, Intel, Compag and so on. These experiments show that the upper bound for BMC using SAT solver is better "not more than 60 to 80 cycles". Now, SAT solver is the key to the performance of the online reachability checking.

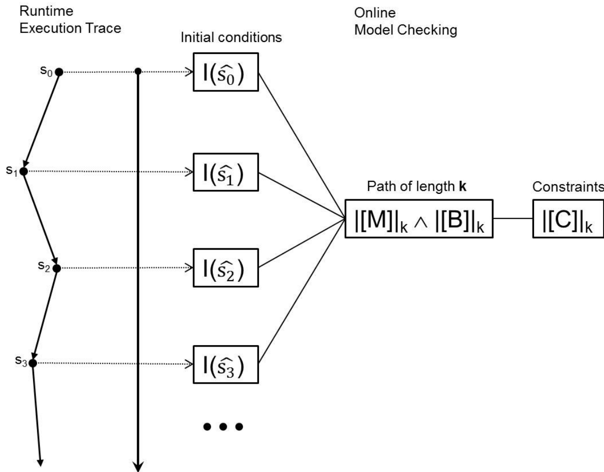

Fig. 3: Online Symbolic Model Checking

We convert the online reachability problem into a propositional satisfiability (SAT) problem as shown in Fig. 3 and then use an efficient SAT solver to search a counterexample if any. For this purpose, we need to convert system variables into *Boolean* variables. Let $\mathcal{B} = \{0, 1\}$ be the *Boolean* domain. The system model $M = (V_M, D_M, R_M, I_M)$ can be redefined as follows:

- $V_M = \{v_1, v_2, \cdots, v_n\}$ is the set of *Boolean* variables;
- $D_M = \mathcal{B}^n$ is the state space of M;
- $R_M \subseteq D_M \times D_M$ is the transition relation of M;
- I_M is the initial condition of M.

We can redefine $B_{\neg f} = (V_B, D_B, R_B, I_B, F_B)$ and $M \times B_{\neg f} = (V, D, R, I, F)$ in a similar way.

Generally speaking, let s_i $(i = 0, 1, 2, \cdots)$ be the concrete states observed at runtime. Then $\widehat{s_i} = \alpha(s_i)$ is the initial assignment to the *Boolean* variables in V_M for the i'th checking cycle. Recall that here α is the mapping function from concrete states to abstract states. Let $\overline{p}_t$, $\overline{q}_t$ and $\overline{r}_t$ be *Boolean* vectors that assign truth values at time t to the state variables in $V_M - V_B$, $V_B - V_M$ and $V_M \cap V_B$ respectively. The initial condition derived from $\widehat{s_i}$ for the i'th checking cycle is denoted as $\mathcal{I}(\widehat{s_i}) = \mathcal{I}_i(\overline{p}_0, \overline{r}_0)$. The path of length k in M is denoted as $|[M]|_k = \bigwedge_{t=1}^{k} R_M(\overline{p}_{t-1}, \overline{r}_{t-1}, \overline{p}_t, \overline{r}_t)$. Similarly, the path of length k in $B_{\neg f}$ is denoted as $|[B]|_k = \bigwedge_{t=1}^{k} R_B(\overline{q}_{t-1}, \overline{r}_{t-1}, \overline{q}_t, \overline{r}_t)$. Then $|[M]|_k \wedge |[B_{\neg f}]|_k$ is the product path of length k in $M \times B_{\neg f}$. We need to check if some state at time t $(0 \leq t \leq k)$ satisfies the extended unsafe condition C_u^+. Therefore, the constraint on the product path of length k is denoted as $|[C]|_k = \bigvee_{t=0}^{k} C_u^+(\overline{p}_t, \overline{q}_t, \overline{r}_t)$. As a result, the SAT problem for online reachability analysis at the i'th checking cycle is denoted as $|[M, \neg f]|_k^i = \mathcal{I}(\widehat{s_i}) \wedge |[M]|_k \wedge |[B]|_k \wedge |[C]|_k$.

