Optimizing Model Checking Based on BDD Characterization

Bwolen Yang

May 1999

CMU-CS-99-129

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:

David R. O'Hallaron, Chair
Randal E. Bryant
Edmund M. Clarke
Thomas R. Gross
Kenneth L. McMillan, Cadence Berkeley Labs

Effort sponsored in part by the Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287, in part by the National Science Foundation under Grant CMS-9318163, and in part by a grant from the Intel Corporation. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

Keywords: formal verification, model checking, time-invariant constraints, conjunctive partitioning, Symbolic Model Verifier (SMV), Binary Decision Diagram (BDD), BDD evaluation methodology, BDD performance study

Abstract

Symbolic model checking has been successfully applied in verification of various finite state systems, ranging from hardware circuits to software protocols. A core technology underlying this success is the Binary Decision Diagram (BDD) representation. Given the importance of BDDs in model checking, it is surprising that there has been little or no work on studying BDD computations in the context of model checking. As a result, the computational aspects of BDDs are not well understood and many BDD-based algorithms tend to be unstable in terms of performance. This thesis addresses the performance instability issue both by developing a general evaluation methodology for studying BDD computations and by proposing new BDD-based optimizations to stabilize and to improve the overall performance.

The evaluation methodology consists of two parts: (1) a set of evaluation metrics characterizing key components of BDD computations, and (2) a trace-based evaluation platform for generating realistic benchmarks from computational traces of BDD-based tools and for replaying these traces on BDD packages. This methodology allows BDD-package designers to study and tune their packages based on realistic computations. This is the *first* evaluation methodology for studying BDD computation systematically.

Based on this evaluation methodology, we have designed and conducted a BDD performance study in the context of model checking. This study is a collaborative effort among six BDD designers using their own BDD packages. The study has resulted in significant performance improvements (with some speedups over 100) and several characterizations of model-checking computations; e.g., this study showed that computational characteristics of model checking are inherently different from those of combinational equivalence checking. These results demonstrate the importance of systematic evaluation and validate our methodology.

Using a similar systematic approach, I have stabilized and improved the performance of an important model-checking optimization called *conjunctive partitioning* and have derived a new algorithm for verifying constraint-rich systems. In both cases, the information encoded in the BDD representation is used to drive the optimizations. For conjunctive partitioning, the set of variables, the graph sizes, and *tentative* BDD operations are used to heuristically order and merge the partitions of the transition relation. For verifying constraint-rich systems, I developed a new BDD-based algorithm, called *assignment-extraction algorithm*, to establish relationships between state variables. This assignment-extraction algorithm decomposes any Boolean expression into assignment expressions. From these assignments, we can more precisely determine the set of variables that can be replaced with equivalent expressions (*macro expansion*). The goal is to remove unnecessary state variables to reduce the overall state space. As with the improvements to conjunctive partitioning, BDD characteristics are used here to stabilize the optimization.

Our systematic approach to study and to improve the underlying BDD computations culminated in a significantly improved version of the SMV model checker that has helped other researchers tackle real-world applications. In particular, our approach has enabled the verification of the fault diagnosis model of NASA's Deep Space One spacecraft.

Acknowledgements

First and foremost, a big *thank you* to my advisor Dave for his support and enthusiasm. He has painstakingly taught me how to do research and tutored me on how to present research results effectively. I am particularly grateful to him for giving me the freedom in research topics and for believing in my ability.

I thank Randy, my virtual advisor, for giving me guidance on research directions and for always being available to answer questions. He has been a great supporter of my research. I am also grateful to Ed, Thomas, and Ken for their enthusiasm and for their comments and suggestions about this work.

Jim has been a big help to my research on the Fx parallelizing compiler. I thank Yirng-An for starting my thesis work on BDDs four years ago. He has greatly facilitated my transition from parallelizing compiler to BDDs.

I thank Claudson and Henry for being my sounding board for research ideas and random complaints. They have been amazing friends throughout this PhD process and the completion of their theses has been a great motivating force for me to finish sooner. I thank James for teaching me about the American culture and for sharing his horror stories, and most of all, for relentlessly quoting me out of context. I thank Sharon, Karen, and Catherine for being extremely nice and for taking care of administrative details. I thank Lisa for keeping my wrists healthy. Sam and Agnes have been a great source of yummy homemade food. Tammy has brought special meanings to the words *purple* and *pecan*. They and other friends—Jennifer, Arup, Nita, Girija, Xuemei, Jas, Chi, Margret, Huifen, Sanjit, Peter, Shipra, Armin, Yunshan, Yuan, and several others—have made this past seven years a rather fun experience. And despite what you might have heard about Pittsburgh, I really enjoyed living in it.

I am very grateful to my parents and my sisters, particularly Heng-Yin, for their love and support. I thank my grandparents, uncles and aunts for providing me a home-away-from-home. Finally, special thanks to Shun-Lien for helping me through many difficult times.

Contents

Chapter 1

Introduction

Model checking is an automatic verification paradigm that checks temporal properties (e.g., safety and liveness) of finite state systems. It has been successfully applied to a variety of applications, ranging from hardware designs to software protocols. A core technology enabling these successes is the use of *Binary Decision Diagrams* (BDDs) to model the state space and to perform the underlying computation.

Research efforts in this field focus mainly on high-level optimizations, such as *abstraction*, *compositional reasoning*, and *symmetry reduction* to reduce the size of the state space, while the computational aspects of BDDs have largely been ignored. As a result, the performance of many model checking algorithms can be unstable; i.e., small changes to the parameters can have dramatic performance impact. This performance instability diminishes model-checking's advantage of being fully automatic, because expert knowledge is often needed to fine tune the parameters. Furthermore, the performance instability has impeded research progress because the impact of new algorithms is difficult to evaluate.

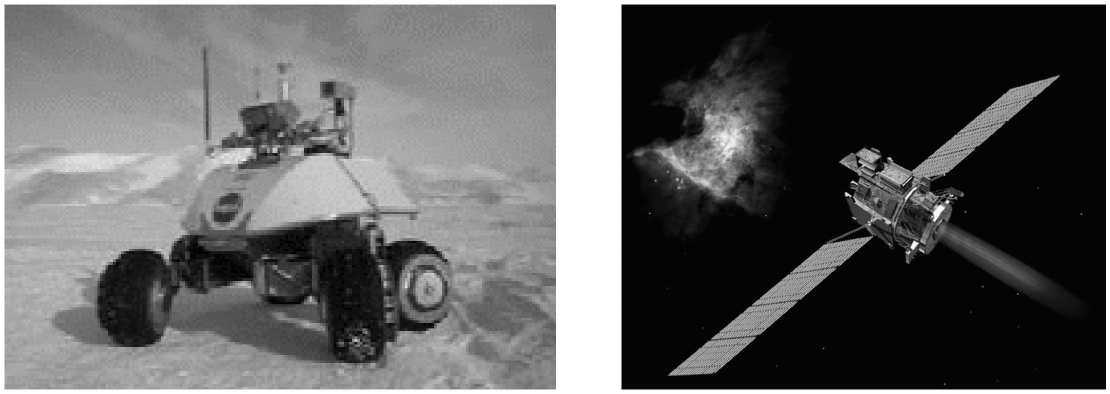
In this thesis, I address the performance instability problem by systematically studying BDD computations. The main contribution is the first evaluation methodology for general BDD computations. Using this evaluation methodology and in collaboration with other BDD package designers, I have conducted a BDD performance study in the context of model checking. This study resulted in better understanding of model-checking computations and significant performance improvements. This evaluation methodology has helped to raise BDD evaluations to a more scientific basis, and it is now gradually being adopted by the BDD community. For example, researchers have used this methodology to extend a BDD variant called MORE to cover model checking computations.

Using a similar approach, I have systematically studied computational aspects of various stages of model-checking computations. In particular, I have stabilized the performance of an important model-checking optimization called *conjunctive partitioning*, and have derived a new algorithm for verifying constraint-rich systems. This thesis culminated in a much improved SMV model checker [58] that has enabled the verification of complex systems such as the fault diagnosis models of the Nomad robot (an Antarctic meteorite explorer) and the NASA Deep Space One spacecraft (Figure 1.1).

In the rest of this introduction chapter, I will explain the importance of model checking and describe the performance instability problem that motivates my thesis.

1.1 Why Model Checking?

Today, the complexity of both hardware and software designs is growing very rapidly. Ensuring their correctness has correspondingly become a much more difficult problem. On the other hand, hardware and software are now an integral part of our daily life. Their correctness is also correspondingly becoming an indispensable part of the design process. In particular, for security and safety critical applications, such as



(a) (b)

Figure 1.1: Autonomous systems: (a) Nomad (b) Deep Space One.

automobiles, medical instruments, highway and air traffic control systems, and electronic commerce, failure can have catastrophic consequences. For example, on June 4th, 1996, a software error caused the Ariane 5 rocket to explode shortly after it was launched. Even for non-safety critical devices, design errors can have huge economic cost, as demonstrated by the Pentium bug that costed Intel almost half a billion dollars in 1994.

There are four principal verification techniques: simulation, testing, theorem proving, and model checking. Before we proceed to describe each of these techniques, it is important to note that the concept of correctness is not absolute. We can only ensure a design satisfies a given specification. We cannot determine if the specification covers all the necessary properties.

Traditionally, simulation and testing are the validation techniques of choice. Both techniques validate the design by observing the outputs of test input patterns. The simulation is performed on a model of the design, while the testing is performed on the actual product. These two methods are very effective, particularly early in the design process when there are many errors. However, they often cannot guarantee the absence of error as checking all input combinations is rarely possible for complex systems. Theorem proving and model checking are formal techniques used to address this problem.

Theorem proving uses axioms and proof rules to prove the correctness of systems. Historically, such proofs were constructed entirely by hand. Today, there are powerful software tools (*theorem provers*), such as HOL [43] and PVS [65], that enforce the correct use of axioms and proof rules and suggest possible ways to make progress in the proof. Theorem proving is a very powerful approach and can be applied to infinite state systems. However, proving the correctness of a single circuit or a protocol can involve manual guidance in proving hundreds and thousands of lemmas in detail. For example, over 1600 definitions and lemmas are involved to prove the correctness of the floating-point division algorithm used in the AMD 5K86 processor [14]. Even for the more mathematically inclined designers, this task can be prohibitively daunting. Thus designers often choose to rely on simulation and testing techniques instead. Another disadvantage of the theorem-proving approach is that there is no bound on the amount of time and memory needed to find a proof.

Model checking, on the other hand, is an automatic technique for verifying finite state systems. By restricting the scope to finite state systems, this technique automatically and exhaustively explores the state space to check whether or not the specification holds. Given enough computational resources, this procedure will always terminate. If a property does not hold, an error trace can often be generated automatically as a witness. For example, if a safety property can be violated, a model checker will generate an error trace indicating how the system can transition from an initial state to a fault state where the safety property fails to hold.

The fully automatic aspect of model checking makes it a preferable choice over theorem proving. Even though model checking is limited to finite-state systems, many real-world applications are either finite state or can be reduced to finite-state systems. In particular, model checking has become an important part of the verification process in hardware companies such as Intel, IBM, and Motorola. Note that there will always be important applications where model checking fails due to large or infinite state space and theorem proving is necessary. A recent research trend has been to merge these two approaches by using model checking for verifying subcomponents and theorem proving to compose the results back together [68].

1.2 State Space Explosion

Historically, model checking was performed by explicit state traversal. An inherent problem of this approach is that the number of states is exponential in the number of system components. *Symbolic model checking* [58] addresses this state explosion problem by using a graph representation called *Binary Decision Diagram* (BDD) [3, 17]. In symbolic model checking, BDDs are used throughout the entire model checking process: from modeling the system to verifying properties and generating counter-examples.

The use of BDDs has greatly increased the power of model checking roughly from 10^5 to 10^{30} states (i.e., from 17 to around 100 state bits). Occasionally, systems with up to 1000 state bits have been successfully verified. However, in terms of real-world applications, 100 state bits (or even 1000) are fairly small. For example, the number of latches in today's digital systems are easily orders of magnitude larger. Thus, there have been many research efforts in reducing the state space. The main focus has been in designing high-level methodologies such as abstraction, compositional reasoning, partial ordering reduction and symmetry reduction.

Abstraction [6, 25, 34, 33, 52] reduces the state space by mapping complex components to simple abstract representations. This is particularly useful for data paths, where large set of data values are mapped to a small set of abstract values.

Compositional reasoning [27, 44, 45, 51, 50, 53, 77] exploits the modular structure in the design by verifying each part of a complex system separately and then composes the results to infer the correctness of the overall system. By focusing the verification on the components rather than the overall system, the size of the state space can be significantly reduced, because the size is exponential in the number of components being verified. When there are mutual dependencies between components, a special strategy called *assume-guarantee reasoning* [45, 51, 50, 62] is used. In this approach, properties of each part are verified by making assumptions of other parts' behaviors. These assumptions must be proved later when the correctness of other parts are proved to be correct. Since these parts may be mutually dependent, this approach seems circular; i.e., proving the correctness of one part depends on making assumptions about the correctness of another part of the system and vice versa. This seemingly circular reasoning is addressed by using induction on time [1, 59]. More specifically, the properties of one part are verified for the current time step by making the assumptions based on the behavior of other parts in the previous time step.

Partial order reduction [41, 42, 79] is targeted to reduce the state space for concurrent asynchronous systems. This optimization is based on the idea that if different execution sequences of asynchronous processes cannot be distinguished by the property we want to check, we only need to consider one representative sequence. Using this approach, the state space can be greatly reduced because we no longer need to consider all possible execution sequences.

Symmetry reduction [24, 37, 48] reduces the state space by replacing symmetric structures in the system by a representative. Symmetry often occurs when components are replicated. This replication arises naturally in both software protocols and hardware designs; e.g., many communication protocols involve interactions between identical processes; in superscalar processors, the functional units are replicated; the memory hierarchy is also full of replicated components such as registers, cache lines, and the main memory.

These state-space reduction techniques target different inherent structures in a model and are complementary to each other. Currently, one disadvantage of these approaches is that they often require human intervention to identify the inherent structures and to describe how to exploit these structures. This process can be labor intensive and may even be error-prone. Much of current research work focuses on how to automate these techniques.

1.3 BDD Performance Instability

Using optimizations described in the previous section (Section 1.2), symbolic model checking has been successfully applied to a variety of applications, ranging from the computer hardware domain, including digital circuit designs, cache coherency protocols, and instruction scheduling algorithms in superscalar processors, as well as to other domains such as the protocols in the airplane Traffic Alert and Collision Avoidance System (TCAS), telephone systems, and reactor systems. The use of BDDs has played a key role in this success.

Given the importance of BDDs in model checking, it is surprising that there has been little research effort to study the computational aspects of BDD and BDD-based algorithms in the context of model checking. Since BDD computations are generally very time and memory intensive, this lack of attention often resulted in performance instability in model-checking algorithms. For example, Figure 1.2 plots the performance of a model checker while varying the partition-size limit of a popular *conjunctive-partitioning algorithm* [69]. Conjunctive partitioning is used to represent the overall transition relation of a finite state system as a set of conjuncts. The partition-size limit (ranging from 1 to 10 million BDD nodes in this experiment) is used to limit the graph size of the BDD representation for each conjunct. The results show that for each partition-size limit, there are always some models that cannot be verified. Furthermore, for many models, improper choice of the partition-size limit can result in a factor of 10 to 100 slowdown and sometimes even result in complete failure.

In general, performance instability is often due to poor evaluation methodologies where validations of new algorithms are based on a few synthetic benchmarks or the use of running time as the only evaluation metric. When evaluating heuristic solutions to complex problems, this lack of systematic analysis often leads to algorithms that perform well only with expert users fine tuning the parameters. As a result, today's model checkers often come with many run-time parameters and manually fine-tuning these parameters is a routine part of the model-checking process. Overall, this performance instability not only diminishes the model-checking's advantage of being fully automatic, it also impedes research progress because it makes the impact of new algorithms difficult to evaluate.

