Improved Symoblic Simulation By Dynamic Funtional Space Partitioning

Tao Feng, Li-C.Wang, Kwang-Ting Cheng
Department of ECE, UC-Santa Barbara, U.S.A
{tfeng,licwang, timcheng}@ece.ucsb.edu

Andy C-C. Lin
Cadence Design Systems, Inc. U.S.A
cclin@verplex.com

Abstract

In this paper, we provide a flexible and automatic method to partition the functional space for efficient symbolic simulation. We utilize a 2-tuple list representation as the basis for partitioning the functional space. The partitioning is carried out dynamically during the symbolic simulation based on the sizes of OBDDs. We develop heuristics for choosing the optimal partitioning points. These heuristics intend to balance the tradeoff between the time and space complexity. We demonstrate the effectiveness of our new symbolic simulation approach through experiments based on a floating point adder and a memory management unit.

1. Introduction

Symbolic simulation based on Ordered Binary Decision Diagram(OBDD) [1] has been shown various successes in formal verification. However, a traditional symbolic simulation approach may easily suffer from the memory size explosion problem. Even worse, a little modification of the circuit or the initial variable ordering can result in order-of-magnitude difference in run-time and memory usage.

The limited and unstable behavior of a symbolic simulator often leads to tremendous frustration for verification engineers. Many attempts have been done to reduce/control the OBDD sizes in symbolic simulation [2]. The partition of functional space (case split) is one promising approach. In the paper [3], the authors split the verification task into subcases based on the understanding of the design. The parametric constrains have been applied to verify each subcase. The authors in [4] decompose the monolithic OBDD into some partitioned-OBDDs based on the control conditions which is the combination of primary inputs or internal variables.

In this paper, we provide a different way to partition the functional space. The contribution of this paper comes from two aspects. First, we provide a 2-tuple list symbolic simulation engine which can represent the boolean function in control and datapath domains separately. During the course of the simulation, the functional space in control domain can be partitioned into subspaces and the corresponding data for each subspace is evaluated in the datapath domain. Second, the paper discusses in detail on how to find the good points for functional space partitioning. For the "hard-to-verify" circuits, we observe the dramatic changes of memory usage during the course of symbolic simulation. These dramatic changes provide hints to find the key partitioning points where OBDD size reduction methods should be applied.

2. Motivations and the baseline study

The curve 1 of Figure 4 in our experiments shows the level-by-level total OBDD sizes by simulating a floating point adder whose netlist is levelized. The monolithic OBDD is built from input to output in the ordinary symbolic simulation. We observe the following aspects:

- The floating point adder is a typical example hard for a traditional symbolic simulator to simulate [9]. Notice that the OBDD sizes does not increase linearly as the circuit level increases. The size may sharply increase at certain levels although the dynamic variable ordering was enabled.
- The points where OBDD size change dramatically can be in any place of the circuit, not necessarily only at the place where symbolic simulation aborts. This may explain the "unpredictable" OBDD performance for the symbolic simulation. To find out the problem source, it would be good to know the earliest point where the sudden change of OBDD sizes occur.
- Past experiences indicate that these points where OBDD sizes change significantly often locate along the boundary between the datapath and the control part of a design. For example, a comparator output from the datapath is often considered as a control signal. This output represents the result of "merging" multiple word-level data and can be a place for OBDD size to blow up. Large circuits usually have complex control and datapath logic. When they converge at the control-datapath interface, it often causes OBDD sizes to change dramatically.

2.1. The optimal partition points inside the circuit

The boundary of the datapath and control part can usually be modeled explicitly or implicitly using the MUX primitives. The MUX inputs and outputs stay on the datapath while the MUX select lines stay on the control. It provides a natural partitioning point at which we can separate the logic.

A simple heuristic is to choose all MUX primitives as the space partition point. Unfortunately it may generate too many trivial subspaces and increase the time complexity of the problem. Thus we need to use the heuristics to find the *key* points for functional space partition.

Our baseline study on monitoring the OBDD performance during the symbolic simulation above gives some hints to the solution. The points where the OBDD size change dramatically have the high influence of the simulation performance. The MUX primitives related to these points will have the higher priority to be selected.