Online reachability checking problem has its own specific features. Provided that T is the time limit allocated to online model checking for each checking cycle, after the initialization at the very beginning, the SAT solver needs to *restart* itself every T time units with a new initial condition $\mathcal{I}(\widehat{s_i})$. At each checking cycle, the SAT solver needs to search a solution (i.e., path) starting from the given initial condition in the model space derived from $|[M]|_k \wedge |[B]|_k \wedge |[C]|_k$. This solution (if any), i.e., a path of length not more than k, indicates some possible error in the system model. Therefore, for the question: is there some error in the system model? The SAT solver will answer "Yes" if a solution is found; "No" if no solution is found; or "Unknown" if the searching process is terminated due to the time constraint. In case of "Unknown", some statistics can be output if required.

It is usually hard for SAT solver to finish the work within the predefined limited time allocated for online model checking. Our goal is to improve the performance of the SAT solver so that it can search as large the model space as possible within the given time limit. Thus, we have more chance to find a solution or counterexample (if any) in time. We have optimized and customized the SAT solver zChaff for online model checking [17].

Converting a general propositional logic expression into a CNF (Conjunctive Normal Form) representation usually needs to introduce many auxiliary variables, which will lead to a large formula with excessive variables. We distinguish the dominant variables (i.e., original variables) from the auxiliary variables and give dominant variables priority over auxiliary variables in the assignment process. That is, we modified the decision heuristic VSIDS of zChaff in favor of the dominant variables.

It is easy to see in Fig. 3 that the only difference among the SAT problems of different checking cycles is the initial condition $\mathcal{I}(\widehat{s_i})$, which indicates the valuation of the state variables. Notice that each conflict clause learned by the SAT

solver is an implication of some clauses of the SAT problem, it is redundant and has nothing to do with the valuation of the variables. Therefore, the conflict clauses learned at previous checking cycles can be directly reused at the later checking cycles to reduce the searching space.

From one checking cycle to the next checking cycle we do not need to *reset* the SAT solver, which will remove all the learned clauses from the clause database. We'll instead *restart* the SAT solver in an efficient manner at each checking cycle. The restart operation in zChaff will simply undo those assignments of variables with decision level greater than 0. Let the state variables $v_1, v_2, \cdots, v_n$ at time frame 0 be ordered in this way. We observe that the initial states $\mathcal{I}(\widehat{s_i})$ and $\mathcal{I}(\widehat{s_{i+1}})$ of any two consecutive checking cycles usually have common part, i.e., the valuation of some variables keep unchanged. Therefore, we can backtrack to the first variable v_j (at decision level j) whose valuation is changed at $\mathcal{I}(\widehat{s_{i+1}})$. Of course, if v_1 is such a variable, then we have to backtrack to v_1 at decision level 1 in this case. However, as long as $j > 1$, we can reuse the deduction results done for $v_1, v_2, \cdots, v_{j-1}$ at the next checking cycle. In particular, if the initial states $\mathcal{I}(\widehat{s_i})$ and $\mathcal{I}(\widehat{s_{i+1}})$ happen to be the same, then we can simply continue the solving process as usual.

C. Underlying Hardware Architecture

With the rapid advancement of the electronic techniques, nowadays many realtime and critical embedded systems are equipped with Field Programmable Gate Arrays (FPGA). Compared to microprocessor, the use of FPGAs for heavy computational tasks is much faster and consumes less power due to the parallel computing feature. We've implemented a prototype of online model checking with SAT solver using FPGA as the underlying hardware architecture as shown in Fig. 4 [17]. The prototype works on the single soft-core processor MicroBlaze implemented in the Xilinx Virtex-5 XUPV5-LX110T FPGA with on-board memory and industry standard connectivity interfaces.

