Strengthening Model Checking Techniques with Inductive Invariants

Short Papers

Strengthening Model Checking Techniques With Inductive Invariants

Gianpiero Cabodi, Sergio Nocco, and Stefano Quer

***Abstract*—This paper describes optimized techniques to efficiently compute and reap benefits from inductive invariants within satisfiability (SAT)-based model checking. We address sequential circuit verification and consider both equivalences and implications between pairs of nodes in the logic networks. First, we present a very efficient dynamic procedure, based on equivalence classes and incremental SAT, specifically oriented to reduce the set of checked invariants. Then, we show how to effectively integrate the computation of inductive invariants within state-of-the-art SAT-based model-checking procedures. Experiments (on more than 600 designs) show the robustness of our approach on verification instances on which stand-alone techniques fail.**

***Index Terms*—Formal verification, model checking, satisfiability (SAT), symbolic techniques.**

I. Introduction

Logic dependencies and implications between signals in a circuit have long been receiving substantial attention [1]. Verification techniques, based on satisfiability (SAT), often consider equivalences and implications among circuit signals as typical cases of simple invariants. The use of induction to compute invariants is also well known in formal verification, as shown in recent literature [2]–[8].

In this paper, we analyze an efficient integrated approach to compute and represent combined inductive equivalences and implications. Given a set of mutually equivalent circuit nodes, we express all equivalences by means of an equivalence class, thus avoiding a quadratic number of relationships. Implications are kept reduced by filtering out equivalences, representing implications between equivalence classes only, and pruning out transitive implications. Moreover, equivalences and implications are proven under external care set conditions, which allow a tighter integration (and intertwining) with other verification engines and/or strategies.

We also explore interactions between inductive invariants and SAT-based verification approaches. In our scheme, invariants are considered as a restriction of the state space, which we exploit in SAT-based (noninductive) unbounded model checking. We specifically address circuit-based quantification [9]–[11] and interpolant (ITP)-based verification [12].

This paper is a revised version of [13]. The theory has been reformulated, and experiments have been completely rerun. Data on SAT-based property verification show the efficacy of the integrated method in terms of performance and scalability.

II. Background

The state transition systems we address are sequential synchronous circuits modeled as finite-state machines. In our notation, S is the state space, $Init \subseteq$ S is the set of initial states, and TR is the transition relation of the system: S $\times$ S $\rightarrow \{0,1\}$. The subscripts indicate time frames. Let $s_0, s_1, \ldots, s_k$ be the state variables for time frames $0, 1, \ldots, k$. S_i and TR_i represent $S(s_i)$ and $TR(s_i, s_{i+1})$, respectively.

An inductive invariant is a property p proven by induction [14]. Given a depth k, a k-step inductive check verifies the following:

1) a *base case*, i.e., starting from the initial states of the system, p holds for k consecutive time steps;
2) an *inductive step*, i.e., if p holds for k consecutive time steps, starting from an arbitrary state, then it also holds at time step $k+1$.

A model-checking approach completely based on SAT and inductive invariants is proposed in [5] and [14]. The method is complete, due to an additional *uniqueness* (or *loop free*) condition for state paths. However, as we do not primarily address the completeness of inductive model checking, we avoid explicitly representing such a condition in the sequel.

