1. Introduction

Content Addressable Memories (CAMs) play an important role in many modern digital systems. CAMs are widely used wherever fast parallel search operations are required. Some examples of CAMs found on modern processors are translation-lookaside buffers (TLBs), branch prediction buffers, branch target buffers and cache tags. CAMs have also been used in such applications as data compression, data-base accelerators, and network routers.

CAMs in microprocessors are usually custom designed at the transistor level, as these circuits are often in the critical path of a chip and it is necessary to optimize area and performance. These circuits often include self timed components and other complex forms of circuitry, and they typically have complex internal timing. For these reasons it becomes necessary to verify such designs at the transistor level. All the designs in this paper have been verified at the transistor level using a switch-level model.

While work on verification of memory arrays has been reported in [2], [6], and [3], there has been little published on the particular needs of CAMs. In [2, pp. 102], Bryant comments on the difficulty of CAM verification as follows: "...Other classes of memory designs can also be verified by simulating a linear, or nearly-linear number of patterns. ... On the other hand, content-addressable memories do not seem to fit into this class, since it is not easy to identify where a particular datum will be stored."

In this paper, we describe how we leveraged the formal verification technique of symbolic trajectory evaluation (STE), along with new Boolean encoding techniques, to verify CAMs. The new encodings were needed to contain the exponential growth in the space requirements with increasing CAM sizes, which could occur with a naive use of variables in STE. Our work shows that we were able to solve this problem and formally verify these types of circuits, with very modest space and time requirements.

In the remainder of this paper we discuss background material on STE and CAMs (Section 2), and then describe experiments done on small, generic CAMs in which we perfected the needed Boolean encodings (Section 3). Finally, we describe how we used our techniques, with success, on two complex CAMs (Sections 4 and 5) from a recent PowerPC processor. This verification was carried out at the joint Motorola-IBM PowerPC design center, Somerset, located in Austin, Texas. In our work, we utilized the Voss STE system [8].

2. Background

2.1. Symbolic trajectory evaluation

STE [7] is a ternary symbolic simulation based technique for verifying behaviors of a system over bounded, finite time intervals. Specifications are trajectory assertions of the form \([\text{Antecedent} \Rightarrow \text{Consequent}]\), where \(\text{Antecedent}\) and \(\text{Consequent}\) are trajectory formulae. Intuitively, the antecedent defines an initial setting and a stimulus pattern for the circuit nodes, while the consequent defines the expected response.

The basic element of a trajectory formula (TF) is a simple predicate, e.g., \((\text{node}_i = 0)\), which states that node\(_i\) of a circuit contains the value 0 at the present time. Using conjunction, case restriction and a next-time operator, trajectory formulas can be constructed from the simple predicates. If \(G_1\) and \(G_2\) are TFS, then their conjunction \(G_1 \wedge G_2\) is also a TF. If \(G\) is a TF, then \(\neg G\) is a TF, where \(\neg\) is the next-time temporal operator, and \(\neg G\) means that \(G\) holds in the next time step. Finally, if \(G\) is a TF, and \(E\) is a Boolean expression, then \((\text{when} \ (E) \ G)\) is a TF, where \text{when}\ is the domain restriction operator. \((\text{when} \ (E) \ G)\) means that \(G\) must hold when \(E\) is true. This simple logic is sufficient to verify the class of systems, such as arrays, for which functionality can be partitioned into a set of operations each of which update the system.
state in a deterministic manner. Given an assertion \([A \implies C]\), the circuit is simulated with ternary symbolic simulation patterns derived from the antecedent \(A\). During simulation, at each time step the circuit state is checked against the expected response specified in \(C\). The ternary symbolic values generated in the process are represented by pairs of Ordered Binary Decision Diagrams (OBDDs).

The temporal logic of STE is quite restricted as compared to the temporal logics of model checkers like SMV [4]. However, the use of such a logic obviates the need to represent a system’s transition relation and calculate its reachable state set, two very expensive operations. Because of this, STE excels in the verification of large, data intensive systems, such as memory arrays, having tens of thousands of state holding elements.