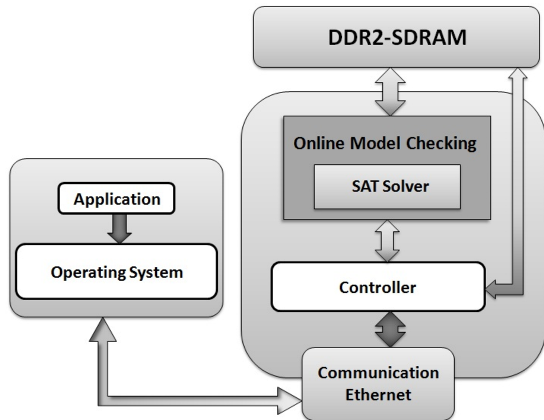

Fig. 4: Hardware Architecture with single SAT solver

Simply speaking, the system application to be checked and the underlying (realtime) operating system are located in one node, while online model checker in another node. The two nodes communicate with each other through Ethernet connection. The checking requirement together with the system model, the dominant variables and other necessary data is sent to the controller by the operating system. The controller loads the data into memory and then signals the online model checker to work. Inside the operating system there is a special system service named "monitor". The monitor observes the runtime state information s_i of the system execution from time to time, then maps it to the corresponding abstract state $\widehat{s_i}$ and finally sends $\widehat{s_i}$ to the controller. Accordingly, the controller put each $\widehat{s_i}$ to a (ring) buffer, a specific region in the memory. Every T time unit the SAT solver tries to take a state from the buffer. If there is a state available, the SAT solver goes to search a path (counterexample) starting from this state; otherwise, it either continues the work of the last checking cycle, provided the work has not finished yet, or simply waits for the next checking cycle begins. The checking results (Yes/No/Unknown) then is sent to the controller, which in turn sends the results to the operating system. In practice, the controller can inform the operating system only when some error is found by the SAT solver.

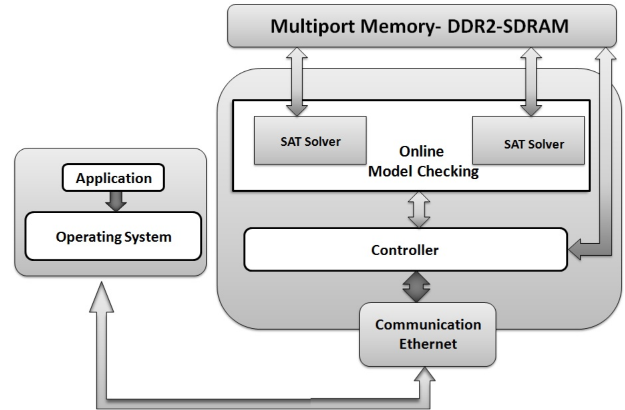

Fig. 5: Hardware Architecture with 2 SAT solvers

To better exploit the parallel feature of FPGA as well as the multi-port memory, we present a new architecture which can support 2 SAT solvers as verification engine for online model checking as shown in Fig. 5. We use 2 soft-core MicroBlaze processors, each with 1454 Flip Flops, 1558 LUT (not includes local instruction and data memory). We use Xilinx Platform Studio 13.1 Software to build our architecture. Each processor runs one SAT solver. The system model (CNF formula) and the dominant variables are stored in multi-port memory whose content can be accessed through different ports simultaneously. Each SAT solver is connected to the multi-port memory through Processor Local Bus. We use Round Robin arbitration algorithm and give each port equal priority. The controller is responsible for the synchronization with the 2 SAT solvers. The communication between the 2 SAT solvers

and the controller is established using mailbox, bidirectional channel. The 2 SAT solvers work in a similar way as the single solver case but in a pipelined manner (see Section IV).

IV. Experimental Results

We implement a quick prototype of the new hardware architecture for online model checking using 2 SAT solvers as verification engine on the 64 bit Windows platform with 2.13GHz i3 CPU and 4GB RAM. The SAT solver is zChaff optimized for online model checking [17].