A state-of-the-art algorithm for generating inductive invariants is shown in Fig. 1. The top-level function INDUCTIVEINV starts from the transition relation of the system TR, the initial state set $Init$, the set of properties under check I, and a set of external constraints Care. In the algorithm, depth is a user-selected value for limiting the induction depth, and P is the set of proven invariants, which is finally returned. The function first extends I with more potential invariants, which are obtained by random simulation from the set of all possible invariant candidates (line 2). The worst-case cardinality of this set is quadratic in the number of circuit nodes, as the equivalences and/or implications between all node pairs are included, if not disproven by simulation. The potential invariant set is then further refined, first by a preliminary

```
1   INDUCTIVEINV (TR, Init, I, Care)
2     I = I ∪ RANDOMSIMINV (TR, Init)
3     // Combinational step
4     P = CHECKINV (TR, Care, I, 0)
5     I = I \ P
6     for k = 0 to depth
7       // Base case
8       I = CHECKINV (TR, Init, I, k)
9       // Inductive step
10      P = P ∪ CHECKINV (TR, Care, I, k + 1)
11      I = I \ P
12    return (P)

13  CHECKINV (TR, S, I, k)
14    do
15      // Conjoin all candidates and express unrolling
16      Î = ⋀_{i∈I} i
17      f = S_0 ∧ ( ⋀_{j=0}^{k-1} TR_j ∧ Î_j ) ∧ (TR_k ∧ ¬Î_k)
18      // Call SAT solver
19      cex = SAT (f)
20      // Rule-out false invariants
21      if (cex ≠ ∅)  I = VALIDINV (I_k, cex)
22    while (cex ≠ ∅)
23    return (I)

24  VALIDINV (I, cex)
25    for each i ∈ I
26      // Evaluate candidates on the counter-example
27      if (¬i|_cex)  I = I \ i
28    return (I)
```

Fig. 1. Finding inductive invariants.

Manuscript received March 6, 2008; revised July 22, 2008. Current version published December 17, 2008. This paper was recommended by Associate Editor V. Bertacco.

The authors are with the Dipartimento di Automatica e Informatica, Politecnico di Torino, 10129 Turin, Italy (e-mail: stefano.quer@polito.it).

Digital Object Identifier 10.1109/TCAD.2008.2009147

combinational check (which is able to capture all invariants held in the whole state space, as shown in line 4) and then by iteration through couples of base case and inductive step with increasing depth k (line 6). Care, which is the set of external constraints (e.g., a binary decision diagram (BDD)-based overapproximate reachable states [6]), is used to restrict the search in the inductive step.

A unique SAT-based proof function CHECKINV is used for both the base case and the inductive step. Following [15], we organize a set of multiple checks as an iterative procedure, where, at each iteration, we refine the set of potential invariants by ruling out all candidates disproven by the same counterexample. In function CHECKINV, for each iteration of the outer loop (line 14), we evaluate the conjunction of all candidate invariants $\widehat{I}$ and compute the propositional formula f expressing either the base case or the inductive step, depending on the S parameter. A SAT solver then checks f (line 19). When f is satisfiable (line 21), function VALIDINV removes from I all the candidate invariants falsified by the current counterexample (line 27).

As noticed in previous works, a key feature for efficiency is the use of incremental SAT. It essentially consists of reusing conflict clauses across different SAT calls. We adopt this mechanism using the *unit assumption strategy* [16]. For any clause c that we may need to remove from the SAT solver database, we actually store the constraint $e \Rightarrow c$, with e being a fresh new variable. The e literal is then passed to the solver as an assumption, thus forcing c to be true. When c has to be removed, it is enough to add the unit clause $\neg e$.

```
1   VALIDINV (E, cex)
2     for each C ∈ E
3       C_new = ∅
4       for each n ∈ C
5         // Compare node and leader
6         if (n|_cex ≠ ĉ|_cex)
7           C = C \ n
8           C_new = C_new ∪ n
9       // Eventually add the new class to E
10      if (|C_new| > 0)  E = E ∪ C_new
11    return (E)
```

Fig. 2. Class splitting, given a SAT counterexample.

TABLE I
IMPLICATIONS AND CLASS SPLITTING

p	q	$P \Rightarrow P'$	$P' \Rightarrow P$	$Q \Rightarrow Q'$	$Q' \Rightarrow Q$
0	0	new	–	new	–
0	1	new	–	–	new
1	0	–	new	new	–
1	1	–	new	–	new
p	q	$P \Rightarrow Q$	$P \Rightarrow Q'$	$P' \Rightarrow Q$	$P' \Rightarrow Q'$
0	0	kept	new	–	new
0	1	kept	new	new	–
1	0	removed	new	new	new
1	1	kept	–	new	new

III. EQUIVALENCE AND IMPLICATION

The algorithm presented in Section II does not specifically address any particular kind of invariant. In this section, we describe our choices in order to efficiently deal with invariants expressing equivalences and implications. To improve the previous algorithm, we group equivalences into equivalence classes (with a linear, rather than quadratic, representation cost) and express implications between equivalence classes (rather than between individual circuit nodes). Our main contribution in this direction is an integrated approach to effectively represent and manipulate a minimal set of equivalences and implications, rather than just computing equivalences [8], or first computing a nonminimal set of implications and then reducing it [17].

