To appear in the *First International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS '98*), Portugal, March-April 1998.

Efficient Modeling of Memory Arrays in Symbolic Ternary Simulation[1]

Miroslav N. Velev[*]
mvelev@ece.cmu.edu

Randal E. Bryant[‡, *]
randy.bryant@cs.cmu.edu

[*]Department of Electrical and Computer Engineering
[‡]School of Computer Science
Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.

Abstract. This paper enables symbolic ternary simulation of systems with large embedded memories. Each memory array is replaced with a behavioral model, where the number of symbolic variables used to characterize the initial state of the memory is proportional to the number of distinct symbolic memory locations accessed. The behavioral model provides a conservative approximation of the replaced memory array, while allowing the address and control inputs of the memory to accept symbolic ternary values. Memory state is represented by a list of entries encoding the sequence of updates of symbolic addresses with symbolic data. The list interacts with the rest of the circuit by means of a software interface developed as part of the symbolic simulation engine. This memory model was incorporated into our verification tool based on Symbolic Trajectory Evaluation. Experimental results show that the new model significantly outperforms the transistor level memory model when verifying a simple pipelined data path.

1 Introduction

Ternary simulation, where the "unknown" value X is used to indicate that a signal can be either 0 or 1, has proven to be very powerful for both validation and formal verification of digital circuits [10]. Given that the simulation algorithm satisfies a monotonicity property to be described later, any binary values resulting from simulating patterns with X's would also result when the X's are replaced by any combination of 0's and 1's. Hence, employing X's reduces the number of simulation patterns, often dramatically. However, ternary simulators will sometimes produce a value X, when an exhaustive analysis would determine the value to be binary (i.e., 0 or 1). This has been resolved by combining ternary modeling with symbolic simulation [1], such that the signals can accept symbolic ternary values, instead of the scalar values 0, 1, and X. Each symbolic ternary value is represented by a pair of symbolic Boolean expressions, defined over a set of symbolic Boolean variables, that encode the cases when the signal would evaluate to 0, 1, or X. The advantage of symbolic ternary simulation is that it efficiently covers a wide range of circuit operating conditions with a single symbolic simulation pattern that involves far fewer variables than would be required for a complete binary symbolic simulation.

One of the hurdles in simulation has been the representation of memory arrays. These have been traditionally modeled by explicitly representing every memory bit.

1. This research was supported in part by the SRC under contract 97-DC-068.

While this is not a problem for conventional simulation, symbolic simulation would require a symbolic variable to denote the initial state of every memory bit. Furthermore, bit-level symbolic model checking [4][5] would need two symbolic variables per memory bit, in order to build the transition relation. Therefore, in both methods the number of variables is proportional to the size of the memory, and is prohibitive for large memory arrays.

This limitation is overcome in our previous work [11] by replacing each memory array with an Efficient Memory Model (EMM). The EMM is a behavioral model, which allows the number of symbolic variables used to be proportional to the number of distinct symbolic memory locations accessed rather than to the size of the memory. It is based on the observation that a typical verification execution sequence usually accesses only a limited number of distinct symbolic locations. However, it was assumed that the memory address and control inputs can accept only symbolic binary values.

To our knowledge, there has not been previous research on how to define a behavioral memory model for the cases when any of its address or control inputs has the value X in symbolic ternary simulation. Our experiments with Version 2.5 of the Cadence Design Systems VERILOG-XL indicated that a *Read* operation performed with an address containing X's returned the contents of the memory location determined when the X's are replaced by 1's. Also, a *Write* operation performed with an address containing X's did not alter the contents of any memory location. Such behavior might be sufficient in conventional informal logic simulation, where performance is of greater concern than functionality when simulating X values. However, it is not adequate for ternary simulation combined with formal verification, where such behavior might result in false positive verification results. The goal of this work is to enable the EMM to accept symbolic ternary values at its address and control inputs, while providing a conservative approximation of the replaced memory array. Conservative approximation means that false positive verification results are guaranteed not to occur, although false negative verification results are possible.

This paper builds on [11] with the following contributions: 1) an extended EMM which can have symbolic ternary values at its control and address inputs, and 2) an EMM-circuit interface which guarantees that the EMM would behave as a conservative approximation of the replaced memory array. Since symbolic ternary values are a superset of symbolic binary values, the extended EMM defined in this paper is a superset of the one from [11].

Experimental results for the EMM were obtained using the Symbolic Trajectory Evaluation (STE) [10] technique for formal verification. STE is an extension of symbolic simulation that has been used to formally verify circuits, including a simple pipelined data path [3]. Incorporation of the EMM in STE enabled the verification of the pipelined data path with a significantly larger register file than previously possible.

A symbolic representation of memory arrays has been used by Burch and Dill [6]. They apply uninterpreted functions with equality, which abstract away the details of the data path and allow them to introduce only a single symbolic variable to denote the

initial state of the entire memory. Each *Write* or *Read* operation results in building a formula over the current memory state, so that the latest memory state is a formula reflecting the sequence of memory writes. In our method, the memory state is represented with a list of entries encoding the sequence of updates of symbolic addresses with symbolic data. Our *Write* operation modifies this list. However, we perform the verification at the circuit level of the implementation and need bit-level data for symbolic word-level memory locations in order to verify the data path. This requires the user to introduce symbolic variables proportional to both the number of distinct symbolic memory locations accessed and the number of data bits per location.

