Vectorized Symbolic Model Checking of Computation Tree Logic for Sequential Machine Verification

Hiromi Hiraishi[†], Kiyoharu Hamaguchi[‡]
Hiroyuki Ochi[‡] and Shuzo Yajima[‡]
[†] Department of Information and Communication Sciences
Kyoto Sangyo University
Kita-ku, Kyoto 603, JAPAN
[‡] Department of Information Science, Kyoto University
Sakyo-ku, Kyoto 606, JAPAN

Abstract

The major goal of this paper is to clarify how large and practical sequential machines can be verified with the current most powerful supercomputers. The basic algorithm used is an implicit symbolic model checking algorithm, which is shown to be 100 times and 40 times more efficient in time and space than the conventional symbolic model checking algorithms. Based on the algorithm, a vectorized symbolic model checking algorithm, which is suitable for execution on vector processors, is also proposed. Some benchmark results show that it achieves about 6 $\sim$ 20 acceleration ratio and it can verify a 16 bit pipelined ALU with 4 word register file, which supports 16 arithmetic/logical operations, in around 12 minutes on a vector processor HITAC S-820/80.

1 Introduction

Various kinds of formal methods for automatic verification have been widely studied. Among them, the symbolic model checking approaches based on a branching time temporal logic called CTL (Computation Tree Logic) are one of the most efficient approaches [5, 6, 7]. It uses a Boolean characteristic function, which is efficiently represented and manipulated by using *Shared Binary Decision Diagrams (SBDD)*[1, 3], to express the state transition relation of a state machine explicitly.

The size of the SBDD representation of the characteristic function, however, is apt to become very large even if the size of the SBDD representation of the state transition functions of a sequential machine is small. In order to avoid this problem, new improved algorithms (we call them as implicit symbolic model checking) are proposed [4, 10, 15]. They do not use the Boolean characteristic function to represent the state transition relation explicitly. Instead, they use the state transition functions of a sequential machine directly without generating the characteristic function representing the state transition relation of a sequential machine explicitly.

Our major goal is to clarify how large and practical sequential machines can be verified based on the symbolic model checking algorithm of CTL by using one of the most advanced current supercomputers. Aiming at verification of sequential machines, we adopted a kind of implicit symbolic model checking algorithm of CTL. It is efficiently executed not only on supercomputers but also on the current conventional workstations.

First, we implemented both algorithms, i.e. explicit and implicit ones, on SPARC Station 1+ to see the effect of the implicit symbolic model checking algorithm. Experimental results show that the implicit version achieves up to 100 times and 40 times improvements in time and space respectively compared with the explicit version.

Next, we vectorized the implicit symbolic model checking algorithm so that it can be executed efficiently on a vector processor. Since the most time consuming parts of the symbolic model checking algorithm are manipulations of SBDD, we concentrated on vectorizing manipulations of SBDD [14]. Although many SBDD manipulators have been developed up to now, most of them are implemented on workstations [2, 13]. In order to handle much larger SBDD in a reasonable time, the use of parallel machines or connection machines is studied [12]. Their algorithms are based on depth first search recursive algorithms and it is difficult to vectorize such recursive algorithms for efficient execution on vector processors. Our vectorized algorithm is based on breadth first search algorithm, instead, to enjoy the power of vector processors.

We also implemented and evaluated our vectorized symbolic model checking algorithm of CTL on a vector processor HITAC S-820/80 at the University of Tokyo. It achieves 6 to 20 acceleration ratio and it takes about 12 minutes to verify a 16 bit pipelined ALU with 4 word register file which supports 16 arithmetic/logical operations.

This paper is organized as follows: Section 2 summarizes CTL, notations of sequential machines, and SBDD. Section 3 describes our symbolic model checking algorithm of CTL for sequential machine verification. Vectorization of our algorithm is discussed in section 4. In section 5 we explain the implementation of our algorithm and show some benchmark results. Section 6 concludes this paper.