1) Equivalence Classes: Let us partition all circuit nodes into equivalence classes (denoted as E). Classes are nonoverlapping, and their union is the full set of nodes, i.e., each node belongs to exactly one class. More formally, given any two circuit nodes n_i and n_j, they are (potentially) equivalent[1] *iff* both the nodes belong to the same class $C \in E$, i.e.,

$$(n_i \simeq n_j) \Leftrightarrow \exists C \in E : (n_i \in C) \wedge (n_j \in C).$$

Given the initial set E, which was computed by simulation, we randomly select one node $\widehat{c}$ for each class as the class representative or leader. Then, we explicitly represent, in all SAT problems, just a sufficient (linear) number of equivalences between each node and the related class leader. All other equivalences can transitively be derived from the aforementioned minimal set. Refinement is done by possibly splitting every class $C \in E$ into two subclasses, keeping all the nodes in C with the same (0 or 1) value grouped in the same subclass in the SAT counterexample.

We report a new procedure, a variant of the function VALIDINV presented in Fig. 1, in Fig. 2. The outermost loop (line 2) iterates through all classes. For each class C, we keep in C all nodes n with the same value of class leader $\widehat{c}$. The remaining ones (if any) are removed from C (line 7) and collected into a new class (line 8), to be finally added back to E (line 10).

The approach has two advantages. The complexity of function VALIDINV is linear with the number of nodes of the network, whereas its original version (see Fig. 1) is potentially quadratic. More importantly, it avoids choking the SAT solver with all (redundant) clauses and variables expressing candidate invariants that can be obtained by transitivity.

2) Implications: In general, since implications are not symmetric, we cannot reduce a quadratic set of implications to a linear one. Although transitive reductions can be applied, the worst-case number of implications is still quadratic. Our contribution is to reduce them as much as possible, by explicitly representing only implications between class leaders.

Given two nodes p and q belonging to classes P and Q, respectively, if $p \Rightarrow q$ is an implication invariant, by transitivity, each node $p_i \in P$ implies every node $q_j \in Q$. All those implications actually represent the same Boolean relation. We thus explicitly represent only one of them, i.e., the *class implication* $P \Rightarrow Q$.

The inductive proof algorithm may proceed exactly as previously described, until the SAT counterexample analysis. In this phase, function VALIDINV refines the equivalence classes and also updates the candidate implications. To this respect, our reduced representation of implications causes an additional complication. Some implicit implications, which are subsumed by potential equivalences, may need to be explicitly represented as a consequence of class splitting. Furthermore, new implications may arise as a result of the class-splitting process.

Let us consider a candidate class implication $P \Rightarrow Q$ and suppose that the current SAT counterexample causes both P and Q to split into $\{P, P'\}$ and $\{Q, Q'\}$, respectively. Then, Table I schematically reports the set of newly generated, kept, or removed implications, due to the values of class leaders $p \in P$ and $q \in Q$ in the counterexample.

From the implementation point of view, the set of candidate implications is modeled as a directed graph, where every node corresponds to an equivalence class and each edge represents one implication.

[1]Equivalences are considered modulo complementation. For the sake of simplicity, we will not explicitly represent negations in the rest of this paper.

```
1   APPROXTRAVIMPROVED (TR, Init, p)
2     R+= 1
3     do
4       // Compute Invariants within R+
5       I = INDUCTIVEINV (TR, Init, p, R+)
6       // Optimize TR
7       TR = OPTIMIZE (TR, I)
8       // Compute a new R+
9       R+ = R+∧ I
10      R+ = R+∧ APPROXTRAV (TR, Init, R+)
11    while (¬ (fixPoint ∨ timeOut ∨ spaceOut))
12    return (R+)
```

Fig. 3. Improved approximate traversal.