1.4 Scope and Overview of The Thesis

This thesis addresses the performance instability issue by systematically studying the computational aspects of BDD-based model-checking algorithms and deriving new heuristics and algorithms to stabilize and improve the overall performance. This systematic approach culminated in a significantly improved SMV model checker both in terms of stability (Figure 1.3) and running time (Figure 1.4). Figure 1.3 shows the stability-improvement results for the conjunctive-partitioning algorithm described in the previous section. In comparison to the results from the original version (Figure 1.2), our improvements have greatly stabilized this algorithm. In particular, the results show that for a wide range of partition-size limits (from 10 thousand to 1 million BDD nodes), we can verify all the models. Furthermore, within this range, the worst case penalty is reduced to a factor of 15 slowdown in comparison to manually choosing the best partition-size limit for each benchmark model. Figure 1.4 shows the performance improvement over the latest version of

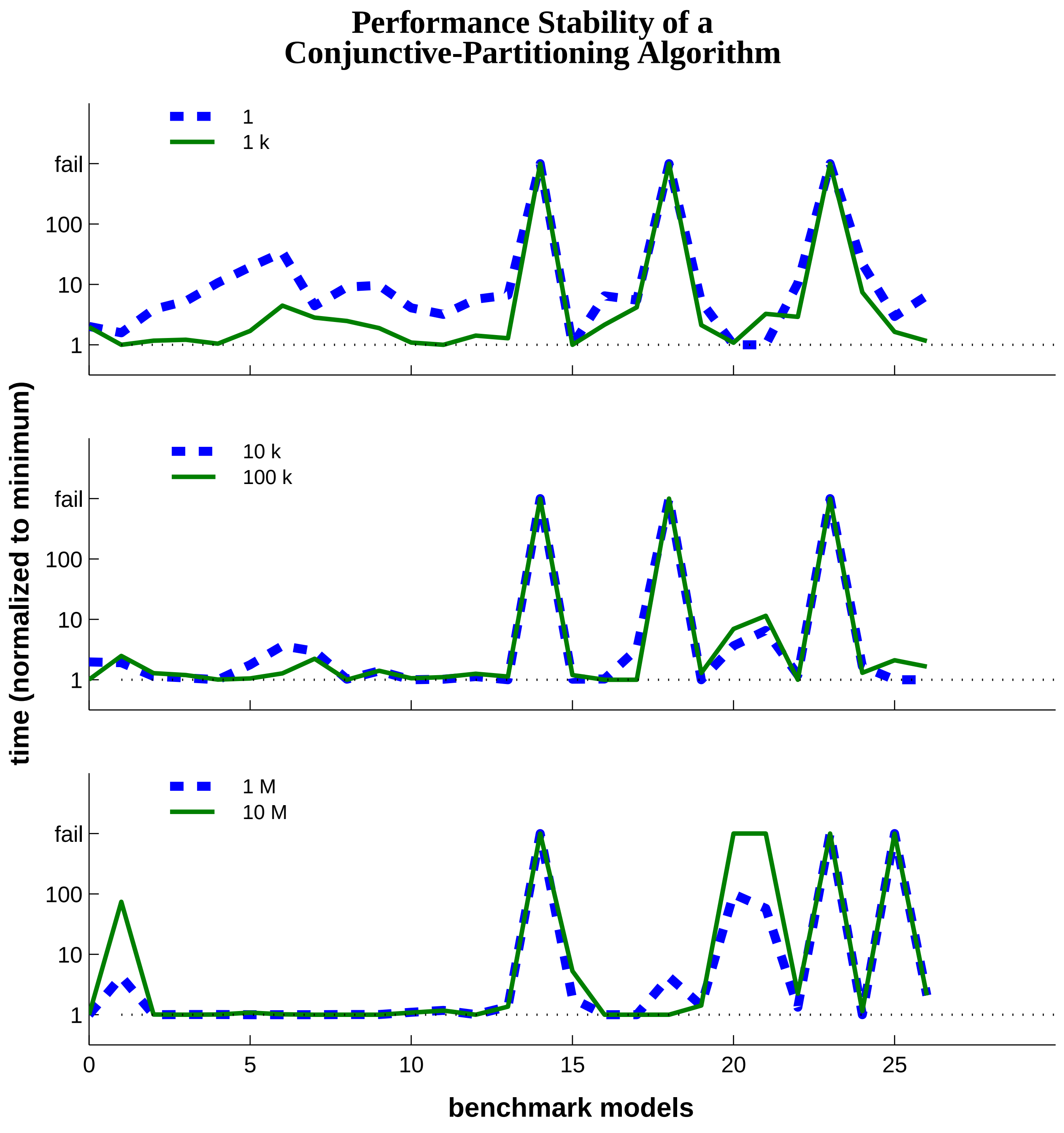


Figure 1.2: Normalized running time of a popular conjunctive-partitioning algorithm on 27 benchmark models. Six partition-size limits used are: 1, 1000, 10,000, 100,000, 1,000,000, and 10,000,000 BDD nodes. For each benchmark model, the running time is normalized against the best running time for that model across different partition-size limits. Note that the six curves shown logically belong to one chart. It is separated into 3 charts for clarity.

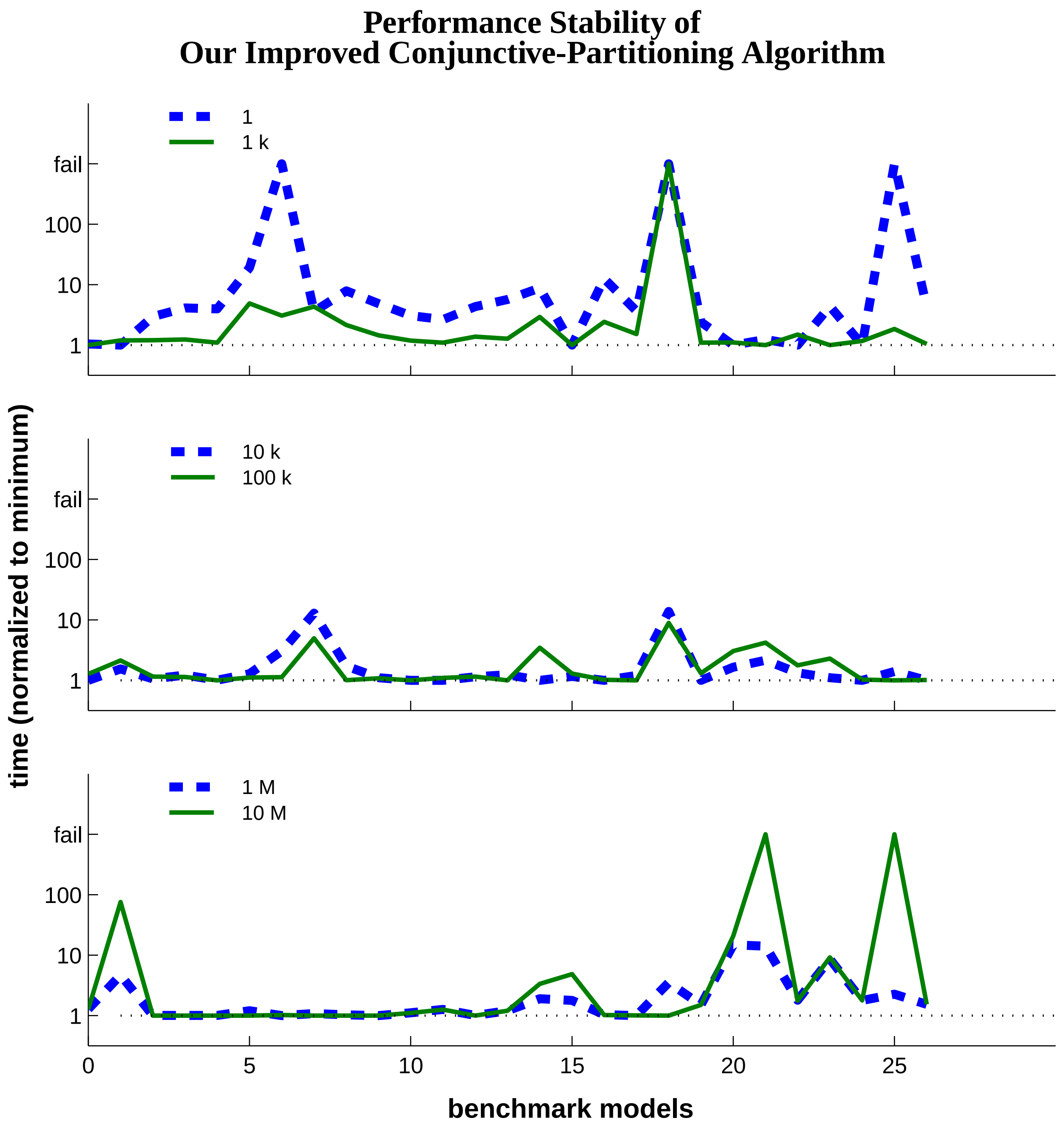


Figure 1.3: Normalized running time of our improved conjunctive-partitioning algorithm on 27 benchmark models. Six partition-size limits used are: 1, 1000, 10,000, 100,000, 1,000,000, and 10,000,000 BDD nodes. For each benchmark model, the running time is normalized against the best running time for that model across different partition-size limits. Note that the six curves shown logically belong to one chart. It is separated into 3 charts for clarity.

the SMV model checker from Carnegie Mellon University. The results show that we have achieved over an order of magnitude performance improvement for most of cases. In this dissertation, I will describe the main methodology and optimizations that are responsible for such significant improvements in both stability and running time.

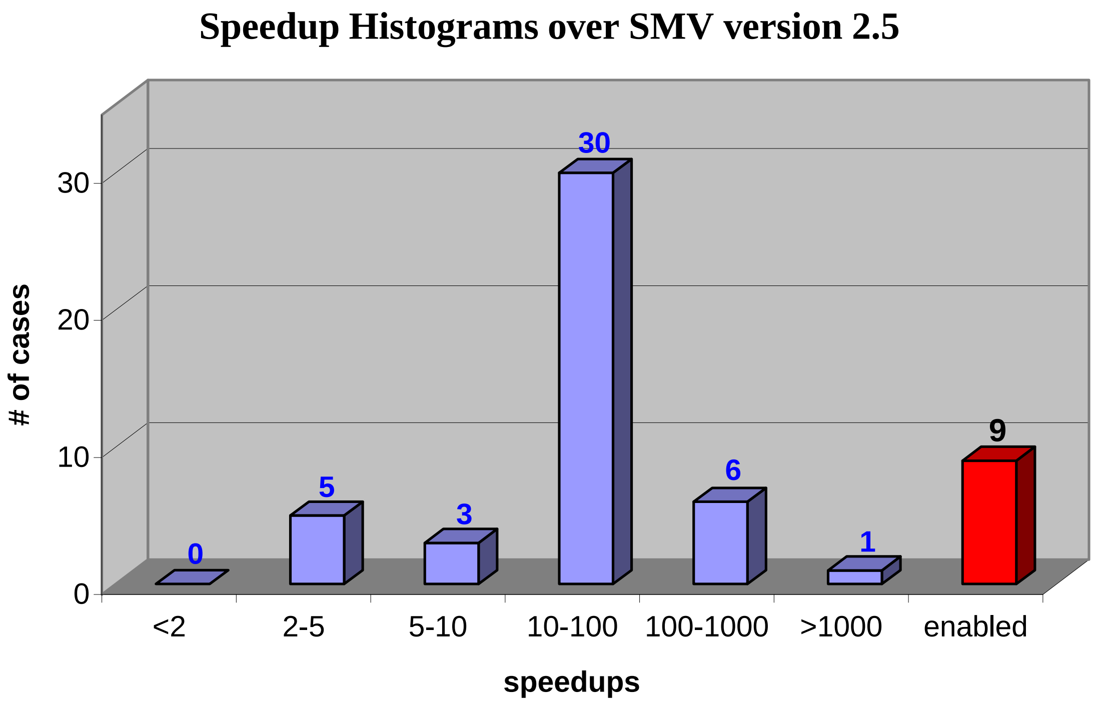


Figure 1.4: Overall performance improvement of our optimizations over SMV 2.5.

Before presenting the contributions of this thesis, Chapter 2 first provides an overview of BDDs and symbolic model checking.

Chapter 3 describes a general evaluation methodology to study BDD computations. This methodology has two parts: the evaluation metrics and an evaluation platform. In this chapter, the metrics are presented as a dependence graph to indicate how one metric influences another. Using this dependence graph, we can systematically diagnose the impact of a new BDD algorithm and identify performance bottlenecks. The evaluation platform consists of a trace recorder for recording computation traces and a trace player for replaying these traces. The purpose of the trace recorder is to generate benchmarks based on real BDD computations. The purpose of the trace player is to provide a simple interface and an uniform platform for studying various BDD packages. Note that in other fields such as dynamic storage allocation and network protocols, the trace-based evaluation methodology has been demonstrated as a powerful technique to study, characterize, and optimize real computations. The BDD trace-based evaluation platform and the set of BDD evaluation metrics together form the *first* evaluation methodology to systematically study BDD computations.

Chapter 4 describes a BDD performance study where the evaluation methodology is used to study BDD computations in the context of model checking. This work is a collaboration of six BDD-package designers from both the industry and the academia, each using their own BDD packages. Each BDD package has its own distinguishing features. By systematically studying these packages, we are able to determine what features are important for model-checking computations. Using this systematic approach, our collaborative efforts have resulted in several characterizations of modeling checking computations and significant performance improvement (two orders of magnitude speedups in some cases) across all the BDD packages involved. This not only is the *first* time that BDD computations are studied in the context of actual model-checking computations, but it is also the *first* time that BDD-package designers collaborated to share ideas and to help each other with performance debugging.

Chapter 5 describes improvements to address the issue of performance instability in the conjunctive-partitioning algorithm described in Figure 1.2. In any conjunctive-partitioning algorithm, there are two main phases: ordering and merging. The ordering phase computes the order among the conjuncts (i.e., compute the associativity). After the conjuncts are ordered, the merging phase combines neighboring conjuncts to reduce the number of partitions. The main idea behind my improvements is to pre-merge *strongly interacting* conjuncts together before the ordering phase. The concept of *strongly interacting* is defined based on BDD characteristics such as the number of common variables between two conjuncts and the merged result's graph size. An extensive experimental evaluation is used to demonstrate that our improvements has made the conjunctive-partitioning algorithm much more stable.

Chapter 6 presents a case study where by studying BDD computations, I derived new BDD-based optimizations and together, with other improvements mentioned earlier, this work has enabled the verification of a class of constraint-rich applications. This research is motivated by the symbolic models developed by NASA for on-line fault diagnosis [81] of autonomous systems such as spacecraft and robot explorers. The models for these systems are very large with the number of state bits ranging from 600 to 1200, whereas the capacity of today's model checker is around 100 state bits. One common characteristic of these models is that they are dominated by time-invariant constraints that are used to specify the relationship between expected and the actual behavior of the system and to encode interconnections between system components. To verify such constraint-rich systems, we introduce two new optimizations. The first optimization is a simple extension of our improved conjunctive-partitioning algorithm. The second is a collection of BDD-based macro-extraction and macro-expansion algorithms to remove state variables. We show that these two optimizations are essential in verifying constraint-rich problems; in particular, these optimizations have enabled the verification of fault diagnosis models for the Nomad robot and the NASA Deep Space One spacecraft.

Chapter 7 finishes with concluding remarks on the overall contributions of this thesis and future directions for research. In particular, even though our approach has resulted in a much improved model checker, the performance instability issue is by no mean solved. For example, the results in Figure 1.3 show that improper choice of parameters can still cost a factor of 15 in performance. This chapter describes another fundamental cause for performance instability: the lack of accurate objective functions. For example, for conjunctive-partitioning algorithms, the ordering and the merging heuristics use objective functions that are based on BDD characteristics such as the number of variables and the graph sizes. However, we do not know exactly how these objective functions or even the resulting conjunctive-partitioning schedules affect the overall performance. Thus, even if we find the optimal solution for these objective functions, it may still perform poorly in comparison to other not-so-optimal solutions. In this chapter, I discuss ongoing research that tries to address this issue along with other interesting directions for future research.

Chapter 2

BDDs and Symbolic Model Checking

In 1978, Akers introduced *Binary Decision Diagrams* (BDDs) as compact representations for Boolean functions [2]. In the mid-1980s, Bryant proposed *Ordered* Binary Decision Diagrams (OBDDs) as canonical representations for Boolean functions and OBDD algorithms for computing Boolean operations efficiently [17]. Since Bryant's work, there has been an explosion in OBDD related research, particularly in the field of formal verification. In this process, a large number of BDD-variants have been proposed ranging from ADD to ZDD [18]. In general, we will refer to all these BDD variants as BDDs unless otherwise specified.

Parallel to the BDD development, Clarke and Emerson invented model checking in early 1980s [22, 36]. They introduced algorithms to automatically reason about temporal properties of finite state systems by exploring the state space. The properties are specified in a temporal logic called CTL—Computation Tree Logic. Even though the early model checkers were fairly limited in the size of the problems they could solve (about $100,000$ states), they were able to verify a number sequential circuits and network protocols [15, 16, 23, 35, 61].

In late 1980s and early 1990s, several researchers independently realized that BDDs can be used to extend the scope of model checking [10, 30, 58, 66]. The use of BDDs to represent finite state systems and to perform symbolic state traversal is called *symbolic model checking*. The use of BDDs has greatly extended the model checker's capacity, where models with up to 2^{100} states are routinely being verified.

This chapter gives an overview of BDDs and symbolic model checking in a bottom-up fashion. We first define what BDDs are and describe the BDD construction process. We then describe the main components and common implementation features in a BDD package. We then describe how to map finite-state systems into Boolean domain so that BDDs can be used to perform model checking. Finally, we describe a CTL model-checking algorithm to tie all these concepts together.