2 Preliminaries

2.1 Computation Tree Logic

Computation Tree Logic (CTL)[8] is a branching time temporal logic. Let AP be a set of atomic propositions. Let p be an atomic proposition and η, ξ be CTL formulas. Then, $p, \neg\eta, \eta \vee \xi, EX\eta, EG\eta$ and $E[\eta \mathcal{U} \xi]$ are also CTL formulas.

The semantics of CTL is defined over a Kripke structure $K = (S, R, I)$, where S is a non-empty finite set of states; $R \subseteq S \times S$ is a total binary relation on S; $I : S \rightarrow 2^{AP}$ is an interpretation function which labels each state with a set of atomic propositions true at that state.

An infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ is called a *path* from s_0 if $(s_i, s_{i+1}) \in R$ for $\forall i \geq 0$. $\pi(i)$ denotes the i-th state of the sequence π (i.e. $\pi(i) = s_i$).

The truth-value of a CTL formula is defined at a state of a Kripke structure and $K, s \models \eta$ denotes that a CTL formula η hold at a state s of a Kripke structure K. If there is no ambiguity, we will omit K and just write as $s \models \eta$. The relation $\models$ is recursively defined as follows: $s \models p$ ($\in AP$) iff $p \in I(s)$; $s \models \neg\eta$ iff $s \not\models \eta$; $s \models \eta \vee \xi$ iff $s \models \eta$ or $s \models \xi$; $s \models EX\eta$ iff there exists some next state s' of s (i.e. $(s, s') \in R$) such that $s' \models \eta$; $s \models EG\eta$ iff there exists some path π on K starting from the state s such that $\pi(i) \models \eta$ for $\forall i \geq 0$; $s \models E[\eta\mathcal{U}\xi]$ iff there exists some path π on K starting from the state s such that $\exists i \geq 0$, $\pi(i) \models \xi$ and $\pi(j) \models \eta$ for $0 \leq \forall j < i$.

2.2 Sequential Machines

Let $x_i (1 \leq i \leq l)$, $y_j (1 \leq j \leq m)$ be input variables and state variables over $B = \{1, 0\}$ respectively. **x** and **y** are vectors $< x_1, x_2, \cdots x_l >$ over B^l and $< y_1, y_2, \cdots y_m >$ over B^m respectively. A sequential machine with l binary input signals, m binary state variables and n binary output signals is defined by the set of Boolean functions as follows:

- State transition functions: $f_j \in [B^l \times B^m \rightarrow B]$ $(1 \leq j \leq m)$
  $\mathbf{f}(\mathbf{x},\mathbf{y}) =< f_1(x_1, x_2, \cdots x_l, y_1, y_2, \cdots y_m), \cdots, f_m(x_1, x_2, \cdots x_l, y_1, y_2, \cdots y_m) >$ gives the next state $\mathbf{y}'$ of a current state $\mathbf{y}$ for an input $\mathbf{x}$.
- Output functions:
  - $z_k \in [B^m \rightarrow B]$ $(1 \leq k \leq n)$ for a Moore-type machine;
    $\mathbf{z}(\mathbf{y}) =< z_1(y_1, y_2, \cdots y_m), \cdots, z_n(y_1, y_2, \cdots y_m) >$ gives the current output at a state $\mathbf{y}$.
  - $z_k \in [B^l \times B^m \rightarrow B]$ $(1 \leq k \leq n)$ for a Mealy-type machine;
    $\mathbf{z}(\mathbf{x},\mathbf{y}) =< z_1(x_1, x_2, \cdots x_l, y_1, y_2, \cdots y_m), \cdots, z_n(x_1, x_2, \cdots x_l, y_1, y_2, \cdots y_m) >$ gives the current output at a state $\mathbf{y}$ for an input $\mathbf{x}$.

In order to associate binary input signals, binary state variables, and binary output signals of sequential machines with atomic propositions, p_{x_i} $(1 \leq i \leq l)$, p_{y_j} $(1 \leq j \leq m)$, and p_{z_k} $(1 \leq k \leq n)$ are used as atomic propositions corresponding to x_i, y_j and z_k respectively. $x_i = 1$ means p_{x_i} is true and so on.