We call this method a *dynamic heuristic* for selecting partitioning points because it is based on the OBDD performance during the course of the simulation. On the contrary, static heuristics can be based on the circuit topological structure to determine which MUX primitives are for partitioning (discussed in section 4.2).

The dynamic functional space partitioning concept, in essence, follows the same principle as that makes the dynamic variable ordering successful. In both approaches, the adjustment of OBDD size occurs only when a problem has been observed.

As an analogy, a fixed initial variable ordering is a static ordering before the symbolic simulation. They are based on the circuit topological structure to estimate the best initial variable ordering. Alternatively, dynamic variable ordering could be more efficient to reduce the OBDD sizes, but it is quite time consuming. Hence, the heuristic needs to invoke the dynamic ordering as few times as possible with the reduction of the total OBDD size as much as possible. This is similar to our dynamic heuristic for choosing MUX primitives to partition the functional space. We want to invoke the partitioning only at the key points with the reduction of total OBDD size as much as possible.

3. Basic concept of 2-tuple list representation

[Definition 1: 2-tuple]
The simulated result on each signal line a is stored as a 2-tuple that is of the form (C^a, D^a), where the first tuple C^a is called a *control* and the second tuple D^a is called a *data*. (C^a, D^a) is read as "node a has the data D^a when the control C^a is true, otherwise the value on node a is unknown X." C^a and D^a could be a single variable or a boolean expression. Intuitively, D^a simulates the results for a site on the datapath while the corresponding C^a tells the control signal's combinations for it to happen.

[Definition 2: 2-tuple list]
Initially, each input port a of the circuit will be assigned with the 2-tuple $(control, data) = (1, D^a)$. The *data* part is assigned a new variable D_a, and the *control* part is denoted as 1 which represents the whole functional space. During the course of symbolic simulation, the whole functional space in the *control* domain can be split into several subspaces. The internal wire a is represented as a list of 2-tuples that is of the form

$$L^a(n) = \{(C_1^a, D_1^a), \ldots, (C_n^a, D_n^a)\} = \biguplus_{i \in [1,n]} (C_i^a, D_i^a) \qquad (1)$$

where n is the number of the split subspaces for the wire a. If the circuit has no unknown states, then the union of the control parts in the 2-tuple list is the whole functional space:

$$C_1^a \vee C_2^a \ldots \vee C_n^a = \sum_{i \in [1,n]} C_i^a = 1 \qquad (2)$$

Here we use the symbol $\biguplus$ to represent the concatenate of multiple 2-tuples and $\sum$ for multiple boolean OR operations.

[Definition 3: mutually exclusive in 2-tuple list]
In the 2-tuple list $L(n) = \{(C_1, D_1), \ldots, (C_n, D_n)\}$, if $C_i \wedge C_j = \emptyset (i \neq j, 1 \leq (i,j) \leq n)$, each 2-tuple in the list is called mutually exclusive with other 2-tuples.

[Theorem 1: 2-tuples list merge rule]
In the list $L = \{(C_1, D_1), \ldots, (C_n, D_n)\}$ which contains n 2-tuples. We can merge multiple 2-tuples (in the same list) into a single 2-tuple. If L is a mutually exclusive 2-tuple list, the following rule can be applied.

$$\begin{aligned} &\biguplus_{i \in [1,n]} (C_i, D_i) = \{(C_1, D_1), (C_2, D_2), \ldots (C_n, D_n)\} \\ &= (C_1 \vee C_2 .. \vee C_i,\ C_1 D_1 \vee C_2 D_2 .. \vee C_n D_n) \\ &= (\sum_{i \in [1,n]} C_i,\ \sum_{i \in [1,n]} C_i D_i) \qquad (3) \end{aligned}$$

Here $C_i D_i$ represents $(C_i \wedge D_i)$.

3.1. The 2-tuple list construction rule

When the signals go through the gates such as AND or OR, the output result can be obtained by exploring the data values under the intersection of the control domains from the fanin wires. The following is the algorithm for the 2-input OR gate.