We take the "MSI protocol with transient states" from the NuSMV software package as our case study. This protocol specifies that "There are three processors, each with one level of cache that stores 1-bit of data and has a 1-bit tag. The caches are write-back, write-allocate. The bus arbitration is round-robin. There is a memory with two 1-bit locations."

From the NuSMV specification of the protocol we can generate the CNF representations of transition relation (without initial condition) and the initial condition respectively. In addition, we can obtain dominant variables as by-product. By unrolling the transition relation up to k steps we get the CNF representation $|[M]|_k$ of the path of length k of the protocol model. The property to be checked is an invariant, from which we generate the CNF representation of the constraint $|[C]|_k$ of the path of length k. The runtime state information is simply generated by "executing" the model itself. Therefore, we have $s_i = \widehat{s_i}$. The SAT problem for online model checking is $|[M, \neg f]|_k^i = \mathcal{I}(\widehat{s_i}) \wedge |[M]|_k \wedge |[C]|_k$.

k	Total Variables	Dominant Variables	Total Clauses
35	6662	1620	22663
40	7602	1845	25873
45	8542	2070	29083
50	9482	2295	32293

TABLE I: MSI model with path of different lengths

Should we use different decision strategies or use the same decision strategy for the SAT solvers working in parallel? We make the 2 SAT solvers have different decision strategies: modified VSIDS and original VSIDS. Then, we do online model checking for the MSI protocol with path of length $k = 35, 40, 45$ and 50 from 201 different initial states, i.e., 201 checking cycles. Table I lists the total variables, the dominant variables and the total clauses of the MSI model with different lengths of path. In this experiment, we did not set any time limit to each checking cycle, just let the SAT solvers run to the end and then start the new checking cycle. From one checking cycle to the next checking cycle, the learned clauses are reused by the SAT solvers. If one SAT solver works faster than the other one, it will deal with more initial states than the other SAT solver, i.e., it will run more checking cycles. The experimental result in Fig. 6 shows that the SAT solver with modified VSIDS decision strategy can run more checking cycles in each case. The experimental result in Fig. 7 shows the total execution time in seconds of the SAT solvers in each case. The performance of the modified VSIDS decision strategy is better than the VSIDS decision strategy. This result is consistent with the one we've done in [17]. Therefore, we prefer using the same decision strategy (i.e., modified VSIDS) for the SAT solvers to do online model checking.