Recently a methodology for application of STE was developed [1] and we have adapted this for verification of CAMs. In this methodology, the desired system behavior is specified as a set of assertions over abstracted system space. Each abstract assertion, which is of the form \([A \implies C]\) describes how the system operations transform the abstract system space. Intuitively, the abstract assertion’s antecedent, \(A\), specifies the current abstract state and inputs, and the consequent, \(C\), specifies the abstract state and outputs after a system operation. In addition to the abstract assertions, the user provides an implementation mapping, giving the correspondence of the abstract state to the nodes in the transistor-level circuit, and giving the timing of signal transitions. Given this mapping, the abstract specification is mapped into a set of trajectory assertions, to be checked by the STE decision procedure. Details of this methodology, the reader is referred to Beatty’s thesis [1].

The abstract assertions shown in this paper use a pseudo-code notation. The actual assertions were written in the FL language, which is the functional language front-end to the Voss verifier.

2.2. An Illustration of the Methodology

To illustrate the methodology described above, consider the abstract assertion below. It describes a tag write operation for the CAM circuit in section 2.3.

\[
\{ \text{op} = \text{tagwrite} \} \land (T[d]adr = adr) \land (Tagin = newtag) \land (T[i] = tag) \rightarrow \text{LEADSTO} \]

\[
(\text{when}(i \neq adr)(T[i] = tag)) \land (\text{when}(i = adr)(T[i] = newtag))
\]

The antecedent (from left to right) states that a tag write operation is being done to address \(adr\) with value \(newtag\), and in the initial state of the verification, the \(i^{th}\) tag entry contains the symbolic value \(tag\). The consequent specifies that the addressed tag entry is changed to \(newtag\), and all other tag entries are unchanged.

The implementation mapping relates the abstract state of each “phrase” in the abstract assertion to concrete signal timing and evaluations on actual circuit nodes. It captures implementation details such as that two phase clocking is used, and that the \(T\text{write}\) signal is asserted, and the tag data and addresses provided, with appropriate setup and hold times, when \(Ph1\) is high. For an example of such details, see [6, pp. 651].

Note that the variables \(i\) and \(j\) in the abstract assertion above are used as array indices. The implementation mapping represents each of them in binary form as a vector of symbolic Boolean variables. From the phrase, \((T[i] = tag)\), in the antecedent of the abstract assertion, the implementation mapping will initialize each tag storage node \(k\) in

![Figure 1: Content Addressable Memory. Tag size = \(t\), Number of entries = \(n\), Data size = \(d\)](image-url)

the CAM with a symbolic ternary function, \(f_{x}(i, tag)\), which returns \(tag\), when \(i = k\), and \(X\) otherwise (\(X\) represents the “unknown” or “don’t care” value of switch-level simulation). This technique, called symbolic indexing, is critical to the efficiency of STE on memory-based circuits [1]. It is responsible for reducing the number of variables in an STE verification to a number logarithmic in the number of array locations.

2.3. A CAM design

Generally, CAMs employ as an identifier a bit field called a tag, each tag identifying a particular data entry stored in the array. CAMs vary depending upon data and tag size, techniques to read and write contents and mark contents as valid, tag masking fields, etc. In spite of all this diversity, CAMs all have in common the associative read capability. The associative read operation consists of searching, in parallel, all tags in the CAM to determine if there is a match to a particular tag of interest, and then sending the associated data entry to an appropriate read port of the memory.

The high-level design shown in Figure 1 is a very basic CAM. We implemented this design as an experimental vehicle for finding better Boolean encodings for CAM verification (see Section 3). This design has \(n\) tag entries, \(T[0], T[1], \ldots, T[n - 1]\). Corresponding to each tag entry, \(T[i]\), there is a data entry \(D[i]\). The most distinctive operation of this circuit is the associative read operation. In this operation \(Tagin\) is compared in parallel with all the \(T[i]\) tag entries, and if there is a match on the \(i^{th}\) tag entry, then \(Hit\) rises, and \(D[i]\) appears at \(Dataout\). If there is no match to \(Tagin\), i.e., a miss, \(Hit\) remains low (and, the surrounding circuitry would ignore \(Dataout\)). We have implemented this design as a transistor-level netlist using a tool called cmu-netlist. Each \(n\)-bit tag consists of \(n\) tag cells. Each tag cell contains 9 transistors and its design is based on the one in [9, pp.590].