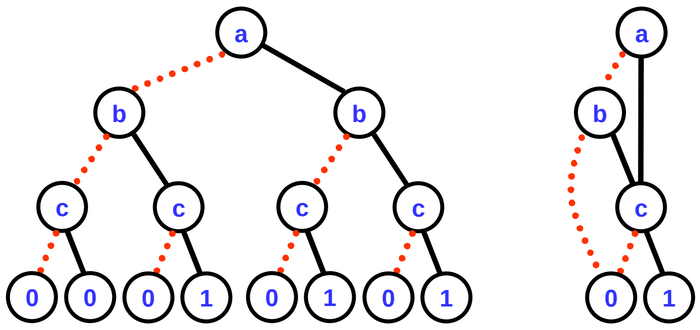
2.1 BDD

A Binary Decision Diagram (BDD) is a directed acyclic graph (DAG) representation of a Boolean function where equivalent Boolean subexpressions are uniquely represented. Due to this uniqueness property, a BDD can be exponentially more compact than its corresponding truth table representation. Figure 2.1 illustrates this concept with three representations of the Boolean function $f = c \wedge (a \vee b)$. Note that we are using 0 to represent *false* and 1 to represent *true*. Figure 2.1(a) is the truth table representation for this function and Figure 2.1(b) is the corresponding *binary decision tree* representation. In a binary decision tree, each internal vertex is labeled with a variable and has edges directed toward two children: the 0-branch (shown as a dashed line) corresponds to the case where the variable is assigned 0, and the 1-branch (shown as a solid line) corresponds to the case where the variable is assigned 1. Each leaf node is labeled 0 or 1 to correspond to the value of the function. Each path from the root to a leaf node corresponds to a truth table entry where the edges in the path corresponds to the assignment of the Boolean variables, and the value of the leaf node is the value of the function under that assignment. Notice that binary decision tree representations can have a lot of unnecessary nodes. In this example, there are multiple copies of 0's and 1's, and three of the c subgraphs are isomorphic to each other. One of the main ideas behind the BDD representation is to remove all the redundancies and represent the graph as a DAG. Figure 2.1(c) shows the BDD representation for the same function. Clearly, the BDD representation is more compact than both the truth table and the binary decision tree representation.

$$f = c \wedge (a \vee b)$$

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(a)



(b) (c)

Figure 2.1: A Boolean function represented with (a) truth table, (b) binary decision tree, (c) binary decision diagram. The dashed-edges are 0-branches and the solid-edges are the 1-branches.

One criterion for guaranteeing the uniqueness of the BDD representation is that all the BDDs constructed must follow the same variable order; i.e., all variables must appear in the same order for any path from the root to a leaf. More formally, for any two variables x and y that are on a path from the root to a leaf, if x precedes y ($x \prec y$) based on the chosen variable order, then x must appear before y on this path. For example, the variable order used in Figure 2.1(c) is $a \prec b \prec c$.

The variable order used can have a significant impact on the size of the BDD graph, from linear to exponential in the number of variables. Let us illustrate the impact of variable order with a comparator-circuit example. In n-bit comparator circuit, there are two sets of n-bit inputs $a = (a_0, ..., a_{n-1})$ and $b = (b_0, ..., b_{n-1})$ and the circuit returns 1 if and only if $(a == b)$, i.e., $\bigwedge_{i=0}^{n-1}(a_i \leftrightarrow b_i)$, where the "==" represents the equality operator in a predicate (similar to the C programming language). Figure 2.2(a) shows a 2-bit comparator circuit. By interleaving the variable orders between a_i's and b_i's (Figure 2.2(b)), essentially, the BDD is computing the equality comparisons (the exclusive-nor gates in the circuit) one bit at a time and only moves on to compare the next bit if all previous comparisons returns true. As Figure 2.2(b) shows, with the interleaved variable order, the BDD structure for each comparison requires only 3 internal nodes. For n-bit comparator, this comparison structure is duplicated n times and thus the BDD size is linear in the number of input variables. On the other hand, if the variable order used is all the a_i's before all the b_i's (Figure 2.2(c)), then we need to first encode all possible assignments of a_i's as a complete binary tree. We then perform the comparisons as b_i's are introduced. Because we need to construct a complete binary tree for a_i's, the BDD size for this variable order is exponential in the number of input variables.

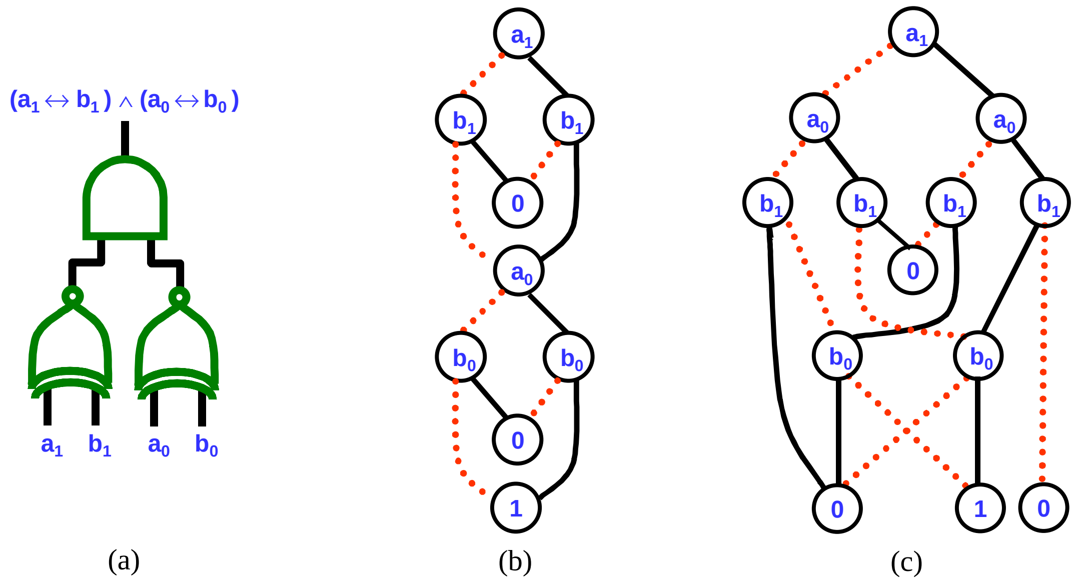


Figure 2.2: Effects of variable order on the BDD for the 2-bit comparator circuit: (a) the circuit, (b) BDD for the interleaved variable order, and (c) BDD for the variable order with all the a_i's before all the b_i's. Note that the BDD node 0 is duplicated to avoid cross edges. In practice, there is only one node for the constant 0.

2.2 Terminology and Notation

Before describing how BDDs are built, we first introduce some terminology and notation. We use 0 to represent *false* and 1 to represent *true*. We use addition to represent Boolean OR and multiplication to represent Boolean *AND*. We use $==$ to represent the the equality operator in a predicate to distinguish from the equality operator $=$ in deriving equations.

Let V be the set of Boolean variables used in the system and let $n = |V|$. We define $f|_{v\leftarrow 0}$ and $f|_{v\leftarrow 1}$ to be *cofactors* of function f with respect to Boolean variable $v \in V$, where $f|_{v\leftarrow 0}$, the *0-cofactor*, is equal to f with the variable v assigned the value of 0, and $f|_{v\leftarrow 1}$, the *1-cofactor*, is equal to f with v assigned the value of 1. We define logical operations on Boolean functions as follows. Given $v \in V$, $f : \{0,1\}^n \rightarrow \{0,1\}$, and $g : \{0,1\}^n \rightarrow \{0,1\}$, then $\forall x \in \{0,1\}^n$,

$$(f \textit{ op } g)(x) := f(x) \textit{ op } g(x),$$

$$(\exists v.f)(x) := (f|_{v\leftarrow 0})(x) + (f|_{v\leftarrow 1})(x),$$

where *op* is any binary logical operator.

A *reachable subgraph* of a node w is defined to be all the nodes that can be reached from w by traversing 0 or more (directed) edges. *BDD nodes* are defined to be internal vertices of BDDs. Given a BDD b, the function f represented by b is recursively defined by Shannon decomposition:

$$f = \overline{v} \cdot f|_{v\leftarrow 0} + v \cdot f|_{v\leftarrow 1} \tag{2.1}$$

(illustrated in Figure 2.3) where v is the variable in b's root node and the 0-cofactor $f|_{v\leftarrow 0}$ is recursively defined by the reachable subgraph of b's 0-branch child. Similarly, the 1-cofactor $f|_{v\leftarrow 1}$ is recursively defined by the reachable subgraph of b's 1-branch child.

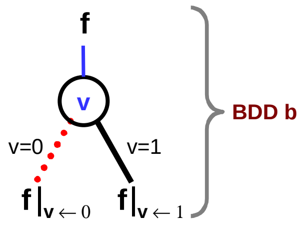


Figure 2.3: The Boolean function f represented by the BDD b.

We say that a function f *depends* on a Boolean variable v if $f|_{v\leftarrow 0} \neq f|_{v\leftarrow 1}$. We define the *support variables* of a function to be the variables that the function depends on, and the *support* of a function to be the set of support variables.

We define the *ITE* operator (if-then-else) as follows: given any two sets D and R, and three functions $f : D \to R$, $g : D \to R$, and $p : D \to \{0, 1\}$, then for all $x \in D$,

$$\mathit{ITE}(p, f, g)(x) := \begin{cases} f(x) & \text{if } p(x) \text{ is true;} \\ g(x) & \text{otherwise.} \end{cases}$$

We define a *care-space optimization* as any algorithm *care-opt* that has following properties: given an arbitrary function f where f may be a non-Boolean function, and a Boolean function c, then

$$\mathit{care\text{-}opt}(f, c) := \mathit{ITE}(c, f, d),$$

where d is defined by the particular algorithm used and may depend on the function f and c. The usual interpretation of this is that we only *care* about the values of f when c is true. We will refer to c as the *care space* and $\neg c$ as the *don't-care space*. The goal of care-space optimizations is to heuristically minimize the representation for f by choosing a suitable d in the don't-care space. Descriptions and a study of some care-space optimizations, including the commonly used *restrict* algorithm [31], can be found in [76].

2.3 BDD Construction

BDD construction is a memoization-based dynamic programming algorithm. A cache known as the *computed cache* is used to record previously computed results. We use a cache instead of a traditional memoization table because the number of distinct subproblems is so large (some are over a billion) that using a complete table is infeasible. Given a Boolean operation, the construction of its BDD representation consists of two main phases. In the top-down *expansion phase*, the Boolean operation is recursively decomposed into subproblems. In the bottom-up *reduction phase*, the result of each subproblem is put into the canonical form. The uniqueness of the result's representation is generally enforced by hash tables known as *unique tables*.

This section describes each of these concepts in detail.

2.3.1 Basis of BDD Construction

Let us first look the theory behind BDD construction. Given a variable order, two BDDs f and g, and any logical operator *op*. The resulting BDD r of the operation f *op* g is constructed based on the Shannon expansion

$$r = f \; op \; g = \overline{\tau} \cdot (f|_{\tau \leftarrow 0} \; op \; g|_{\tau \leftarrow 0}) + \tau \cdot (f|_{\tau \leftarrow 1} \; op \; g|_{\tau \leftarrow 1}) \tag{2.2}$$

where τ, the *top variable*, is the variable with the highest precedence among all the variables in f and g. This equation splits the operation $f \; op \; g$ into two subproblems based the value of the variable τ. If τ is false ($\overline{\tau}$ is true), then the value of this operation can be computed from the 0-cofactors, i.e., $(f|_{\tau \leftarrow 0} \; op \; g|_{\tau \leftarrow 0})$. Otherwise, if τ is true, then the value of this operation is $(f|_{\tau \leftarrow 1} \; op \; g|_{\tau \leftarrow 1})$. The BDDs for these cofactors can be easily obtained. If the top variable is the root of a graph, then the BDD representations for its cofactors are simply the children of that root node. Otherwise, by definition, the top variable does not appear in the graph; thus the BDD representation for both cofactors is just the graph itself.

In the top-down *expansion phase*, this Shannon-expansion process repeats recursively following the given variable order for all the Boolean variables in f and g. The base case (also called the *terminal case*) of this recursive process is when the operation can be trivially evaluated. For example, the Boolean operation $f \wedge f$ is a terminal case because it can be trivially evaluated to f. Similarly, $f \wedge 0$ is also a terminal case. The recursive process will terminate because restricting all the variables of a function produces a constant function, and all Boolean operations involving only constant operand(s) can be trivially evaluated. At the end of the expansion phase, there may be reducible subexpressions such as $(\overline{x} \cdot h + x \cdot h)$. Thus, to ensure uniqueness, a *reduction phase* is necessary to reduce expressions like $(\overline{x} \cdot h + x \cdot h)$ to h. This bottom-up reduction phase is performed in the reverse order of the expansion phase.

Shannon expansions of subproblems can be performed in any order. In particular, the depth-first construction always expands the subproblems with the greatest depth (lowest variable-order precedence); similarly, the breadth-first construction expands the subproblems with the smallest depth (highest variable-order precedence).

For the rest of this document, we will refer to the Boolean operations issued by a user of a BDD package as the *top-level operations* to distinguish them from *sub-operations* (subproblems) generated internally by the Shannon-expansion process.

2.3.2 BDD Algorithms: Depth-First Approach

Typical BDD algorithms are based on the depth-first traversal introduced in Bryant's original BDD publication. Here, I will describe three important BDD algorithms used in symbolic model checking.

The first algorithm computes any basic logical operations such as *AND*, *OR*, and *XOR*. Bryant has shown that the complexity of these operations is quadratic in the graph sizes of the input arguments. Figure 2.4 shows a typical depth-first algorithm used in modern BDD packages for this class of operations [11, 55]. This algorithm takes a logical operator (*op*) and its two BDD operands (f and g) as inputs and returns a BDD for the result of the operation $f \, op \, g$. The result BDD is constructed by recursively performing Shannon expansion in the depth-first manner. This recursive expansion ends when the new operation created is a terminal case (line 1) or when it is found in the computed cache (line 2). A computed cache stores previously computed results to avoid repeating work that was done before. Line 3 determines the top variable of f and g. Line 4 and 5 recursively perform Shannon expansion on the cofactors. At the end of this recursive expansion, the reduction step (line 6) ensures that the BDD result is a reduced BDD node. Then the uniqueness of the resulting BDD node is checked against a *unique table* which records all existing BDD nodes (lines 7–11). Finally, the operation with its result is inserted into the computed cache (line 12) and the BDD result is returned (line 13). Typically, both the computed cache and the unique table are implemented with hash tables. The computed cache replacement policy varies from direct map, n-way associative, to LRU.

The second algorithm is called *relational product* (also known as *AndExists* or *and-smooth*) [19] that computes $\exists \vec{v}.f \wedge g$. The relational product is the core operation in symbolic model checking as it is used to perform both forward and backward state traversal. This process is described in Section 2.5.1. The

```
df_op($op$, $f$,$g$)
    /* compute $f$ $op$ $g$, where $f$ and $g$ are two BDDs and $op$ is a logical operator. */
1       if (terminal case) return simplified result
2       if the operation ($op$,$f$,$g$) is in computed cache, return result found in cache
3       let $\tau$ be the top variable of $f$ and $g$
4       $r_0 \leftarrow$ df_op($op$, $f|_{\tau\leftarrow 0}$, $g|_{\tau\leftarrow 0}$)
5       $r_1 \leftarrow$ df_op($op$, $f|_{\tau\leftarrow 1}$, $g|_{\tau\leftarrow 1}$)
6       if ($r_0 == r_1$) return $r_0$
7       $b \leftarrow$ BDD node ($\tau$, $r_0$, $r_1$)
8       $result \leftarrow$ lookup(unique table, $b$)
9       if (BDD node $b$ does not exist in the unique table)
10          insert $b$ into the unique table
11          $r \leftarrow b$
12      insert this operation and its result into the computed cache
13      return $r$
```

Figure 2.4: Depth-first BDD algorithm.

basic idea is that the product $f \wedge g$ selects the set of valid transitions and the existential quantification $\exists\vec{v}$ extracts and unions destination states from the selected transitions. This operation has been proven to be NP-hard [58].

Figure 2.5 shows a typical BDD algorithm for computing the relational-product operation. This algorithm is structurally very similar to the above algorithm for basic Boolean operations. The main difference is that when the top variable (τ) needs to be quantified, a new BDD operation (*OR*(r_0, r_1)) is generated (lines 5–11). Due to this additional recursion, the worst case complexity of this algorithm is exponential in the graph size of the input arguments.

The third algorithm is a care-space optimization algorithm (as defined in Section 2.2) called *restrict* [31]. It is a commonly used algorithm to minimize the BDD representation of a Boolean function by choosing appropriate values for the don't-care space. Figure 2.6 shows a typical BDD algorithm for computing the *restrict* operation. This algorithm is also structurally very similar to the *df_op* algorithm. There are two main differences. First, when the top variable (τ) only appears in the care space c (line 4), this algorithm avoids introducing the new variable into the result by quantifying out τ (*OR*($c|_{\tau\leftarrow 0}$, $c|_{\tau\leftarrow 1}$)) and then recurses on the result (line 5). The other difference is that when one of f's cofactor is in the don't-care space, then the *restrict* algorithm tries to reduce the graph size by replacing both f's root node and its *don't-care* cofactor with the minimized result of f's other cofactor (lines 6–9). For example, line 6 and line 7 show that if the 0-cofactor $f|_{\tau\leftarrow 0}$ is in the don't-care space, i.e., $c|_{\tau\leftarrow 0} == 0$, then the result graph is the *restrict* of the 1-cofactor $f|_{\tau\leftarrow 1}$. The worst case complexity of this algorithm is exponential in the graph size of the input arguments because the additional recursion introduced by the quantification in line 5.