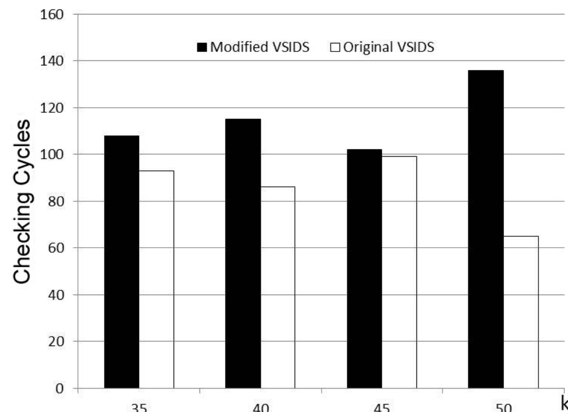

Fig. 6: Performance Comparison of 2 Decision Strategies

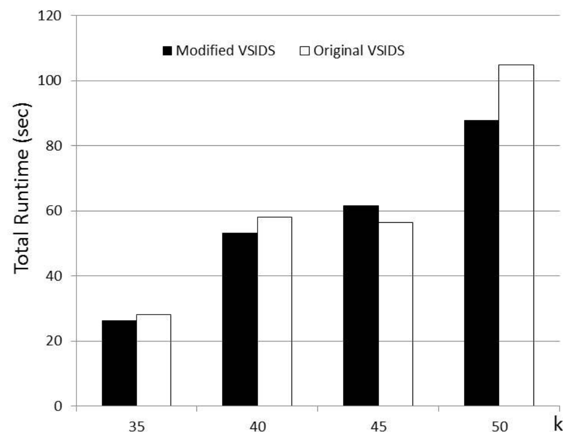

Fig. 7: Performance Comparison of 2 Decision Strategies

How many SAT solvers work in parallel can get the best performance? We've used 1, 2, 3, 4 and 5 SAT solvers respectively to do online model checking for the MSI protocol with path of length $k = 50$ from 201 different initial states, i.e., 201 checking cycles. In this experiment, we also did not set any time limit to each checking cycle, just let the SAT solvers run to the end and then start the new checking cycle. The experimental result in Fig. 8 shows the maximum execution time in seconds of the SAT solver(s) in each case. For example, in the case of 5 SAT solvers, we take the maximum execution time among the 5 SAT solvers. It is easy to see that the 2 SAT solvers' case outperforms all the other 4 cases. The possible reasons are as follows:

- The experimental results show that the learned clauses

shared between different checking cycles can improve the efficiency of the SAT solvers. But the more SAT solvers we use, the less states each SAT solver can deal with, thus the less learned clauses are produced to prune the state space for each SAT solver.

- Each SAT solver has to access cache and memory very frequently. The more SAT solvers we use, the higher is the probability of access conflicts due to requesting data at the same time. We use a Round Robin arbitration algorithm, which gives each port equal priority. In this case, our memory is considered as a shared resource between different processors (SAT solvers). If we run more than two SAT solvers in parallel, it means more synchronization overhead.

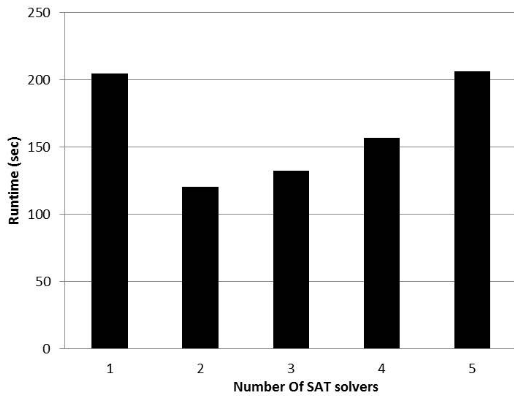

Fig. 8: Performance Comparison of Multi-SAT Solvers

From this experiment, we found that the learned clauses have a large impact on the performance of the SAT solver. In the 5 SAT solvers' case, one SAT solver deals with only 8 states, but its total execution time is even more than that of the single solver case, by which 201 states are processed. It is reasonable to use 2 SAT solvers for online model checking. In addition, we have let the 2 SAT solvers share the shortest learned clauses with each other, but the performance improvement is not very satisfying at least for this example. Therefore, we keep the 2 SAT solver independent of each other.

The 2 SAT solvers work in a pipelined manner as shown in Fig. 9. Every T time unit only one SAT solver tries to take a state from the buffer. Each SAT solver has $2T$ time unit to do the search work from the given state. For each SAT solver, if there is a state available, the SAT solver will go to search a path (solution) starting from this state. Otherwise, the SAT solver will resume the search work of the last checking cycle, provided that the work has not finished yet; or it will wait until the next checking cycle begins.

To simulate a general buffer setting, we use a randomly generated Boolean value to decide if a SAT solver can take a state from the buffer or not. The experimental result in Fig. 10 shows the performance of one SAT solver for online model