It is an assumption that, among valid tag entries there would be at most one tag that would match \(Tagin\). This property, the at most one tag match property is an important system invariant. However, it is usually not enforced in hardware. Rather, CAMs generally depend upon surrounding circuitry, or the software manipulating the entire chip, to maintain this invariant. For example in the PowerPC BAT array [5], the responsibility of maintaining the invariant is with the operating system. In Section 3, we show how we used this invariant to efficiently verify a CAM. Even when the circuit design is enhanced to handle multiple matches, we can verify CAMs efficiently using techniques

\(^{1}\)In some instances CAMs also have an associative write capability. The PowerPC Branch Target Address Cache circuit is one such example and it is described in Section 4.
based on those outlined here. However, due to space limitations, we will not discuss that in this paper.

3. CAM properties and CAM encodings

Below we show how a well chosen encoding can dramatically reduce the number of variables, and therefore the number of OBDD nodes, required for the verification.

3.1. CAM Encodings

We will discuss the CAM encoding problem in the context of verifying the associative read operation of CAMs. We will refer to a generic CAM modeled after that of Figure 1, in Section 2.3.

The most obvious approach to verifying the associative read operation is to introduce a Boolean variable for each bit of state in the $T[i]$ and $D[i]$ tag and data entries. We illustrate this below with an example trajectory assertion. Assume the number of CAM entries, $n$, is 3. Let $t_0$, $i_0$, and $t_1$ and $i_1$ be vectors of Boolean variables of size $t$, the width of the $T[i]$ entries. Let $d_0$, $d_1$, and $d_2$ be vectors of Boolean variables of size $d$, the width of the $D[i]$ entries. The following assertion specifies the associative read operation under these conditions.

\[
\begin{align*}
\text{when}(\text{match}[0] \land \text{match}[1] \land \text{match}[2] \land (i = 0)) & \land (HIT = 1) \land (dataout \neq \text{data}).
\end{align*}
\]

This encoding requires only $(n+1) \cdot t+d+\lceil log_2 n \rceil$ Boolean variables. We call this the plain encoding. For identical data and tag sizes, the number of variables goes down by half, as compared to the full encoding. However, as later results will show, with increasing $n$, memory requirements can still grow rapidly with the plain encoding scheme. So we must improve on it.

We can reduce the number of variables further, by taking advantage of the at most one tag match system invariant. Let $Tag$ be $\text{tin} = \{\text{tin}_1, \text{tin}_2, ..., \text{tin}_n\}$. In order that the tag entry $T[0]$ not match $\text{tin}$, it should be one of the following $t$ ternary vectors: $\{\text{time}_1, X, ..., X\}$, $\{X, \text{time}_2, X, ..., X\}$, $\{X, X, ..., \text{time}_n\}$. The position at which the tag in $T[0]$ is unequal can be encoded by, $p$, a vector of $\lceil \log_2 t \rceil$ variables. So the condition that $T[0]$ is not equal to $\text{tin}$ can be written as

\[
\begin{align*}
\text{when}(p = 0) & \land (T[0] = \text{time}_1) \land (T[0] = \text{time}_2) \land \text{when}(p = 1) \land (T[0] = \text{time}_1) \land \text{when}(p = 1) \land (T[0] = \text{time}_2) \land \text{when}(p = 1) \land (T[0] = \text{time}_n) \\
\end{align*}
\]

We abbreviate this as $3 \beta_i T[0][p] = \lnot \text{tin}_i p$, i.e. there exists a $p$ such that the $p^{th}$ bit position there is a mismatch between $\text{tin}$ and the tag entry $T[0]$.