2.3 Shared Binary Decision Diagram

Boolean functions are efficiently represented by using a *Shared Binary Decision Diagram* (SBDD)[1, 3]. SBDD is a kind of labeled acyclic directed graph representing Shannon's expansion theorem according to a given fixed variable ordering, in which all isomorphic subgraphs are shared and nodes corresponding to redundant variables are removed. Each node is labeled by its corresponding variable name and has two outgoing edges called *'0' edge* and *'1' edge* respectively. It represents a Boolean function $f = xf_1 + \overline{x}f_0$, where x is its label; f_1 and f_0 are Boolean functions pointed to by its '1' edge and '0' edge respectively.

SBDD has various useful properties. If the ordering of the variables is fixed for the whole graph, the graph is canonical, i.e. there are no two different nodes representing a same Boolean function [1, 3]. In addition, the size of the graph is feasible for many practical Boolean functions [11]. The manipulations for various operations on Boolean functions represented by SBDD can be performed in time proportional to the size of the SBDD [3].

In order to guarantee the uniqueness of SBDD representation, we need to manage SBDD nodes so that no two different nodes represent a same function. This is usually done by using a hash table called *node table* [13]. In addition, in order to perform various operations on Boolean functions in time proportional to the size of their corresponding SBDD, same operations on same Boolean functions should be prevented. This is usually done by using another hash table called *operation result table*[13].

3 Symbolic Model Checking for Sequential Machines

For a Boolean function $f \in [B^n \to B]$ and a vector of variables $\mathrm{x} = < x_1, x_2, \cdots x_n >$ over B^n, we use the following notations:

$$\begin{aligned} \exists x_i.f(\mathrm{x}) &\overset{\mathrm{def}}{=} f(x_1, x_2, \cdots x_{i-1}, 0, x_{i+1}, \cdots, x_n) \vee f(x_1, x_2, \cdots x_{i-1}, 1, x_{i+1}, \cdots, x_n) \\ \exists \mathrm{x}.f(\mathrm{x}) &\overset{\mathrm{def}}{=} \exists x_1 \exists x_2 \cdots \exists x_n.f(\mathrm{x}) \end{aligned}$$

A subset S of B^n is represented by a Boolean characteristic function $F_S \in [B^n \to B]$ such that $F_S(\mathrm{s}) = 1$ if and only if $\mathrm{s} \in S$.

3.1 Basic Algorithm

The algorithm shown in this sub-section is based on the symbolic model checking algorithm proposed in [5, 6, 7].

Since the semantics of CTL is defined over Kripke structure, a given sequential machine has to be transformed to the corresponding Kripke structure for model checking.

Let x and y be a input vector and a state vector of a sequential machine respectively. Let s be a state vector of the corresponding Kripke structure. Since a state transition of a sequential machine corresponds to a state of the corresponding Kripke structure, s can be expressed as $\mathrm{x}\#\mathrm{y}$, where $\mathrm{x}\#\mathrm{y}$ represents a concatenation of two vectors x and y (i.e. $\mathrm{x}\#\mathrm{y} \overset{\mathrm{def}}{=} < x_1, x_2, \cdots x_l, y_1, y_2, \cdots y_m >$). The set of states of the Kripke structure is $B^l \times B^m$.

By introducing new vectors of Boolean variables $\mathrm{x}' = < x'_1, x'_2, \cdots x'_l >$ and $\mathrm{y}' = < y'_1, y'_2, \cdots y'_m >$ corresponding to x and y, s' is defined to be $\mathrm{x}'\#\mathrm{y}'$. We use x', y', and s' to represent the input vector and the state vector of the sequential machine and the state vector of the Kripke structure at the next time.