Construction Rule1: for 2-inputs OR gate
Input: $L^a = (C_1^a, D_1^a), \ldots, (C_n^a, D_n^a)$
$L^b = (C_1^b, D_1^b), \ldots, (C_m^b, D_m^b)$
Output: $L^{out} = \biguplus_{(i \in [1,n], j \in [1,m])} (C_i^a \wedge C_j^b,\ D_i^a \vee D_j^b)$

[Proof of construction rule1:] We prove the above 2-tuple list construction rule can evaluate the same function as the ordinary symbolic method. The difference between them is that the function space in the *control* domain is always 1 in the ordinary symbolic method, while in our construction rule the control space of input signals has been mutual exclusively partitioned and represented in a 2-tuple list. The 2-tuples in a list can be merged with the equation 3 and becomes the representation of the ordinary method. $(\sum_i C_i^a = 1, \sum_j C_j^b = 1)$

The ordinary symbolic simulation evaluates the OR function as:

$$\begin{aligned} L^a \vee L^b &= (1, \sum_i C_i^a D_i^a) \vee (1, \sum_j C_j^b D_j^b) \\ &= (1,\ \sum_i C_i^a D_i^a \vee \sum_j C_j^b D_j^b) \end{aligned} \tag{4}$$

Our 2-tuple list construction rule1 evaluates function as:

$$\begin{aligned} L^a \vee L^b &= \biguplus_{i\in[1,n]} (C_i^a, D_i^a) \vee \biguplus_{j\in[1,m]} (C_j^b, D_j^b) \\ &= \biguplus_{(i\in[1,n],j\in[1,m])} (C_i^a \wedge C_j^b,\ D_i^a \vee D_j^b) \\ &= (\sum_{(i\in[1,n],j\in[1,m])} (C_i^a \wedge C_j^b), \sum_{(i\in[1,n],j\in[1,m])} (C_i^a \wedge C_j^b) \wedge (D_i^a \vee D_j^b)) \\ &= (1,\ \sum_i C_i^a D_i^a \vee \sum_j C_j^b D_j^b) \end{aligned} \tag{5}$$

By applying the equation 2 and 3, we derive that given the same inputs, our construction rule evaluates the same function as the ordinary symbolic simulation in equation 4. The above construction rule can be extended to the other gates such as XOR and AND gates. In the above rule, the variables in the control and data domains are handled independently, thus the control variables do not go into the data domain and vice versa.

For the MUX primitive, the variables in the control and date domains can be exchanged with the following construction rule.

Construction Rule2: for 2:1 MUX gate
Input: $L^a = (C_1^a, D_1^a), \ldots, (C_n^a, D_n^a)$
$L^b = (C_1^b, D_1^b), \ldots, (C_m^b, D_m^b)$
$L^c = (C_1^c, D_1^c), \ldots, (C_p^c, D_p^c)$
Output:

$$\begin{aligned} L^{out} = \{ & \uplus_{(i\in[1,n],j\in[1,p])} (C_j^c \wedge D_j^c \wedge C_i^a,\quad D_i^a), \\ & \uplus_{(i\in[1,m],j\in[1,p])} (C_j^c \wedge (!D_j^c) \wedge C_i^b,\ \ D_i^b) \} \end{aligned}$$

The 2:1 MUX has two data-input signals (L^a and L^b) and one select-input signal(L^c). The output signal L^{out} will have the value of L^a if the value of the select-input signal L^c is true, otherwise, L^{out} will have the value of L^b. We note that the variables in the *data* domain of L^c will be moved to the *control* domain in the L^{out}. For the space of the paper, we omit the proof of the construction rule2. A demonstration example on hidden weighted function has been shown in [6].

3.2. The 2-tuple list in the verification flow

Our symbolic simulator consists of the following three steps: (1) extraction of the MUX primitives from a gate-level circuit, (2) symbolic simulation with the 2-tuple list construction rules, and (3) consistency checking of the output result in the 2-tuple list.

A gate-level netlist can usually be synthesized from its high-level (RTL) model. The RTL statements such as "if", "case" are the decision points, and are usually synthesized as MUXes in the low level circuit. In addition to the high-level information, the MUX can also be extracted in a low-level circuit where the signal and its negated signal have re-convergent fanout. In our symbolic simulator, we distinguish the MUX primitive with other primitives such as AND/OR gates, because on the MUX primitive the variables in the *control* and *data* domains can be interchanged and adjusted in our 2-tuple list representation(according to its construction rule).