2.3.3 Breadth-First Approach

The other popular approach of performing BDD construction is the breadth-first approach. In this approach, the Shannon expansion for a top-level operation is performed top-down from the highest to the lowest variable order. Thus, all operations with the same variable order will be expanded at the same time. Similarly, the reduction phase is performed bottom-up from the lowest to the highest variable order where all operations of the same variable order will be reduced at the same time. Note that this *levelized* access pattern is a little different from the traditional notion of breadth-first traversal. We will continue to refer to this access

```
RP($\vec{v}$, $f$, $g$)
    /* compute relational product: $\exists\vec{v}.f \wedge g$, where $\vec{v}$ is a set of variables to be quantified and
     * $f$ and $g$ are two BDDs.
     */
1   if (terminal case) return simplified result
2   if the result of (RP, $\vec{v}$, $f$, $g$) is cached, return the result
3   let $\tau$ be the top variable of $f$ and $g$
4   $r_0 \leftarrow$ RP($\vec{v}$, $f|_{\tau\leftarrow 0}$, $g|_{\tau\leftarrow 0}$)  /* Shannon expansion on 0-cofactors */
5   if ($\tau \in \vec{v}$)  /* existential quantification on $\tau \equiv$ OR($r_0$, RP($\vec{v}$, $f|_{\tau\leftarrow 1}$, $g|_{\tau\leftarrow 1}$)) */
6       if ($r_0 == 1$)  /* OR(1, RP($\vec{v}$, $f|_{\tau\leftarrow 1}$, $g|_{\tau\leftarrow 1}$)) $\equiv 1$ */
7           $r \leftarrow 1$.
8       else
9           $r_1 \leftarrow$ RP($\vec{v}$, $f|_{\tau\leftarrow 1}$, $g|_{\tau\leftarrow 1}$)  /* Shannon expansion on 1-cofactors */
10          $r \leftarrow$ df_op(OR, $r_0$, $r_1$)
11  else
12      $r_1 \leftarrow$ RP($\vec{v}$, $f|_{\tau\leftarrow 1}$, $g|_{\tau\leftarrow 1}$)  /* Shannon expansion on 1-cofactors */
13      $r \leftarrow$ reduced, unique BDD node for ($\tau$, $r_0$, $r_1$)
14  cache the result of this operation
15  return $r$
```

Figure 2.5: A typical relational product algorithm.

```
restrict($f$, $c$)
    /* compute care-space optimization:
     *       if $c$ then $f$ else choose some value to minimize the result
     * Precondition: $c \neq 0$.
     */
1   if ($f$ is constant) or ($c == 1$) return $f$
2   if the result of (restrict, $f$, $c$) is cached, return the result
3   let $\tau$ be the top variable of $f$ and $c$
4   if ($\tau$ is not the top variable of $f$)  /* $c$ has variables not in $f$. quantify, then recurse */
5       $r \leftarrow$ restrict($f$, df_op(OR, $c|_{\tau\leftarrow 0}$, $c|_{\tau\leftarrow 1}$))
6   else if ($c|_{\tau\leftarrow 0} == 0$)  /* don't care about 0-branch */
7       $r \leftarrow$ restrict($f|_{\tau\leftarrow 1}$, $c|_{\tau\leftarrow 1}$)
8   else if ($c|_{\tau\leftarrow 1} == 0$)  /* don't care about 1-branch */
9       $r \leftarrow$ restrict($f|_{\tau\leftarrow 0}$, $c|_{\tau\leftarrow 0}$)
10  else  /* normal Shannon decomposition */
11      $r_0 \leftarrow$ restrict($f|_{\tau\leftarrow 0}$, $c|_{\tau\leftarrow 0}$)  /* 0-cofactors */
12      $r_1 \leftarrow$ restrict($f|_{\tau\leftarrow 1}$, $c|_{\tau\leftarrow 1}$)  /* 1-cofactors */
13      $r \leftarrow$ reduced, unique BDD node for ($\tau$, $r_0$, $r_1$)
14  cache the result of this operation
15  return $r$
```

Figure 2.6: *restrict* care-space optimization algorithm.

pattern as breadth-first to be consistent with previous work. Based on this structured access pattern, we can exploit memory locality by using per-variable memory managers and per-variable breadth-first queues to cluster the nodes of the same variable together. This clustering is beneficial only if many nodes are processed for each breadth-first queue during each expansion and reduction phase.

In this subsection, we will illustrate the breadth-first BDD construction process by introducing a graphical approach to visualize the BDD construction process. Figure 2.7 illustrates the Shannon expansion (Equation 2.2) for the operation f *op* g. On the left side of this figure, the operation is represented with an *operator node* (the rectangular node) which refers to BDD representations of f and g as this operation's left and right operands, respectively[1]. The right side of this figure shows the Shannon expansion of this operation with respect to the variable τ. The original operator node now stores the top variable τ. The two newly created children operator nodes represent the operation on the 0-cofactors (dash edge) and the operation on the 1-cofactors (solid edge). A small circle is placed on the edges to these newly generated operations to distinguish these edges from internal edges of actual BDD nodes.

Notice that this expansion is very much like *sifting down* the operator along both the left and right branches. This view of BDD construction is a generalization of the coding variable approach in [46] which essentially is an *OR* operation. This generalization allows representation of different types of operations, including bit-level and word-level operations. Furthermore, this sifting view integrates BDD construction process with the *level exchange* method [39] in dynamic variable reordering. The only difference here is that the level exchange method sifts down BDD variables, while BDD construction sifts down operators.

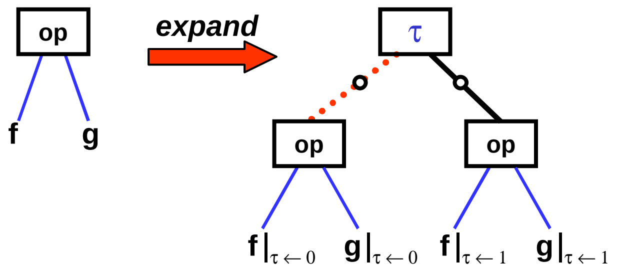


Figure 2.7: Graphical view of Shannon expansion. The dashed edge with a circle represents 0-cofactors sub-operation and the solid edge with a circle represents 1-cofactors sub-operation.

Figure 2.8 illustrates the two rules used in the reduction phase. Figure 2.8(a) illustrates the case when the results of both branches (bdd_0 and bdd_1) are distinct. Figure 2.8(b) illustrates the case when both branches are the same and thus the resulting BDD does not depend on the variable τ. In both cases, the reduction step changes the operator node to forward to the unique BDD result for this operation.

The rest of this section illustrates the breadth-first BDD construction with an example. Figure 2.9(a) and (b) show the BDDs for the operands f and g. Figure 2.9(c) shows the initial graph representation for $f + g$. Note that in the following figures, the constant node 0 is duplicated multiple times to avoid excessive cross edges. In practice, there is only one node for the constant 0.

Figure 2.10 illustrates the top-down expansion phase. Figure 2.10(a) shows the result after Shannon expanding the variable a. Figure 2.10(b.1) illustrates Shannon expansion on the variable b. In this figure, note that the second and the fourth *OR* operator nodes can be trivially evaluated. Furthermore, the first and the third *OR* operator node are identical. Figure 2.10(b.2) shows the result after these simplifications. Figure 2.10(c.1) shows the expansion on the variable c and Figure 2.10(c.2) shows the result of simplifying (c.1). At this point, the expansion phase ends since there are no more unexpanded operations.

[1]The depth-first algorithms does not explicitly store the operations as operator nodes. Instead, the operation is implicitly stored in the stack as arguments to the recursive calls.

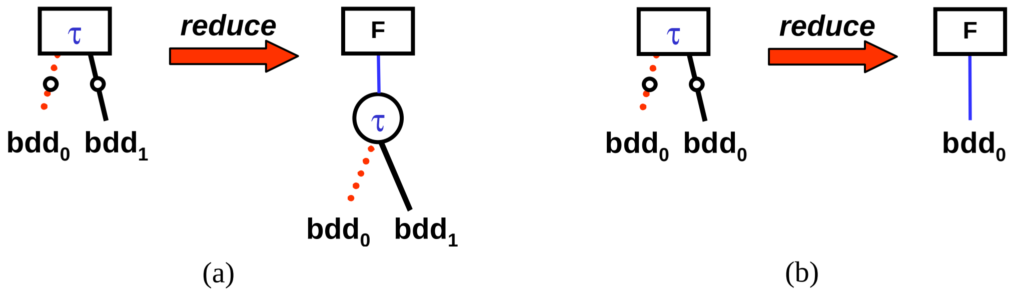
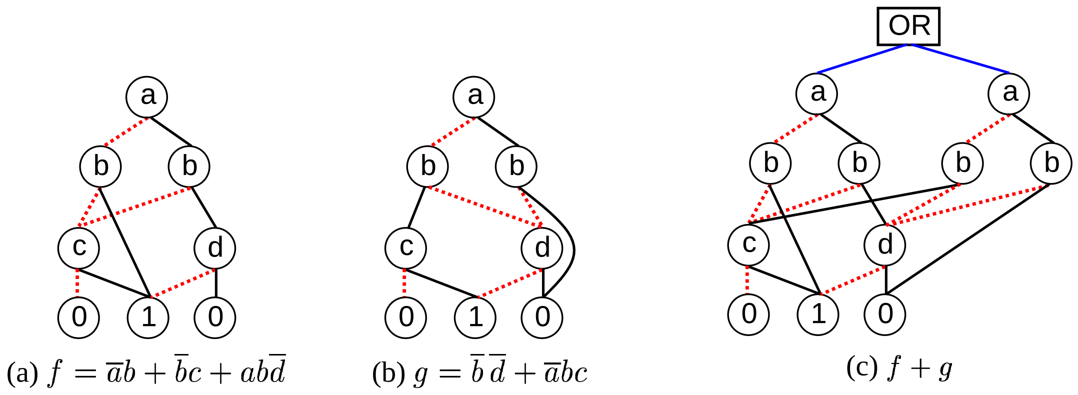


Figure 2.8: Reduction. A reduction changes an operator node to a *forwarding node* and forwards to the resulting BDD: (a) reduction when the children's result BDDs are distinct, and (b) reduction when the children's result BDDs are the same.



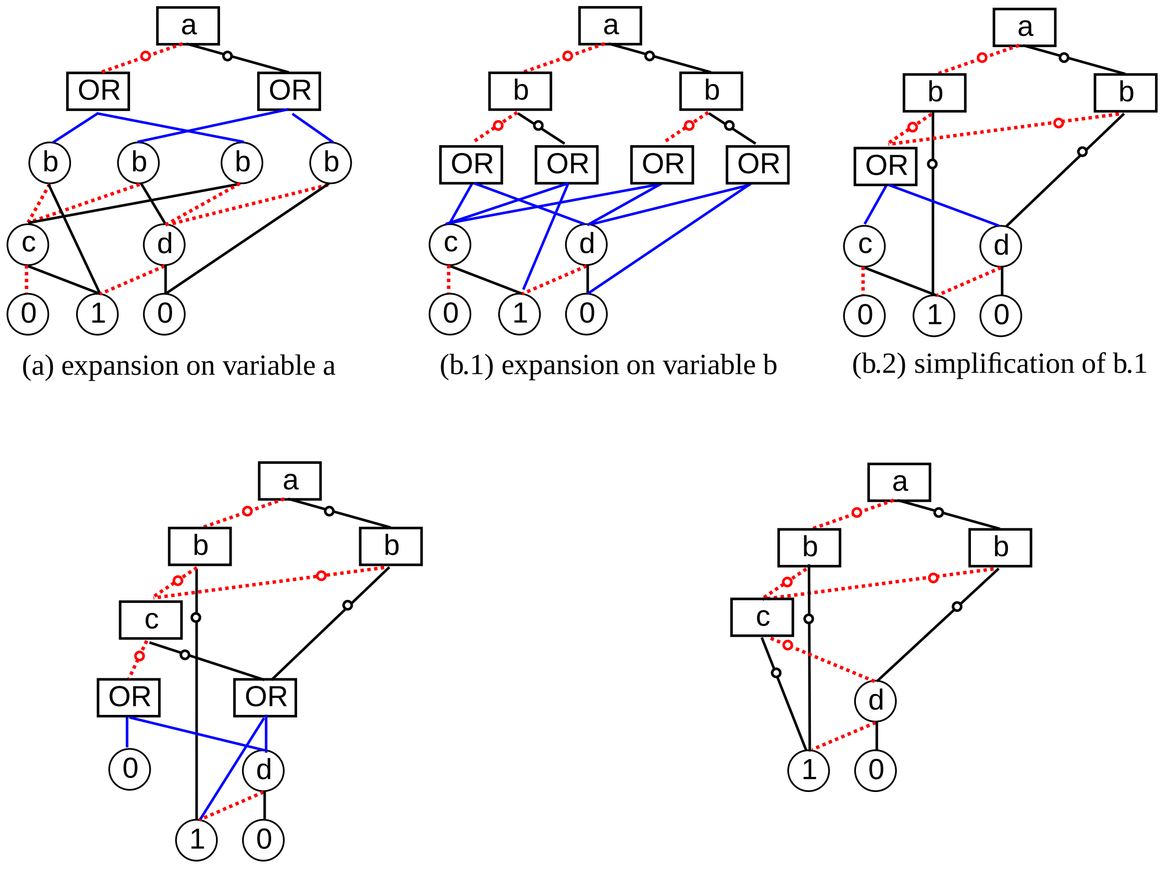
(a) $f = \overline{a}b + \overline{b}c + ab\overline{d}$
(b) $g = \overline{b}\,\overline{d} + \overline{a}bc$
OR
(c) $f + g$

Figure 2.9: BDD construction example: (a) BDD for f, (b) BDD for g, and (c) graphical representation for the operation $f + g$.

Figure 2.11 illustrates the bottom-up reduction phase. Figure 2.11(a.1) shows the result after reduction on variable c. In this figure, a new BDD node (labeled c) is created and the corresponding operator node is then changed to a forwarding node so that the parents can access the newly created result. Figure 2.11(a.2) shows the parents updating the forwarding pointer. Figure 2.11(b) shows the result after reduction on variable b. Finally, Figure 2.11(c) shows the result after updating the forwarding pointers and performing reduction on variable a. In this final figure, the BDD graph referred to by the forwarding pointer is the result of the operation $f + g$ in Figure 2.9(c).

2.3.4 Depth-First or Breadth-First?

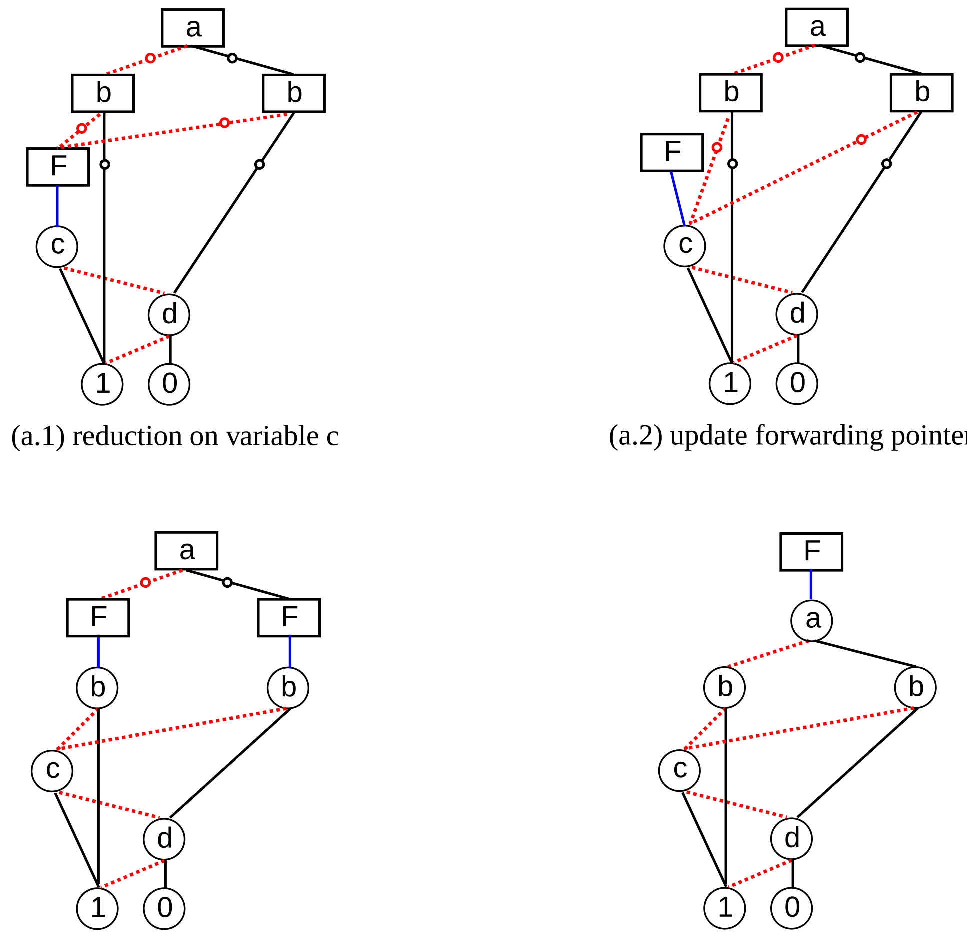
BDD construction can be very memory intensive, especially when large graphs are involved. It not only requires a lot of memory, it also requires frequent accesses to many small data structures (the node size is typically 16 bytes on 32-bit machines). Conventional BDD construction algorithms are based on depth-first traversal of the BDDs [11, 55]. This approach may have poor memory behavior as there is not a good way to ensure that the BDD nodes accessed are close in memory. The performance impact is especially severe for BDDs larger than the physical memory. Recently, there has been much interest in BDD construction based on breadth-first traversal [4, 46, 63, 64, 71, 83]. In a breadth-first traversal, both the expansion phase and the reduction phase process all the nodes of the same variable at the same time. The breadth-first construction exploits this structured access by clustering nodes (for both BDD and operator nodes) of the same variable



(a) expansion on variable a (b.1) expansion on variable b (b.2) simplification of b.1

(c.1) expansion on variable c (c.2) simplification of c.1

Figure 2.10: Breadth-first BDD construction: expansion phase.



