Efficient Modeling of Embedded Memories in Bounded Model Checking

Malay K. Ganai, Aarti Gupta, and Pranav Ashar

NEC Laboratories America, Princeton, NJ USA 08540
Fax: +1-609-951-2499
{malay,agupta,ashar}@nec-labs.com

Abstract. We describe a viable approach for memory abstraction that preserves memory semantics, thereby augmenting the capability of SAT-based BMC to handle designs with large embedded memory without explicitly modeling each memory bit. Our method does not require examining the design or changing the SAT-solver and is guaranteed not to generate false negatives. The proposed method is similar, but with key enhancements, to the previous abstract interpretation of memory that captures its forwarding semantics, *i.e.*, a data read from a memory location is same as the most recent data written at the same location. In our method, we construct an abstract model for BMC by eliminating memory arrays, but retaining the memory interface signals and adding constraints on those signals at every analysis depth to preserve the memory semantics. The size of these *memory-modeling constraints* depends quadratically on the number of memory accesses and linearly on the bus widths of memory interface signals. Since the analysis depth of BMC bounds the number of memory accesses, these constraints are significantly smaller than the explicit memory model. The novelty of our memory-modeling constraints is that they capture the exclusivity of a read and write pair explicitly, *i.e.*, when a SAT-solver decides on a valid read and write pair, other pairs are *implied invalid immediately*, thereby reducing the solve time. We have implemented these techniques in our SAT-based BMC framework where we demonstrate the effectiveness of such an abstraction on a number of hardware and software designs with large embedded memories. We show at least an order of magnitude improvement (both in time and space) using our method over explicit modeling of each memory bit. We also show that our method of adding constraints boosts the performance of the SAT solver (in BMC) significantly as opposed to the conventional way of modeling these constraints as nested *if-then-else* expressions.

1 Introduction

Formal verification techniques like SAT-based Bounded Model Checking (BMC) [1-4] enjoy several nice properties over BDD-based symbolic model checking [5, 6]; their performance is less sensitive to the problem sizes and they do not suffer from space explosion. Due to the many recent advances in DPLL-style SAT solvers [7-11], SAT-based BMC can handle much larger designs and analyze them faster than before.

R. Alur and D.A. Peled (Eds.): CAV 2004, LNCS 3114, pp. 440–452, 2004.

Designs with large embedded memories are quite common and have wide application. However, these embedded memories add further complexity to formal verification tasks due to an exponential increase in state space with each additional memory bit. In typical BMC approaches [1-4], the search space increases with each time-frame unrolling of a design. With explicit modeling of large embedded memories, the search space frequently becomes prohibitively large to analyze even beyond a reasonable depth. In order to make BMC more useful, it is important to have some abstraction of these memories. However, for finding real bugs, it is sufficient that the abstraction techniques capture the memory semantics [12] without explicitly modeling each memory bit. In the following, we discuss some of the existing abstraction techniques.

In bottom-up abstraction and refinement techniques [13-18], starting from a small abstract model of the concrete design, counter-examples discovered are used to refine the model iteratively. In practice, several iterations are needed before a property can be proved correct or a real counter-example can be found. Note that after every iterative refinement step the model size increases, making it increasingly difficult to verify. In contrast to the bottom-up approaches, top-down approaches [19, 20] use resolution-based proof techniques in SAT to generate the abstract model, starting from a concrete design. As shown in [20], one can use iterative abstraction to progressively reduce the model size. Since these approaches use the concrete model to start with, it may not be feasible to apply them on designs with large memories. Moreover, both approaches, in general, are not geared towards extracting the memory semantics.

To capture the memory semantics, Burch and Dill introduced the interpreted *read* and *write* operations in their logic of equality with un-interpreted functions (EUF) [12] instead of naïve un-interpreted abstraction of memories. These interpreted functions were used to represent the memory symbolically by creating nested *if-then-else* (ITE) expressions to record the history of writes to the memory. Such partial interpretation of memory has also been exploited in later derivative verification efforts [21-24]. Specifically in [21], Velev *et al.* used this partial interpretation in a symbolic simulation engine to replace memory by a behavioral model that interacts with the rest of the circuit through a software interface that monitors the memory control signals. Bryant *et al.* proposed the logic of Counter arithmetic with Lambda expressions and Un-interpreted functions (CLU) to model infinite-state systems and unbounded memory in the UCLID system [25]. Memory is modeled as a functional expression whose body changes with each time step. Similar to [12], the memory is represented symbolically by creating nested ITE expressions. One can use BMC to verify safety properties with this restricted CLU logic.