We now verify the associative read operation in two parts. First, we verify the case where no CAM entries match the input tag, and then we verify the case where the $i^{th}$ entry does match the input tag. For the case where no hit occurs the new assertion is:

\[
\begin{align*}
\text{when}(\text{matchonly}[0] \land (HIT = 1) \land (\text{dataout} = \text{data})) & \land \text{when}(\text{matchonly}[1] \land (HIT = 1) \land (\text{dataout} = \text{data})) \\
\end{align*}
\]

The first line of the antecedent specifies that an associative read is being done and the input data is $\text{tin}$. The second line specifies that the three tag registers initially contain $i_0$, $i_1$, and $i_2$. The three data registers are specified as initially containing $d_0$, $d_1$, and $d_2$. To simplify the assertion of the following Boolean functions, match[0] = $(\text{tin} = i_0)$, match[1] = $(\text{tin} = i_1)$, match[2] = $(\text{tin} = i_2)$, match[0,1] = $\lnot \text{match}[0] \lor \text{match}[1] \lor \text{match}[2]$, match[0,1,2] = $\lnot \text{match}[0] \land \text{match}[1] \land \text{match}[2]$, match[0,1,2] = $\lnot \text{match}[0] \land \text{match}[1] \land \text{match}[2]$. The first line in the consequent checks that there are no matching entries in the CAM. The second line checks for $\text{HIT}$ and $\text{Dataout}$ when only the first entry matches. Note that we do not check for conditions inconsistent with the at most one tag match system invariant. For example, we do not check for what happens if $(\text{tin} = i_0)$ and $(\text{tin} = i_1)$ are both true. A total of $(t + d) n + t$ Boolean variables are needed for this assertion. We call this encoding, where every circuit state bit has a corresponding Boolean variable, the full encoding.

We can reduce the variable count, however, by using symbolic indexing. At this point we will use it just for the data entries. To effect this, the antecedent should be changed to contain $(D[i] = \text{data})$ instead of $(D[0] = d_0 \land (D[1] = d_1) \land (D[2] = d_2))$. $\text{data}$ is a vector of Boolean variables $d$ bits wide, and $i$ is a vector of Boolean variables $\lceil \log_2 n \rceil$ bits wide. The consequent is also changed. Line 2 of the consequent is changed to (lines 3 and 4 are changed similarly):

\[
\begin{align*}
\text{when}(\text{match}[0] \land \lnot \text{match}[1] \land \lnot \text{match}[2] \land (i = 0) \land (HIT = 1) \land (\text{dataout} = \text{data}).
\end{align*}
\]

This encoding requires only $(\log_2 n \cdot n + \log_2 t + t)$ Boolean variables, a substantial savings over the two earlier encodings. We refer to this encoding as the CAM encoding. As will be seen in section 3.3, verification of even moderate sized CAMs would be intractable without an encoding at least as efficient as the CAM encoding.

It is instructive to compare the number of Boolean variables required for the different encodings, with the number required for representing a transition relation. For a 16 entry CAM, with 16 bit tag and data sizes, the number of Boolean variables required for the full, plain and CAM encodings are 528, 292, and 100 respectively. For the transition relation the required number of Boolean variables is over 1024, which is double the number of state elements.

\[\text{Some parts of the assertion necessary for verification thoroughness, e.g. that the tag}
\text{and data bits are unchanged on a read, have been omitted.}\]
3.2. Experimental Results and Discussion

In Figures 2 and 3 we have plotted the results from verification of

different size CAMs, using the CAM encoding and the plain encoding. The

full encoding is not included here, as it usually performs much

worse than the other two encodings. We have plotted the memory taken

by the OBDDs generated when verifying the associative read operation. All

other verified CAM operations take less space, and have not been

included here. The OBDD variable ordering for the experiments was

carefully chosen, so that, as much as possible, we would avoid unfair

comparisons between the two encodings. For each encoding, we chose

an initial variable ordering that, from our understanding of the circuit,

function, would result in small OBDDs. Upon running STE with the

initial variable ordering, the OBDD package dynamically reordered

some of the variables. We used this reordering information to improve

our understanding of the variable interaction and further tuned the

variable ordering to minimize the OBDD sizes before running STE

again.

Figure 2 shows how the OBDD sizes for the plain and CAM en-

coding vary for CAMs with varying associativities (tag and data sizes

are constant). As the graph shows, there is a dramatic difference in

the space taken by the two encoding approaches. As the number of

tag entries increases, the plain encoding requires substantially more

memory than the CAM encoding. Many TLBs are highly associative,

and for such circuits the plain encoding approach will clearly not work.

