Disjunctive Image Computation for Embedded Software Verification

Chao Wang
NEC Laboratories America
Princeton, NJ, U.S.A.

Zijiang Yang
Western Michigan University
Kalamazoo, MI, U.S.A.

Franjo Ivančić, Aarti Gupta
NEC Laboratories America
Princeton, NJ, U.S.A

Abstract

Finite state models generated from software programs have unique characteristics that are not exploited by existing model checking algorithms. In this paper, we propose a novel disjunctive image computation algorithm and other simplifications based on these characteristics. Our algorithm divides an image computation into a disjunctive set of easier ones that can be performed in isolation. Hypergraph partitioning is used to minimize the number of live variables in each disjunctive component. We use the live variables to simplify transition relations and reachable state subsets. Our experiments on a set of real-world C programs show that the new algorithm achieves orders-of-magnitude performance improvement over the best known conjunctive image computation algorithm.

1 Introduction

Symbolic model checking [1], a widely accepted technique in hardware verification, is also showing promises for verifying embedded software programs and device derivers [2]. In this paper, we consider verifying C programs that include integer arithmetic, pointers, arrays, function calls, and bounded memory allocation. Although program verification in general is undecidable—it is equivalent to the halting problem of a Turing machine—the problem becomes decidable under certain conditions. Here we consider the case where the number of recursive function calls and the data size are bounded. Note that in practice both recursive functions and dynamic memory allocation are strongly discouraged in embedded software programs that demand a higher degree of reliability. With these assumptions, we can build a finite state model from the software program and apply model checking.

Software models have some characteristics that make them significantly different from hardware models. For instance, software models often have larger sequential depths and many more state variables. Program variables also have a higher degree of locality—many are effective only in parts of the program. At any program location, only a very limited number of variables can change their values. In contrast, most state variables (or latches) in hardware are updated at every clock cycle and can not be easily localized. Due to these differences, existing symbolic model checking algorithms [3, 4, 5], although fine-tuned for handling hardware, do not work well on software models.

In this paper, we propose a symbolic image computation algorithm that exploits the unique characteristics of software models. It disjunctively decomposes the computation into a set of steps that can be performed in isolation on submodules. Breaking the expensive computation into a set of cheaper ones can significantly reduce the peak memory size during image computation. Our algorithm for creating the submodules is geared towards exploitation of variable locality. Using a hypergraph partitioning heuristic, we are able to produce a small set of submodules and at the same time minimize the number of live variables. We further improve the performance by preventing irrelevant variables from appearing in the transition relations, and by existentially quantifying dead variables from the reachable state subsets.

We have implemented and evaluated our new algorithm using the public domain symbolic model checker VIS [6]. We demonstrate, on a set of typical embedded software programs, that our new algorithm outperforms the best known conjunctive image computation algorithms in terms of both CPU time and memory usage. The improvement is both consistent and significant (up to orders-of-magnitude).

After establishing notation in Section 3, we present our disjunctive image computation algorithm in Section 4. The application of relevant and live variables to simplify the computation is explained in Section 5. We illustrate in Section 6 the use of hypergraph partitioning to minimize the number of live variables. We give the experimental results in Section 7, and then conclude in Section 8.

2 Related Work

Partitioned transition relations for symbolic image computation were proposed in [7, 9] in both disjunctive and conjunctive forms. In [8], multiple variable orders were used together with partitioned ROBDDs [10] to reduce the peak

3-9810801-0-6/DATE06 © 2006 EDAA

memory usage in reachability analysis. However, these works were not targeted to software models. Note that previous applications of disjunctively partitioned transition relation were not successful for hardware models, since creating a good disjunctive partition itself is a non-trivial task. Our work demonstrates that disjunctive partitioning is naturally suited for software models. It is different from the prior work in the criteria we use for decomposition and in our software-specific simplifications.

In [11], Edwards, Ma and Damiano applied a commercial model checker to software by synthesizing C programs into circuits. However, only very small programs can be directly verified. Although they pointed out that model checking algorithms must be re-engineered, they did not provide any solution. Ball and Rajamani presented in [2] a tool for verifying Boolean programs abstracted from C code. Their underlying algorithm was a generalization of an inter-procedural data flow analysis algorithm [12]. Our work is different since it builds upon symbolic model checking and therefore takes the full advantage of the decade-long research in BDD based algorithms and matured implementations (e.g. SMV[1] and VIS[6]).