This paper advocates a two step approach for the verification of circuits with large embedded memories. The first step is to use STE to verify the transistor level memory arrays independently from the rest of the circuit. Pandey and Bryant have combined symmetry reductions and STE to enable the verification of very large memory arrays at the transistor level [9][8]. The second step is to use STE to verify the circuit after the memory arrays are replaced by EMMs and is the focus of this work.

In the remainder of the paper, Sect. 2 describes the symbolic domain used in our algorithms. Sect. 3 gives a brief overview of STE. Sect. 4 presents the EMM and Sect. 5 introduces its underlying algorithms. Sect. 6 explains the way to incorporate the EMM into STE. Experimental results and conclusions are presented in Sect. 7.

2 Symbolic Domain

We will consider three different domains - control, address, and data - corresponding to the three different types of information that can be applied at the inputs of a memory array. A control expression c will represent the value of a node in ternary symbolic simulation and will have a high encoding $c.h$ and a low encoding $c.l$, each of which is a Boolean expression. The ternary values that can be represented by a control expression c are shown in Table 1. We would write [$c.h$, $c.l$] to denote c. It will be assumed that $c.h$ and $c.l$ cannot be simultaneously **false**. The types **BExpr**, **CExpr** will denote respectively Boolean and control expressions in the algorithms to be presented.

Ternary value	$c.h$	$c.l$
0	**false**	**true**
1	**true**	**false**
X	**true**	**true**

Table 1. 2-bit encoding of ternary logic

The memory address and data inputs, since connected with circuit nodes, will receive ternary values represented as control expressions. Hence, addresses and data will be represented by vectors of control expressions having width n and w, respectively, for a memory with $N = 2^n$ locations, each holding a word consisting of w bits. Observe that an X at a given bit position represents the “unknown” value, i.e., the bit

can be either 0 or 1, so that many distinct addresses or data will be represented. To capture this property of ternary simulation, we introduce the type **ASExpr** (address set expression) to denote a set of addresses. Similarly, the type **DSExpr** (data set expression) will denote a set of data. Note that in both cases, a set will be represented by a single vector of ternary values. We will use the notation $\langle a_1, \ldots, a_n \rangle$ to explicitly represent the address set expression a, where a_i is the control expression for the corresponding bit position of a. Data set expressions will have a similar explicit representation, but with w bits. Symbolic variables will be introduced in each of the domains and will be used in expression generation.

The symbols $\mathcal{U}_{\mathcal{A}}$ and $\mathcal{U}_{\mathcal{D}}$ will designate the universal address and data sets, respectively. They will represent the most general information about a set of addresses or data. Similarly, the symbols $\varnothing_{\mathcal{A}}$ and $\varnothing_{\mathcal{D}}$ will denote the empty address and data sets, respectively. In ternary logic, $\mathcal{U}_{\mathcal{A}}$ and $\mathcal{U}_{\mathcal{D}}$ can be represented by vectors of control expressions consisting entirely of Xs.

We will use the term *context* to refer to an assignment of values to the symbolic variables. A Boolean expression can be viewed as defining a set of contexts, namely those for which the expression evaluates to **true**.

A symbolic predicate is a function which takes symbolic arguments and returns a symbolic Boolean expression. The following symbolic predicates will be used in our algorithms, where c is of type **CExpr**, and a is of type **ASExpr**:

$$Zero(c) \doteq \neg c.h \wedge c.l \,, \tag{1}$$

$$Hard(c) \doteq c.h \wedge \neg c.l \,, \tag{2}$$

$$Soft(c) \doteq c.h \wedge c.l \,, \tag{3}$$

$$Unique(a) \doteq \bigwedge_{i=1}^{n} \neg\, Soft(a_i) \,. \tag{4}$$

The predicates *Zero*, *Hard*, and *Soft* define the conditions for their arguments to be the ternary 0, 1, and X, respectively. The predicate *Unique* defines the condition for the address set expression a to represent a *unique* or single address.

The selection operator *ITE* (for "If-Then-Else"), when applied on three Boolean expressions, is defined as:

$$ITE(b,\ t,\ e) \doteq (b \wedge t) \vee (\neg b \wedge e) \,. \tag{5}$$

Address set comparison with another address set is implemented as:

$$a_1 = a_2 \doteq \neg \bigvee_{i=1}^{n} [(a_1.h_i \oplus a_2.h_i) \vee (a_1.l_i \oplus a_2.l_i)] \,, \tag{6}$$

where $a_1.h_i$ and $a_1.l_i$ represent the high and low encodings of the control expression for bit i of address set expression a_1. Address set comparison with the universal address set is implemented as:

$$a = \mathcal{U}_{\mathcal{A}} \doteq \bigwedge_{i=1}^{n} Soft(a_i) \,. \tag{7}$$

Address set selection $a_1 \leftarrow ITE(b,\ a_2,\ a_3)$ is implemented by selecting the corre-

sponding bits:

$$a_1.h_i \leftarrow ITE(b,\ a_2.h_i,\ a_3.h_i), \quad a_1.l_i \leftarrow ITE(b,\ a_2.l_i,\ a_3.l_i), \quad i = 1, \ldots, n\,. \tag{8}$$

Checking whether address set a_1 is a subset of address set a_2 is done by:

$$a_1 \subseteq a_2 \ \doteq\ \neg \bigvee_{i=1}^{n} (a_1.h_i \wedge \neg a_2.h_i \ \vee\ a_1.l_i \wedge \neg a_2.l_i)\,, \tag{9}$$

and checking address sets a_1 and a_2 for overlap is implemented by:

$$Overlap(a_1, a_2) \ \doteq\ \bigwedge_{i=1}^{n} (a_1.l_i \wedge a_2.l_i \ \vee\ a_1.h_i \wedge a_2.h_i)\,. \tag{10}$$

Computing $l \in a$, where a is an address set expression and l is a vector of Boolean expressions, is implemented by:

$$l \in a \ \doteq\ \bigwedge_{i=1}^{n} ITE(l_i,\ a.h_i,\ a.l_i)\,. \tag{11}$$

The definition of symbolic predicates over data set expressions is similar, but over vectors of width w.