These results motivated us to use CAM encodings in all our further

CAM verifications (Sections 4 and 5).

In Figure 3, we have shown the OBDD size trends for the two

encodings when the tag size changes (others parameters remaining

constant). The space savings with the CAM encoding are similar to

that in Figure 2. Although these results are not as dramatic as those of

Figure 2, they show that use of the CAM encoding still results in

at least an order of magnitude space savings, as compared to the plain

encoding.

We can explain the trends in these results in terms of circuit struc-
ture, and the interactions of the circuit Boolean functions. Consider

the 3-entry CAM described in Section 3.1, and let the tag size be k.

In this design the \( i^{th} \) match line, \( \text{match}_i \) contains the result of a

match between the tag input and the \( i^{th} \) tag entry. When the plain

encoding is used, the \( i^{th} \) match line contains the result of the match

between the input tag \( tagin \) and the \( i^{th} \) tag entry \( tag_j \). After the com-

pare, the Boolean function associated with \( \text{match}_i \) is

\[
\text{match}_i = \neg((tagin[k-1] \oplus \text{tag}[k-1] \lor \ldots \lor (\text{tagin}[0] \oplus \text{tag}[0])).
\]

The value on each dataout line, \( \text{Dataout}_j \), is a function of all the func-
tions on the match lines, bit \( d[j] \) (used in the associative read

assertion), and \( \hat{0} \). So, potentially there are interactions among all the

Boolean variables associated with the tag and data entries and the tag

input.

When the CAM encoding is used, the antecedent fragment (Sec-
tion 3.1) specifying the 0th tag entry is given by (when \( i = 0 \) (\( T[0] =

\text{tin} \lor (\exists p_i) T[0,p_i] = \neg \text{tin} \lor 0 \)). When the tag input

is \( \text{tin} \), then the 0th tag entry matches only if \( i = i_{10} = 0 \). This

is the information conveyed by the Boolean function on \( \text{match}_0 \).

Therefore, \( f_{\text{match}_0} = i_1 \cdot i_0 \). So, the functions on the dataout lines

depend only on the Boolean variables in \( d \), and \( \hat{0} \). Thus, the use of

CAM encoding minimizes the variable interaction and this results in

substantial space savings, especially when the number of entries is

large. We have not shown the running times of the assertions here,

most of which finish in a few seconds on a RS/6000™ model 250

workstation.

4. PowerPC Branch Target Address Cache Array

The Branch Target Address Cache (BTAC) array is part of the

speculative instruction fetch mechanism on some PowerPC processors.

The particular BTAC we verified, from a recent PowerPC processor,

was a 64 entry content addressable memory, where each entry consists

of a 30-bit tag and a 32-bit data part (Figure 4). The branch address

is used to access the BTAC array, which contains the target address of

previously executed branch instructions that are predicted to be taken.

The primary task of this unit is an associative read operation, i.e.,
given a branch instruction address presented at the \( \text{rdofadr} \) input,
to determine if there is a matching tag entry, and if so give out the

corresponding data entry, which is the branch target address. The

verification of this operation is similar to that of the CAM associative

read operation of Section 3. There are also a number of other operations

this unit performs, including reset, and initialization of its round-robin

register. Our discussion, however, will focus on the \textit{replace}, or \textit{CAM}

write operation.
4.1. BTAC Replace operation

In the replace operation, a TAG-DATA pair is updated with new values. The selection of an entry for updating is not necessarily based on the address of the entry, rather, it can also be based on a round-robin replacement policy. This operation is essentially a CAM write operation.

The first step in this operation is to select the entry to be replaced. An associative read is done on a tag value presented at read port 1, i.e., \( rd1_{\text{fadr}} \). This input tag is compared to all the stored tags in parallel, and if there is a match, \( \text{hit1} \) rises and the matching entry is updated with the new values at \( wr_{\text{fadr}} \) (the new tag) and \( btac_{\text{data_in}} \) (the new data). If there is no match, then a round-robin replacement policy is enforced. This replacement policy is implemented with a 64-bit round-robin register (right side of Figure 4) which is a one-hot encoded ring counter. The bit position in the ring counter which is 1 points to the BTAC entry to be replaced in the case of a miss on the address at \( rd1_{\text{fadr}} \). Irrespective of the value on \( \text{hit1} \), all entries which are not replaced remain unchanged.