In this paper, we describe a viable approach for memory abstraction that preserves memory semantics, thereby augmenting the capability of SAT-based BMC to handle designs with large embedded memory without explicitly modeling each memory bit. Our method does not require examining the design or changing the SAT-solver and is guaranteed not to generate false negatives. The proposed method is similar, but with key enhancements, to the abstract interpretation of memory [12, 21] that captures its forwarding semantics, *i.e.*, a data read from a memory location is the same as the most recent data written at the same location. In our method, we construct an abstract

model for BMC by eliminating memory arrays, but retaining the memory interface signals and adding constraints on those signals at every analysis depth to preserve the semantics of the memory. The size of these *memory-modeling constraints* depends quadratically on the number of memory accesses and linearly on the bus widths of memory interface signals. Since the analysis depth of BMC bounds the number of memory accesses, these constraints are significantly smaller than the explicit memory model. The novelty of our memory-modeling constraints is that they capture the exclusivity of a read and write pair explicitly, *i.e.*, when a SAT-solver decides on a valid read and write pair, other pairs are *implied invalid immediately*, thereby reducing the solve time. We have implemented these techniques in our SAT-based BMC framework where we demonstrate the effectiveness of such an abstraction on a number of hardware and software designs with large embedded memories. We show at least an order of magnitude improvement (both in time and space) using our method over explicit modeling of each memory bit. We also show that our method of adding constraints boosts the performance of the SAT solver (in BMC) significantly as opposed to the conventional way of modeling these constraints as nested *if-then-else* expressions [12].

Outline. In Section 2 we give basic idea used in our approach, in Section 3 we give relevant background on BMC and memory semantics, in Section 4 we discuss our approach in detail, in Section 5 we discuss our experiments, and in Section 6 we conclude with summarizing remarks and future work.

2 Basic Idea

For typical designs with embedded memory, a *Main* module interacts with the memory module *MEM* using the interface signals as shown in Figure 1. For a single-port memory at any given clock cycle, the following observations can be made: a) *at most one address is valid*, b) *at most one write occurs*, and c) *at most one read occurs*.

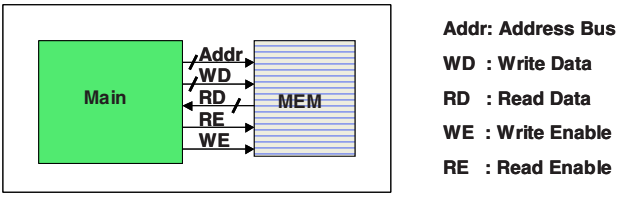

Fig. 1. Design with embedded memory

As BMC-search-bound k becomes larger, the unrolled design size increases linearly with the size of memory bits. For designs with large embedded memories, this increases the search space prohibitively for any search engine to analyze.

It can be observed in this memory model that memory read at any depth depends only on the most recent data written previously at the same address. Therefore, to enable the SAT-based BMC to analyze deeper on such designs, we replace an explicit memory model with an efficient memory model as follows:

a) We *remove* the *MEM* module but *retain* the memory interface signals and the input-output directionality with respect to the *Main* module.
b) We *add constraints* at every analysis depth *k* on the memory interface signals that preserve the forwarding semantics of the memory.

To improve the SAT solver (in BMC), we also do the following:

c) We *add constraints* such that when the SAT-solver decides on a valid read and write pair, other pairs are *implied invalid immediately.*

Note that although a) and b) are sufficient to generate an efficient model that preserves the validity of a correctness property, c) makes the performance of the SAT-based BMC superior as observed in our experiments. Moreover, we do not have to examine the *Main* module while adding these memory-modeling constraints. Though we consider a single-port memory for the ensuing discussion, our method is also extendable to multi-port memories. In the next two sections, we will give some relevant background and then formalize our contribution.

3 Background

Bounded Model Checking

In BMC, the specification is expressed in LTL (Linear Temporal Logic). Given a Kripke structure *M*, an LTL formula *f*, and a bound *k*, the translation task in BMC is to construct a propositional formula $[M, f]_k$, such that the formula is satisfiable if and only if there exists a witness of length *k* [26, 27]. The satisfiability check is performed by a backend SAT solver. Verification typically proceeds by looking for witnesses or counter-examples (CE) of increasing length until the *completeness threshold* is reached [26, 27]. The overall algorithm of a SAT-based BMC method for checking (or falsifying) a simple safety property is shown in the Figure 2. Note that P^i denotes the property node at i^{th} unrolling. The SAT problems generated by the BMC translation procedure grow bigger as *k* increases. Therefore, the practical efficiency of the backend SAT solver becomes critical in enabling deeper searches to be performed.

```
BMC(k,P){//Falsify safety property P within bound k
 for (int i=0; i<=k ; i++) {
   P^i= Unroll(P,i);//Property node at i^th unrolling
   if (SAT_Solve(P^i=0)=SAT) return CE;//Try to falsify
 }
 return NO_CE; } //No counter-example found
```