Let f_j be a state transition function corresponding to a state variable y_j. The Boolean function representing the Kripke structure K, denoted by F_K, is constructed as follows:

$$F_K(\mathrm{s}', \mathrm{s}) = \prod_{0 \leq j \leq m} (y'_j \equiv f_j(\mathrm{x}, \mathrm{y}))$$

This function means that $F_K(\mathrm{s}', \mathrm{s}) = 1$ if and only if $(\mathrm{s}', \mathrm{s})$ is an edge of the corresponding Kripke structure. It is easy to see that $F_K(\mathrm{s}', \mathrm{s})$ does not depend on x' and we can also regard it as a Boolean characteristic function which represents state transition relation of the sequential machine.

Let $F_\eta(\mathrm{s})$ be a characteristic function of a CTL formula η. It represents a set of states where η holds. We can get $F_\eta(\mathrm{s})$ in a bottom up manner as follows:

- For atomic propositions, $F_{p_{x_i}}(\mathrm{s}) \overset{\mathrm{def}}{=} x_i$, $F_{p_{y_j}}(\mathrm{s}) \overset{\mathrm{def}}{=} y_j$ and $F_{p_{z_k}}(\mathrm{s}) \overset{\mathrm{def}}{=} z_k$.
- $F_{\eta \cdot \xi}(\mathrm{s}) \overset{\mathrm{def}}{=} F_\eta(\mathrm{s}) \cdot F_\xi(\mathrm{s})$, where '$\cdot$' is any Boolean operator.

- $F_{EX\eta}(s) \overset{\text{def}}{=} \exists s'.(F_\eta(s') \wedge F_K(s', s))$.
- $F_{EG\eta}(s)$ is obtained by the following fixed point calculations.

$$A_0(s) \overset{\text{def}}{=} F_\eta(s), \quad A_{i+1}(s) \overset{\text{def}}{=} A_i(s) \wedge \exists s'.(A_i(s') \wedge F_K(s', s))$$

- $F_{E[\eta \mathcal{U} \xi]}(s)$ is obtained by the following fixed point calculations.

$$A_0(s) \overset{\text{def}}{=} F_\xi(s), \quad A_{i+1}(s) \overset{\text{def}}{=} A_i(s) \vee \exists s'.(A_i(s') \wedge F_\eta(s) \wedge F_K(s', s))$$

3.2 Implicit Manipulation of Transition Relation

The size of an SBDD representing the characteristic function $F_K(s', s)$ for the transition relation can be very large, even if the total size for f_j is small. In order to improve the efficiency of the above algorithm, it is desired to prevent the calculation of $F_K(s', s)$.

Note that $F_K(s', s)$ is used only in the form of $\exists s'.(C(s') \wedge F_K(s', s))$. This function is equivalent to

$$\exists y'.((\exists x'.C(s')) \wedge \prod_{1 \leq j \leq m} (y'_j \equiv f_j(x, y)))$$

Therefore, we can get this function without constructing $F_K(s', s)$ explicitly as follows:
[Implicit Calculation of $\exists s'.(C(s') \wedge F_K(s', s))$]

Obtain the following $m+1$ functions D_i from D_0 to D_m sequentially. D_m is the result.

$$\begin{array}{lcl} D_0(y', s) & \overset{\text{def}}{=} & \exists x'.C(s') \\ D_{i+1}(y_{i+2}, y_{i+3}, \cdots, y_m, s) & \overset{\text{def}}{=} & (D_i(1, y_{i+2}, y_{i+3}, \cdots, y_m, s) \wedge f_{i+1}(s)) \vee \\ & & (D_i(0, y_{i+2}, y_{i+3}, \cdots, y_m, s) \wedge \neg f_{i+1}(s)), \quad 0 \leq i \leq m-2 \\ D_m(s) & \overset{\text{def}}{=} & (D_{m-1}(1, s) \wedge f_m(s)) \vee \\ & & (D_{m-1}(0, s) \wedge \neg f_m(s)) \end{array}$$

4 Vectorization of SBDD Manipulation

Because most time consuming part of the symbolic model checking algorithm is the manipulation of Boolean functions represented by SBDD, we concentrate on vectorization of SBDD manipulation.