Note that all of the above predicates are symbolic, i.e., they return a symbolic Boolean expression and will be true in some contexts and false in others. Therefore, a symbolic predicate cannot be used as a control decision in algorithms. The function *Valid*(), when applied to a symbolic Boolean expression, will return **true** if the expression is valid or equal to **true** (i.e., true for all contexts), and will return **false** otherwise. We can make control decisions based on whether or not an expression is valid.

We will also need to form a data set expression that is the union of two data set expressions, d_1 and d_2. If these differ in exactly one bit position, i.e., one of them has a 0 and the other a 1, then the ternary result will have an X in that bit position and will be an exact computation. However, if d_1 and d_2 differ in many bit positions, these will be represented as Xs in the ternary result and that will not always yield an exact computation. For example, if $d_1 = \langle 0, 1\rangle$ and $d_2 = \langle 1, 0\rangle$, the result will be $\langle X, X\rangle$ and will not be exact, as it will also contain the data set expressions $\langle 0, 0\rangle$ and $\langle 1, 1\rangle$, which are not subsets of d_1 or d_2. We define the operation *approximate union* $d_1 \mathbin{\widetilde{\cup}} d_2$ of two data set expressions as:

$$[d_1 \mathbin{\widetilde{\cup}} d_2\ ]_i \ \doteq\ [d_1.h_i \vee d_2.h_i,\ d_1.l_i \vee d_2.l_i], \qquad i = 1, \ldots, w\,. \tag{12}$$

Finally, we will define the operation *data merge*, $\widetilde{\bigcup}_{l \in a} d$, where l is a vector of symbolic variables, a is an address set expression, and d is a data set expression, as:

$$[\widetilde{\bigcup}_{l \in a} d]_i \ \doteq\ [\exists_l\ (l \in a) \wedge d.h_i,\ \exists_l\ (l \in a) \wedge d.l_i], \qquad i = 1, \ldots, w\,. \tag{13}$$

We have used Ordered Binary Decision Diagrams (OBDDs) [2] to represent the Boolean expressions in our implementation. However, any representation of Boolean expressions can be substituted, as long as function *Valid*() can be defined for it.

3 STE Background

STE is a formal verification technique based on symbolic simulation. For the purpose of this paper, it would suffice to say that STE is capable of verifying circuit properties,

described as *assertions*, of the form $A \overset{\text{LEADSTO}}{\Rightarrow} C$. The *antecedent A* specifies constraints on the inputs and the internal state of the circuit, and the *consequent C* specifies the set of expected outputs and state transitions. Both *A* and *C* are formulas that can be defined recursively as:

1) *a simple predicate*: (**node** *n* = *b*), or (**node_vector** *N* = *a*), or (**node_vector** *N* = *d*), where *b*, *a*, and *d* are of types **BExpr**, **ASExpr**, and **DSExpr**, respectively, and in the last two cases each node of the node vector *N* gets associated with its corresponding bit-level control expression of the given address-set or data-set expression;

2) *a conjunction of two formulas*: $F_1 \wedge F_2$ is a formula if F_1 and F_2 are formulas;

3) *a domain restriction*: $(b \rightarrow F)$, where *b* is of type **BExpr**, is a formula if *F* is a formula, meaning that *F* should hold for the contexts in which *b* is **true**;

4) *a next time operator*: $\mathbf{N}F$ is a formula if *F* is a formula, meaning that *F* should hold in the next time period;

5) *a memory array indexing predicate*: (*mem*[*a*] = *d*), where *mem* is a memory name, *a* is of type **ASExpr**, and *d* is of type **DSExpr**.

A shorthand notation for *k* nested next time operators is $\mathbf{N}^k$. A formula is said to be *instantaneous* if it does not contain any next time operators. Any formula *F* can be rewritten into the form $F_0 \wedge \mathbf{N}F_1 \wedge \mathbf{N}^2F_2 \wedge \ldots \wedge \mathbf{N}^kF_k$, where each formula F_i is instantaneous. For simplicity in the current presentation, we will assume that the antecedent is free of self inconsistencies, i.e., it cannot have a node asserted to two complementary logic values simultaneously.

STE maintains two global Boolean expressions OK_A and OK_C, which are initialized to be **true**. The STE algorithm updates the circuit node values and the global Boolean expressions at every simulation time step. The antecedent defines the stimuli and the consequent defines the set of acceptable responses for the circuit. The expression OK_A maintains the condition under which the circuit node values are compatible with the values specified by the antecedent. The expression OK_C maintains the condition under which the circuit node values belong to the set of acceptable values specified by the consequent. The Boolean expression $\neg OK_A \vee OK_C$ defines the condition under which the assertion holds for the circuit.