(a.1) reduction on variable c

(a.2) update forwarding pointers

(b) reduction on variable b

(c) reduction on variable a

Figure 2.11: Breadth-first BDD construction: reduction phase.

together in memory with specialized node managers.

Despite its poorer memory locality, the depth-first construction has very low memory overhead. The number of unsolved subproblems that it keeps tracks of at any given time is the depth of the recursion, which is at most the number of variables. Since the number of variables is typically a very small constant, the depth-first construction does not require much memory to store these subproblems. In contrast, for each top-level operation, the breadth-first construction keeps all the subproblems (operator nodes) generated by Shannon expansion of this top-level operation until the result for this top-level operation is constructed. Because the number of subproblems can be quadratic in the size of the BDD operands, the breadth-first approach can incur a large memory overhead. In particular, for borderline cases where the depth-first construction fits in the physical memory while the breadth-first construction does not, the performance of the breadth-first construction can degrade significantly due to page faults.

To limit memory overhead, we have introduced a partial breadth-first construction [83] based on *context switch*. Within each evaluation context, the breadth-first expansion is used until the number of operator nodes (unsolved subproblems) reaches a fixed evaluation threshold. Upon reaching this threshold, the current context is pushed onto a *context stack* and a new child context is started. Then the yet-to-be-expanded operator nodes of the parent context are partitioned into small groups and the child context evaluates these operations one group at a time. If the child context reaches its evaluation threshold, it will also context switch and this process repeats. When all yet-to-be-expanded operations from the parent context are evaluated, the child context terminates. This technique bounds the memory overhead to *O(number of variables* $\times$ *threshold)*. By keeping the evaluation threshold to be a small fraction of available physical memory, we can limit the memory overhead and control the working set size to gain better memory locality. Finally, this approach can be naturally parallelized by distributing the contexts across the processors [84].

Other than the memory overhead, the breadth-first based approaches have other performance drawbacks. In terms of running time, one drawback is in the implementation of the cache. In the breadth-first approach, the sub-problems are explicitly represented as operator nodes and the uniqueness of these nodes is ensured by using a hash table with chaining for collision resolution. Accesses to this hash table are inherently slower than accesses to a typical direct mapped (1-way associative) computed cache used in the depth-first approaches. Furthermore, handling of both the computed and yet-to-be-computed operator nodes adds even more overhead. Depending on the implementation strategy, this overhead could be in the form of an explicit cache garbage collection phase as a cache replacement strategy or as a transfer of computed results from operator nodes' hash table to a computed cache. Maintenance of the breadth-first queues is another source of overhead. This overhead can be higher for operations such as relational products because of the possible additional recursion (line 7 in Figure 2.5). Given that each sub-problem requires only a couple hundred cycles on modern machines, these overheads can have a non-negligible impact on the overall performance.

2.4 BDD Package Components and Their Common Features

Modern BDD packages typically share common implementation features based on [11, 72]. There are three main components in a BDD package: the BDD algorithm component, the dynamic variable reordering component, and the garbage collection component. In this section, we describe the common features in each of these components.

BDD Algorithm

This component computes the result BDDs for various Boolean operations (as described in previous sections). The implementation of these algorithms is typically based on depth-first traversal. The unique tables are hash tables with the hash collisions resolved by chaining. A separate unique table is associated with each variable to facilitate the dynamic-variable-reordering process. The computed cache is a hash-based direct mapped (1-way associative) cache. BDD nodes support *complement edges* [3] where for each edge, an extra bit is used to indicate whether or not the target function should be complemented (Boolean negation). The advantage of this encoding is that a function and its complement can be represented by the same BDD and use this extra bit in the reference edge to interpret the BDD either in the positive or the negated form. Implementation-wise, this extra bit is typically encoded in the least significant bit of the address pointer (the reference edge) to avoid incurring extra memory cost. This encoding exploits the property that address pointers in modern machines are always at least 4-byte aligned, which means the least significant bit is always 0. Thus it can be used to encode the complement information.

Dynamic Variable Reordering

As the variable order can have significant impact on the size of a BDD graph, dynamic variable reordering is an essential part of all modern BDD packages. The goal for this component is to dynamically establish a good variable order as the computation progresses. Typically, when a variable reordering algorithm is invoked, all top-level operations that are currently being processed are aborted. When the variable reordering algorithm terminates, these aborted operations are restarted from the beginning. The dynamic variable reordering algorithms are generally based on the *sifting* algorithm [72]; i.e., the variable orders are changed by exchanging nodes in one level with nodes in the adjacent level. Figure 2.12 illustrates this process. This figure shows the sifting process for exchanging the orders of the variable a and the variable b. We tag each node with a number so that we can refer to them easily. To understand this operation, let us first recall Equation 2.1 that defines the function represented by a BDD in terms of its cofactors. Using this definition, the BDD in Figure 2.12(a) can be represented as

$$\overline{a} \cdot (\overline{b} \cdot f_0 + b \cdot f_1) + a \cdot (\overline{b} \cdot g_0 + b \cdot g_1).$$

By rearranging this formula, we get

$$\overline{b} \cdot (\overline{a} \cdot f_0 + a \cdot g_0) + b \cdot (\overline{a} \cdot f_1 + a \cdot g_1),$$

which is the BDD after the sifting process (Figure 2.12(b)). Implementation-wise, node 4 and node 5 are created to represent the new children of node 1. Node 1 is updated to reference these new children and is relabeled with variable b. As for node 2 and node 3, they remain unchanged and if there are no references to them, they will be garbage collected later. Note that because node 1 might be referenced by others, it is important that node 1 is reused with its reachable graph representing the same function. Without this reuse, we will need to create a new node in place of node 1 and will have to locate all the references to node 1 and update them to reference this new node, which is very inefficient. The node-reuse technique allows the sifting algorithm to be a local operation involving only the node and its children.

Figure 2.13 illustrates the sifting process for the 2-bit comparator example from Section 2.1. Figure 2.13(a) is the BDD representation for the variable order with all the a_i's before all the b_i's. By exchanging the orders of variable a_0 and b_1, we obtain the BDD for the interleaved variable order (Figure 2.13(b)). In this example, node 1 and node 2 are reused so that we do not need to update references from their parent (node 0). As for nodes 3, 4, 5, and 6, they are no longer part of the comparator function and if there are no other references to them, they can be garbage collected. Note that the reachable subgraph of node 0 in Figure 2.13(b) has an interleaved variable order and is identical to the BDD in Figure 2.2(c).

Garbage Collection

BDD computations are inherently memory intensive because after all, it is all about traversing and constructing graphs. Furthermore, in verification, many intermediate BDD results are created to arrive at a simple

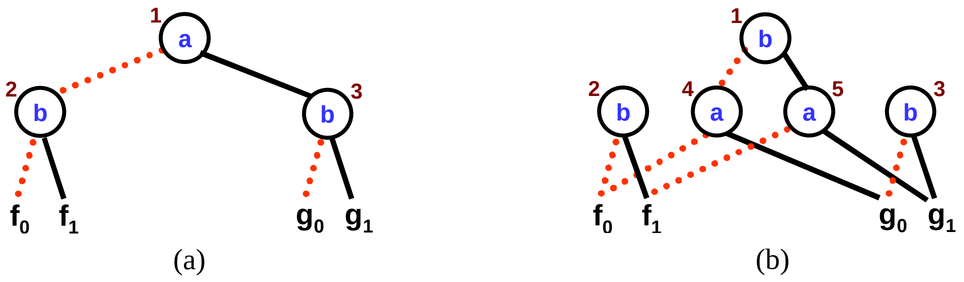


Figure 2.12: Sifting operation for dynamic variable reordering: (a) before sifting and (b) after sifting down variable a. The nodes are tagged with numbers to show which nodes are reused (node 1) and which are newly created (node 4 and 5).

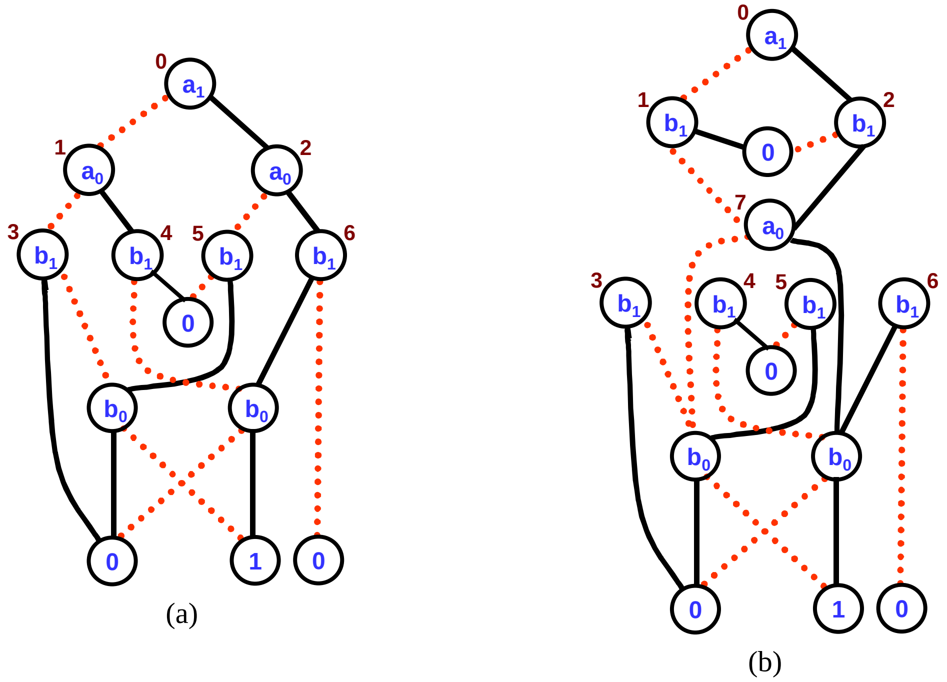


Figure 2.13: Variable reordering example for 2-bit comparator. (a) BDD for a_i's before b_i's variable order. (b) Exchange the variable orders of variable a_0 and b_1 to obtain the interleaved variable order.

final answer—true or false. Thus, it is important to have a good garbage collector to automatically remove BDD nodes that are no longer useful. We will refer to a BDD node as *reachable* if it is in some BDD that external user has a reference to. As external users free references to BDDs, some BDD nodes may no longer be reachable (*deaths*). We will refer to these nodes as *unreachable* BDD nodes.

Typical garbage collectors used in modern BDD packages are based on reference counting and the reclaimed nodes are maintained in a *free-list* for later reuse. Garbage collection is invoked when the percentage of the unreachable BDD nodes exceeds a preset threshold. Because the unreachable BDD nodes are not garbage collected as soon as their reference counts reach zero, garbage collection in the BDD world has an unusual concept called *rebirth*, where unreachable nodes become reachable again. This rebirth further introduces an unexpected property that the garbage collection algorithm not only has impact on the memory usage, but can also have significant impact on the overall running time. In the following, we will describe how rebirth occurs and how it can influence running time.

The rebirth of a BDD node can occur as follows. BDD nodes are referenced both externally by users of a BDD package and internally by the unique tables and also possibly by the computed cache. When BDD nodes become unreachable from external references, they are not garbage collected immediately. Thus, some of these unreachable BDD nodes may become reachable again (rebirths) if they end up being the results for new subproblems.

So, in presence of rebirths, how does garbage collection affect the running time? Let us use an example to demonstrate this. Figure 2.14 shows a snap shot of a BDD package. In this example, nodes in the BDD f are not reachable from any external references and f has an internal reference from the i^{th} entry of the computed cache. Since f has no external references, the entire BDD can be garbage collected. During garbage collection, we must also remove all dangling references. In particular, we must clear the cache entry i. Let us assume that later on, f is reborn as a result of a new subproblem, and we need to compute the same operation that was previous recorded in cache entry i. In this case, because the garbage collection has already cleared that cache entry, we will need to recompute the entire operation. Thus, in the case that BDD nodes toggle between reachable and unreachable frequently, garbage collection can reduce the effectiveness of the computed cache and significantly degrade the overall performance. In fact, for model checking computations, we found that this is one main source of the performance bottlenecks in modern BDD packages (Chapter 4).

2.5 Boolean Representations of Finite State Systems

To use BDDs in model checking, we will need to map the model checking problem to the Boolean domain. This means mapping the set of states, the transition relation, and the state traversal to Boolean functions.

In general, we may also need to model non-Boolean functions or non-Boolean variables to represent a finite state system. These non-Boolean functions can be represented using variants of BDDs (e.g., MTBDD [28]) that extend the BDD concept to include non-Boolean values. These extensions also include algorithms for Boolean operators with non-Boolean arguments such as the "$<$", "$>$", "$==$" and "$\in$". Unless clarification is necessary, we will continue to refer to all these BDD-variants simply as BDDs.

2.5.1 State Sets and State Transitions

For finite state systems, a state typically describes the values of many components (e.g., latches in digital circuits) and each component is represented by a *state variable.* Let $V = \{v_0, ..., v_{n-1}\}$ be the universe of state variables in a system and K_{v_i} be the set of possible values for variable v_i, then a state can be described by assigning values to all the variables in V and the set of possible states S_A is

$$S_A := K_{v_0} \times K_{v_1} \times ... \times K_{v_{n-1}}.$$

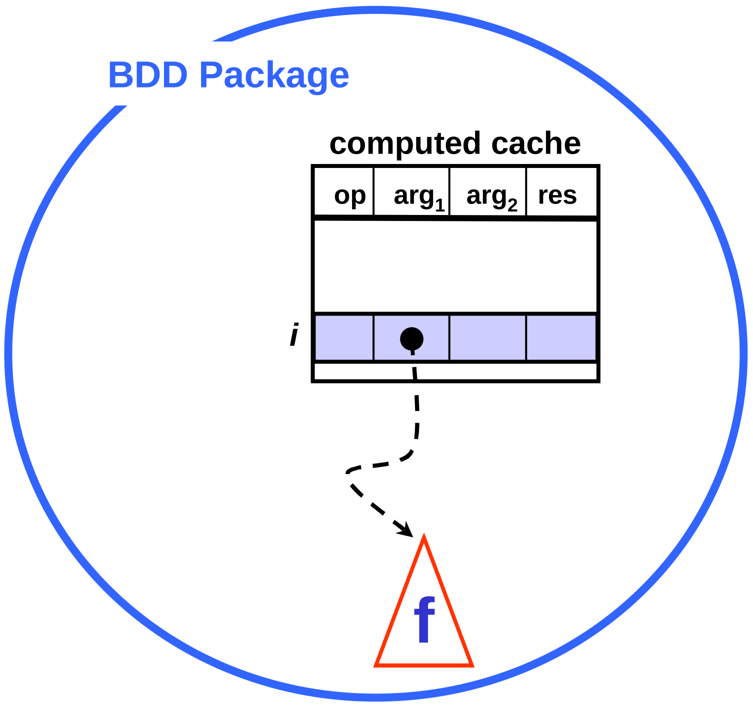


Figure 2.14: A snapshot of a BDD package: f is a BDD whose nodes are not reachable from external references.

This valuation can in term be written as a Boolean function that is true exactly for the valuation as

$$\bigwedge_{i=0}^{n-1} (v_i == c_i),$$

where $c_i \in K_{v_i}$ is the value assigned to the variable v_i. A set of states can be represented as a disjunction of the Boolean functions that represent the states. We denote the BDD representation for a set of states $S \subset S_A$ by $S(V)$ to indicate that the set of variables V is used in the BDD representation.

Let us illustrate this using a 3-bit counter (Figure 2.15). A state in a 3-bit counter is a 3-tuple representing the values of three state variables v_2, v_1, and v_0. For example, a state $(0, 1, 1)$ represents the state where the value of v_2 is 0, v_1 is 1, and v_0 is 1. Its corresponding Boolean representation is $(v_2 == 0) \wedge (v_1 == 1) \wedge (v_0 == 1)$ or more concisely, $\overline{v}_2 v_1 v_0$. Representing a set of states as a Boolean function is just as straightforward. For example, the set $\{(0, 0, 1), (0, 1, 1)\}$ can be represented as $\overline{v}_2\overline{v}_1 v_0 + \overline{v}_2 v_1 v_0$. Essentially, for Boolean variables, a state can be represented by a minterm and a set of states can be represented by a sum of minterms.

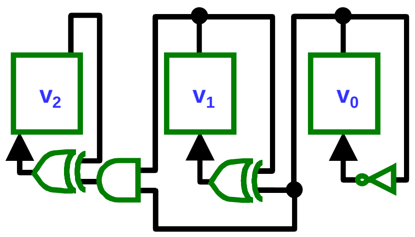


Figure 2.15: 3-bit counter.