The algorithm by Barner and Rabinovitz [13] was also based on symbolic model checking and used disjunctively partitioned transition relations. However, their partitioning method is completely different, since it requires a conjunctive transition relation and expensive *and-quantify* operations in order to build disjunctive transition relations. In contrast, we do not need the entire transition relation nor quantification operation in order to build the disjunctive transition relations. Furthermore, they did not exploit the variable locality which we use both for decomposition and for subsequent optimizations.

To summarize, the main contributions of our paper are to propose a new method for deriving and using disjunctively partitioned transition relations for software model checking, and further optimizations using variable locality information derived from the static analysis of the given program.

3 Software Model

We first explain how the verification model is constructed from a software program, and then review symbolic model checking in this context.

In our software modeling approach [14, 15], C programs are first preprocessed through source code rewriting to remove complex control structures and expressions. After that, only bounded integer variables, assignment statements and simple arithmetic expressions remain. Arithmetic expressions are modeled by instantiating pre-defined Boolean logic components (e.g. adders and multipliers). Next, we group statements into basic blocks, for each of which we assign a program location. A set of binary variables called

Table 1. An example of the control flow table.

present state				next state		
p_3	p_2	p_1	guard	q_3	q_2	q_1
0	0	0	$(x \equiv 5)$	0	0	1
0	0	0	$(x \not\equiv 5)$	1	0	0
0	0	1	true	0	1	0
0	1	0	true	0	1	1
...				...		

Program Counter (PC) variables are created to encode the program locations. The set of all program variables and PC variables, together with their next-state functions, define the finite state verification model.

Let the model be represented in terms of (1) a set of present-state program variables $X = \{x_1, ..., x_N\}$ and PC variables $P = \{p_1, ..., p_M\}$, and (2) a set of next-state program variables $Y = \{y_1, ..., y_N\}$ and PC variables $Q = \{q_1, ..., q_M\}$. Let δ_{y_i} and δ_{q_i} denote the next-state functions of y_i and q_i, respectively. We have

$$\delta_{y_i}(X, P) = \bigvee_j (P \equiv j) \wedge e_{i,j}(X) \ ,$$

where $j \in \{1, 2, ..., K\}$ is a PC location and $e_{i,j}(X)$ is the right-hand side of an assignment to y_i at location j. Note that $e_{i,j} = x_i$ if there is no assignment to y_i at location j. We build the next-state functions of PC variables similarly. For example, for the program flow in Table 1 (*guard* is the condition under which a transition is made), we have

$$\begin{aligned} \delta_{q_3} &= (P \equiv 0 \wedge x \not\equiv 5) \vee \ldots \ , \\ \delta_{q_2} &= (P \equiv 1) \vee (P \equiv 2) \vee \ldots \ , \\ \delta_{q_1} &= (P \equiv 0 \wedge x \equiv 5) \vee (P \equiv 2) \vee \ldots \ . \end{aligned}$$

The model can be represented symbolically as $\langle T, I \rangle$, where $T(X, P, Y, Q)$ is the transition relation and $I(X, P)$ is the initial state predicate. Both are Boolean functions that can be represented by BDDs.

$$T = \prod_{1 \le i \le N} T_{y_i}(X, P, y_i) \wedge \prod_{1 \le l \le M} T_{q_l}(X, P, q_l) \ ,$$

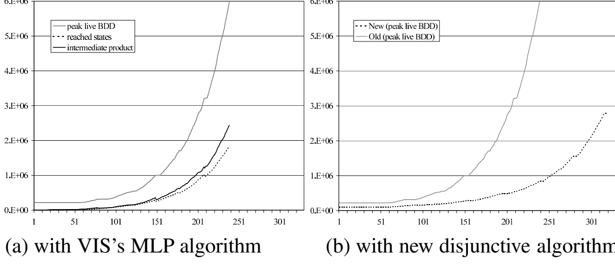
where T_{y_i} and T_{q_l} are the bit-relations defined as follows,

$$\begin{aligned} T_{y_i}(X, P, y_i) &= y_i \leftrightarrow \delta_{y_i}(X, P) \ , \\ T_{q_l}(X, P, q_l) &= q_l \leftrightarrow \delta_{q_l}(X, P) \ . \end{aligned}$$