Verification of the replace operation, required verifying a number of different cases, many of these similar to the memory write operation. One of the more interesting cases is that outlined above, when there is no hit on \( rd1_{\text{fadr}} \) and the TAG-DATA entry pointed to by the round-robin register is written to (and all other TAG-DATA entries are unchanged). This case is discussed below.

To verify this case, we encoded the TAG value to be unequal to the symbolic value \( tag \). In order to do this, we could use a CAM encoding where the \( i^{th} \) bit position of a TAG[] entry equals \( i\text{tag} \), and all other bit positions of the TAG[] entry are X. The problem with this is that if we have to show that TAG[] remains unchanged, then it is not sufficient to show that it still has its earlier value which is of the form \( <X,X,...,\text{tag},X,...> \). The bit positions which are X can change, and we would not be able to detect it, since X denotes an absence of information. Therefore, in the assertion below, we have a vector of symbolic values called \( val \), which we use to encode a value of the form \( <\text{val[0]},\text{val[1]},...\text{val[i-1]},\text{tag[i]},\text{val[i+1]},...\text{val[n-1]}> \). This value is unequal to \( tag \), but, we can also detect whether the value of TAG[] remains unchanged in an operation, since none of the bit positions contain X. In this manner, we verified that only the tag entry pointed to by the round-robin register was updated, and the rest remained unchanged.

4.2. Results

The most complex BTAC assertion takes 40MB of memory and 5 minutes to run, on a RS/6000 model 350 workstation. Of this 40MB, 24 MB is taken up by the OBDDs, and the remaining space is taken up by other run-time data structures. The total run time for all assertions was 20 minutes. All the BTAC assertions passed, and no bugs were uncovered in this circuit. If a more naive Boolean encoding had been used for the BTAC verification, the OBDD growth trends of Figures 2 and 3 predict that a memory of several GB, and a 32-bit address space, would not have been sufficient for this verification!

5. PowerPC Block Address Translation array

The PowerPC architecture includes a block address translation (BAT) mechanism which maps ranges of effective addresses larger than a single page into contiguous areas of physical memory [5]. Such areas are used for data not subject to normal virtual memory handling, such as a memory-mapped display buffer. This translation mechanism is implemented as an array consisting of software controlled registers.

The DBAT array implements the BAT translation mechanism for data memory references. It is a CAM containing 4 tag entries and 4 data entries. Each tag-data entry pair is organized as a pair of registers called the Upper DBAT Register and the Lower DBAT register (Figure 5). The two operations this array performs are the SPR ("special purpose register") operation, and the non-SPR operation. In the SPR operation, this array behaves like a register file where in a single clock cycle reads and writes are done on the Special Purpose Registers (SPRs) constituting the upper and lower DBAT registers.

In the non-SPR mode of operation, the DBAT array behaves like a CAM and it translates the 9 to 15 most significant bits of the logical address (bit 0 is the MSB) into the physical address. The remaining bits pass unchanged. In Figure 5, the incoming logical address (top 15 bits, i.e. \( EA(0:14) \)) is compared to the block effective page index (BEPI) entry. The block length field (BL) contains a 11-bit mask, used to determine which bits are to be compared. If the mask is all 0’s, then all 15 bits are compared, and the corresponding 15-bit data entry, the Block Real Page Number (BRPN), is sent out as the upper 15 bits of the physical address. If the mask entry is all 1’s, then only the top most 4 bits are compared, and a match only the top most 4 bits of the
also, for each register pair we need a distinct Boolean variable.
Over any of these 12 bit positions. This mismatch is expressed as
comparison is done over bits 0 through 11, and the mismatch can be
expressed symbolically as a symbolic vector,
the (12) legal values the mask bits can hold.

DBA T registers. In order to express that a register contains a data value
thermore, the bits masked out can be different for all four of the upper
entries. The system invariant specified in the PowerPC programming
environment manual [5] is that at most one DBAT entry should match
the incoming logical address. More details on this complex unit can
be found in [5]. While we have verified all the DBAT operations, here
we describe only the verification of some aspects of the interesting
“non-SPR” mode of operation.