In addition to the set of states, we also need to map the system's state transitions to the Boolean domain. We extend the above concept of representing a set of states to representing a set of ordered-pairs of states. To represent a pair of states, we need two sets of state variables: V the set of *present-state variables* for the first tuple and V' the set of *next-state variables* for the second tuple. Each variable v in V has a corresponding next-state variable v' in V'. A valuation of variables in V and V' can be viewed as a state transition from a present state (valuation of variables in V) to its next state (valuation of variables in V'). A transition relation can then be represented as a set of these valuations, or a disjunction of their corresponding Boolean representations. We denote the BDD representation of a transition relation T as $T(V, V')$.

For the 3-bit counter, one valid transition is from $(0, 0, 1)$ to $(0, 1, 0)$. This transition can then be represented as $\overline{v}_2\overline{v}_1v_0\overline{v}'_2v'_1\overline{v}'_0$, where $\overline{v}_2\overline{v}_1v_0$ encodes the present state $(0, 0, 1)$ and $\overline{v}'_2v'_1\overline{v}'_0$ encodes the next state $(0, 1, 0)$. This process can be applied to obtain the minterms representing all the valid transitions. The transition relation can then be computed as the sum of these minterms.

In modeling finite state systems, the overall state transitions are generally specified by defining the valid transitions for each state variable separately. To support non-deterministic transitions, a state variable v_i's transition relation is defined by a function $\mathbf{f}_i$ that maps each state of the system to a set of possible next-state values for variable v_i, i.e., $\mathbf{f}_i : K_{v_0} \times K_{v_1} \times ... \times K_{v_{n-1}} \rightarrow 2^{K_{v_i}}$, where K_{v_i} is the set of all possible values for variable v_i. We use the bold face font to emphasize that the function $\mathbf{f}_i$ returns a *set* possible next-state values (instead of a single next-state value for the deterministic case). Let $\mathbf{f}_i$ be the function that defines the state transitions of the state variable v_i. Then the BDD representation for v_i's transition relation T_i can be defined as

$$T_i(V, V') := (v'_i \in \mathbf{f}_i(V)).$$

For deterministic systems with f_i's as the transition functions, we can replace the set operator $\in$ with $==$ as

$$T_i(V, V') := (v'_i == f_i(V)).$$

For synchronous systems, all the state variables transition at the same time. Thus, the BDD for the overall state transition relation T is the conjunction of all the state variables' transition relations; i.e.,

$$T(V, V') := \bigwedge_{i=0}^{n} T_i(V, V').$$

Detailed descriptions on this formulation, including mapping of asynchronous systems, can be found in [19, 58].

Let us revisit the 3-bit counter example to illustrate this mapping. The Boolean functions for the latches' transition relations are

$$\begin{array}{rcl} T_0(V, V') & := & (v'_0 == \overline{v}_0), \\ T_1(V, V') & := & (v'_1 == (v_1 \oplus v_0)), \\ T_2(V, V') & := & (v'_2 == (v_2 \oplus (v_0 \wedge v_1)). \end{array}$$

T_0 says that the next state value of the least significant bit v_0 is always the complement of its current value; i.e., it toggles at every clock. T_1 says v_1 will toggle in the next state if v_0 is currently 1. T_2 says v_2 will toggle in the next state if all the lower bits are currently 1. Using this formulation, the overall transition relation T is

$$T(V, V') := [(v'_0 == \overline{v}_0) \wedge (v'_1 == (v_1 \oplus v_0)) \wedge (v'_2 == (v_2 \oplus (v_0 \wedge v_1)))].$$

2.5.2 Symbolic State Traversal

To reason about temporal properties, the *image* and the *pre-image* of the transition relation are used for forward and backward state traversal, respectively. Based on the BDD representations of a state set S and the transition relation T, we can compute the *image* and the *pre-image* of S as follows:

$$\begin{aligned} \textit{image}_{V'}(S) &:= \exists V.[T(V,V') \wedge S(V)], \\ \textit{pre-image}_{V}(S) &:= \exists V'.[T(V,V') \wedge S(V')]. \end{aligned}$$

where $\textit{image}_{V'}(S)$ denotes the BDD result for $\textit{image}(S)$ represented with variables in V', and $\textit{pre-image}_{V}(S)$ denotes the BDD result for $\textit{pre-image}(S)$ represented with variables in V. The intuition is that for both cases, the conjunction $T \wedge S$ selects the set of valid transitions and the existential quantification extracts and unions the destination states together. Note that both the image and the pre-image operation can be computed by the relational-product algorithm ($\exists\vec{v}.f \wedge g$) in Figure 2.5.

In Figure 2.16, we illustrate how the image operation can be computed using the relational-product operation. In this figure, the leftmost table represents the set of states $S = \{(0,0,0),(1,1,0)\}$ that we want to compute the image from. The corresponding Boolean function is

$$S(V) = \overline{v}_2\overline{v}_1\overline{v}_0 + v_2v_1\overline{v}_0.$$

The transition-relation table T in the center represents the set of valid transitions for the 3-bit counter. Each row represents a valid transition with the left-hand-side of the table representing the current state and the right-hand-side of the table representing the next state. The corresponding Boolean function is

$$\begin{aligned} T(V,V') \;=\; & \overline{v}_2\overline{v}_1\overline{v}_0\overline{v}'_2\overline{v}'_1v'_0 + \overline{v}_2\overline{v}_1v_0\overline{v}'_2v'_1\overline{v}'_0 + \overline{v}_2v_1\overline{v}_0\overline{v}'_2v'_1v'_0 + \overline{v}_2v_1v_0v'_2\overline{v}'_1\overline{v}'_0 + \\ & v_2\overline{v}_1\overline{v}_0v'_2\overline{v}'_1v'_0 + v_2\overline{v}_1v_0v'_2v'_1\overline{v}'_0 + v_2v_1\overline{v}_0v'_2v'_1v'_0 + v_2v_1v_0\overline{v}'_2\overline{v}'_1\overline{v}'_0. \end{aligned}$$

The rightmost table is the set of states that can be reached from S in one step , i.e., the *image*(S). The conjunction $S(V) \wedge T(V,V')$ uses the states in S to select the valid transition relations by matching S to the present-state part (the left part) of table T. The existential quantification extracts the result by removing the present-state part of table T and by unioning the selected next-state part (the right part) of T. For the pre-image computation, we simply reverse this process and from right to left, perform the selection on the next-state part of T and the extraction from present-state part of T.

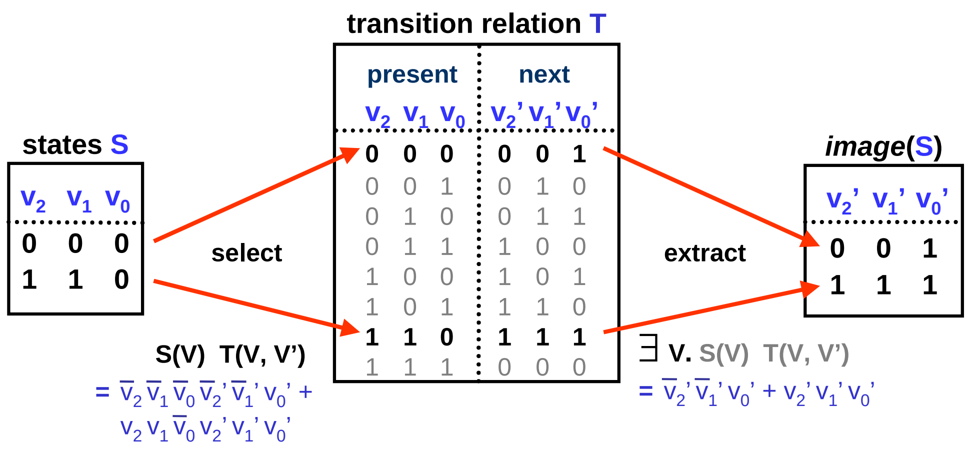


Figure 2.16: Forward state traversal for 3-bit counter.

2.5.3 Conjunctive Partitioning of Transition Relations

One limitation of the BDD representation is that the monolithic BDD for the transition relation T is often too large to build. A solution to this problem is the *conjunctive partitioning* [19] of the transition relation. In conjunctive partitioning, the transition relation is represented as a conjunction $P_1 \wedge P_2 \wedge ... \wedge P_k$ with each conjunct P_i represented by a BDD. Then, the pre-image can be computed by conjuncting with one P_i at a time, and by using *early quantification* to quantify out variables as soon as possible. The early-quantification optimization is based on the property that sub-formulas can be moved out of the scope of an existential quantification if they do not depend on any of the variables being quantified. Formally, let V_i, a subset of V, be the set of variables that do not appear in any of the subsequent P_j's, where $1 \leq i \leq k$ and $i < j \leq k$. Then the image can be computed as

$$\begin{array}{rcl} p_1 & := & \exists V_1.[P_1(V,V') \wedge S(V)] \\ p_2 & := & \exists V_2.[P_2(V,V') \wedge p_1] \\ & \vdots & \\ p_k & := & \exists V_k.[P_k(V,V') \wedge p_{k-1}] \\ \mathit{image}_{V'}(S) & := & p_k \end{array}$$

For example, in the 3-bit counter, if we choose $P_1 = T_2(V,V')$, $P_2 = T_1(V,V')$, and $P_3 = T_0(V,V')$, (where T_i's for the 3-bit counters are defined in Section 2.5.1) then the *image*(S) is equal to

$$\exists v_0.[T_0(V,V') \wedge \exists v_1.[T_1(V,V') \wedge \exists v_2.[T_2(V,V') \wedge S(V')]\ ]\ ].$$

Here, we can quantify the variable v_2 early because for the 3-bit counter, neither $T_0(V,V')$ nor $T_1(V,V')$ depends on v_2. Similarly, v_1 can be early quantified because $T_0(V,V')$ does not depend on it. Now, if we were to reverse the order of conjunctions with $P_1 = T_0(V,V')$, $P_2 = T_1(V,V')$, and $P_3 = T_2(V,V')$ then both V_0 and V_1 are empty, thus early quantification is not possible and the *image*(S) is equal to

$$\exists v_0.\exists v_1.\exists v_2.[T_2(V,V') \wedge (T_1(V,V') \wedge (T_0(V,V') \wedge S(V')))].$$

This shows that the ordering of the partitions can affect the number of variables that we can quantify out early.

In general, the determination and ordering of partitions (the P_i's in above) can have significant performance impact. Commonly used heuristics [40, 69] treat the state variables' transition relations (T_i's) as the partitions. The ordering step then greedily schedules the partitions to quantify out more variables as soon as possible, while introducing fewer new variables. Then, the ordered partitions are tentatively merged with their predecessors to reduce the number of intermediate results. Each merged result is kept only if the resulting graph size is less than a pre-determined limit called *partition-size limit*.

The conjunctive partitioning for the pre-image computation is performed similarly with next-state variables in V' being the variables to be quantified instead of present-state variables in V. Note that because the set of variables to be quantified is different, the resulting conjuncts for the image computation are typically different from those for the pre-image computation.

2.6 CTL Temporal Logic and Model Checking

In model checking, temporal logic is used to specify the behavior of a system as time progresses. The particular temporal logic that we are interested in is a logic called CTL [22, 36]. In this section, we informally describe CTL and model checking based on CTL. Descriptions of other temporal logics such as

CTL* and LTL and more detailed descriptions of CTL model checking, including modifications to restrict the verification only to *fair* computation paths[2], can be found in [19, 21, 58].

2.6.1 CTL

In CTL, each temporal operator has two parts: a path quantifier followed by a tense operator. There are two path quantifiers: **A** representing *for all computation paths*, and **E** representing *there exists a computation path*. There are four main tense operators:

X : the *next time* operator,

F : the *eventually* operator,

G : the *globally* operator, and

U : the *until* operator.

Combining the two path quantifiers with the four tense operators, there is a total of eight CTL operators. In the following, we first define CTL formula using the three basic CTL operators: **EX**, **E [U]**, and **EG**. We then show how the remaining five operators can be expressed using these three basic operators.

Let P be the set of *atomic propositions*, then a CTL formula is defined as follows:

- Every atomic proposition $p \in P$ is a formula in CTL.
- If f and g are CTL formulas, then so are $\neg f$, $f \vee g$, $\mathbf{EX}\, f$, $\mathbf{E}\,[f\ \mathbf{U}\ g]$, and $\mathbf{EG}\, f$.

The propositional operator $\neg$ and $\vee$ are negation and disjunction, respectively. The other propositional operators can be defined in term of these two operators. The definitions of the temporal operators are as follows:

- $\mathbf{EX}\, f$: f holds at some successor state of the current state.
- $\mathbf{E}\,[f\ \mathbf{U}\ g]$: for some computation path starting from current state, there exists a state such that g holds, and f holds for all previous states in this computation path.
- $\mathbf{EG}\, f$: there exists a computation path such that f holds on every state of the path.

The remaining five CTL operators can be expressed as follows:

- $\mathbf{AX}\, f \equiv \neg\mathbf{EX}\, \neg f$: f holds for all successor state of the current state.
- $\mathbf{EF}\, f \equiv \mathbf{E}\,[true\ \mathbf{U}\ f]$: for some computation path starting from current state, there exists a state such that f holds.
- $\mathbf{AF}\, f \equiv \neg\mathbf{EG}\, \neg f$: for every computation path starting from the current state, there exists a state such that f holds.
- $\mathbf{AG}\, f \equiv \neg\mathbf{EF}\, \neg f$: for every path, f holds for every state on the path.
- $\mathbf{A}\,[f\ \mathbf{U}\ g] \equiv (\neg\mathbf{E}\,[\neg g\ \mathbf{U}\ (\neg f \wedge \neg g)]) \wedge (\neg\mathbf{EG}\, \neg g)$: for every computation path starting from the current state, there exists a state such that g holds, and f holds for all previous states in this computation path.

[2]A computation path is *fair* under a constraint c if c is true infinitely often in this path. One usage of the *fairness constraint* c is to restrict the verification to paths where every process executes infinitely often; i.e., no starvation.

The following are some typical CTL formulas:

- **EF** $($*thruster.state* $==$ `nominal`$)$: it is possible to reach the state where the thruster's status is nominal.
- **AG** $($**EF** *restart*$)$: from any state, we can always reach the restart state.
- **AG** $($*req* $\rightarrow$ **AF** *ack*$)$: if a request occurs, it will eventually be acknowledged.
- **AG** $($**AF** *deviceEnabled*$)$: the device is enabled infinitely often along all computational paths.

2.6.2 CTL Model Checking

In model checking, a CTL property is expressed by a notation "$(M, s) \models f$" which means that the CTL formula f is true in state s of the finite state model M. A CTL formula f can be interpreted as a set of states which satisfies f, i.e., $\{s \mid (M, s) \models f\}$. In the following, we will use a formula f to represent both the formula itself and the set of states where the formula is true. Under this interpretation, the CTL operators can be computed as follows [22]:

$$\begin{aligned}
\textbf{EX}\, f &= \textit{pre-image}(f) \\
\textbf{E}\, [f\ \textbf{U}\ g] &= \textbf{lfp}\ Z\ [g \vee (f \wedge \textbf{EX}\ (Z))] \\
\textbf{EG}\, f &= \textbf{gfp}\ Z\ [f \wedge \textbf{EX}\ (Z)]
\end{aligned}$$

where **lfp** $Z\ [h(Z)]$ and **gfp** $Z\ [h(Z)]$ compute the least and the greatest fixed-point, respectively, of the function h. In the following, we describe algorithms for computing each of these three operations and provide intuition on how it works. For detailed proofs, please see Clarke and Emerson's work [22].

EX f computes the set of states such that f holds for some successor state. Thus, if we interpret a formula f as the set of states that f is hold, then **EX** f computes the predecessors of f, i.e., *pre-image*(f). The BDD algorithm for computing the pre-image operation is shown in Section 2.5.

Similarly, **E** $[f$ **U** $g]$ computes the set of states that for any of these states s, there exists a computation path such that along this path starting from s, f must be true until the first state where g is true. Thus, we can compute **E** $[f$ **U** $g]$ by starting from g and include all the states that can reach g while satisfying f. In another words, this is the backward reachability computation starting from g and traverses the state space backward while restricting the valid predecessor states to f.

Figure 2.17 shows the algorithm for computing **E** $[f$ **U** $g]$. Note that all the Boolean operations can be computed using the BDD algorithm *df_op* in Figure 2.4 and the pre-image operation can be computed using the BDD algorithm shown in Section 2.5. In the *eval_eu()* subroutine, the result of the least fixed-point is stored in variable Y and it is initially set to g (line 1), the starting states for the reverse reachability computation. The variable ΔY stores the set of newly reached states and it is also initially set to g (line 2). Then this algorithm computes the set of states that satisfies f and can be reached from ΔY in one backward step (line 4). Then it updates the set of newly reached state (line 5) and the set of reached states (line 6). This process repeats until no new states are reached (line 7). At this point, we have reached a fixed point and the process terminates. Note that because we are traversing a finite state space, sooner or later, we will have explored all the reachable states. Thus this algorithm always terminates.

EG f computes the set of states that for any of these states s, there exists a computation path p such that along this path starting from s, f is always true. Based on this definition, we can define **EG** f recursively as the following:

EG f is true in state s if and only if f is true in s and **EG** f is true in some successor state of s.

```
eval_eu($f$, $g$)
    /* compute $\mathbf{E}\,[f\ \mathbf{U}\ g]$ */
    $Y \leftarrow g$   /* initial results for the least fixed-point */
    $\Delta Y \leftarrow g$   /* changes in Y from one iteration to the next */
    do
        $pY \leftarrow (f \wedge \textit{pre-image}(\Delta Y))$
        $\Delta Y \leftarrow pY \wedge (\neg Y)$
        $Y \leftarrow Y \vee \Delta Y$
    while ($\Delta Y \neq \texttt{false}$)
    return $Y$
```

Figure 2.17: Algorithm for evaluating the **E** $[f\ \mathbf{U}\ g]$.