Fig. 2. SAT-based BMC for Safety Property P

Memory Semantics

Embedded memories are used in several forms such as RAM, stack, and FIFO with at least one port for data access. For our discussion, we will assume a single-port memory, as shown in Figure 1, with the following interface signals: Address Bus (Addr), Write Data Bus (WD), Read Data Bus (RD), Write Enable (WE), and Read Enable (RE). The *write* phase of the memory is a two-cycles event, *i.e.*, in the current clock

cycle *when* the data value is assigned to *WD* bus, the write address location is assigned to the *Addr* bus and the *WE* signal is made active, the new data is *then* available in the next clock cycle. The *read* phase of memory is a one-cycle event, *i.e.*, when the read address location is assigned to the *Addr* bus and *RE* is made active, the read data is assigned to the *RD* bus in the same clock cycle.

Assume that we unroll the design up to depth k (starts from 0). Let S^j denote a memory interface signal variable S at time frame j. Let the Boolean variable $E^{i,j}$ denote the address comparison between time frames i and j and be defined as $E^{i,j}=(Addr^i=Addr^j)$. Then the forwarding semantics of the memory can be expressed as:

$$RD^k = \{WD^j \mid E^{j,k}=1 \wedge WE^j=1 \wedge RE^k=1 \wedge \forall_{j<i<k}(E^{i,k}=0 \vee WE^i=0)\}, \text{ where } j<k \quad (1)$$

In other words, data read at depth k equals the data written at depth j if addresses are equal at k and j, write enable is active at j, read enable is active at k, and for all depths between j and k, either addresses are different from that of k or no write happened.

4 Our Approach

In our approach, we augment SAT-based BMC (a part of our formal verification platform) with a mechanism to add memory-modeling constraints at every unroll depth of BMC analysis. We use a hybrid SAT solver [10] that uses hybrid representations of Boolean constraints, *i.e.*, 2-input OR/INVERTER gates to represent the original circuit problem and CNF to represent the learned constraints. In this work, we extend the use of hybrid representation to model memory constraints efficiently. We specifically compare it with only a 2-input uniform gate (instead of a multi-input gate) representation, as it was already shown in [11] that efficient Boolean constraint propagation can be achieved using a fast table lookup scheme on such a representation.

In order to add the constraints for the forwarding semantics of memory as in (1), one can use a conventional approach based on the selection operator ITE[1] in the following way. Let the Boolean variable $s^{j,k}$ denote the *valid read signal* and be defined as $s^{j,k} = E^{j,k} \wedge WE^j$. Then the data read at depth k (given $RE^k=1$) is expressed as:

$$RD^k = ITE(s^{k-1,k}, WD^{k-1}, ITE(s^{k-2,k}, WD^{k-2}, \ldots ITE(s^{0,k}, WD^0, WD^{-1}))\ldots) \quad (2)$$

where WD^{-1} denotes the initial data value, same in all memory locations. We can extend (2) to handle non-uniform memory initialization as well.

Note that when constraints are added as above, the decision $s^{i,k}=1$ does not necessarily imply $RD^k=WD^i$; other pairs need to be established invalid through decision procedures as well, *i.e.*, $s^{i+1,k}=0$, $s^{i+2,k}=0,\ldots,s^{k-1,k}=0$. Instead we add explicit constraints to capture the read and write pairs exclusively. Once a read and write pair is chosen by the SAT-solver, the other pairs are implied invalid immediately. Let the Boolean

[1] ITE when applied on three Boolean variables is defined as ITE(s,t,e) = (s & t) | (!s & e).

variable $S^{i,k}$ denote the *exclusive valid read signal* and the Boolean variable $PS^{i,k}$ denote the intermediate exclusive signal. They are defined as follows:

$$\forall_{0\le i<k}\ PS^{i,k} = !s^{i,k} \wedge PS^{i+1,k} \quad (= RE^{k} \text{ for } i=k)$$
$$\forall_{0\le i<k}\ S^{i,k} = s^{i,k} \wedge PS^{i+1,k} \quad (= PS^{0,k} \text{ for } i=-1) \tag{3}$$

Now equation (1) can be expressed as

$$RD^{k} = (S^{k-1,k} \wedge WD^{k-1}) \vee (S^{k-2,k} \wedge WD^{k-2}) \vee \ldots \vee (S^{0,k} \wedge WD^{0}) \vee (S^{-1,k} \wedge WD^{-1}) \tag{4}$$

Note that $S^{i,k}=1$, immediately implies $S^{j,k}=0$ where $j \neq i$ and $i,j < k$.