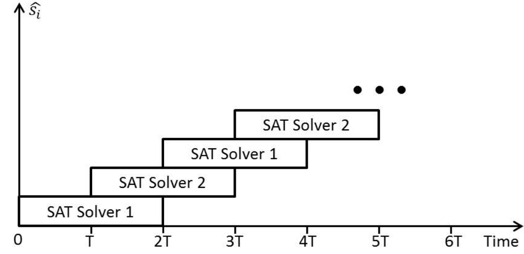

Fig. 9: 2 SAT solvers work in a pipeline manner

checking the MSI protocol with the path of length $k = 35$ from the 33rd to 65th checking cycles with time limit $2T = 0.08s$ for each checking cycle. At the checking cycles 45, 47, 49 and 53 the SAT solver can not take a state from the buffer and the work of the last checking cycle has been done, therefore, it does nothing and has to wait for the next checking cycles. At the checking cycles 37 upto 40 the SAT solver also can not take a state from the buffer, since the work at the checking cycle 36 has not finished yet, therefore, the SAT solver resumes the work of the checking cycle 36. At the checking cycles 44 the SAT solver also has to resume the work of the checking cycle 43 and finally gets a definite result this time, which means no error path of length $k \leq 35$ starting from the given state is found.

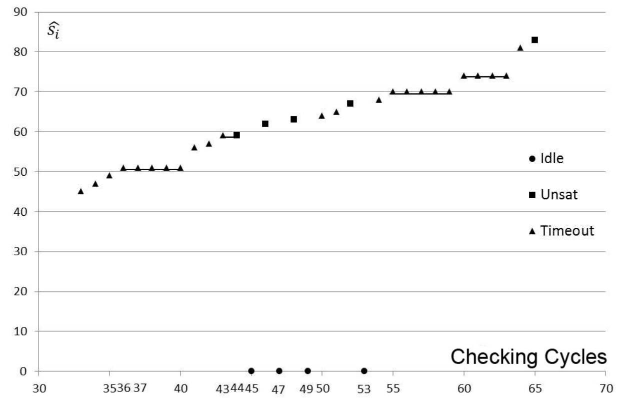

Fig. 10: Performance of SAT solver with random buffer setting

V. Related Work

The related work of online model checking can be found in [15]. SAT-based model checking is currently enjoying a substantial success in several industrial fields. Dramatic improvements in SAT solver technology over the last decade have fueled research in verification methods based on SAT solvers. [18] presents a methodology for applying Bounded Model Checking Using Satisfiability Solving in industry for invariance checking. [19] presents tuning the static order decision heuristics over GRASP for SAT procedures in the

context of bounded model checking of industrial designs. [20] improves SAT performance for Bounded Model Checking by tuning the VSIDS decision heuristic. There are other related works based on applying SAT for verification techniques as reported in several academic and industrial case studies [21] [22], other efforts in bounded model checking based on SAT have been applied in both hardware verification [23] and specification logics [24]. In addition, SAT have been used in the formal verification of railway control systems [25]. Our work is the first one to apply SAT solver for online model checking exploiting its own features.

VI. Conclusion

In this paper, we first explain the basic idea of online model checking, and then focus on accelerating online model checking from the following three aspects:

- Reduce the workload of online model checking by introducing offline backward exploration;
- Optimize and customize SAT solver specific for online model checking;
- Present a new underlying architecture based on FPGA using two SAT solvers working in a pipelined manner for online model checking.

We've implemented a quick prototype of the new underlying architecture and done several experiments to test the performance of the online model checking. Of course, this doesn't mean that our online model checking can definitely predict subtle errors before they have happened for any real world application. We believe that online model checking technique provides a simple way to "look" into the near future of the system application under test. Therefore, it has the potential capability to predict the errors if the implementation of our online model checking is enough efficient.

References

[1] D. M. Cummings, "Haven't found that software glitch, toyota? keep trying," *Los Angeles Times*, March 11, 2010.

[2] E. M. Clark, O. Grumberg, Jr, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[3] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, J. v. Leeuwen, J. Hartmanis, and G. Goos, Eds. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1996, foreword By-Wolper, Pierre.

[4] S. Berezin, S. V. A. Campos, and E. M. Clarke, "Compositional reasoning in model checking," in *COMPOS'97: Revised Lectures from the International Symposium on Compositionality: The Significant Difference*. London, UK: Springer-Verlag, 1998, pp. 81–102.

[5] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, 1994.

[6] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 118–149, 2003.

[7] G. J. Myers and C. Sandler, *The Art of Software Testing*. John Wiley & Sons, 2004.