Or more concisely,

$$\mathbf{EG}\, f = f \wedge \mathbf{EX}\,(\mathbf{EG}\, f).$$

This equation says that **EG** f is a fixed point for the function $h(Z) = f \wedge \mathbf{EX}\ Z$. In Figure 2.17, we show the algorithm for computing this fixed point.

```
eval_eg($f$)
    /* compute $\mathbf{EG}\ f$ */
    $Y \leftarrow f$   /* initial results for the greatest fixed-point */
    do
        $Y' \leftarrow Y$   /* remember previous value of Y */
        $Y \leftarrow (Y \wedge \textit{pre-image}(Y))$
    while ($Y \neq Y'$)
    return $Y$
```

Figure 2.18: Algorithm for evaluating the **EG** f.

2.7 Summary

Symbolic model checking verifies temporal properties of finite state systems by traversing the state space to ensure the properties hold for all valid states. The BDDs have been proven to be an efficient representation for model-checking computations.

Because this thesis focuses on the *computational aspects* of BDDs in the context model checking, we have devoted this chapter to describe BDD characteristics, BDD construction, BDD packages, and symbolic state traversal with a fair amount of detail. However, on the subject of the CTL temporal logic and CTL model checking, we have provided only enough detail to demonstrate how basic Boolean operations and the state traversal can be used to perform model checking. For more detailed descriptions of the CTL model checking process, we refer interested readers to Burch et al.'s paper on symbolic model checking for sequential circuit verification [19]. Also, McMillan's book on symbolic model checking [58] provides a more complete treatment of symbolic model checking, including a description of the SMV model checker.

Finally, Clarke et al.’s upcoming book [21] is perhaps the most complete guide to date on the subject of model checking.

Chapter 3

BDD Performance Evaluation Methodology

Given the importance of the BDD representation, there have been many research efforts focused on developing new BDD construction algorithms. Traditionally, the validations are often based on a set of benchmark circuits (e.g., building BDDs for the combinational parts of ISCAS'85 [13] or ISCAS'89 [12]) with the running time as the evaluation metric for comparing different algorithms. This approach has two major drawbacks: (1) the BDD computations for constructing benchmark circuits can be very different from other applications such as model checking, and (2) validations solely based on running time often fail to account for implementation-dependent effects. These drawbacks can result in misleading conclusions about the impact of new optimizations.

Recently, in studying memory locality issues, hardware specific metrics such as *hardware-cache miss rate* and *translation-lookaside buffer (TLB) miss rate* have been introduced [57]. These metrics are important in studying memory locality, but by themselves, they still do not sufficiently decouple the effects of implementation details. For example, different implementations can have very different strategies for controlling the garbage-collection frequency and the computed-cache size. These strategies can greatly affect the amount of work performed, which, in turn, can have significant impact on hardware metrics such as cache and TLB miss rates. Thus, these hardware metrics alone are still not sufficient for drawing conclusions about memory locality of different algorithms.

This chapter addresses the issues of evaluation metrics and realistic benchmarks by proposing a general evaluation methodology for studying BDD performance. This methodology proposes a set of evaluation metrics for characterizing key components of BDD computations and a trace-based evaluation platform to study BDD computations based on real workloads.

In general, this methodology is designed to put BDD evaluations on a more scientific basis.

3.1 Evaluation Metrics

This section defines a set of metrics for characterizing BDD computations. Our goal is to find metrics that are useful for more accurately measuring the performance impact of different techniques used to perform BDD computations. To achieve this goal, we try to account for other secondary effects (such as implementation details) as much as possible, and use machine- and implementation-independent metrics to characterize the BDD computations whenever possible.

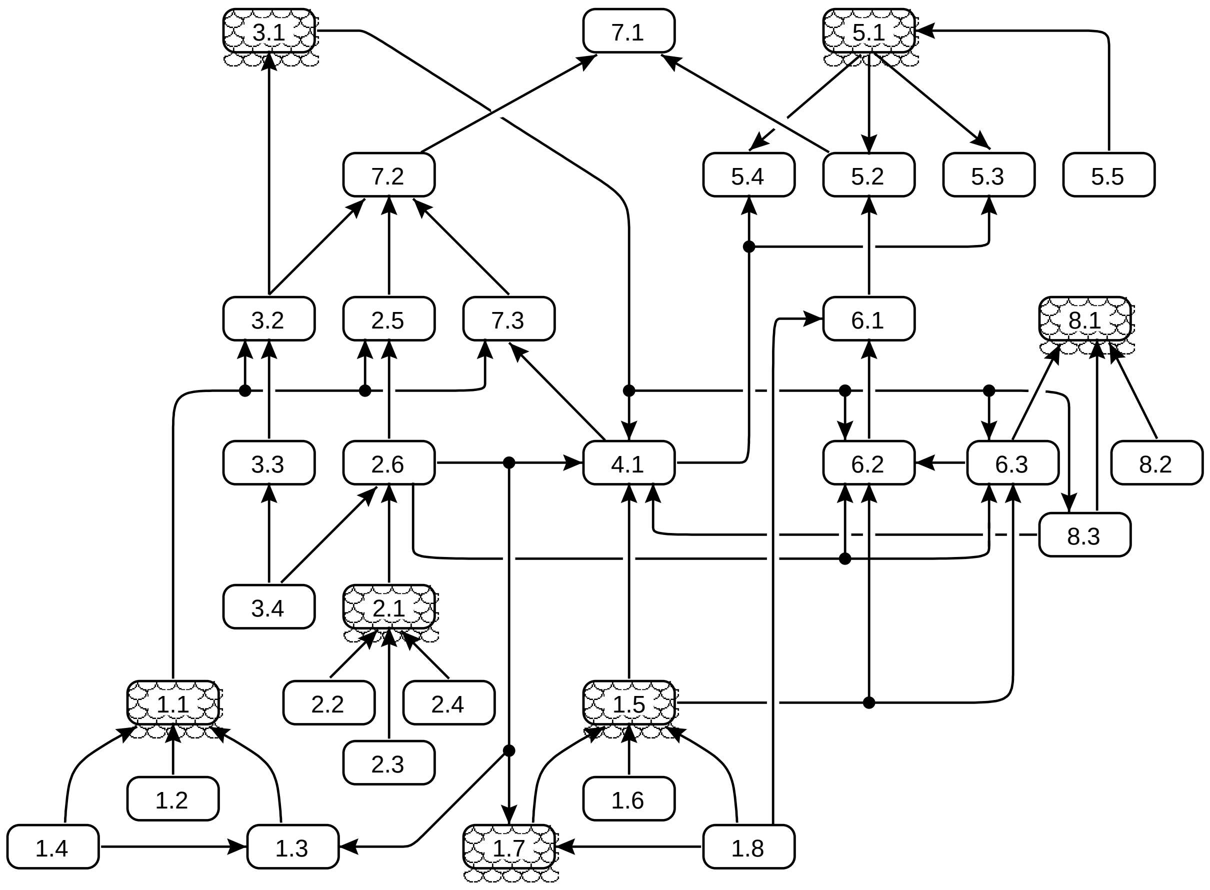
Along with descriptions of metrics, we use dependence graphs to illustrate how the metrics influence each other. In these dependence graphs, each metric is represented by a node and there is a directed edge between two nodes when the source node's metric influences the destination node's metric. For example, the effectiveness of caching strongly influences the number of subproblems generated in BDD construction; thus, in the corresponding dependence graph, we put an edge from the node for the effectiveness of caching to the node for the number of sub-operations. In drawing the dependence graphs, we use a rectangular node ▭ to represent a metric that can be directly measured. A shaded rectangular node ▨ represents a qualitative metric that cannot be directly measured, but can be approximated by studying the metrics in its descendants. A double-rectangular node ▭ or ▨ indicates the descendants of that node are not shown in the figure and can be found in other dependence graphs shown in this section. This simplification makes the dependence graphs easier to read and emphasize the primary metrics for studying each components of BDD computations.

In Figure 3.1, we show the overall dependence graph of all the metrics to be defined in this section. This dependence graph provides an overview of what the metrics are and how they all fit together. Detailed descriptions of these metrics and explanations on their usage will be presented later. In this figure, each node is labeled with a number $x.y$ to indicate that the metric is the y^{th} new metric defined in Section 3.1.x. For consistency, the metrics will be labeled the same way throughout this section.

A dependence graph of metrics provides a simple way to determine what set of metrics is necessary to evaluate new BDD algorithms. For example, in studying the impact of the breadth-first BDD construction, the common practice is to compare a breadth-first BDD package with a state-of-the-art depth-first BDD package and use the running time (node 7.1) to demonstrate the impact of the breadth-first approach. And whenever applicable, the page fault rate (node 5.2), the hardware cache miss rate (node 5.3), and the TLB miss rate (node 5.4) are used to explain that the main cause is due to memory locality (node 5.1). However, what is often ignored is that the number of sub-operations (node 4.1) generated can differ greatly depending on the computed-cache management algorithms (node 1.5) and the garbage-collection frequency (node 2.5). Thus without first ensuring that the results of these metrics are accounted for, we cannot be sure that the performance impact we observed is truly due to memory locality. This information is encoded in the dependence graph as edges from the number of sub-operations (node 4.1) to the hardware cache miss rate (node 5.3) and the TLB miss rate (node 5.4).

Using a similar approach, this dependence graph can be used for systematic performance diagnosis and performance tuning. For example, the graph shows that the overall performance (node 7.1) depends on CPU time (node 7.2) and the page fault rate (node 5.2). Thus, we can measure these two metrics to determine the causes for performance differences. Suppose the discrepancy is in CPU time, then we can follow down the dependence graph and measure the metrics that the CPU time depends on, namely time spent on dynamic variable reordering (node 3.2), garbage collection (node 2.4), and BDD algorithms (node 7.3). We can repeat this process until the main causes for performance differences are identified. Once the main causes are identified, we can make improvements to address performance issues due to implementation details and repeat this process until the performance differences are mainly due to the inherently algorithmic differences. Thus this dependence graph serves as a road map for performing systematic performance tuning and for understanding the true impact of the underlying algorithms. For example, in the BDD performance study to be presented in the next chapter, we used this approach to identify the impact of the breadth-first BDD construction to model-checking computations and found that the breadth-first based approach has no measurable advantage over the depth-first based approach.

Each of the following subsections describes the metrics for evaluating a component of BDD computations. We present the components in a bottom-up fashion, from metrics for node hashing to metrics for the space and the time usage and for characterizing the size (or the difficulty) of the given BDD computations. The metrics for each component are also presented as a dependence graph. The root of each graph represents the component that the corresponding subsection focuses on. Nodes that have some paths to the root represent the metrics that can influence that component. Nodes that are reachable from the root represent the metrics for measuring that component's impact. Note that the definition of each metric is labeled to correspond to the nodes in Figure 3.1. Some metrics are repeated in different subsections as they contribute to several components of BDD computations (sometimes in different ways). The repeated versions of metrics are annotated with square brackets as in $[x.y]$ to reflect that the metric's main use is described in



1.1: BDD-node hashing
1.2: cost of BDD hash function
1.3: # of BDD nodes visited per BDD table access
1.4: max BDD table size
1.5: cache-node hashing
1.6: cost of cache hash function
1.7: quality of cache hash fn
1.8: max computed cache size

2.1: GC effectiveness
2.2: rebirth rate
2.3: # of dead BDD nodes
2.4: death rate
2.5: GC time
2.6: # of GCs

3.1: reordering effectiveness
3.2: reordering time
3.3: # of nodes sifted
3.4: # of reorderings

4.1: # of ops

5.1: memory locality
5.2: page-fault rate
5.3: hardware-cache miss rate
5.4: TLB miss rate
5.5: # of ops processed per BF queue visit

6.1: max memory usage
6.2: max # of BDD nodes
6.3: max # of live BDD nodes

7.1: elapsed time
7.2: CPU time
7.3: BDD alg time

8.1: size of BDD computations
8.2: # of BDD vars
8.3: min # of ops

Figure 3.1: The overall dependence graph of metrics for studying BDD computations. The descriptions of metrics and how they are used are described in the rest of this section one component at a time. The node label $x.y$ indicates that the metric is the y^{th} new metric defined in Section 3.1.x.

Section 3.1.x as the y^{th} new metric in that subsection.

3.1.1 Metrics for Hashing

Hashing is a fundamental part of typical BDD packages. It is used both to ensure the uniqueness of BDD nodes and to cache the results of previously computed subproblems. For each subproblem generated, there can be up to 3 hash-table accesses (a computed-cache lookup and a computed-cache insert plus a combined BDD unique-table lookup-and-insert). Because the time spent processing each subproblem is only a few hundred CPU cycles, hashing can be a significant part of the overall cost in the BDD computation. Figure 3.2 shows the dependence graphs of metrics for monitoring and comparing the impact of the hash functions on both BDD unique tables and computed caches. The nodes in this graph are:

1.1 **BDD-node hashing:** the quality of the hashing algorithm used for the BDD unique tables. This includes both the quality and the cost of the hash function plus the table resizing strategy. This qualitative metric cannot be directly measured.

1.2 **cost of BDD hash function:** the time needed to compute the BDD hash function. The metric measures the cost of arithmetic operations used to compute the hash function excluding the time spent on fetching the operands from memory. It can be used to show the tradeoff between the quality and the cost of hash function. This metric is independent of BDD packages and hash key values. It can be measured by averaging over the time used for computing the hash value of some arbitrary input keys a large number of times.

1.3 **# of BDD nodes visited per BDD table access:** the total number of BDD nodes examined during accesses to the BDD unique tables / the total number of accesses to the BDD unique tables. This metric measures the degree of node clustering and thus indicates the quality of the hash function used.

1.4 **max BDD table size:** the maximum number of total entries in the BDD unique tables. The table size directly affects the clustering of the nodes. Thus, to compare the quality of the hash functions, we must ensure the table sizes are roughly the same.

1.5 **cache-node hashing:** the quality of the hashing algorithm used for the computed cache. This value includes both the quality and the cost of the hash function plus the table resizing strategy. This qualitative metric cannot be directly measured.

1.6 **cost of cache hash function:** the time needed to compute the hash function for the computed cache. The metric is used to measure the cost of arithmetic operations used to compute the hash function excluding the time spent on fetching the operands from memory. It can be used to show the tradeoff between the quality and the cost of hash function. Similar to the "*cost of BDD hash function*", this metric is independent of BDD packages and hash key values. It can be measured by averaging over the time used for computing the hash value of some arbitrary input keys a large number of times.

1.7 **quality of cache hash fn:** the quality of the cache's hash function. For a fully associative cache, the "total number of cache nodes examined during accesses to the computed caches / the total number of accesses to the computed caches" serves as a good metric to measure the degrees of collision. For a direct-map cache (also known as 1-way associative cache), we measure the percentage of empty cache entries and compare it with $n(\frac{n-1}{n})^m$, where n is the size of the computed cache and m is the number of the cache insertions so far. The formula $n(\frac{n-1}{n})^m$ (provided by Fabio Somenzi from the University of Colorado) is the expected number of empty cache entries with a completely random hash function. For other types of cache (e.g., k-way associative where $1 < k < n$), both of the above measures can be used to quantify the quality of the hash function.

The formula for the expected number of empty entries for a direct-map cache can be derived as follows: for each insertion, the probability for a cache entry not to be chosen is $\frac{n-1}{n}$ for a cache of n entries. Thus, for m insertions, the probability that a cache entry is still empty is $(\frac{n-1}{n})^m$. Since there are n entries, the expected number of empty entries is $n(\frac{n-1}{n})^m$.

1.8 **max computed cache size:** the maximum number of total entries of the computed caches. The table size directly affects the clustering of the nodes. Thus, to compare the quality of the hash functions, we must ensure the table sizes are roughly the same.