Vector processors achieve more than several GFLOPS by vector instructions which execute uniform operations on array-structured data using pipelined functional units, and they usually have large main memory of several hundred mega bytes. In conjunction with floating-point operations, they also support integer and bit-wise logical operations. Since the performance of programs on vector processors are strongly affected by *vectorization ratio* and *vector length*, we usually need to devise new algorithms suitable for vector execution to enjoy power of vector processors.

4.1 Vectorized Algorithm for SBDD Manipulation

The conventional algorithm for manipulating SBDD's is based on a recursive procedure (or depth-first operation), which is not suitable for vector processing. In this subsection, we propose a breadth-first algorithm for manipulating SBDD's [14].

The proposed algorithm consists of two parts; an *expansion phase* and a *reduction phase*. In the expansion phase, new nodes sufficient to represent the resultant function are generated in a breadth-first manner from the root node toward leaf nodes. In the reduction phase, the nodes generated in the expansion phase are checked and the redundant nodes and the equivalent nodes are removed in a breadth-first manner from nodes nearby leaf nodes toward the root node. The nodes generated in the expansion phase are called *temporary nodes*, while the nodes which already exist are called *permanent nodes.*

4.1.1 Expansion Phase

The input for the expansion phase is a triple (op, f, g), where op is a Boolean operator to be executed, and f and g are the root edges for operand Boolean functions. We refer to this triple as a *requirement.* The requirement (op, f, g) requires to compute the root edge for the resultant function of $op(f, g)$. During processing a requirement, new requirements will be generated for computing the operations between subfunctions or subsubfunctions $\cdots$ of the operand functions. Actually a requirement corresponds to a procedure call in the depth-first algorithm. We introduce a queue called.a *requirement queue* to manage these requirements, which makes our procedure breadth-first. (The procedure would be depth-first if we use a stack instead of the queue.)

For a given requirement (op, f, g), a new root node is not always generated. We should not generate a new node if a node representing the result of $op(f, g)$ already exists. For example, if the result of $op(f, g)$ is found trivially, or found by looking up the operation result table, we do not generate a new node. These judgment can be done immediately from f and g. However, we can not tell, in general, the existence of the node of the same function as $op(f, g)$ until we construct the whole graph for the subfunctions of $op(f, g)$. In our breadth-first algorithm, we once generate a temporary node in such cases. Whether the temporary node is actually essential or not is examined in the reduction phase.

[Algorithm of the Expansion Phase]

Put the given requirement (op, f, g) to the requirement queue and repeat the following operations for every requirement in the queue until the queue becomes empty.

(1) If the root node representing the result of $op(f, g)$ is found trivially, return the edge pointing to the node.

(2) If the root node representing the result of $op(f, g)$ is found in the operation result table, return the edge found in the table.

(3) Otherwise, generate a new temporary node and return the edge pointing to the temporary node. At the same time, register the edge pointing to the temporary node to the operation result table as the result of $op(f, g)$ and put the new requirements (op, f_0, g_0) and (op, f_1, g_1) to the requirement queue, whose result will be '0' edge and '1' edge of this temporary node respectively.

Since the total number of requirements processed in the above procedure is exactly the same as the number of procedure calls in the conventional depth-first algorithm, there is no serious increase on the computation cost. The only drawback of our algorithm is the increase of the storage required for temporary nodes.

This procedure is suitable for vector processing because it is a simple reptition of processing all requirements in the queue simultaneously and all the repeated operations are vectorized.

4.1.2 Reduction Phase

After the expansion phase is finished, there may be the following type of temporary nodes:

- *Redundant node*: A temporary node whose '0' and '1' edges point to the same node.
- *Equivalent node*: A temporary node whose label, '0' and '1' edges are the same as one of the permanent nodes.