IV. Inductive Invariant and Model Checking

In this section, we show how to exploit inductive invariants as a source for optimizations and/or simplifications of SAT-based model-checking tasks. We will obviously avoid discussing inductive verification [14], as already widely known and described. By viewing invariant generation as a preprocessing step, any kind of subsequent verification could exploit circuit simplifications due to equivalent node collapse. We will show that other enhancements are possible in two directions: 1) the invariants used to restrict the search space of SAT-based model checking and 2) the intermediate results of other model-checking techniques adopted to further tighten the invariant sets. We will address BDD-based approximate reachability (as an orthogonal technique to make state space stricter), circuit-based quantification in backward reachability (as an example of mutual tightening), and ITP-based verification (as a case where invariants are used to restrict the search space).

1) Approximate Reachability: Invariants have been used by Case *et al.* [17] to compute an overapproximation of the reachable states, with more modular effort than BDD-based reachability. We obtain further benefits by intertwining the two approaches. We iteratively exploit inductive invariants as an external constraint to tighten BDD-based overapproximate reachability and BDD-based approximate states as an additional constraint to find more inductive invariants.

The reachability procedure is shown in Fig. 3. The function iterates through inductive invariant computations (line 5) and approximate traversals (line 10). Both procedures accept the current overapproximated reachable state set R^+ as a "care" parameter, which is used to enforce invariant proofs and make reachability tighter.

Function OPTIMIZE (line 7) returns an optimized transition relation by merging equivalent circuit nodes and exploiting implications for redundancy removal. Function APPROXTRAV (line 10) performs BDD-based approximate reachability analysis, following the frame-by-frame or machine-by-machine approach by Cho *et al.* [18].

2) Circuit-Based Quantification: Fig. 4 shows how we exploit and dynamically tighten invariants within a circuit-based quantification backward verification algorithm.

p is the property to be proven on model TR, starting from the set of states $Init$. An initial Care set is evaluated in terms of overapproximated BDDs and inductive invariants.[2] Then, a backward traversal iteration is performed, where the PREIMAGE operator (line 8) is based on circuit quantification [9]–[11]. The Care set is refined at each step by removing newly reached states from it (line 13). A further refinement of Care is done at the end of each iteration by computing new inductive invariants (line 14), as described in Section III. Backward traversal and inductive invariant computation thus have a bidirectional interaction, as tighter care sets help in finding new invariants, whereas tighter invariant sets help in simplifying state set representations.

[2]Heuristics to enable/disable BDD-based reachability, based on the number of state elements and the circuit size, are adopted but not discussed here.

```
1   BACKVERIF (TR, Init, p)
2     Reached = From = New = ¬p
3     Care = APPROXTRAVIMPROVED (TR, Init, p)
4     while (SAT (New))
5       // Check for intersection
6       if (SAT (From ∧ Init))  return (FAIL)
7       // Circuit-based pre-image computation
8       To = PREIMAGE (TR, From, Care)
9       New = (To \ Reached) ∧ Care
10      From = BESTAIG (To, Care)
11      Reached = Reached ∨ To
12      // Refine the care set
13      Care = Care \ Reached
14      Care = INDUCTIVEINV (TR, Init, p, Care)
15    return (PASS)
```

Fig. 4. Circuit-based backward verification with inductive invariants.

```
1   INTERPOLANTMC (TR, Init, p)
3     Care = APPROXTRAVIMPROVED (TR, Init, p)
3     k = 0
4     while (true)
5       R+= Init
6       k = k + 1
7       while (true)
8         f = R+∧ TR0
9         g = ( ⋀_{j=1}^{k} TRj ) ∧ ¬pk
10        if (SAT (f ∧ g))
11          if (R+≠ Init)  break
12          else  return (FAIL)
13        To = ITP(f, g) ∧ Care
14        if (To ⇒ R+)  return (PASS)
15        R+= R+∨ To
```

Fig. 5. ITP-based verification with inductive invariants.