When the symbolic simulation finishes, an output signal is represented by a 2-tuple list $L^a(n) = \{(C_1^a, D_1^a), \ldots, (C_n^a, D_n^a)\}$. If we want to verify this result by comparing it with the result obtained by simulating another model (another gate-level model or an assertion), we need to perform *consistency checking*. In consistency checking, we compare $L^a(n)$ to another list $L^b(m) = \{(C_1^b, D_1^b), \ldots, (C_m^b, D_m^b)\}$. We first check if the union of their *control* domain covers the whole functional space: $(C_1^a \vee C_2^a \vee \ldots \vee C_n^a) = (C_1^b \vee C_2^b \vee \ldots \vee C_m^b) = 1$. We further check, under the intersection of *control* domains between the 2-tuple lists, their values are the same: $D_i^a = D_j^b$ when $C_i^a \wedge C_j^b \neq \emptyset$ $(1 \le i \le n, 1 \le j \le m)$.

4. Heuristics for selecting partition point

As the partitioning is based on the input-select signals of the MUXes, different values of the input-select signals will partition the control space in different ways. When the signals go through AND/OR gates, the control space can be further partitioned by the intersection of the subspaces from the fanin wires (by applying the 2-tuple construction rules). Although the OBDD size for each subspace becomes smaller, it may take more time to handle all of the subcases if the number of the partitioned subspaces is very large. To control the size of 2-tuple lists, we need to carefully select the MUX primitives for partitioning.

4.1. Remodel the MUX primitives

Instead of partitioning all MUX primitives in a circuit, our method selects some of the MUXes as the partitioning points based on the objective to control both the OBDD size and the simulation run time.

For those MUX primitives which is not chosen for partition, we implicitly remodel the MUX primitives using AND/OR gates (see Figure 1). Hence, instead of applying the construction rule2 to interchange the control and data variables, we apply the construction rule1 for the AND/OR gates so that the output of the MUX would keep the original partitioned subspaces as those given at its fanin wires. In this way, the control space will not be partitioned into too many diverse subspaces.

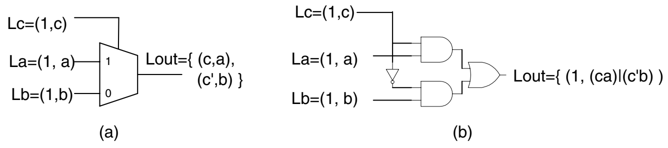

Figure 1: Remodel the MUX primitives

4.2. Static heuristic to choose the MUX primitives

Our static heuristic is based on the structure of a circuit [6]. If the logic cones of the data-input signals of a MUX overlap significantly with the logic cones of the select-input signal, OBDD could have trouble in finding a good ordering. We select the MUX primitive as a partitioning point to separate these variables in control and data domains.

4.3. Dynamic heuristic to choose the MUX primitives

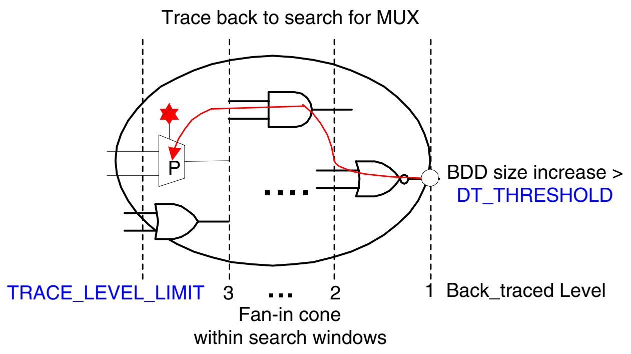

Figure 2: Dynamic heuristic to choose MUX primitives

The dynamic heuristic is embedded in our 2-tuple list symbolic simulation engine. We levelize a given circuit and symbolically simulate the gates level by level. For MUX primitives, originally all of them are reset with the "mux_nosplit" flag. This means that they will be evaluated using the construction rule1. We then use the dynamic heuristic to select some MUX primitives as the partitioning points by setting their "mux_split" flags. These MUX primitives will be evaluated using the construction rule2.

During the course of constructing OBDD for the gate output, we record the total OBDD size obtained so far. Once we find that the total OBDD size is beyond a given threshold "DT_THRESHOLD", we trace back from this point to search for the MUX partitioning points. The procedure "trace_back" will mask the "mux_split" flag for the MUX primitives in the search window. The symbolic simulation then goes back to these masked MUX primitives and re-evaluate them using the construction rule2 to partition the functional space.