4.1 Efficient Representation of Memory Modeling Constraints

As mentioned earlier, we use a hybrid representation for building constraints. We capture equations (3) and (4) efficiently

1) by not representing the constraints as a large tree-based structure since such a structure adversely affects the BCP performance as observed in the context of adding large conflict clauses [10], and
2) by not creating unnecessary 2-literal clauses since they too adversely affect a CNF-based SAT-solver that uses 2-literal watch scheme for BCP [28].

We implemented the addition of memory modeling constraints as part of the procedure *EMM_Constraints,* which is invoked after every unrolling as shown in the modified BMC algorithm (*m-BMC*) in Figure 3. The procedure *EMM_Constraints,* as shown in Figure 4, generates the constraints at every depth k using 3 sub-procedures: *Gen_Addr_Equal_Sig, Gen_Valid_Read_Sig,* and *Gen_Read_Data_Constraints*. It then returns the accumulated constraints C^{i} up to depth i. As we see in the following detailed discussion, these constraints C^{k} capture the forwarding semantics of the memory very efficiently up to depth k.

```
//BMC with efficient memory modeling
 m-BMC(k,P) {
  C^-1=ϕ; //initialize memory modeling constraints
  for (int i=0; i<=k ; i++) {
   P^i=Unroll(P,i);//Get property node at i^th unrolling
   C^i=EMM_Constraints(i,C^i-1);//update the constraints
   if (SAT_Solve(P^i=0∧C^i=1)=SAT) return CE;}//Try to falsify
  return NO_CE; } //No counter-example found
```

Fig. 3. Improved SAT-based BMC with Efficient Memory Modeling

Gen_Addr_Equal_Sig: Generation of Address Comparison Signals

Let m denote the bit width of the memory address bus. We implement the address comparison as follows: for every address pair comparison ($Addr^{j}=Addr^{k}$) we introduce new variables $E^{j,k}$ and $e^{j,k}_{i}$ $\forall_{0\le i<m}$ (for every bit i). Then we add the following CNF clauses for each i:

$(!E^{j,k} + Addr^{j}_{i} + !Addr^{k}_{i})$, $(!E^{j,k} + !Addr^{j}_{i} + Addr^{k}_{i})$,
$(e^{j,k}_{i} + Addr^{j}_{i} + Addr^{k}_{i})$, $(e^{j,k}_{i} + !Addr^{j}_{i} + !Addr^{k}_{i})$

Finally, one clause to connect the relation between $E^{j,k}$ and $e^{j,k}_{i}$, *i.e.*,

$(!e^{j,k}_{0} + \ldots + !e^{j,k}_{i} + \ldots + !e^{j,k}_{m-1} + E^{j,k})$

Note that these clauses capture the relation that $E^{j,k}=1$ if and only if $(Addr^{j}=Addr^{k})$. The naïve way to express the same equivalence relation structurally would be to use an AND-tree of X-NOR ($\otimes$) gates as follows:

$E^{j,k} = (Addr^{j}_{0} \otimes Addr^{k}_{0}) \wedge \ldots \wedge (Addr^{j}_{m-1} \otimes Addr^{k}_{m-1})$

Clearly, this representation would require *4m-1* 2-input OR gates, amounting to *12m-3* equivalent CNF clauses (3 clauses per gate). Our representation on the other hand, requires only *4m+1* clauses and does not require any 2-literal clause. Thus, at every depth k, we add only *(4m+1)k* clauses for address comparison rather than the *(12m-3)k* gates clauses required by the naïve approach.

```
//Modeling of memory constraint at depth k
//where C is cumulative constraints up to depth k-1
EMM_Constraints(k, C){
 //Generate address equal signals
 Gen_Addr_Equal_Sig(k);
 //Generate exclusive valid read signals
 Gen_Valid_Read_Sig(k);
 //Generate constraints on read data
 C(k) = Gen_Read_Data_Constraints(k);
 return C ∪ C(k);}
```

Fig. 4. Efficient Modeling of Memory Constraint

Gen_Valid_Read_Sig: Generation of exclusive valid read signals

To represent the exclusive valid read signals as in equation (3), we use a 2-input gate representation rather than CNF clauses. Since each intermediate variable has fan-outs to other signals, we cannot eliminate them. If we were representing those using CNF clauses, it would introduce too many additional 2-literal clauses. This representation adds *3k* 2-input gates (or *9k* gate clauses) at every depth k.