Image computation is the most fundamental step in symbolic model checking. The *image* of a set of states D consists of all the successors of D with respect to T. Denoted by $\mathsf{EY}_T\, D$, the image of D is given as follows,

$$\mathsf{EY}_T\, D = \exists X, P \,.\, T(X, P, Q, Y) \wedge D(X, P) \ .$$

Our discussion in this paper will be focused on checking reachability properties, since the extension to other properties is straightforward. The reachability analysis can be performed by starting from I and then repeatedly adding the image of already reached states until a fix-point.



(a) with VIS's MLP algorithm (b) with new disjunctive algorithm

Figure 1. PPP: BDD sizes in reachability analysis.

4 Disjunctive Image Computation

The best known symbolic algorithms [3, 4] do not work well when they are applied directly to the software models. Fig. 1-(a) shows the data on a C program from a Linux implementation of Point-to-Point Protocol (PPP), whose verification model has 1435 binary state variables. We encoded the model in BLIF-MV format and then ran reachability analysis with VIS [6] (with dynamic reordering). The three exponentially growing curves represent at each step the total BDD size, the size of reached states, and the peak size of intermediate products. CPU time also grows in a similar fashion. Due to the large number of program variables in software models, such an exponential growth can quickly deplete the memory resources.

4.1 Decomposition of Transition Relation

The transition relation T of a software model can be decomposed naturally into a union of disjunctive components, one for each program location. Since existential quantification $\exists$ distributes over $\vee$, we can compute individual images with smaller transition relation components. This can significantly reduce the peak memory usage at each reachability step.

Let (j/P) represent the substitution of P with the integer value j; similarly, let $f(X/Y)$ represent the substitution of Y variables with X variables inside Function f. By definition, we have

$$\delta_i(X, j/P) = e_{i,j}(X) \ ,$$

where $e_{i,j}(X)$ is actually the cofactor of $\delta_i(X, P)$ with respect to $(P \equiv j)$. The cofactors of transition bit-relations with respect to $(P \equiv j)$ are given as follows,

$$\begin{aligned} (T_{y_i})_{(P\equiv j)} &= (y_i \leftrightarrow \delta_{y_i})_{(P\equiv j)} \\ &= (y_i \wedge \delta_{y_i} \vee \neg y_i \wedge \neg\delta_{y_i})_{(P\equiv j)} \\ &= y_i \wedge (\delta_{y_i})_{(P\equiv j)} \vee \neg y_i \wedge \neg(\delta_{y_i})_{(P\equiv j)} \\ &= y_i \wedge e_{i,j} \vee \neg y_i \wedge \neg e_{i,j} \\ &= y_i \leftrightarrow e_{i,j} \end{aligned}$$

and similarly $(T_{q_l})_{(P\equiv j)} = q_l \leftrightarrow e_{l,j}$.

```
for ( i = 0; i < K; i++ ) {
    new = EY_{T_i} ( R[i] );
    for ( j = 0; j < K; j++ ) {
        R[j] = R[j] ∪ (new ∧ P ≡ j);
    }
}
```

Figure 2. Distribution of image computation results.

By definition $T = \bigvee_j (P \equiv j) \wedge (T)_{(P\equiv j)}$, where

$$(T)_{(P\equiv j)} = \prod_{1\le i\le N} (y_i \leftrightarrow e_{i,j}) \wedge \prod_{1\le l\le M} (q_l \leftrightarrow e_{l,j}) \ .$$

Since existential quantification distributes over $\vee$,

$$\begin{aligned} \mathsf{EY}_T\, D &= \exists X, P \,.\, D \wedge T \\ &= \exists X, P \,.\, D \wedge \textstyle\bigvee_j (P \equiv j) \wedge (T)_{(P\equiv j)} \\ &= \textstyle\bigvee_j \exists X, P \,.\, D \wedge (P \equiv j) \wedge (T)_{(P\equiv j)} \end{aligned}$$

There can be one disjunctive component for every PC location j. However, for efficiency purposes, we often merge multiple locations and then build a disjunctive component for each cluster. In Section 6 we will give a heuristic algorithm for the merging, which simultaneously minimizes the number of live variables in each cluster.