The main tasks of the reduction phase are to find redundant or equivalent nodes and to remove them. They are performed in a breadth-first manner from the nodes nearby the leaf nodes toward the root node. In addition, temporary nodes which are neither redundant nor equivalent are registered to the node table. In practice, the removal of the redundant nodes and the equivalent nodes should be done at the end of the reduction phase because these nodes could be pointed to by some edges. Therefore, the redundant nodes and the equivalent nodes are marked with *slave nodes*. Every slave node has a pointer to its *master node* which takes the place of the slave node. When a slave node is pointed to by '0' or '1' edges of other nodes, these edges are modified to point to the master node.

[Algorithm of the Reduction Phase]

Repeat the following operations while there are temporary nodes. For every temporary node whose '0' and '1' edges are not temporary nodes (i.e. permanent nodes or leaf nodes), execute the followings:

(1) If its '0' and '1' edges are the same, mark the node as a slave node whose master node is the node pointed to by its '0' edge.

(2) If there is an equivalent node registered in the node table, mark the temporary node as a slave node whose master node is the node registered in the node table.

(3) Otherwise, register the node to the node table, and change its attribute to permanent from temporary.

This procedure is also suitable for vector processing because all temporary nodes whose '0' and '1' edges are not temporary nodes can be processed at a time, and almost all operations are vectorizable.

5 Experimental Results

5.1 Pipelined ALU

The sequential machine we used as an example is an n bit pipelined ALU with a register file. Its structures is similar to the sequential circuit used by Burch et al[5, 6]. Fig. 1 shows its block diagram. The solid lines represent data paths and the dotted lines represent control signals. The register file consists of 4 registers of n bits. PRA, PRB and PRC are n bit pipeline registers. This pipelined ALU performs one of 16 arithmetic/logical operations on the register file according to the given input signals. There are 11 bit input signals; 1 bit *Enable* signal, 4 bit *Op_Code* signal, three 2 bit signals specifying source register A (*Src_Reg_A*), source register B (*Src_Reg_B*), and destination register (*Dest_Reg*) respectively. When *Enable* signal is asserted, this pipelined ALU performs its specified operation in 3 stage pipeline. In the first stage, the operands are read from the register file to PRA and/or PRB. Simple modification on the operand data may be performed during this stage if necessary. In the second stage, the specified operation is performed and its