Function BESTAIG returns the smallest AND inverter graph representing a set of states included between To ∧ Care and To ∨¬Care. To avoid a too-expensive procedure, we simply apply redundancy removal on To, which is controlled by time and size thresholds, using Care as an external care set. After that, we make a selection among To, New, and the optimized To, based on the circuit sizes (evaluated with linear cost).

3) ITP-Based Verification: Fig. 5 shows an ITP-based procedure inspired by [12]. Here, we use inductive invariants both to simplify TR (as in the previous cases) and as an additional constraint (Care) to be conjoined with reachable states generated by interpolation.

Given two inconsistent formulas f and g, a Craig ITP $ITP(f, g)$ is an overapproximation of f, which is still inconsistent with g, expressed in terms of the common variables of f and g. Within the framework of Fig. 5, an ITP is equivalent to an overapproximate image of R^+ [12]. The outermost loop (line 4) guarantees the use of deeper and deeper backward unrollings g until the (backward) diameter is possibly reached. Inner iterations (line 7) represent overapproximate traversals, using the ITP as the image operator. The procedure terminates in two possible cases: 1) when the property is disproven (line 12) and 2) when the inner loop reaches a fixed point (line 14). Whenever $f \wedge g$ is satisfiable (with $R^+ \neq Init$, as shown in line 11), we have a false failure and simply restart another traversal at the next outer iteration. The Care set is computed by mixing BDD-based approximate reachability and inductive invariants (line 3), as in the previous algorithms. It is used as a care set for both the SAT checks and the ITP simplifications based on redundancy removal.

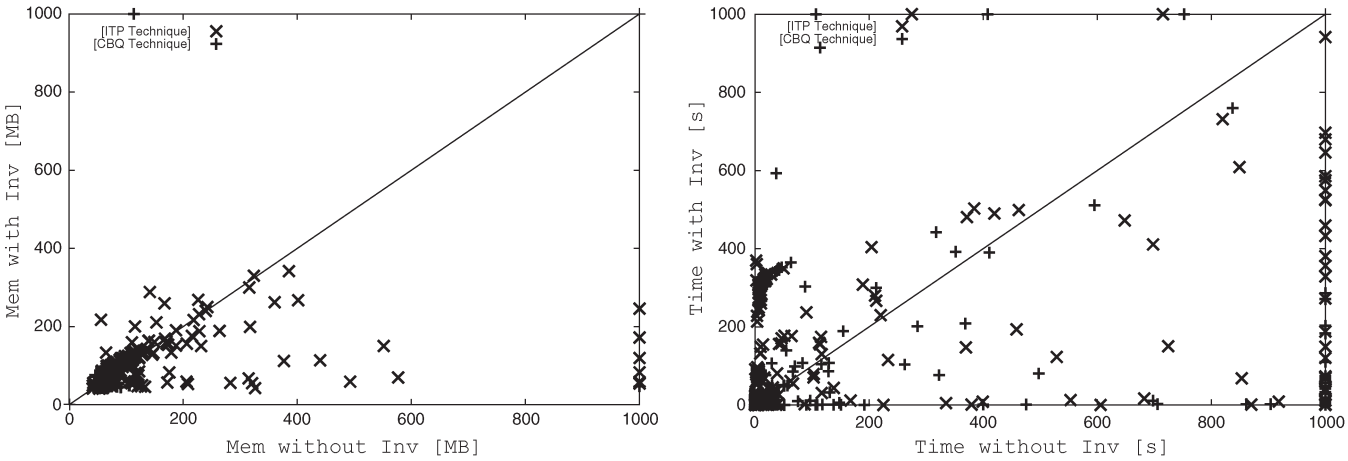

Fig. 6. Verification results with and without invariants on all $(344 + 309 = 653)$ benchmarks.