4 Efficient Modeling of Memory Arrays

The main assumption of our approach is that every memory array can be represented, possibly after the introduction of some extra logic, as a memory with only write and read ports, all of which have the same numbers of address and data bits, as shown in Fig. 1.

The interaction of the memory array with the rest of the circuit is assumed to take place when a port `Enable` signal is not 0. In case of multiple port `Enable`s not being 0 simultaneously, the resulting accesses to the memory array will be ordered according to the priority of the ports.

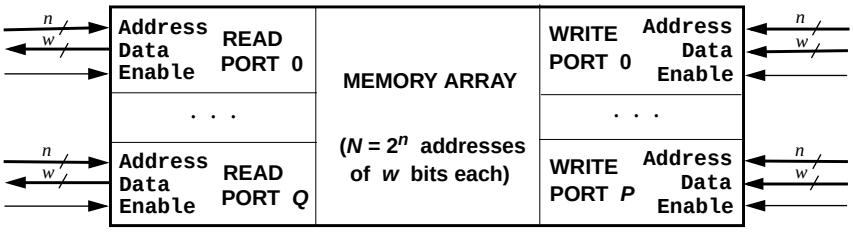

Fig. 1. View of a memory array, according to our model

During symbolic simulation, the memory state is represented by a list containing entries of the form $\langle h, s, a, d \rangle$, where h and s are Boolean expressions denoting the set of contexts for which the entry is defined, a is an address expression denoting a memory location, and d is a data expression representing the contents of this location. The context information is included for modeling memory systems where the *Write* operation may be performed conditionally on the value of a control signal c. The Boolean expression h represents the contexts $Hard(c) \wedge Unique(a)$, when the control signal was 1 and the address a was unique. Under contexts h the location a is definitely overwritten with data d. The Boolean expression s represents the contexts $Soft(c) \vee Hard(c) \wedge \neg Unique(a)$, when the control signal was an X, or it was a 1 and the address was not unique. Under contexts s the location a is uncertainly overwritten with data d. Initially the list is empty. The type **List** will be used to denote such memory lists.

The list interacts with the rest of the circuit by means of a software interface developed as part of the symbolic simulation engine. The interface monitors the memory input lines. Should a memory input value change, given that its corresponding port `Enable` value c is not 0, a *Write* or a *Read* operation will result, as determined by the type of the port. The `Address` and `Data` lines of the port will be scanned in order to form the address set expression a and the data set expression d, respectively. A *Write* operation takes as arguments both a and d, while a *Read* operation takes only a. Both of these operations will be presented in the next section.

After completing a *Write* operation, the software interface checks every read port of the same memory for a possible on-going read (as determined by the port `Enable` value being different from 0) from an address that overlaps the one of the recent write. For any such port, a *Read* operation is invoked immediately. This guarantees that the EMM will behave as a conservative approximation of the replaced memory array.

A *Read* operation retrieves from the list a data set expression rd that represents the data contents of address a. The software interface completes the read by scheduling the `Data` lines of the port to be updated with the data set expression $ITE(Hard(c), rd, ITE(Soft(c), (rd \mathbin{\widetilde{\cup}} d), d))$. The data set expression d is the one that the `Data` lines will otherwise have.

5 Implementation of Memory Operations

5.1 Support Operations

The list entries are kept in order from *head* (low priority) to *tail* (high priority). The initial state of every memory location is assumed to contain arbitrary data and is represented with the universal data set $\mathcal{U}_{\mathcal{D}}$. Entries in the list from low to high priority model the sequence of memory writes with the tail entry being the result of the latest memory update. Entries may be inserted at the tail end only, using procedure *InsertTail*(), and may be deleted using procedure *Delete*().

5.2 Implementation of Memory *Read* and *Write* Operations

The *Write* operation, shown as a procedure in Fig. 2, takes as arguments a memory list, a control expression denoting the contexts for which the write should be performed, and address set and data set expressions denoting the memory location and its desired contents, respectively. As the code shows, the write is implemented by simply inserting an element into the *tail* (high priority) end of the list, indicating that this entry should overwrite any other entries for this address. An optimized implementation of the *Write* operation will be presented after introducing the *Read* operation.

procedure *Write*(**List** *mem*, **CExpr** *c*, **ASExpr** *a*, **DSExpr** *d*)
/* Write data *d* to location *a* under control *c* */
$h \leftarrow Hard(c) \wedge Unique(a)$
$s \leftarrow Soft(c) \quad \vee \quad Hard(c) \wedge \neg Unique(a)$
$InsertTail(mem, \langle h, s, a, d \rangle)$

Fig. 2. Implementation of the *Write* operation

Two implementations of the *Read* operation are shown in Figures 3 and 4 as functions which, given a memory list and an address set expression, return a data set expression indicating the contents of this location. The purpose of both implementations is to construct a data set expression giving the contents of the memory location denoted by its argument address set expression. They do this by scanning through the list from lowest to highest priority. For each list entry, a Boolean expression *hard_match* is built that indicates the contexts for which the entry is hard (definite) and its (unique) address equals the read address *a*. Under these contexts, that element's data *ed* is selected. Else, under the contexts expressed by the Boolean expression *soft_match*, the approximate union of the element's data and the previously formed data is selected. Finally, under the contexts when both *hard_match* and *soft_match* are false, the previously formed data is kept.