Note that the decomposition into $(T)_{(P\equiv j)}$ is based on the PC locations, not on individual program variables as in [13]. The method in [13] builds one transition relation disjunct for each variable, which often defeats the purpose of exploiting variable locality. Another significant difference is that, we create $(T)_{(P\equiv j)}$ directly from the software program, while they rely on the existing conjunctive transition relation and expensive existential quantification operations. In practice, building the conjunctive transition relation itself may be computationally expensive or even infeasible.

4.2 Decomposition of Reachable States

The reachable states are also represented as the union of many subsets, one for each cluster,

$$R(X, P) = \bigvee_j (P \equiv j) \wedge R(X, j/P) \ .$$

Since the image of $R(X, i/P)$ may be in a different location j, we need to redistribute images after every step (shown in Fig. 2). Note that an optimization of the algorithm based on control flow structure can make the complexity of redistribution $O(E)$, where E is the number of edges in the control flow graph.

The procedure in Fig. 2 computes reachable states Frame By Frame (FBF). Alternatively, we can compute reachable states Machine by Machine (MBM); that is, the analysis is

performed on one individual cluster until it converges, after which the result is propagated to other clusters. MBM minimizes the traffic (data transfer) among different clusters and therefore is appealing when a distributed implementation is used. This is analogous to the approximate FSM traversal algorithm of [16], with the difference that we build the transition relations without approximation and our reachable states are alway exact.

There are two different ways of implementing the disjunctive algorithm using BDDs. In the first approach, all transition relations and reachable subsets are represented in a single BDD manager (following the same variable order). Alternatively, we can allocate BDD managers for different clusters so that the variable orders are tailored towards individual clusters. We have implemented both approaches, and found no significant performance improvement of the multi-manager approach. This is due to the large overhead of transfering BDDs from one variable order into another.

5 Simplification Using Variable Locality

5.1 Relevant Variables

Definition 1 *The set of* relevant *variables with respect to location j, denoted by X_j^R, are those appearing in either the assignments or conditional expressions of block j. The set $X_j^I = X \setminus X_j^R$ consists of* irrelevant *variables.*

The contribution of an irrelevant variable to the transition relation is of the form $(y_i \leftrightarrow x_i)$; hence $(T)_{(P\equiv j)}$ is

$$\prod_{x_i \in X_j^I} (y_i \leftrightarrow x_j) \wedge \prod_{x_i \in X_j^R} (T_{y_i})_{(P\equiv j)} \wedge \prod (T_{q_l})_{(P\equiv j)} \quad .$$

Although $(y_i \leftrightarrow x_i)$ can be represented by a BDD with 3 nodes, conjoining many of them together is known to produce BDDs with exponential number of nodes in the worst case. On the other hand, a good BDD order for these constraints may be bad for other Boolean formulae encountered in reachability analysis.

Inside image computation, the equality constraints facilitate the substitution of x_i with y_i for all irrelevant variables. Unfortunately, existing quantification scheduling algorithms [3, 4] often fail to identify these variables. Since quantification of X_j^I has the same effect as substitution, we choose not to include these constraints in $(T)_{(P\equiv j)}$ in the first place, to avoid the potential BDD blow-up during quantification. Note that $\mathsf{EY}_{(T)_{(P\equiv j)}} D(X,P) =$

$$\exists X_j^R, P \,.\, D(X_j^I / Y_j^I) \wedge \prod_{x_i \in X_j^R} (T_{y_i})_{(P\equiv j)} \wedge \prod (T_{q_l})_{(P\equiv j)} \quad .$$

Let $R(Y,Q)$ be the result of the above EY operator, we still need to substitute all the Y and Q with the corresponding present-state variables to get $R(X,P)$. Therefore, for irrelevant variables the substitutions need to be done twice. In our actual implementation, we remove the equality constraints from the transition relation and avoid substitutions in both directions. Our experimental studies show that this significantly reduces the peak BDD sizes of the intermediate products in image computation.

```
{                         L1: x = y = 0;
    x = a;
    z = x + b;            L2: x = 7;
}                         L3: s = x;
...                       L4: y = 8;
{                         L5: s = y;
    x = c;
    z = x + d;            L6: goto L2;
}                         L7: ERROR
```

(a) global variable x; (b) early convergence.