The "trace_back" procedure searches the MUX primitives backward level by level within the fanin cone from the point where the explosion of OBDD size was observed. The search window is restricted with the parameter "LEVEL_LIMIT" and the search will stop when the backward traced level goes beyond the "LEVEL_LIMIT". In our experiments, we set the search window range be 4 levels.

The MUX primitives in the search window will be marked with "mux_split" flag. Meanwhile, all the other MUX primitives which share the same select-input wire with them will also be marked. Usually, these shared input-select signals of the MUX primitives are the global control variables in a circuit. Identifying these variables ensures that the partitioning can be done systematically across the entire circuit. This usually helps to reduce the sizes of the 2-tuple lists in the simulation [6].

5. Applications and experimental results

All experiments were run on a Pentium 4 1.5G machines with 512M memory.

5.1. Experimental Example I: Floating Point Adder

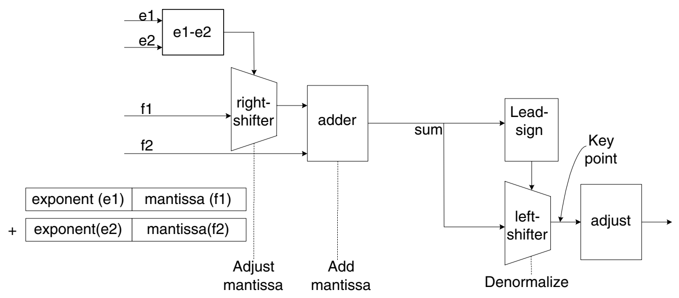

Figure 3: FADD implementations

5.1.1 The procedure of symbolic simulation

The FP adder is described in a hierarchical manner[7] and synthesized into flattened netlist. Figure 3 shows how two floating point numbers are added together. Each floating number is represented in the form of exponent($e1/e2$) and mantissa($f1/f2$). At first, the two exponents $e1$ and $e2$ are compared, the difference $e1 - e2$ is the amount number to right shift(align) the smaller mantissa. After alignment, the two aligned mantissas are added together as the sum result. It should be normalized by left shifting the sum result.

In our experiments, we first symbolically simulate the circuit without using the 2-tuple list partition. We levelize the circuit and monitor the total OBDD size at each level. As shown in figure 4(curve 1), the total OBDD size would exponentially increase even with dynamic ordering enabled. Actually the symbolic simulation could abort when it reaches the recourse limitation at the last level when the exponent bits is large enough.

At the level of 142 in the circuit, the significant increment of OBDD size corresponds to the places where normalization of the sum result is done (marked as key point in figure 3). The amount of left shift in the shifter depends on the most significant non-sign bit of the input data which is to be shifted.

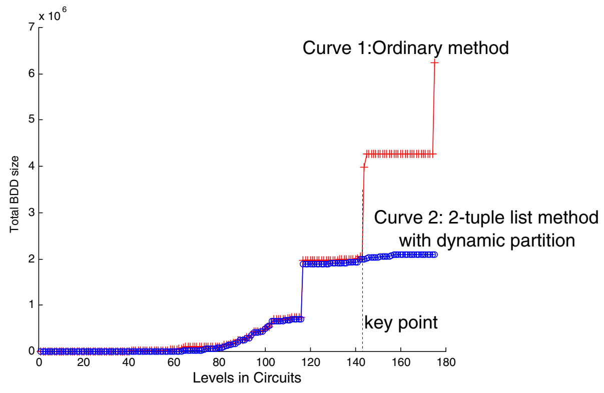

Figure 4: Total OBDD size for simulation FADD(fadd_e5_m24) with exponent(5bits), mantissa(24bits)

Figure 4(curve 2) shows the total OBDD size at each level using our 2-tuple list method with dynamic heuristic for selecting the partition points. We notice that the curve becomes flat as they reach the primary outputs (large level numbers). The dynamic heuristic successfully found the MUX primitives which influence the key points of figure 3. The curves indicate that our method has effectively decomposed the functional space to avoid the OBDD blow-up problem.

Table 1 summarizes the run-time and OBDD size of each method. We did the experiments on the adder of floating point numbers with 24 bits mantissa and different bits (from 3 to 7 bits) in exponent.