[8] M. Barnett and W. Schulte, "Spying on components: A runtime verification technique," in *Workshop on Specification and Verification of Component-Based Systems*, G. T. Leavens, M. Sitaraman, and D. Giannakopoulou, Eds., Oct. 2001, published as Iowa State Technical Report 01-09a.

[9] K. Arkoudas and M. Rinard, "Deductive Runtime Certification," in *Proceedings of the 2004 Workshop on Runtime Verification (RV 2004)*, Barcelona, Spain, April 2004.

[10] S. Tasiran and S. Qadeer, "Runtime Refinement Checking of Concurrent Data Structures," in *Proceedings of the 2004 Workshop on Runtime Verification (RV 2004)*, Barcelona, Spain, April 2004.

[11] F. Chen and G. Rosu, "Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation," in *Proceedings of the 2003 Workshop on Runtime Verification (RV 2003)*, Boulder, Colorado, USA, 2003.

[12] D. Drusinsky, "The Temporal Rover and the ATG Rover," in *SPIN*, 2000, pp. 323–330.

[13] K. Havelund and G. Rosu, "Java PathExplorer — a runtime verification tool," in *Proceedings 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space (ISAIRAS'01)*, Montreal, Canada, Jun. 2001.

[14] A. Easwaran, S. Kannan, and O. Sokolsky, "Steering of discrete event systems: Control theory approach," *Electr. Notes Theor. Comput. Sci.*, vol. 144, no. 4, pp. 21–39, 2006.

[15] Y. Zhao and F. Rammig, "Online model checking for dependable real-time systems," in *15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*. Shenzhen, China: IEEE Computer Society, April 2012, pp. 154–161.

[16] A. Cimatti, E. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Integrating bdd-based and sat-based symbolic model checking," in *Frontiers of Combining Systems*, ser. Lecture Notes in Computer Science, A. Armando, Ed. Springer Berlin / Heidelberg, 2002, vol. 2309, pp. 265–276.

[17] M. Qanadilo, "Sat solver for online model checking," Master Thesis, Department of Computer Engineering, An Najah National University, Nablus, Palestine, January 2013.

[18] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Form. Methods Syst. Des.*, vol. 19, no. 1, pp. 7–34, Jul. 2001.

[19] O. Shtrichman, "Tuning sat checkers for bounded model checking," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, E. Emerson and A. Sistla, Eds. Springer Berlin / Heidelberg, 2000, vol. 1855, pp. 480–494.

[20] O. Shacham and E. Zarpas, "Tuning the vsids decision heuristic for bounded model checking," in *Microprocessor Test and Verification: Common Challenges and Solutions, 2003. Proceedings. 4th International Workshop on*, may 2003, pp. 75 – 79.

[21] A. Biere, E. Clarke, R. Raimi, and Y. Zhu, "Verifying safety properties of a powerpc microprocessor using symbolic model checking without bdds," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, N. Halbwachs and D. Peled, Eds. Springer Berlin / Heidelberg, 1999, vol. 1633, pp. 686–686.

[22] P. Bjesse, T. Leonard, and A. Mokkedem, "Finding bugs in an alpha microprocessor using satisfiability solvers," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, G. Berry, H. Comon, and A. Finkel, Eds. Springer Berlin / Heidelberg, 2001, vol. 2102, pp. 454–464.

[23] L. Guerra e Silva, L. M. Silveira, and J. Marques-Silva, "Algorithms for solving boolean satisfiability in combinational circuits," in *Proceedings of the conference on Design, automation and test in Europe*, ser. DATE '99. New York, NY, USA: ACM, 1999.

[24] O. Grumberg and D. E. Long, "Model Checking and Modular Verification," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 843–872, 1994.

[25] A. Borälv, "The industrial success of verification tools based on stalmarck's method," in *Proceedings of the 9th International Conference on Computer Aided Verification*, ser. CAV '97. London, UK: Springer-Verlag, 1997, pp. 7–10.