Figure 3. Two examples on the variable live scope.

5.2 Live Variables

Even a reachable state subset may have all the program variables in its support, making it hard to find a compact BDD with dynamic reordering (a major reason for the blow up in Fig. 1). However, many variables are local to certain program locations and their values are meaningless at other locations. Locally defined variables, for instance, should be considered as state-holding only inside the blocks where they are defined, since elsewhere their values do not affect the control flow nor the data path. However, by default their values are carried on as Boolean functions in the reachable state subsets, which makes the BDD representation of reachable states unnecessarily large.

Local variables can be easily identified and removed from state subsets. However, even globally defined variables may not be live at all program locations — they may be used only in certain segments of the program. A variable is *live* at a certain program location if its value affects the program's control flow and/or data path. In Fig. 3-(a), for instance, if x appears only in these two blocks, it is considered as dead elsewhere. More formally,

Definition 2 *Program location j is within the* live scope *of variable x_i if and only if there is an execution path from j to a location where the value of x_i is used.*

The live scope of a variable x_i is computed as follows using standard static analysis: First, find all program locations where x_i is used (i.e. in an assignment or a conditional expression); then trace back from these locations until reaching locations where the value of x_i is changed. All locations visited during the process are in the live scope of x_i. Let K be the number of program locations, E be the number of

Table 2. New states after every reachability step.

Line#	With all program variables	With live variables only
1	$P \equiv 2 \wedge x \equiv 0 \wedge y \equiv 0$	$P \equiv 2$
2	$P \equiv 3 \wedge x \equiv 7 \wedge y \equiv 0$	$P \equiv 3 \wedge x \equiv 7$
3	$P \equiv 4 \wedge x \equiv 7 \wedge y \equiv 0 \wedge s \equiv 7$	$P \equiv 4 \wedge s \equiv 7$
4	$P \equiv 5 \wedge x \equiv 7 \wedge y \equiv 8 \wedge s \equiv 7$	$P \equiv 5 \wedge s \equiv 7 \wedge y \equiv 8$
5	$P \equiv 6 \wedge x \equiv 7 \wedge y \equiv 8 \wedge s \equiv 8$	$P \equiv 6 \wedge s \equiv 8$
6	$P \equiv 2 \wedge x \equiv 7 \wedge y \equiv 8 \wedge s \equiv 8$	$\emptyset$
2'	$P \equiv 3 \wedge x \equiv 7 \wedge y \equiv 8 \wedge s \equiv 8$	$\emptyset$
3'	$P \equiv 4 \wedge x \equiv 7 \wedge y \equiv 8 \wedge s \equiv 7$	$\emptyset$
4'	$\emptyset$	$\emptyset$

transitions in the control flow, and N be the number of state variables, the complexity of this process is $O((K+E)\times N)$. Compared to model checking, the overhead is negligible.

We use the live variable analysis for the following optimization. During the redistribution of image results (Fig. 2), all variables that are not live (called dead variables) in that destination location can be existentially quantified out.

Removing dead variables from reachable subsets not only reduces the BDD sizes, but also leads to a potentially faster convergence of reachability analysis. Take Fig. 3-(b) as an example, where we assume that each statement is a basic block and all variables are global. Variable x is live in L2-3 and y is live in 4-5. By removing x and y from the reachable state subsets wherever they are dead, one can declare the termination of reachability analysis after going from L1 through L6 only once (shown in Table 2). Otherwise, reachability analysis needs two more steps to converge, since after L6, the new state ($P \equiv 2 \wedge x \equiv 7 \wedge y \equiv 8 \wedge s \equiv 8$) is not covered by the already reached state ($P \equiv 2 \wedge x \equiv 0 \wedge y \equiv 0$).