Table 1: Run time and OBDD size comparison results

Circuits	Run time(s)		Total OBDD size		Max split subspaces	
	-ord	-tp	-ord	-tp	-tp(dynamic)	-tp(static)
fadd_e3_m24*	288s	154s	969878	442255	18	137
fadd_e4_m24	5129s	1434s	6079878	2001076	27	185
fadd_e5_m24	7931s	1779s	6358884	2145178	27	257
fadd_e6_m24	abort	1984s	abort	2825484	27	313
fadd_e7_m24	abort	2714s	abort	2994318	27	345

-ord: ordinary method, -tp: our 2-tuple list method with dynamic partition
fadd_e3_m24* is with 3bits exponent and 24bits mantissa

Table 2 compares the run-time and OBDD size with different partitioning heuristics. The Max OBDD size in the table is the max OBDD nodes used to represent the function of a signal. The total OBDD size is the total OBDD nodes allocated. The "all split" heuristic selects all MUXes for partitioning.

Figure 5 shows the max number of subspaces split by the 2-tuple list at each level during the simulation. First, with the "all split" heuristic, although the OBDD size might be reduced much, the 2-tuple list size will increase dramatically (the upper curve in the figure 5). In this case, each signal could have a large-size 2-tuple list to be processed by the simulator. As a result, the run time can be slow.

The static heuristic for partitioning, as explained before, is based on circuit structure. Only the MUX primitives with overlapping logic cones are chosen. The number of partitioned subspaces is reduced but could still grow dramatically. The dynamic heuristic only chooses the MUX primitives which could greatly influence the OBDD performance. As a result, it could limit the partition points and at the same time, reduce the total OBDD size.

Table 2: Comparison of partition heuristics for fadd_e3_m24

Heuristic	Run time(s)	Total OBDD size	Max OBDD size	MUXs for partition
all split	337s	410303	8415	528
static	209s	445441	8415	399
dynamic	154s	442252	48522	97

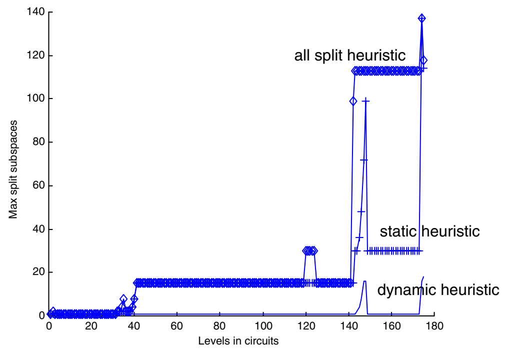

Figure 5: The number of subspaces split at each level for fadd_e3_m24 by different heuristics

5.1.2 The procedure of consistency checking

When the symbolic simulation finishes, the consistency checking needs to be performed to compare the output signal with other model. Due to the different partition strategy and the circuit implementation, the 2-tuple list representation of output signal in each circuit model can be different. One method for equivalence checking is to use the merge rule(Theorem 1) to convert the 2-tuple list into one monolithic OBDD. In some cases, the final merge of the 2-tuple list at the output can avoid the intermediate OBDD peak size in the middle of the simulation compared with the ordinary symbolic simulation which builds the monolithic OBDD for every internal signal.

For the complex circuits, the output signal could be too complex to be represented by a monolithic OBDD. Hence, we keep the output signal in 2-tuple list format and use the method proposed in section 3.2 for equivalence checking. In out experiments, the implementation of the circuit has been modified as the revised model. Table 3 shows the result of equivalence checking by using our 2-tuple list method and the ordinary method respectively.