Gen_Read_Data_Constraints: Generation of constraints on read data signals

By virtue of equation (3), we know that for a given k, at most one $S^{j,k}=1$, $\forall_{-1 \le j < k}$. We use this fact to represent the constraint in equation (4) as CNF clauses. Let n denote the bit width of the data bus. We add the following clauses $\forall_{0 \le i < n,}\ \forall_{-1 \le j < k}$

$(!S^{j,k} + !RD^{k}_{i} + WD^{j}_{i})$, $(!S^{j,k} + RD^{k}_{i} + !WD^{j}_{i})$

To capture validity of read signal, we add the following clause

$(!RE^{k} + S^{-1,k} + \ldots + S^{j,k} + \ldots + S^{k-1,k})$

Thus we add *2n(k+1)+1* clauses at every depth k. On the other hand, if we use gate representation, it would require *n(2k+1)* gates and therefore, *3n(2k+1)* gate clauses.

Overall, at every depth k, our *hybrid exclusive select signals* representation adds *(4m+2n+1)k+2n+1* clauses and *3k* gates as compared to *(4m+2n+2)k +n* gates in a purely circuit representation. Note that though the size of these accumulated constraints grows quadratically with depth k, they are still significantly smaller than the explicit memory-model.

5 Experiments

For our experiments, we used three well-known recursive software programs *Fibonacci, 3n+1,* and *Towers-of Hanoi* with an embedded stack as shown in Figure 5 and one hardware design with embedded RAM. In each of these cases, we chose a safety property that makes the modeling of the entire memory imperative, *i.e.*, we simply cannot remove away the memory from the design. We translated each of the software programs into equivalent hardware models using Verilog HDL using a stack model. For each of the software designs, we use an inverse function to describe the negated safety property that requires a non-trivial state space search, e.g., given a certain value of *fibonacci* number does there exist a corresponding n? (Similar queries are made for a given number of recursive calls to terminate *3n+1*, and for a given number of legal moves required in *Towers-of-Hanoi*.)

```
1.  //Fibonacci
2.  //cache and recursion
3.  fib(n) {
4.    if (n<2) return n;
5.    //cache lookup
6.    if(lookup(n,&f))
7.      return f;
8.    f = fib(n-1)+fib(n-2);
9.    //insert cache
10.   store(n,f);
11.   return f; }

12. //3n+1
13. //period tracks # of
14. //calls req. to converge;
15. //initialized to 0
16. 3nPlus1(n) {
17.   if (n==1) return;
18.   if ((odd(n))
19.     3nPlus1(3*n+1);
20.   else
21.     3nPlus1(n/2);
22.   period++; }

23. //Towers of Hanoi
24. //count tracks # of moves
25. //req.; initialized to 0
26. toh(n,s,d,a) {
27.   if (n==0) return;
28.   toh(n-1,s,a,d);
29.   count++;
30.   toh(n-1,a,d,s); }
```

Fig. 5. Software programs with embedded stack used in experiments

We conducted our experiments on a workstation composed of dual Intel 2.8 GHz Xeon Processors with 4GB physical memory running Red Hat Linux 7.2 using a 3 hours time limit for each BMC run. We compare the performance of augmented SAT-based BMC (m-BMC) for handling embedded memory with basic SAT-based BMC using *explicit* memory modeling. We also compare the performance of m-BMC using our hybrid exclusive select signal representation (*hESS*), with that of a hybrid nested ITE representation (*hITE*). In addition, we show the effect on their performance with increasing memory sizes for a given property and design.

We performed our first set of experiments on the hardware models of the software programs with several properties selected as described above. Each of the properties has a non-trivial witness and is listed in Tables 1-4 in the order of increasing search complexity. We used a fixed memory size in each of the models. We also used one industrial hardware design with a safety property that is not known to have a counter example. For these properties, we show the performance and memory utilization comparison of the memory modeling styles, *i.e.*, explicit memory modeling, memory modeling using our *hESS* representation and that using *hITE* representation in the Tables 1-4. In Tables 1-3, we show comparison results for *Fibonacci*, *3n+1,* and *Towers-of-Hanoi,* respectively. We used address width (AW)=12, data width (DW)=32 for *Fibonacci,* AW=12, DW=2 for *3n+1*, and AW=12, DW=22 for *Towers-of-Hanoi* models. In Table 4, we show comparison results for the industrial hardware design with a given safety property S for various intermediate analysis depths as the property was not violated within the resource limit. Without the memory, the design has ~400 latches and ~5k gates. The memory module has AW=12 and DW=12.