6 Creating the Disjunctive Partition

We now explain how to merge basic blocks into disjunctive clusters. Although considering each block as a separate cluster maximizes the number of dead variables and irrelevant variables, the often large number of basic blocks encountered in practice (which can easily be in the thousands) may negate any benefits. Therefore, we want to make fewer clusters but still retain the benefit of variable locality. We formulate it as a recursive bi-partitioning problem.

We start with all basic blocks in a single cluster and then perform recursive bi-partitioning. A cost function and a predetermined threshold are needed to specify when bi-partitioning should be stopped. The actual BDD size of the transition relation disjunct is an ideal cost function since we want to keep individual transition relations small. However, it would be too expensive to build the actual BDD in order to get its size. Instead, we use the number of relevant variables as cost, since the number of support BDD variables is often a good indicator of the BDD size.

Next, we define what to minimize during bi-partitioning. The candidate optimization criteria are: (1) the number of variables live in both clusters (*liveVar*); (2) the number of variables changing their values in both clusters (*asgnVar*); and (3) the number of control flow transitions between the two clusters (*cfgEdge*). Note that less shared live variables indicates a higher degree of variable locality, since more dead variables can be removed from the reachable state subsets. Less shared assigned variables means that less bit-relations are grouped together. Less shared CFG edges means less traffic among different clusters during the redistribution of image results.

All three optimization criteria can be cast into hypergraph partitioning problems, which differ only in the way hyperedges are defined. We represent individual basic blocks as hypernodes, and use hyperedges to represent shared live variables, assignment variables, or control flow transitions. For *liveVar* and *asgnVar*, we add a hyperedge for each variable to connect all blocks where it is live or assigned; for *cfgEdge*, we add for each transition an edge to connect the head and tail blocks. We then compute a partition of this hypergraph such that the number of hyperedges across different groups is minimized.

Although our live variable based partitioning method is similar to the MLP algorithm of [4], they are designed for different applications. MLP computes a quantification schedule for a set of conjunctively partitioned transition relations; while our algorithm groups disjunctive transition relations into larger clusters. Nevertheless, just like the impact of quantification schedule on image computation, a good disjunctive partition is also important for the performance of disjunctive image computation.

7 Experiments

We have implemented our new algorithm using the symbolic model checker VIS [6] and the hMeTis hypergraph partitioning package [17]. We encode our verification models in VIS's BLIF-MV format. The performance evaluation was conducted by comparing to the best known image computation method [4] in VIS 2.0. All the experiments were run on a workstation with 2.8 GHz Xeon processors and 4GB of RAM running Red Hat Linux 7.2. BDD variable reordering was enabled with method *sift*. For the purpose of controlled experiments, all image computation related parameters in VIS were kept unchanged.

Our benchmarks are C programs from public domain as well as industry, including Linux device drivers, file systems, and software in portable devices. For all test cases we check reachability properties. We also apply range analysis to reduce the number of bits needed to encode the program variables. To test the sheer capacity of our algorithm, verification models are generated without predicate abstraction, which means our models are significantly more com-

Table 3. Comparison in reachability analysis.

Test Cases			Completed		CPU Time (s)		Peak BDD (k)	
name	vars	dep.	old	disj	old	disj	old	disj
ssdf	37	11	Y	Y	0.01	0.02	0.8	0.8
sfi	105	47	Y	Y	0.5	0.6	4	4
sirpp	169	52	Y	Y	8	6	10	10
srb	343	43	Y	Y	2766	146	925	106
core1	416	211	Y	Y	1115	89	155	51
core2	445	192	70	Y	>2h	80	490	70
srr	856	316	Y	Y	5426	151	990	57
smhb	888	104	25	Y	>2h	341	1219	120
siic	967	162	Y	Y	4020	260	1188	79
spr	1001	617	85	Y	>2h	1617	428	430
sdir	1050	209	136	Y	>2h	394	3135	120
srdo	1211	215	128	Y	>2h	496	2482	145
core3	1213	189	41	Y	>2h	205	139	101
ppp	1435	?	208	277	>2h	>2h	3194	540
core4	4759	?	6	47	>2h	>2h	2949	1021
daisy	8392	?	0	83	>2h	>2h	-	480

plex than the Boolean programs in [2].