V. Experimental Result

We implemented our algorithms on top of the PdTrav tool, a state-of-the-art verification framework that won two of the subcategories (i.e., the "ALL" and the "UNSAT" ranking) at the 2007 Model Checking Competition [19]. The winning tool was a preliminary version of the invariant-based strategy presented in this paper. The tool uses Minisat [20] (version 1.14p) as the underlying SAT solver.

We present the results on two circuit suites: 344 benchmarks from the Model Checking Competition [19] and 309 verification instances from four other sources [i.e., the VIS distribution [21] (148 selected properties), the IBM Formal Verification Library [22] (65 circuits), the Sun PicoJava II examples [12] (20 instances), and the circuits coming from the VTT group of STMicroelectronics (74 properties)]. The benchmarks have latches ranging from tens to thousands (more than 4000) and include some very hard to prove properties.

All experiments ran on a Pentium Core Duo 2.4-GHz Workstation with 2 GB of main memory running Debian Linux. For each experiment, we set a memory limit of 1 GB and a time limit of 1000 s (which includes 300 s for inductive invariant computation). We also adopted a maximum induction depth of 2 for the equivalences and 1 for the implications.

Fig. 6 shows the overall results for both backward (circuit-based quantification [9]–[11]) and forward (ITP based [12]) verifications. The figure includes two graphs that compare the memory and execution time with and without invariants. We omitted cases where all methods ran out of time or memory. The markers below the main diagonal show gains for the methods presented in this paper. Most occurrences of worsening, at low execution times, clearly show that the introduced overhead dominates in "easy" instances, whereas improvements become evident on "difficult" instances with higher execution times. The experiments reported 205 failed properties and 186 inductively proven properties. Overall, the advantages and disadvantages of using inductive invariants were evenly distributed between failed and proven instances (with overhead in easy cases and average improvements in more difficult ones). We thus do not explicitly profile the two subsets of experiments. We deem it is more important to show the execution time for all the "pass" experiments than the 186 properties inductively proven. Fig. 7 plots those cases and clearly supports our claim that invariants can help subsequent model-checking tasks, even in cases where they are not able (alone) to inductively prove a property. A very interesting observation of the aforementioned graphs is that (after excluding too-hard experiments, which are not solved with any technique) out-of-time experiments were reduced from 74 to 3 in backward verification and from 37 to 2 in ITP-based verification. This shows that inductive invariants indeed help the overall verification process.

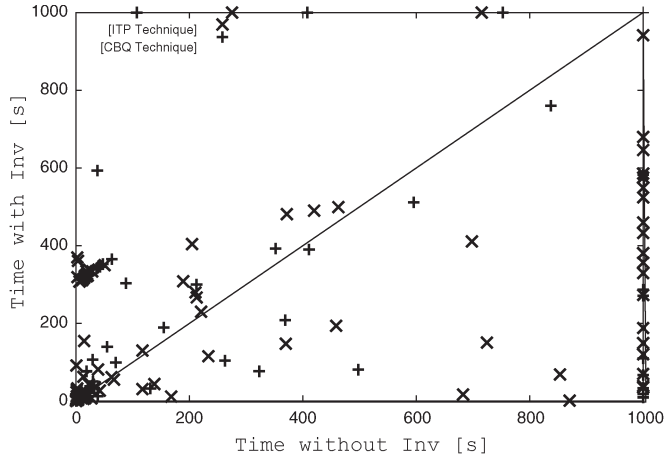

Fig. 7. Verification results for noninductive proofs.

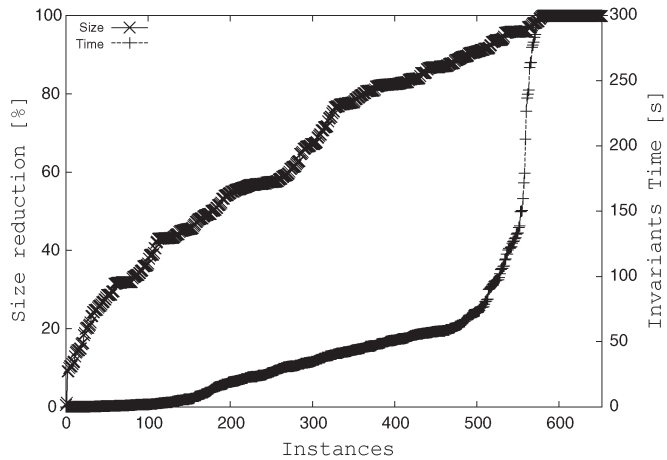

Fig. 8. Invariant computation time and their compaction effect.