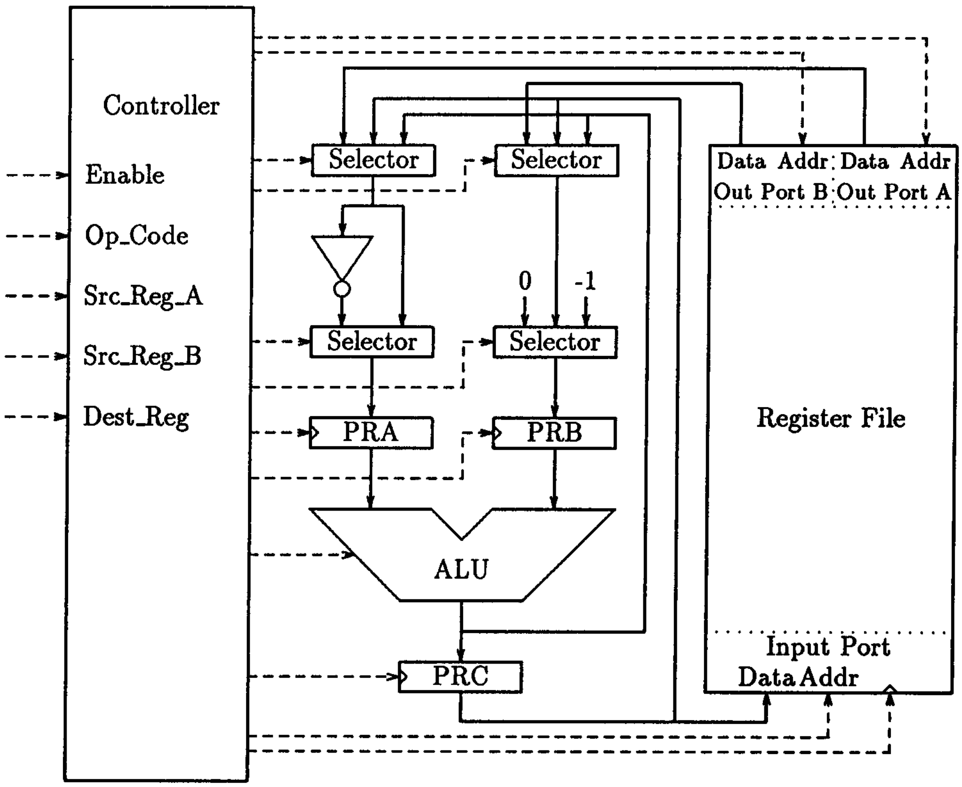

Figure 1: Block Diagram of a Pipelined ALU

result is stored in PRC. In addition, this result can be used immediately as operands of an instruction on the next clock cycle. In the third stage, the content of PRC is written into the register file, as well as it can be used as operands of an instruction on this clock cycle.

The specification of the ALU can be written as CTL formula in the similar manner stated in [5, 6]. Considering that the latency of this pipelined ALU is three, what we should verify are:

- If *Enable* is asserted, the content of the destination register at three clock later will be the result of the specified operation on the source registers at two clock later.
- For any register in the register file, if *Enable* is not asserted or it is not specified as a destination register, its content at three clock later will be the same as its content at two clock later.

We used the n bit pipelined ALU explained above as benchmark tests for our model checking algorithms. It is referred to CALUn hereafter. CALUn contains $7n + 11$ bit memory elements in total (the 4 word register file, 3 pipeline registers, a C-flag, and 10 flipflops in the controller). In addition, PADDn, which is obtained from CALUn by fixing the Op_Code input signals to *ADD* instruction and removing the C-flag. PADDn contains $7n + 6$ bit memory element in total.

Table 1: Size of SBDD representing Sequential Machines and Kripke Structures

Name	Sequential Machine (nodes)	Kripke Structure (nodes)	ratio K.S./S.M.
PADD2	132	7,514	56.9
PADD8	540	54,734	101.4
PADD16	1,084	117,694	108.6
CALU2	541	183,065	338.4
CALU8	2,194	> 1,000,000	—
CALU16	4,566	> 1,000,000	—

Table 2: Verification time and space of the implicit and explicit versions

Name	Implicit version		Explicit version		Ratio	
	time (sec)	size (nodes)	time (sec)	size (nodes)	time	size
PADD2	5.37	3,718	75.60	34,539	14.08	9.29
PADD8	90.67	8,320	6,025.88	279,746	66.46	33.62
PADD16	608.78	14,864	59,851.55	607,354	98.31	40.86
CALU2	24.35	34,527	——†	> 1,000,000	—	—
CALU8	1,879.72	252,997	——†	> 1,000,000	—	—
CALU16	16,648.28	754,465	——†	> 1,000,000	—	—

† Cannot be obtained because more than 1 million nodes are required.

5.2 Effects of the Implicit Manipulation of Transition Relation

In order to evaluate the effects of the implicit manipulation of transition relation stated in Section 3.2, we have implemented two symbolic model checker for sequential machine verification on a SPARC Station 1+: one is based on the algorithm explained in Section 3.1 (*Explicit* version); the other is based on the algorithm stated in Section 3.2 (*Implicit* version). These two model checker can use up to 1 million SBDD nodes by using about 23 M byte user area. We used SBDD package developed by Minato [13] for Boolean manipulations in the model checkers.