In Tables 1-4, the 1st Column shows the properties with increasing complexity, the 2nd Column shows the witness depth (intermediate analysis depth in Table 4), 3-7 Columns show the performance figures and 8-12 Columns show the memory utilization figures. Specifically in the performance columns, Columns 3-5 show the BMC search time taken (in seconds) for explicit memory modeling (P1), using *hITE* (P2), and using *hESS* (P3) respectively; Columns 6 and 7 show the speed up (ratio) using *hESS* over the explicit memory modeling and *hITE* respectively. For the memory

Table 1. Comparison of memory modeling on *Fibonacci* model (AW=12, DW=32)

Prp	Wit Depth	Performance					Memory Utilization				
		Explicit P1(s)	hITE P2(s)	hESS P3(s)	Speed P1/P3	Speed P1/P3	Explicit M1(mb)	hITE M2(mb)	hESS M3(mb)	Red. M3/M1	Red. M3/M2
1-1	14	179	1	1	146	1.1	517	7	6	0.01	0.86
1-2	25	1050	5	4	248	1.3	1411	12	10	0.01	0.83
1-3	38	2835	20	15	184	1.3	2239	22	17	0.01	0.77
1-4	51	NA	79	47	NA	1.7	MO	41	34	NA	0.83
1-5	64	NA	125	100	NA	1.3	MO	63	52	NA	0.83
1-6	77	NA	252	311	NA	0.8	MO	100	75	NA	0.75
1-7	90	NA	587	362	NA	1.6	MO	175	92	NA	0.53
1-8	103	NA	625	557	NA	1.1	MO	163	161	NA	0.99
1-9	116	NA	1060	674	NA	1.6	MO	189	187	NA	0.99
1-10	129	NA	1674	1359	NA	1.2	MO	343	204	NA	0.59
1-11	142	NA	3782	2165	NA	1.7	MO	353	372	NA	1.05
1-12	155	NA	2980	2043	NA	1.5	MO	421	303	NA	0.72
1-13	168	NA	4349	4517	NA	1.0	MO	319	623	NA	1.95
1-14	181	NA	5573	4010	NA	1.4	MO	485	335	NA	0.69
1-15	194	NA	6973	4889	NA	1.4	MO	558	531	NA	0.95
1-16	**207**	**NA**	**> 3hr**	**7330**	**NA**	**NA**	**MO**	**541**	**461**	**NA**	**0.85**

Table 2. Comparison of memory modeling on *3n+1* model (AW=12, DW=2)

Prp	Wit Depth	Performance					Memory Utilization				
		Explicit P1(s)	hITE P2(s)	hESS P3(s)	Speed P1/P3	Speed P2/P3	Explicit M1(mb)	hITE M2(mb)	hESS M3(mb)	Red. M3/M1	Red. M3/M1
2-1	44	2736	85	51	54	1.7	293	25	20	0.07	0.8
2-2	47	3837	109	138	28	0.8	314	30	37	0.12	1.23
2-3	50	2811	167	160	18	1.0	412	44	37	0.09	0.84
2-4	53	3236	205	207	16	1.0	407	42	58	0.14	1.38
2-5	56	5643	258	264	21	1.0	569	48	44	0.08	0.92
2-6	59	4518	312	277	16	1.1	432	56	49	0.11	0.88
2-7	62	9078	324	368	25	0.9	479	58	59	0.12	1.02
2-8	65	9613	426	483	20	0.9	585	72	85	0.15	1.18
2-9	68	10446	487	522	20	0.9	648	73	64	0.10	0.88
2-10	71	9903	562	590	17	1.0	668	82	74	0.11	0.90
2-11	74	> 3hr	674	692	NA	1.0	981	83	92	NA	1.11
2-12	77	> 3hr	910	746	NA	1.2	719	110	83	NA	0.75
2-13	80	> 3hr	820	861	NA	1.0	875	106	89	NA	0.84
2-14	83	> 3hr	969	990	NA	1.0	586	113	80	NA	0.71
2-15	89	> 3hr	1292	1201	NA	1.1	659	127	113	NA	0.89

Table 3. Comparison of memory modeling on *Towers-of-Hanoi* (AW=12, DW=22)

Prp	Wit Depth	Performance					Memory Utilization				
		Explicit P1(sec)	hITE P2(s)	hESS P3(s)	Speed P1/P3	Speed P2/P3	Explicit M1(mb)	hITE M2(mb)	hESS M3(mb)	Red M3/M1	Red. M3/M2
3-1	10	4	0	0	149	1.3	71	3	3	0.04	1.00
3-2	24	182	1	1	264	1.2	664	6	5	0.01	0.83
3-3	52	2587	13	10	255	1.2	2059	16	12	0.01	0.75
3-4	108	NA	229	129	NA	1.8	MO	68	43	NA	0.63
3-5	220	NA	1266	838	NA	1.5	MO	214	143	NA	0.67
3-6	444	NA	8232	6925	NA	1.2	MO	845	569	NA	0.67

utilization columns, Columns 8-10 show the memory used (in MB) by explicit memory modeling (M1), using *hITE* (M2), and using *hESS* (M3) respectively; Column 11 and 12 show the memory usage reduction (ratio) using *hESS* over the explicit memory modeling and *hITE* respectively (Note, *MO* ≡ Memory Out, *NA* ≡ Not Applicable).