Fig. 8 finally gives an additional insight on the invariant generation cost and consequent benefit. The two graphs show circuit size reduction (which is measured as the percentage ratio of the circuit size after/before invariant-based reduction) and the overall time spent

for invariant generation (which is limited by the 300-s bound). Both plots have been drawn, with the instances sorted by growing Y values. We can observe a relevant size compaction in many cases (more than 30% reduction in about 300 instances), whereas the time overhead is highly acceptable in most cases. (In about 550 of the 653 invariants, computation required less than 150 s.)

VI. Conclusion

This paper has addressed two main issues. It has described optimized techniques to speed up the computation of inductive invariants by means of efficient data structures and manipulation algorithms based on equivalence classes. It has then shown how to effectively integrate inductive invariants within state-of-the-art noninductive model-checking procedures. Experimental data show very promising results, improving the results obtained during the 2007 Model Checking Competition.

References

[1] W. Kunz and D. Pradhan, "Recursive learning: A new implication technique for efficient solutions to CAD problems-test, verification, and optimization," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 13, no. 9, pp. 1143–1158, Sep. 1994.

[2] K. van Eijk and J. Jess, "Detection of equivalent state variables in finite state machine verification," in *Proc. Int. Workshop Logic Synth.*, Lake Tahoe, CA, May 1995.

[3] D. Stoffel and W. Kunz, "Record and play: A structural fixed point iteration for sequential circuit verification," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, San Jose, CA, Nov. 1997, pp. 394–399.

[4] C. A. J. van Eijk, "Sequential equivalence checking based on structural similarities," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 7, pp. 814–819, Jul. 2000.

[5] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *Proc. Formal Methods Comput.-Aided Des.*, Austin, TX: Springer-Verlag, 2000, vol. 1954 pp. 372–389.

[6] A. Gupta, M. Ganai, C. Wang, Z. Yang, and P. Ashar, "Abstraction and BDDs complement SAT-based BMC in diver," in *Proc. Comput.-Aided Verification*, W. A. Hunt and F. Somenzi, Eds. Boulder, CO: Springer-Verlag, Jul. 2003, vol. 2725, pp. 206–209.

[7] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, San Jose, CA, Nov. 2004, pp. 50–57.

[8] F. Lu and K. T. Cheng, "IChecker: An efficient checker for inductive invariants," in *Proc. High-Level Des. Validation Test Workshop*, 2006, pp. 176–180.

[9] P. A. Abdulla, P. Bjesse, and N. Een, "Symbolic reachability analysis based on SAT-solvers," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 1785, S. Graf and M. I. Schwartzbach, Eds. Berlin, Germany: Springer-Verlag, Apr. 2000, pp. 411–425.

[10] M. K. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based unbounded symbolic model checking using circuit cofactoring," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, San Jose, CA, Nov. 2004, pp. 510–517.

[11] G. Cabodi, S. Nocco, and S. Quer, "Circuit based quantification: Back to state set manipulation within unbounded model checking," in *Proc. Des. Autom. Test Eur. Conf.*, Munich, Germany, Mar. 2005, pp. 688–689.

[12] K. L. McMillan, "Interpolation and SAT-based model checking," in *Proc. Comput.-Aided Verification*, W. A. Hunt and F. Somenzi, Eds. Boulder, CO: Springer-Verlag, 2003, vol. 2725, pp. 1–13.