Table 1 shows the number of SBDD nodes used to represent a sequential machine (i.e. its state transition functions and output functions) and the size of SBDD representing a

Table 3: Benchmark results of the vectorized symbolic model checking

Name	Scalar (sec)	Vector (sec)	S/V
PADD2	4.251	0.659	6.45
PADD8	56.132	6.718	8.36
PADD16	387.178	38.730	10.00
CALU2	18.686	1.874	9.97
CALU8	833.253	40.991	20.33
CALU16	——†	741.420	—

† Not experimented because it may exceed CPU time limit.

characteristic function for transition relations of their corresponding Kripke structures. Kripke structures require much more space than sequential machines.

Table 2 shows the comparison of the experiments of the two implementations. It shows the required time and the required number of SBDD nodes for verification. We can see from this table that the *Implicit* version is dramatically efficient compared with the *Explicit* version. It achieves up to 100 times and 40 times improvements in time and space respectively. The amount of improvements seems to become much larger if a sequential machine under verification becomes more complex.

5.3 Effects of the Vectorization

Considering the experimental results stated in the previous subsection, we adopt the *Implicit* mode checking algorithm to implement the vectorized symbolic model checker for sequential machines on a vector processor HITAC S-820/80. We call this implementation as *Vector* version. The *Vector* version uses the vectorized manipulation algorithms for SBDD proposed in Section 4.1. It can use 5 million SBDD nodes with 256 M byte user area.

Table 3 shows the scalar execution and the vector execution of the *Vector* version on HITAC S-820/80. It achieves about 6 to 20 vector acceleration ration. It verified CALU16 in about 12 minutes.

6 Concluding Remarks

In this paper, we first compared the implicit and the explicit symbolic model checking algorithms of CTL for sequential machine verifications. It is shown that the implicit one is dramatically efficient and achieved $14 \sim 98$ times and $9 \sim 38$ improvement in time and space. The more complex sequential machines become, the more improvement factor it achieves.

Next we proposed the vectorized symbolic model checking algorithm based on the new algorithm. It achieved $6 \sim 20$ acceleration ratio and succeeded to verify a 16 bit pipelined ALU of 16 arithmetic/logical operations on 4 word register file in only 12 minutes.

Our current implementations do not support *frontier set simplification* [5, 6, 9] which is effective in fixed point calculations. We think it is not difficult to realize it in our model checkers. We would also like to support *fairness constraint* [5, 6] in the near future.

Acknowledgments The authors would like to express their appreciations to Prof. E. M. Clarke of CMU, Dr. N. Takagi of Kyoto Univ., Dr. N. Ishiura of Osaka Univ., and Dr. S. Kimura of Kobe Univ. for their valuable discussions. This research is supported partially by Japan-U.S.A. cooperative research of JSPS and NSF. It is also supported partially by a grant-in-aid for scientific research of the Ministry of Education of Japan.

References

[1] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[2] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. 27th Design Automation Conference*, pages 40–45, June 1990.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[4] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proc. 28th Design Automation Conference*, June 1991.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. Technical report, Carnegie Mellon University, November 1989.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proc. 27th Design Automation Conference*, pages 46–51, June 1990.

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. Logic in Computer Science*, June 1990.

[8] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Proc. Workshop on Logic of Programs*, pages 52–71. Springer-Verlag, 1981.

[9] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using functional vectors. In *Proc. IMEC-IFIP Intrn. Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, November 1989.

[10] O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In *Proc. Workshop on Computer-Aided Verification*, June 1990.

[11] N. Ishiura and S. Yajima. A class of logic functions expressible by polynomial-size binary decision diagrams. In *Proc. the Synthesis and Simulation Meeting and International Interchange*, pages 48–54, October 1990.

[12] S. Kimura and E. M. Clarke. A parallel algorithm for constructing binary decision diagrams. In *Proc. IEEE ICCD'90*, September 1990.

[13] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *Proc. 27th Design Automation Conference*, pages 52–57, June 1990.

[14] H. Ochi, N. Ishiura, and S. Yajima. Breadth-first manipulation of SBDD of boolean functions for vector processing. In *Proc. 28th Design Automation Conference*, June 1991.

[15] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. In *Proc. ICCAD*, 1990.