Observing the performance figures in Column 6 of the Tables 1-4, we see that our approach improves the performance of BMC by *1-2 orders of magnitude* when compared to explicit memory modeling. Similarly, as shown in Column 11 of these tables, there is a reduction in memory utilization by *1-2 orders of magnitude* by the use of our approach. Moreover, our modeling style of using the hybrid exclusive select signals representation is better than the hybrid nested ITE, as shown in Column 7 and 12. Noticeably, in the last row of Table 1 and 4, *hITE* times out while our approach

Table 4. Comparison of memory modeling on industrial hardware design (AW=12, DW=12)

Prp	Inter Depth	Performance					Memory Utilization				
		Explicit P1(s)	hITE P2(s)	hESS P3(s)	Speed P1/P3	Speed P2/P3	Explicit M1(mb)	hITE M2(mb)	hESS M3(mb)	Red. M3/M1	Red. M3/M2
S	68	10680	1264	925	11	1.3	2049	91	64	0.03	0.7
	150	NA	9218	7140	NA	1.3	MO	770	261	NA	0.3
	178	**NA**	**>3hr**	**10272**	**NA**	**NA**	**MO**	**NA**	**908**	**NA**	**NA**

Table 5. Comparison of memory modeling (for *3n+1*) with DW=12 and varying address width

Prp	AW	Performance					Memory Utilization				
		Explicit P1(s)	hITE P2(s)	hESS P3(s)	Speed P1/P3	Speed P2/P3	Explicit M1(mb)	hITE M2(mb)	hESS M3(mb)	Red M3/M1	Red M3/M2
2-1	4	64	85	60	1.1	1.4	23	22	20	0.87	0.9
	5	81	72	47	1.7	1.5	21	22	17	0.81	0.8
	6	110	77	79	1.4	1.0	25	23	24	0.96	1.0
	7	117	99	53	2.2	1.9	28	25	19	0.68	0.8
	8	146	87	78	1.9	1.1	41	24	24	0.59	1.0
	9	265	79	73	3.6	1.1	49	25	22	0.45	0.9
	10	767	86	59	12.9	1.4	95	27	22	0.23	0.8
	11	1490	89	56	26.4	1.6	153	27	20	0.13	0.7
	12	2736	85	51	54.1	1.7	293	25	20	0.07	0.8
	13	3759	83	54	69.6	1.5	569	24	21	0.04	0.9
	14	11583	81	46	249.2	1.7	1452	25	18	0.01	0.7

using *hESS* completes the analysis within the 3 hours time limit. Note that due to tail recursive nature of the *3n+1* program, the search complexity is not severe and therefore, we don't see the consistent benefit of exclusive select signals for this example in Table 2. On average, we see a performance improvement of 30% and a reduction in memory utilization of 20%, noticeably more at higher analysis depths.

In the second set of experiments, we used different memory sizes for the model *3n+1* and the property *2-1*. We varied the address bus width AW from 4 to 14 bits and compare the performance and memory utilization of the three approaches as shown in Column 2 of Table 5. The description of the remaining Columns in Table 5 is same as that in Tables 1-4. As shown in Columns 6 and 10, the performance improvement and memory usage reduction gets more pronounced, about 2 orders of magnitude, with increasing memory size. Clearly, the merits from adding memory-modeling constraints outweigh its quadratic growth. Moreover, our approach show on average 50% performance improvement and 20% memory usage reduction over nested ITE expressions.

6 Conclusions

Verifying designs with large embedded memories is typically handled by abstracting out (over-approximating) the memories. Such abstraction is generally not useful for

finding real bugs. Current SAT-based BMC efforts are incapable of handling designs with explicit memory modeling due to enormously increased search space complexity. In this work, we are the first to use memory-modeling constraints to augment SAT-based BMC in order to handle embedded memory designs without explicitly modeling each memory bit. Our method does not require transforming the design and is also guaranteed not to generate false negatives. This method is similar to abstract interpretation of memory [12, 21], but with key enhancements. Our method can easily augment any SAT-based BMC effort. We demonstrate the effectiveness of our approach on a number of software and hardware designs with large embedded memories. We show about 1-2 orders of magnitude time and space improvement using our method over explicit modeling of each memory bit. We also show that our method of adding constraints boosts the performance of a SAT solver significantly as opposed to the conventional way of modeling these constraints as nested ITE expressions [12]. While the growth of memory modeling constraint clauses is quadratic with the analysis depth, we also observed that though the constraint clauses are sufficient they may not be necessary in every time frame. As an ongoing effort, we are further investigating the possibility of adding these clauses only as and when required.