Both implementations of the *Read* operation use $\mathcal{U}_{\mathcal{D}}$ as the default data set expression. The contexts for which *Read* does not find a matching address in the list are those for which the addressed memory location has never been accessed by a write. The data set expression $\mathcal{U}_{\mathcal{D}}$ is then returned to indicate that the location may contain arbitrary data.

function *Read*(**List** *mem*, **ASExpr** *a*) : **DSExpr**
/* Attempt to read from location *a* */
l ← *GenVectorBoolVars*()
address_containment ← $l \in a$
rd ← $\mathcal{U}_{\mathcal{D}}$
for each ⟨*eh*, *es*, *ea*, *ed*⟩ **in** *mem* from head to tail **do**
match ← $l \in ea \wedge$ *address_containment*
hard_match ← *match* ∧ *eh*
soft_match ← *match* ∧ *es*
rd ← *ITE*(*hard_match*, *ed*, *ITE*(*soft_match*, (*ed* $\widetilde{\cup}$ *rd*), *rd*))
rd ← $\widetilde{\bigcup}_{l \in a}$ *rd*
return *rd*

Fig. 3. First implementation of the *Read* operation

The difference between the two implementations is in the precision of the data retrieved from non-unique addresses. While the second implementation will return $\mathcal{U}_{\mathcal{D}}$ for the contexts when the read address *a* is non-unique, the first implementation will try to extract finer data for the contents of the locations contained in *a*. It does so by building a table of data set expressions at each unique address which is a subset of the read address *a*. This is done by introducing a vector of new Boolean variables *l*, which are used for indexing all the unique addresses that are contained in the read address set expression *a*. After scanning the list, these index address variables are existentially quantified from the bit-level low and high encodings of the retrieved data set expression *rd*. This merges the data set expressions corresponding to the contents of every unique address within *a*.

A useful optimization of the indexing is to introduce as many new variables in *l* as there are non-unique bits (i.e., whose low and high encodings are not complements) in the read address set expression *a*. Then, in forming the Boolean expression *match*, the unique bits of *a* will be required to be equal to the corresponding bits of *ea*. Finally, the existential quantification in $\widetilde{\bigcup}_{l \in a}$ *rd* is done only over the index variables used.

The second implementation of *Read* is designed to be precise only in the contexts when the argument address is unique, and to return $\mathcal{U}_{\mathcal{D}}$ otherwise. However, because of its fewer calculations, it requires less memory and CPU time. The expression *soft_match* is defined so that for any list entry, whose address overlaps the read address *a*, the approximate union of the entry's data set expression and the previously formed data set expression is selected. Note that in the contexts when the currently examined list element is hard, as determined by *eh*, we require that the element's address does not equal the read address (so that it is a proper subset of it). This ensures that the Boolean expressions for *hard_match* and *soft_match* will not be true simultaneously.

function *Read*(**List** *mem*, **ASExpr** *a*) : **DSExpr**
/* Attempt to read from location *a* */
rd ← $\mathcal{U}_{\mathcal{D}}$
if ¬*Valid*(¬*Unique*(*a*)) **then**
for each ⟨*eh*, *es*, *ea*, *ed*⟩ **in** *mem* from head to tail **do**
hard_match ← *eh* ∧ (*ea* = *a*)
soft_match ← (*es* ∨ *eh* ∧ ¬(*ea* = *a*)) ∧ *Overlap*(*ea*, *a*)
rd ← *ITE*(*hard_match*, *ed*, *ITE*(*soft_match*, (*ed* $\widetilde{\cup}$ *rd*), *rd*))
return *rd*

Fig. 4. Second implementation of the *Read* operation

The difference between the two implementations of *Read*() can be illustrated with the following example. Suppose that the list for memory *mem* was initially empty and then updated with *Write*(*mem*, 1, ⟨0, 0⟩, ⟨1, 1⟩) and *Write*(*mem*, 1, ⟨0, 1⟩, ⟨1, 0⟩). Then *Read*(*mem*, ⟨0, X⟩), will return ⟨1, X⟩ when using the first implementation of the function, but ⟨X, X⟩ when using the second one. The work of the first implementation can be viewed as building a table that maps unique addresses contained in the read address to data set expressions, and then finally merging these data set expressions. In the example, the table will associate address ⟨0, 0⟩ with data ⟨1, 1⟩, and the address ⟨0, 1⟩ with data ⟨1, 0⟩, so that merging the data will give ⟨1, X⟩ as the final result.

procedure *Write*(**List** *mem*, **CExpr** *c*, **ASExpr** *a*, **DSExpr** *d*)
/* Write data *d* to location *a* under control *c* */
h ← *Hard*(*c*) ∧ *Unique*(*a*)
s ← *Soft*(*c*) ∨ *Hard*(*c*) ∧ ¬*Unique*(*a*)
/* Optional optimization */
overlap ← **false**
for each ⟨*eh*, *es*, *ea*, *ed*⟩ **in** *mem* **do**
if *Valid*((*eh* ∨ *es*) ⇒
(*ea* ⊆ *a*) ∧ [*h* ∨ *s* ∧ *eh* ∧ (*d* = $\mathcal{U}_{\mathcal{D}}$) ∨ *s* ∧ *es* ∧ (*ed* ⊆ *d*)]) **then**
Delete(*mem*, ⟨*eh*, *es*, *ea*, *ed*⟩)
else
if ¬*Valid*(¬(*d* = $\mathcal{U}_{\mathcal{D}}$)) **then**
overlap ← *overlap* ∨ (*eh* ∨ *es*) ∧ *Overlap*(*ea*, *a*)
if ¬*Valid*((*h* ∨ *s*) ⇒ ¬*overlap* ∧ (*d* = $\mathcal{U}_{\mathcal{D}}$)) **then**
/* Perform Write */
InsertTail(*mem*, ⟨*h*, *s*, *a*, *d*⟩)