First, we give the performance comparison on PPP in Fig. 1-(b), which shows the peak memory usage at each step. Within the 4 hour time limit, the conventional method completed 238 steps and *New* completed 328 steps. The result shows that our new disjunctive image computation method reduces the peak memory usage significantly, and the reduction in BDD size also translates into reduction in CPU time. Table 3 gives our comparison on a larger set of benchmarks (with 2 hour time limit). Columns 1-3 give for each model the name, the number of state variables, and the sequential depth. Columns 4-5 indicate whether reachability analysis was completed (if not, the maximal depth is given). Columns 6-9 compare the CPU time and the peak number of live BDD nodes. Note that for *daisy*, the old method timed out while building the transition relation. The results show that the performance improvement of our new algorithm is both significant and consistent.

We also evaluated the impact of three partitioning heuristics on the performance of disjunctive image computation. The partition threshold was set to 175 relevant variables; the runtime of hMetis was found to be negligible. The result table is omitted due to the space limit. Our results show that partitioning heuristics have a significant impact on the performance. We found that the *liveVar* based heuristic performs better than the others on most of the harder cases and it tends to give a smaller number of disjuncts.

8 Conclusions

We have presented a disjunctive image computation algorithm for software model checking to exploit the unique characteristics of sequential software models. By dividing an expensive computation into a set of easier ones and applying program variable locality to simplify the BDD representation, our new method significantly reduces the CPU time and the peak memory usage required. Although previous experience with hardware verification shows that symbolic model checking often loses robustness when the number of state variables exceeds 200, our work demonstrates that by exploiting the domain-specific characteristics, BDD based algorithms can *directly handle* software models with thousands of variables.

References

[1] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994.

[2] T. Ball and S. K. Rajamani. Bebop: A symboloc model checker for Boolean programs. In *Proc. of the SPIN Workshop (SPIN'00)*, pages 113–130. LNCS 1885.

[3] R. K. Ranjan, A. Aziz, R. K. Brayton, B. F. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. Presented at IWLS95, May 1995.

[4] I.-H. Moon, G. D. Hachtel, and F. Somenzi. Border-block triangular form and conjunction schedule in image computation. In *Formal Methods in Computer Aided Design*, pages 73–90. LNCS 1954.

[5] C. Wang, G. D. Hachtel, and F. Somenzi. The compositional far side of image computation. In *Proc. of Int. Conf. CAD (ICCAD'03)*, pages 334-340.

[6] R. K. Brayton et al. VIS: A system for verification and synthesis. In *Computer Aided Verification (CAV'96)*, pages 428–432. LNCS 1102.

[7] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proc. of Design Automation Conference (DAC'91)*, pages 403–407.

[8] A. Narayan, A. J. Isles, J. Jain, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Reachability analysis using partitioned ROBDDs. In *Proc. of Int. Conf. CAD (ICCAD'97)*, pages 388–393.

[9] G. Cabodi, P. Camurati, L. Lavagno, and S. Quer. Disjunctive partitioning and partial iterative squaring: an effective approach for symbolic traversal of large circuits. In *Proc. of Design Automation Conference (DAC'97)*, pages 728–733.

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

[11] S. Edwards, T. Ma, and R. Damiano. Using a hardware model checker to verify software. Presented at *Int. Conf. on ASIC*, 2001.

[12] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167:131–170, 1996.

[13] S. Barner and I. Rabinovitz. Efficient symbolic model checking of software using partial disjunctive partitioning. In *Correct Hardware Design and Verification Methods (CHARME'03)*, pages 35–50. LNCS 2860.

[14] F. Ivančić, Z. Yang, I. Shlyakhter, M. Ganai, A. Gupta, and P. Ashar. F-SOFT: Software verification platform. In *Computer-Aided Verification*, pages 301–306, 2005. LNCS 3576.

[15] F. Ivančić, I. Shlyakhter, A. Gupta, M. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking C program using F-Soft. In *Proc. of Int. Conf. on Computer Design*, pages 297–308, 2005.

[16] H. Cho, G. D. Hachtel, E. Macii, M. Poncino, and F. Somenzi. Automatic state space decomposition for approximate FSM traversal based on circuit analysis. *IEEE Trans. on CAD*, 15(12):1451–1464, 1996.

[17] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. Technical Report 98-019, University of Minnesota, 1998.