References

1. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *Proceedings of the Design Automation Conference*, 1999, pp. 317-320.
2. P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *Proceedings of Conference on Formal Methods in Computer-Aided Design*, 2000.
3. M. Ganai and A. Aziz, "Improved SAT-based Bounded Reachability Analysis," in *Proceedings of VLSI Design Conference*, 2002.
4. P. A. Abdulla, P. Bjesse, and N. Een, "Symbolic Reachability Analysis based on {SAT}-Solvers," in *Proceedings of Workshop on Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, 2000.
5. E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*: MIT Press, 1999.
6. K. L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*: Kluwer Academic Publishers, 1993.
7. J. P. Marques-Silva and K. A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Transactions on Computers*, vol. 48, pp. 506-521, 1999.
8. H. Zhang, "SATO: An efficient propositional prover," in *Proceedings of International Conference on Automated Deduction*, vol. 1249, *LNAI*, 1997, pp. 272-275.
9. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proceedings of Design Automation Conference*, 2001.
10. M. Ganai, L. Zhang, P. Ashar, and A. Gupta, "Combining Strengths of Circuit-based and CNF-based Algorithms for a High Performance SAT Solver," in *Proceedings of the Design Automation Conference*, 2002.
11. A. Kuehlmann, M. Ganai, and V. Paruthi, "Circuit-based Boolean Reasoning," in *Proceedings of Design Automation Conference*, 2001.

12. J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *Proceedings of the sixth International Conference on Computer-Aided Verification*, vol. 818, D. L. Dill, Ed.: Springer-Verlag, 1994, pp. 68--80.
13. D. E. Long, "Model checking, abstraction and compositional verification," Carnegie Mellon University, 1993.
14. R. P. Kurshan, *Computer-Aided Verification of Co-ordinating Processes: The Automata-Theoretic Approach*: Princeton University Press, 1994.
15. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Proceedings of CAV*, vol. 1855, *LNCS*, 2000, pp. 154-169.
16. E. M. Clarke, A. Gupta, J. Kukula, and O. Strichman, "SAT based abstraction-refinement using ILP and machine learning techniques," in *Proceedings of CAV*, 2002.
17. D. Wang, P.-H. Ho, J. Long, J. Kukula, Y. Zhu, T. Ma, and R. Damiano, "Formal Property Verification by Abstraction Refinement with Formal, Simulation and Hybrid Engines," in *38th Design Automation Conference*, 2001.
18. P. Chauhan, E. M. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang, "Automated Abstraction Refinement for Model Checking Large State Spaces using SAT based Conflict Analysis," in *Proceedings of FMCAD*, 2002.
19. K. McMillan and N. Amla, "Automatic Abstraction without Counterexamples," in *Tools and Algorithms for the Construction and Analysis of Systems*, April 2003.
20. A. Gupta, M. Ganai, P. Ashar, and Z. Yang, "Iterative Abstraction using SAT-based BMC with Proof Analysis," in *Proceedings of International Conference on Computer-Aided Design*, 2003.
21. M. N. Velev, R. E. Bryant, and A. Jain, "Efficient Modeling of Memory Arrays in Symbolic Simulation," in *ComputerAided Verification*, O. Grumberg, Ed., 1997, pp. 388-399.
22. R. E. Bryant, S. German, and M. N. Velev, "Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic," in *Computer-Aided Verification*, N. Halbwachs and D. Peled, Eds.: Springer-Verlag, 1999, pp. 470-482.
23. M. N. Velev, "Automatic Abstraction of Memories in the Formal Verification of Superscalar Microprocessors," in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, 2001, pp. 252-267.
24. S. K. Lahiri, S. A. Seshia, and R. E. Bryant, "Modeling and Verification of Out-of-Order Microprocessors in UCLID," in *Proceedings of Formal Methods in Computer-Aided Design*, 2002, pp. 142-159.
25. R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions," in *Computer-Aided Verification*, 2002.
26. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Proceedings of Workshop on Tools and Algorithms for Analysis and Construction of Systems (TACAS)*, vol. 1579, *LNCS*, 1999.
27. M. Sheeran, S. Singh, and G. Stalmarck, "Checking Safety Properties using Induction and a SAT Solver," in *Proceedings of Conference on Formal Methods in Computer-Aided Design*, 2000.
28. S. Pilarski and G. Hu, "Speeding up SAT for EDA," in *Proceedings of Design Automation and Test in Europe*, 2002, pp. 1081.