Fig. 5. Optimized implementation of the *Write* operation

Based on the definition of the *Read* operation, an optimized version of the *Write* operation can be constructed as shown in Fig. 5. It removes any list elements that for

all contexts are either not selected, as determined by both *eh* and *es* being false simultaneously, or are overwritten by the new entry. The latter category can be subdivided into several classes:

1) Entries with a unique address, that are overwritten by a hard write (i.e., h is true, which implies that a is unique, so that ($ea \subseteq a$) will evaluate to true only for the contexts when *ea* is unique).

2) Entries with a unique address, as determined by *eh* being true, which are overwritten by a soft write (s is true) with data equal to $\mathcal{U}_{\mathcal{D}}$. In this case, reading from the current element's address *ea* will select the element's data *ed*, but will later also form the approximate union of the previously formed data with the new element's data $\mathcal{U}_{\mathcal{D}}$. Hence, $\mathcal{U}_{\mathcal{D}}$ will be returned, so that the current element's data will not affect the result.

3) Entries created by a soft write (*es* is true), whose address and data set expressions are subsets of those of the new entry, which is also the result of a soft write (s is true). Then, reading from an address, which is a subset of the current element's address *ea*, will select the approximate union of the previously formed data with the current element's data *ed*. However, since ($ea \subseteq a$) and s is true, when later scanning the new list element, the approximate union of its data d with the previosly formed data will obscure the effect of *ed*.

Another optimization is to form the Boolean expression *overlap* that will express the condition for the new element's address a overlapping any other element's address. In the case of no overlap, there is no point in inserting the new element when its data is $\mathcal{U}_{\mathcal{D}}$, as that will be identical with the initial state of location a. Finally, when both h and s are false simultaneously, there is no point in inserting the new entry, as it will never be selected.

Note that these optimizations need not be performed, as they are based on the way that the *Read* operation works. We could safely leave any overwritten element in the list and always insert the new one.

6 Incorporation into STE

Efficient modeling of memory arrays in STE requires that formulas of the form ($b \rightarrow (mem[a] = d)$), where b is a Boolean expression, a is an address set expression, d is a data set expression, and *mem* is a memory array, be incorporated into the STE algorithm. When such formulas occur in the antecedent, they should result in treating d as the data of memory location a, given contexts b, and are processed by procedure *AssertMem*(), presented in Fig. 6. OK_A, the Boolean expression indicating the absence of an antecedent failure, is updated with the condition that either b is false, or else the asserted data d is neither more general, nor incompatible with the data currently at a.

Similarly, when such formulas occur in the consequent, they should result in checking that the data at location a is neither more general, nor incompatible with the given data d under contexts b. These formulas are processed by procedure *CheckMem*() - see Fig. 7.

procedure *AssertMem*(**List** *mem*, **BExpr** *b*, **ASExpr** *a*, **DSExpr** *d*)
/* Determine conditions under which location *a* was asserted to data *d* given contexts *b*, and reflect them on OK_A, the Boolean expression indicating the absence of an antecedent failure */
$rd \leftarrow Read(mem, a)$
$OK_A \leftarrow OK_A \wedge (b \Rightarrow (d \subseteq rd))$
if $\neg Valid(b \Rightarrow (d = rd))$ **then**
$c.h \leftarrow b$
$c.l \leftarrow \neg b$
$Write(mem, c, a, d)$

Fig. 6. Implementation of the STE procedure *AssertMem*

procedure *CheckMem*(**List** *mem*, **BExpr** *b*, **ASExpr** *a*, **DSExpr** *d*)
/* Determine conditions under which location *a* was checked to have data *d* given contexts *b*, and reflect them on OK_C, the Boolean expression indicating the absence of a consequent failure */
$rd \leftarrow Read(mem, a)$
$OK_C \leftarrow OK_C \wedge (b \Rightarrow (rd \subseteq d))$

Fig. 7. Implementation of the STE procedure *CheckMem*

7 Experimental Results

Experiments were performed on the pipelined addressable accumulator shown in Fig. 8. The pipeline register `Hold` separates the execution and the write back stages of the pipeline. The control logic stores the previous address and compares it with the present one at the `Addr` input. In case of equality, the control signal of the multiplexor is set so as to select the output of the `Hold` register. Hence, data forwarding takes effect. For a more detailed description of the circuit and its specifications, the reader is referred to [7][11].

For the experiments with the EMM, the dual-ported register file is removed from the circuit. The software interface ensures that a *Read* operation takes place relative to `phi1` and a *Write* operation takes place relative to `phi2`, according to the register file connections shown in Fig. 8.(b).