5.1. DBAT non-SPR operation

In section 3.1 we described a way of encoding that a register
T[0] was not equal to a value tin. We abbreviated this by
\exists p. T[0][p] \neq tin\). This encoding does not work directly for
expressing a mismatch on an upper DBAT register because comparison
can be disabled on some selected register bits by the mask field.
Furthermore, the bits masked out can be different for all four of the upper
DBAT registers. In order to express that a register contains a data value
that does not match the incoming data, we needed to take into account
the (12) legal values the mask bits can hold.

Using 4 symbolic Boolean variables, \(m = m_1 m_2 m_3 m_0\), we created
a symbolic vector, \(\vec{M}\), to encode the 12 legal mask values. Given a
vector of symbolic Boolean variables, \(\vec{u} [0-14]\), all legal BEPI entries
may be expressed symbolically as \(\vec{u} [0-3] \| \vec{u} [4-14] \& \neg \vec{M}\), where
\(\|\) is the bitvector concatenation operator. The position of 1’s in a
mask indicates the BEPI bit positions which are not compared to
an incoming tag, tin. Therefore, if the mask is 00000000111, the
comparison is done over bits 0 through 11, and the mismatch can be
over any of these 12 bit positions. This mismatch is expressed as
\(\exists p. (0 \leq p \leq 11) \wedge T[0][p] = \neg tin\). Combining such information for
all the 12 legal mask values covers all possible cases of a tag mismatch.
Since every register pair can have a different mask, we need a separate
set of Boolean variables, \(\vec{m}\), for encoding the mask value for each pair.
Also, for each register pair we need a distinct Boolean variable, \(v\), to
indicate whether this entry is valid. Using this encoding, verification
of the associative read can be done in a manner similar to that described
earlier.

5.2. Results

We wrote two assertions for verifying the DBAT circuit, one for
SPR operation, and the other for non-SPR operation. On a RS/6000
model 350 workstation, peak memory requirements for running all
the assertions was 16.1 MB, and the total time was 15 minutes.
We also wrote an assertion for the non-SPR operation using the plain
encoding, to compare against these results. This encoding did not
work well. Even with many control signals set to non-symbolic values,
the memory required was over 100 MB!

We discovered two bugs in this circuit, both in the SPR mode
of operation. The first bug was that the signal, rpm_s, should have
been all 0’s, and was not. The second bug involved an incorrect
implementation of the signal, rpm_stat19. It is significant that these
bugs were discovered by running just one assertion specifying the SPR
operation. This is in contrast to the commonly accepted practice of
running a huge number of (non-symbolic) simulation vectors, often
for days, with no certainty that such corner cases will be brought out.

6. Conclusion

We have reported on new techniques to verify CAMs. It is based
on symbolic trajectory evaluation and new Boolean encoding tech-
iques. We have shown that our techniques avoid the OBDD space
explosion problem for CAMs, and the OBDD space scales linearly or
sub-linearly with increasing in various CAM parameters. Using these
techniques we have verified complex CAMs from a recent PowerPC
microprocessor. This work opens the way to the efficient verification
of numerous on-chip CAMs such as TLBs, cache tags and branch
target buffers.

References

with Application to Microprocessors, Ph.D. Thesis, published as
Technical report CMU–CS–93–190, School of Computer Science,
Carnegie Mellon University, August 1993.

Switch-Level Simulation,” IEEE Transactions on Computer-Aided
Design of Integrated Circuits and Systems, Vol.10, no.1,

Methodology using Formal Techniques,” in Proceedings of the
International Test Conference, 1996.

University, 1992.

[5] “PowerPC™ Microprocessor Family: The Programming Envi-

of PowerPC(TM) arrays using symbolic trajectory evaluation,”

evaluation of partially-ordered trajectories,” Formal Methods in

user’s guide,” Technical Report 93-45, Department of Computer
Science, University of British Columbia, 1993.

[9] N. Weste, K. Eshraghian, Principles of CMOS VLSI design, A

PowerPC and RS/6000 are trademarks of the International Business Ma-
nes Corporation used under license therefrom.