Table 3: **Run time and OBDD size for equivalence checking**

Circuits	-tp Vs. -tp		-ord Vs. -ord	
	Time	Total OBDD size	Time	Total OBDD size
fadd_e2_m24*	47s	80738	78s	219084
fadd_e3_m24	495s	554946	1024s	2391480
fadd_e4_m24	4007s	3199882	13592s	14527730

-ord: ordinary method, -tp: our 2-tuple list method with dynamic partition
fadd_e2_m24* is with 2bits exponent and 24bits mantissa

5.2. Experimental Example II: Memory Management Unit

Another application is to verify the memory management unit(MMU) in the high-performance microprocessors. The MMU consists of two on-chip content addressable memory blocks (BAT and TLB blocks) to support the virtual memory address translation. BAT block contains four entries with the tag $T[i]$ and data $D[i]$ ($i \in [0..3]$) in each entry. If one tag $T[i]$ matches the input effective address ea, then $match[i] = 1$ and the corresponding data $D[i]$ in this entry is placed at the output of MMU. The control switches behave similarly as a MUX select line. For the bus structures, the 2-tuple list can be used to partition the functional space based on the control switches. We can see that with our 2-tuple list representation, the partitioning point is not strictly limited to MUX primitives, it can be any place where the concept of select control signal is applied.

The MMU example used in our experiments contains practical custom-design modules at the transistor level. An ordinary symbolic simulator could not handle the MMU due to the interactions between the TLB and the BAT modules [8]. The mixed-level nature of the MMU design (gates and transistors) adds another dimension of difficulty for a symbolic simulation. However, a transistor can be modeled as a latch while the latch enable signal serves as a control signal. Thus the 2-tuple list symbolic simulator can be applied in a smooth way for the mixed-level MMU design.

In our experiments, we initialize the value of memory cells with symbols. Then, symbolic simulation is carried out on the MMU. Table 4 shows the results with our 2-tuple list simulator. Note that an ordinary symbolic simulator could not simulate this design without encountering OBDD size blow up.

There are many other applications which our proposed

Table 4: **Comparison of time and OBDD for MMU blocks**

Circuits	Run time(s)	Total OBDD size	Max OBDD size	Split subspaces
MMU	302s	265429	12657	67
BAT	173s	196085	7927	16
TLB	100s	39436	1054	35
SEG	3s	10897	1595	16

2-tuple list partition method can be applied. In the microprogram controller, the instruction is encoded to generate control signals and the multiplexer selects data from different resources [7]. Greatest common divisor(GCD) is another example in arithmetic unit [10]. Our next goal is to extend our partition method into the sequential circuits.

6. Conclusion

In this paper, we present a functional-space partitioning method based on the construction of 2-tuple lists in the symbolic simulation. The partitioning is done dynamically by selecting MUX (or control points) using the proposed heuristics. The dynamic heuristic monitors the OBDD performance during the symbolic simulation in order to identify the key points where partitioning of the functional space can improve the global OBDD performance greatly. We demonstrate the effectiveness of our simulator by experiments on two known circuit examples, the floating point adder and the memory management unit, which both were shown to be difficult for an ordinary symbolic simulator to handle before.

References

[1] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[2] Alan. J. Hu. Formal hardware verifi cation with BDDs: An introduction. IEEE Pacifi c Rim Conference on Communications, Computers and Signal Processing, 1997

[3] Mark D.Aagaard, Robert B.Jones, Carl-Johan H.Seger. Formal Verifi cation Using Parametric Representations of Boolean Constraints. In *36th ACM/IEEE Design Automation Conference*, 1999.

[4] Amit Narayan, Jawahar Jain, Fujita, Sangiovanni. Paritioned ROBDDs - A Compact, Canonical and Effi ciently Manipulable Representation for Boolean Functions. In *ACM/IEEE Int. Conference on Computer-Aided Design*, 1996.

[5] R.E.Bryant. On the Complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. In *IEEE Trans. on Computer*, 1991.

[6] T.Feng, Li-C. Wang, Kwang-Ting Cheng Improved Symbolic Simulation By Functional Space Decomposition. In *Asia and South Pacifi c Design Automation Conference*, 2004.

[7] K.C.Chang. Digital Systems Design with VHDL and Synthesis, An integrated approach. In *IEEE computer society press*, 1999.

[8] T.Feng, Li-C. Wang, Kwang-Ting Cheng etc Enhanced Symbolic Simulation for Effi cient Verifi cation of Embedded Array Systems. In *Asia and South Pacifi c Design Automation Conference*, 2003.

[9] Yirng-An Chen, Randal E. Bryant Verifi cation of Floating Point Adders In *Proceeding of International Conference of Computer Aided Verifi cation*, 1998.

[10] D.J.Smith Practical Modeling Examples - HDL Chip Design In *Doone Publications, Chapter 12*, 1996