The specifications necessary for verifying the pipelined addressable accumulator, are presented in (14), (15), and (16). Note that `Reg[`i`]` and `Reg[`j`]` in (15) and (16), respectively, are instances of *symbolic indexing* [1]. We construct the antecedents by first defining the operation of the two phase clocks. Shorthand notation for the possible value combinations of the clocks is presented next:

$Clk01 \doteq$ (`phi1` = 0) $\wedge$ (`phi2` = 1),
$Clk00 \doteq$ (`phi1` = 0) $\wedge$ (`phi2` = 0),
$Clk10 \doteq$ (`phi1` = 1) $\wedge$ (`phi2` = 0).

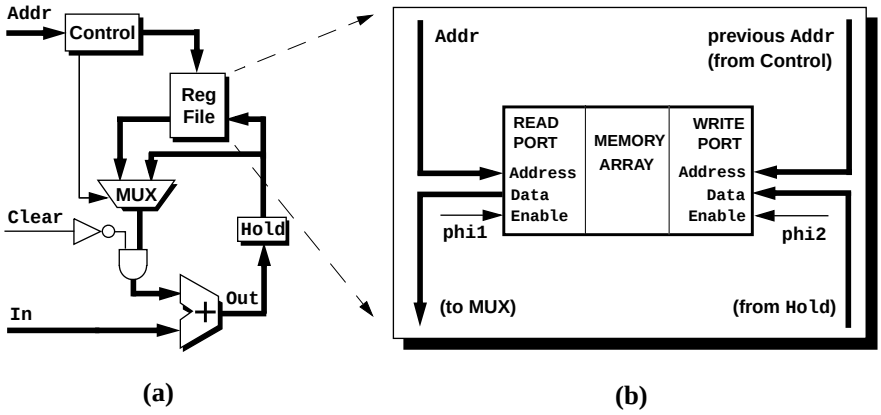

Fig. 8. (a) The pipelined addressable accumulator; (b) the connections of its register file when replaced by an EMM. The thick lines indicate buses, while the thin ones are of a single bit

The clocking behavior of the entire circuit over 4, 8, and 12 time periods, respectively, is described by:

$$Clocks\_4 \doteq Clk01 \wedge \mathbf{N}(Clk00) \wedge \mathbf{N}^2(Clk10) \wedge \mathbf{N}^3(Clk00),$$

$$Clocks\_8 \doteq Clocks\_4 \wedge \mathbf{N}^4(Clocks\_4),$$

$$Clocks\_12 \doteq Clocks\_4 \wedge \mathbf{N}^4(Clocks\_4) \wedge \mathbf{N}^8(Clocks\_4).$$

The first assertion (14) verifies that the `Hold` register can be initialized with data from the input `In` of the pipelined addressable accumulator. The next time operator **N** positions the constraints on the circuit and the desired responses that should follow relative to the phase clocks, given the timing details of the implementation.

$$Clocks\_8 \wedge \mathbf{N}^2((\texttt{Clear} = 1) \wedge (\texttt{Addr} = i) \wedge (\texttt{In} = a)) \stackrel{\text{LEADSTO}}{\Rightarrow} \mathbf{N}^4(\texttt{Out} = a) \wedge \mathbf{N}^5(\texttt{Hold} = a) \quad (14)$$

The second assertion (15) verifies the adder in the pipelined addressable accumulator. The `Hold` register and location i of the register file are initialized in such a way, that if the circuit is correct, the second input to the adder will have the symbolic data set expression b, while its external input has data set expression a. The expected response is that the output `Out` of the adder will get the data set expression $a + b$, and so will the `Hold` register.

$$Clocks\_12 \wedge \mathbf{N}^2(\texttt{Addr} = k) \wedge \mathbf{N}^5(i == k \rightarrow \texttt{Hold} = b) \wedge \mathbf{N}^6((\texttt{Clear} = 0) \wedge (\texttt{Addr} = i) \wedge (\texttt{In} = a) \wedge (i \mathrel{!}= k \rightarrow \texttt{Reg}[i] = b)) \stackrel{\text{LEADSTO}}{\Rightarrow} \mathbf{N}^8(\texttt{Out} = a + b) \wedge \mathbf{N}^9(\texttt{Hold} = a + b) \quad (15)$$

The last assertion (16) verifies that the register file can maintain its state in the

pipelined addressable accumulator.

$$Clocks\_12 \wedge \mathbf{N}^2(i \mathrel{!}= j \rightarrow \texttt{Addr} = k) \wedge \mathbf{N}^5((i \mathrel{!}= j \wedge j == k) \rightarrow \texttt{Hold} = b) \wedge$$
$$\mathbf{N}^6((i \mathrel{!}= j \rightarrow \texttt{Addr} = i) \wedge ((i \mathrel{!}= j \wedge j \mathrel{!}= k) \rightarrow \texttt{Reg}[j] = b))$$
$$\overset{\text{LEADSTO}}{\Rightarrow} \mathbf{N}^{10}(i \mathrel{!}= j \rightarrow \texttt{Reg}[j] = b) \qquad (16)$$

The experiments were performed on an IBM RS/6000 43P-140 with a 233MHz PowerPC 604e microprocessor, having 512 MB of physical memory, and running AIX 4.1.5. Table 2 shows our experimental results for the pipelined data path when verified with two memory models: the transistor-level model (TLM) and the EMM. The latter uses the first (EMM_1) or the second (EMM_2) implementation of the *Read* operation, presented in Section 5. The last three columns of each category contain the ratios of the corresponding quantities.