[13] G. Cabodi, S. Nocco, and S. Quer, "Boosting the role of inductive invariants in model checking," in *Proc. Des. Autom. Test Eur. Conf.*, Nice, France, Apr. 2007, pp. 1–6.

[14] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and SAT solver," in *Proc. Formal Methods Comput.-Aided Des.*, W. A. Hunt and S. D. Johnson, Eds. Austin, TX: Springer-Verlag, Nov. 2000, vol. 1954, pp. 108–125.

[15] R. Fraer, S. Ikram, G. Kamhi, T. Leonard, and A. Mokkedem, "Accelerated verification of RTL assertions based on satisfiability solvers," in *Proc. HLDVT*, 2002, pp. 107–110.

[16] N. Eén and N. Sörensson, "Temporal induction by incremental SAT solving," in *Proc. 1st Int. Workshop BMC*, Boulder, CO, Jul. 2003, pp. 543–560.

[17] M. L. Case, A. Mishchenko, and R. K. Brayton, "Inductively finding a reachable state space over-approximation," in *Proc. Int. Workshop Logic Synth.*, Lake Tahoe, CA, May 2006.

[18] H. Cho, G. D. Hatchel, E. Macii, B. Plessier, and F. Somenzi, "Algorithms for approximate FSM traversal based on state space decomposition," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 15, no. 12, pp. 1465–1478, Dec. 1996.

[19] A. Biere and T. Jussila. (2007). *The Model Checking Competition Web Page*. [Online]. Available: http://fmv.jku.at/hwmcc/index.html

[20] N. Eén and N. Sörensson, *The Minisat SAT Solver*, 2003. [Online]. Available: http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/

[21] R. K. Brayton *et al.*, "VIS," in *Proc. Formal Methods Comput.-Aided Des.*, M. Srivas and A. Camilleri, Eds. Palo Alto, CA: Springer-Verlag, Nov. 1996, vol. 1166, pp. 248–256.

[22] IBM, *The IBM Formal Verification Benchmark Library*, 2003. [Online]. Available: http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/benchmarks.html

Access-Pattern-Aware On-Chip Memory Allocation for SIMD Processors

Hoseok Chang and Wonyong Sung

***Abstract*—The number of cycles for each external memory access in Single Instruction Multiple Data (SIMD) processors is heavily affected by the access pattern, such as aligned, unaligned, or stride. We developed a high-performance dynamic on-chip memory-allocation method for SIMD processors by considering the memory access pattern as well as the access frequency. The access pattern and the access count for an array of a loop are determined by both code analysis and profiling, which are performed on a developed compiler framework. This framework not only conducts dynamic on-chip memory allocation but also generates optimized codes for a target processor. The proposed allocation method has been tested with several multimedia benchmarks including motion estimation, 2-D discrete cosine transform, and MPEG2 encoder programs.**

***Index Terms*—Code generation, dynamic memory allocation, interleaved memory, on-chip memory allocation, Single Instruction Multiple Data (SIMD) processor.**

I. Introduction

Single Instruction Multiple Data (SIMD) processors are very efficient for arithmetic intensive applications; however, they are very susceptible to the performance bottleneck of memory systems [1], [2]. The data access pattern of SIMD processors is usually categorized as aligned, unaligned, and stride accesses; among them, the latter two demand more clock cycles in each memory access [3], [4]. Because of this, most SIMD processors equip an interleaved or a multiport memory unit not only to access multiple data at a time but also to align them efficiently [5], [6]. The interleaved memory unit is employed for this study because this structure is more scalable and economic when compared to the multiport-memory-based one.

Manuscript received January 28, 2008; revised July 19, 2008. Current version published December 17, 2008. This work was supported in part by Samsung Electronics Inc. and in part by the Ministry of Education, Science and Technology, Republic of Korea under the Brain Korea 21 Project. This paper was recommended by Associate Editor M. Poncino.

The authors are with the School of Electrical Engineering, Seoul National University, Seoul 151-742, Korea (e-mail: chs@dsp.snu.ac.kr; wysung@snu.ac.kr).

Digital Object Identifier 10.1109/TCAD.2008.2009145