N	*w*	CPU Time (s)						Memory (MB)					
		TLM	**EMM_1**	**EMM_2**	**TLM / EMM_1**	**TLM / EMM_2**	**EMM_1 / EMM_2**	**TLM**	**EMM_1**	**EMM_2**	**TLM / EMM_1**	**TLM / EMM_2**	**EMM_1 / EMM_2**
16	16	337	45	44	7.5	7.7	1.0	4.2	2.3	1.7	1.8	2.5	1.4
	32	676	88	86	7.7	7.9	1.0	7.3	3.3	2.1	2.2	3.5	1.6
	64	1353	173	169	7.8	8.0	1.0	13.6	5.4	2.9	2.5	4.7	1.9
	128	2716	343	337	7.9	8.1	1.0	26.3	9.5	4.7	2.8	5.6	2.0
32	16	635	51	49	12.5	13.0	1.0	8.2	3.1	1.9	2.6	4.3	1.6
	32	1265	98	93	12.9	13.6	1.1	15.3	4.9	2.5	3.1	6.1	2.0
	64	2538	196	184	12.9	13.8	1.1	29.5	8.6	3.7	3.4	8.0	2.3
	128	5077	392	374	13.0	13.6	1.0	57.7	15.8	6.2	3.7	9.3	2.5
64	16	1227	65	59	18.8	20.8	1.1	16.0	4.7	1.9	3.4	8.4	2.5
	32	2460	126	114	19.5	21.6	1.1	30.7	8.1	2.6	3.8	11.8	3.1
	64	4905	253	224	19.4	21.9	1.1	59.8	14.9	3.8	4.0	15.7	3.9
	128	9853	509	455	19.4	21.7	1.1	118.0	28.6	6.4	4.1	18.4	4.5
128	16	2423	101	87	24.0	27.9	1.2	31.6	7.9	2.3	4.0	13.7	3.4
	32	4867	203	170	24.0	28.6	1.2	61.6	14.5	2.6	4.2	23.7	5.6
	64	9659	405	337	23.8	28.7	1.2	121.1	27.7	4.0	4.4	30.3	6.9
	128	18990	830	691	22.9	27.5	1.2	241.7	54.0	6.6	4.5	36.6	8.2

Table 2. Experimental results for memories with *N* addresses of *w* bits each

As can be seen, both the EMM_1 and the EMM_2 outperform the TLM. In the case of EMM_2, a 8-29× speedup and a 3-37× reduction in memory were obtained, with the EMM_2 advantage increasing with both dimensions of the memory array. EMM_1 has a comparable performance in terms of CPU time, but requires up to 8× more memory. The advantage of EMM_2 over EMM_1 increases with both dimensions of the memory

array - the more precise calculations of EMM_1 come at a premium. The asymptotic growth of time and memory is summarized in Table 3.

Criterion	TLM	EMM_1	EMM_2
Time(*N*)	linear	sublinear	sublinear
Time(*w*)	linear	linear	linear
Memory(*N*)	linear	sublinear	sublinear
Memory(*w*)	linear	linear	sublinear

Table 3. Asymptotic growth comparison of the CPU time and memory as a function of the number of addresses *N* and data bits *w* for the three memory models

Hence, the new method for efficient modeling of memory arrays has proven to be extremely promising. It will enable the symbolic ternary simulation of memory arrays far larger than previously possible.

References

1. D. L. Beatty, R. E. Bryant, and C.-J. H. Seger, "Synchronous Circuit Verification by Symbolic Simulation: An Illustration," *Sixth MIT Conference on Advanced Research in VLSI*, 1990, pp. 98-112.
2. R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," *ACM Computing Surveys*, Vol. 24, No. 3 (September 1992), pp. 293-318.
3. R. E. Bryant, D. E. Beatty, and C.-J. H. Seger, "Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation," *28th Design Automation Conference*, June 1991, pp. 297-402.
4. J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential Circuit Verification Using Symbolic Model Checking," *27th Design Automation Conference*, June 1990, pp. 46-51.
5. J. R. Burch, E. M. Clarke, and D. E. Long, "Representing Circuits More Efficiently in Symbolic Model Checking," *28th Design Automation Conference*, June 1991, pp. 403-407.
6. J. R. Burch, and D. L. Dill, "Automated Verification of Pipelined Microprocessor Control," *CAV '94*, D. L. Dill, ed., LNCS 818, Springer-Verlag, June 1994, pp. 68-80.
7. A. Jain, "Formal Hardware Verification by Symbolic Trajectory Evaluation," Ph.D. thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, August 1997.
8. M. Pandey, "Formal Verification of Memory Arrays," Ph.D. thesis, School of Computer Science, Carnegie Mellon University, May 1997.
9. M. Pandey, and R. E. Bryant, "Exploiting Symmetry When Verifying Transistor-Level Circuits by Symbolic Trajectory Evaluation," *CAV '97*, O. Grumberg, ed., LNCS 1254, Springer-Verlag, June 1997, pp. 244-255.
10. C.-J. H. Seger, and R. E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Formal Methods in System Design*, Vol. 6, No. 2 (March 1995), pp. 147-190.
11. M. Velev, R. E. Bryant, and A. Jain, "Efficient Modeling of Memory Arrays in Symbolic Simulation,"[2] *CAV '97*, O. Grumberg, ed., LNCS 1254, Springer-Verlag, June 1997, pp. 388-399.

2. Available from: `http://www.ece.cmu.edu/afs/ece/usr/mvelev/.home-page.html`