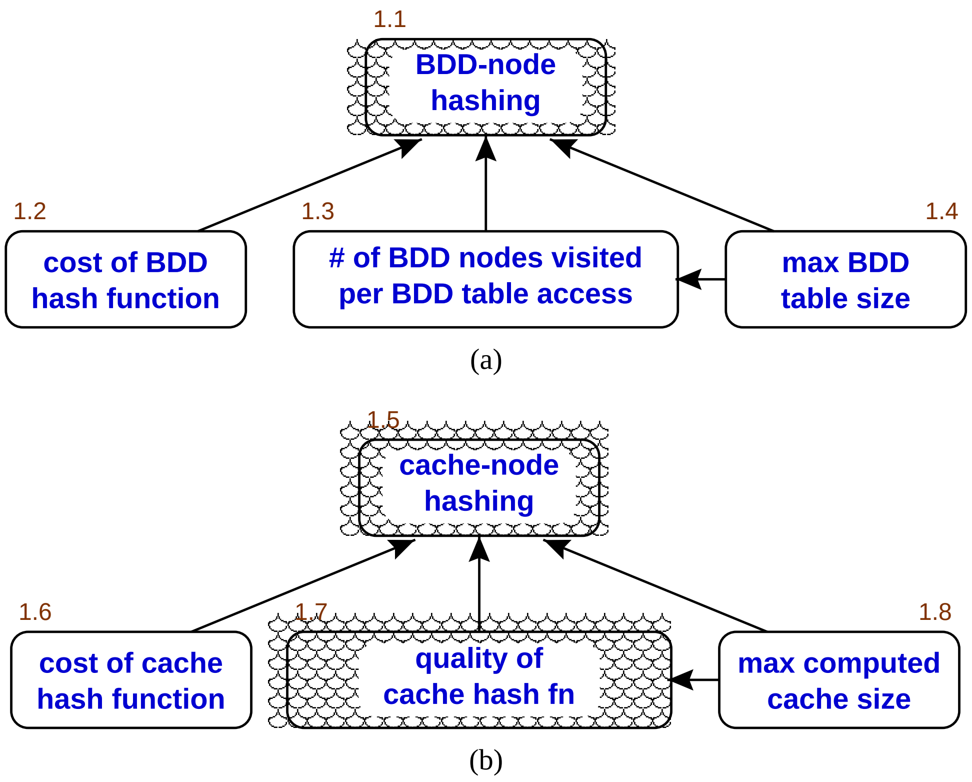


Figure 3.2: Metrics for studying the effectiveness of hashing of (a) BDD nodes, and (b) computed cache nodes.

Note that some researchers use the hit rate of the computed cache as a measure for the effectiveness of the computed cache. This measure has been empirically observed to be unreliable. In the following, we show that when the hit rate is less than 50%, a cache miss can actually increase the overall hit rate.

Let n be the number of cache lookups and h be the number of cache hits at some point of BDD computations. Let o be the next BDD operation to be computed. Consider the following two scenarios. (1) Suppose that the result of operation o was previously computed and is currently stored in the computed cache, then executing operation o will result in the cache hit rate of $\frac{h+1}{n+1}$. (2) On the other hand, if the operation o is not stored in the computed cache, then we have a cache miss and at least two new sub-operations will be generated. In this case, suppose that the result of these two new sub-operations are currently stored in the computed-cache. Then the overall cache hit rate after completing the operation o will be $\frac{h+2}{n+3}$.

Based these two scenarios, a computed-cache miss can increase the computed-cache hit rate when the hit rate for scenario 1 is less than the hit rate for scenario 2; i.e.,

$$\frac{h+1}{n+1} < \frac{h+2}{n+3}$$

$$\begin{aligned}(h+1)(n+3) &< (h+2)(n+1)\\ hn+3h+n+3 &< hn+h+2n+2\\ 2h &< n-1\\ \frac{h}{n} &< 0.5-\frac{1}{2n}\end{aligned}$$

Thus, for sufficiently large number of lookups n, when the cache hit rate ($\frac{h}{n}$) is less than 50%, then a cache miss can result in higher cache hit rate.

3.1.2 Metrics for Garbage Collection

The effectiveness of garbage collection can have significant impact on both space and time usage. The impact on time is a bit unusual in comparison to conventional software systems, such as LISP, that also perform garbage collection automatically. The main difference is that unlike these conventional systems where memory associated with a dead object remains unusable until it is recycled by garbage collection, for BDD packages, a dead object can become live or reachable again and this *rebirth* process can affect the overall running time. This process is described in Section 2.4. Here is a brief summary.

Garbage collection's impact on running time is caused by its effect on the computed cache. When garbage collection reclaims dead BDD nodes, the computed-cache entries that reference these nodes must also be removed to eliminate dangling references. However, if the rebirth rate of BDD nodes is high, then it is very likely that computed-cache entries being removed may actually be useful. Thus, invoking garbage collection too frequently can increase the total number of operations due to cache misses. On the other hand, invoking garbage collection too infrequently can greatly increase the memory usage. Thus, the effectiveness of garbage collection can have significant impact on both the space and the time usages.

Figure 3.3(a) shows the dependence graph of the metrics that can be used to measure the effectiveness of garbage collection. The nodes in the graphs are:

2.1 **GC effectiveness:** the effectiveness of the garbage-collection algorithm in reducing the time and the memory usages. This qualitative metric cannot be directly measured.

2.2 **rebirth rate:** the number of rebirths / the number of deaths. This metric provides a measure on how much memory is truly reclaimed by garbage collection. Note that for BDD packages that use mark-and-sweep garbage collectors, the number of rebirths is not readily available. For reference-count based garbage collectors, the rebirth rate can be measured as the reference counts toggle between 0 and 1. For these BDD packages, rebirth rate is a good metric to use for triggering garbage collection.

2.3 **# of dead BDD nodes:** the number of dead BDD nodes. This metric is the measure on how much memory can be reclaimed by garbage collection at any given point in time. Traditionally, this is the main metric used for triggering garbage collection. However, since rebirths can occur, this metric is not an accurate measure of its effectiveness in reclaiming memory. Thus, this metric should be used with the rebirth rate to determine when to perform garbage collection.

2.4 **death rate:** the number of deaths / the number of operations. This metric measures the frequency of deaths. It is yet another useful metric to determine when to trigger garbage collection.

Figure 3.3(b) shows the dependence graph of metrics used to measure the cost of garbage collection. The nodes in the graph are:

2.5 **GC time:** time spent on performing garbage collection. This is measured in the number of CPU seconds.

2.6 **# of GCs:** the number of times garbage collection is invoked. As mentioned before, the frequency of garbage collection can have significant impact in both space and time.

[3.4] **# of reorderings:** the number of times that the dynamic reordering algorithm is invoked. Since garbage collection is typically invoked before and after the dynamic variable reordering to clear away the dead nodes, the frequency of dynamic variable reordering affects the frequency of garbage collection.

[1.1] **BDD-node hashing:** the quality of the hashing algorithm used for the BDD unique table. This only influences garbage-collection algorithms that need to rehash the BDD unique table (e.g., due to memory compaction).

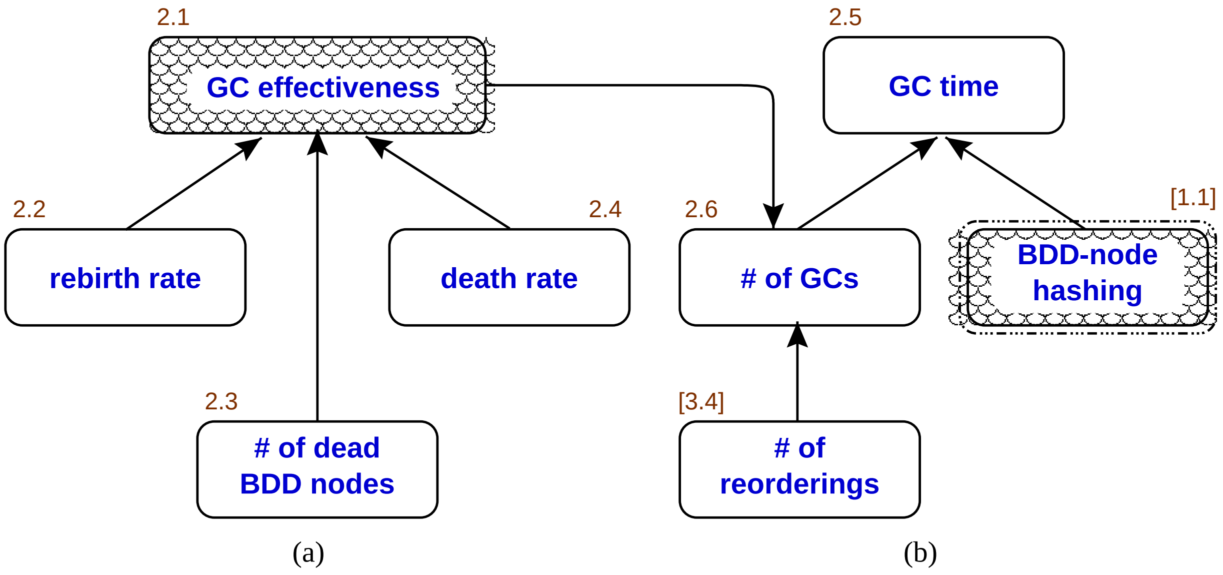


Figure 3.3: Metrics for studying garbage collection: (a) effectiveness of garbage collection, and (b) effort spent on garbage collection.

3.1.3 Metrics for Dynamic Variable Reordering

Dynamic variable reordering is an important part of BDD packages, because the BDD graph size strongly depends on the variable order used. Dynamic-variable-reordering algorithms are generally based on *sifting* and *window permutation* algorithms [72] whose core operation is exchanging nodes in the adjacent level (*level-exchange*). When the initial variable order used is not "good enough" to finish the computation, dynamic variable reordering is invoked to find a better variable order. When this occurs, the time spent in the reordering process typically dominates the overall computation time.

Dynamic variable reordering is inherently a difficult problem for many reasons. First, finding an optimal ordering is NP-hard. Second, small changes in the triggering and termination criteria may have significant impact in both the space and time requirements. Third, some BDD algorithms (e.g., the restrict algorithm in Figure 2.6) produce different results with different variable orders. These different results lead to different subsequent computations. Thus it is difficult to isolate the effects of different reordering algorithms. These difficulties make dynamic variable reordering a very challenging problem and also make this the weakest part of our evaluation methodology. Figure 3.4 shows the dependence graph of the metrics proposed. The nodes in the graph are:

3.1 **reordering effectiveness:** the impact of dynamic variable reordering. This qualitative metric cannot be directly measured.

3.2 **reordering time:** the time spent in dynamic variable reordering. This is measured in the number of CPU seconds. The metric measures the percentage of time spent in the reordering process.

3.3 **# of nodes sifted:** the number of nodes sifted down; i.e., the number of nodes that swapped the variable order with their children during the variable reordering process. The basic operation in variable reordering is sifting down nodes to swap the variable order with their children to reorder the adjacent variables. Thus, this metric provides a good machine-independent measure on the amount of work performed by (or the cost of) the reordering process.

3.4 **# of reorderings:** the number of times that the dynamic-variable-reordering algorithm is invoked.

[1.1] **BDD-node hashing:** the quality of the hashing algorithm used for the BDD unique table. During dynamic variable reordering, the BDD unique table is used to ensure the uniqueness of new nodes created.

[4.1] **# of ops:** the number of operations recursively generated by the BDD computations. This metric can be used to reflect the impact of dynamic variable reordering on the amount of work that BDD algorithms need to performed.

[6.3] **max # of live BDD nodes:** the maximum number of live BDD nodes. This metric can be used to reflect the impact of dynamic variable reordering on the minimum memory requirement.

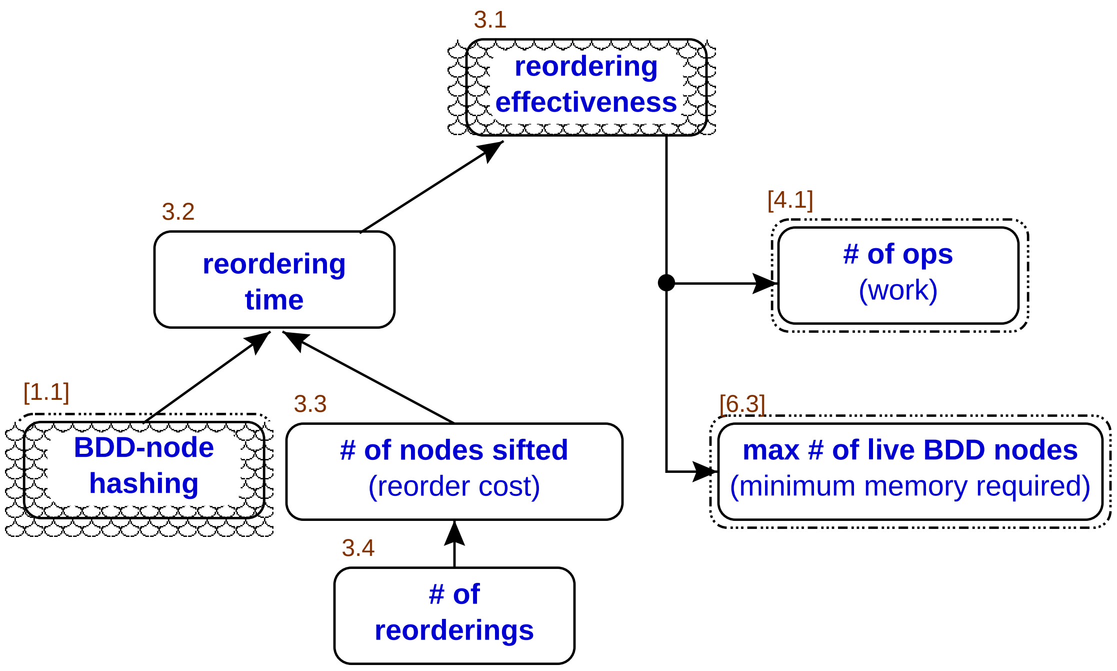


Figure 3.4: Metrics for studying dynamic-variable-reordering algorithms.

3.1.4 Metrics for Work

We define the amount of work performed by BDD computations to be the number of operations recursively generated. Figure 3.5 shows the dependence graph of metrics for work. The nodes in the graph are:

4.1 **# of ops:** the number of operations recursively generated by BDD computations. This metric provides a machine-independent measure for evaluating the impact of different algorithms, such as dynamic-variable-reordering algorithms or computed-cache strategies.

[3.1] **reordering effectiveness:** the impact of dynamic variable reordering. The quality of the variable order produced can have a very strong impact on BDD graph sizes, which in turn, affect the number of operations generated.

[2.6] **# of GCs:** the number of times garbage collection is invoked. The frequency of garbage collection can have a significant impact on the total number of operations because during garbage collection, computed cache entries with references to dead BDD nodes must be cleared. A more detailed description of this effect can be found in Section 2.4 and Section 3.1.2.

[1.5] **cache-node hashing:** the quality of the hashing algorithm used for the computed cache. The computed-cache hashing strategy affects both which and how many computed results are recorded, and thus directly influences the number of operations being repeated.

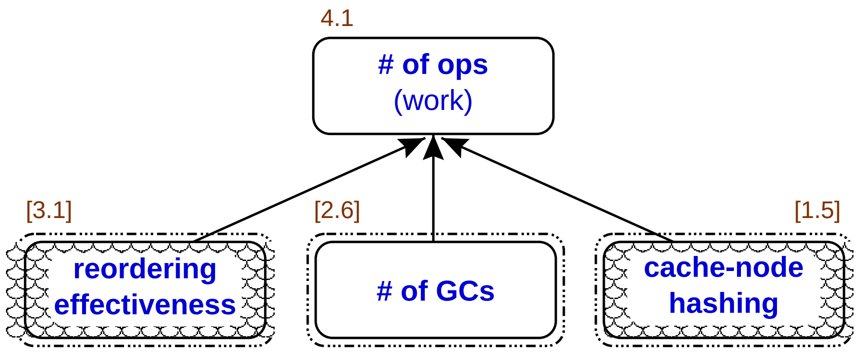


Figure 3.5: Metrics for studying the amount of work performed by the BDD algorithms.

3.1.5 Metrics for Memory Locality

BDD computation is very memory intensive, and thus many research efforts have focused on new algorithms for better memory locality [4, 56, 63, 64, 71, 83] and to study the memory locality of BDD packages [57]. One common drawback of all these approaches is that the evaluation metrics used do not account for implementation details that could have significant overall performance impact. For example, the garbage collection frequency and the computed cache size can impact the overall performance by over 100 times (see Section 4.2). Without accounting for these effects, the conclusions drawn can be misleading.

Figure 3.6 shows the dependence graph of metrics for studying memory locality. The nodes in the graph are:

5.1 **memory locality:** the spatial and temporal locality of memory references. This qualitative metric cannot be easily measured directly.

5.2 **page-fault rate:** the number of page faults / elapsed time. This measures the locality of memory accesses at page level and is effective only when the computation does not fit in the physical memory.

5.3 **hardware-cache miss rate:** the number of hardware-cache misses / CPU time. This measures the locality of memory accesses at cache-line level. This metric requires either hardware support to report the number of cache misses or can be obtained through software simulation. One way to simulate the cache is to instrument the BDD packages with software such as ATOM [38] and measure the cache miss rate through a software simulated cache.

5.4 **TLB miss rate:** the miss rate of the translation-lookaside buffer (TLB). The TLB is used to cache the translation between virtual and physical page. This metric measures the locality of memory accesses at page level. Unlike the page-fault rate, this metric is useful when the computations fits in the main memory. Similar to "*hardware-cache miss rate*", this metric requires either hardware support to report the number of cache misses or can be obtained through software simulation. One way to simulate the TLB is to instrument the BDD packages with software such as ATOM [38] and measure the cache miss rate through a software simulated TLB.

5.5 **# of ops processed per BF queue visit:** the number of operations processed by the breadth-first technique / the total number of visits to the breadth-first queues. The breadth-first techniques can improve memory locality by clustering BDD nodes and operations of the same variable together. Thus, this metric provides an indication on the maximum benefit one can derive from such clustering. If this value is small, then breadth-first techniques have little impact on memory locality. On the other hand, when this number is high, we have an indirect indication that breadth-first techniques may improve memory locality. In this case, the metrics described above can be used to further substantiate their impact.

[6.1] **max memory usage:** the maximum amount of memory used. When the memory usage exceeds the available physical memory, it becomes a major contributing factor to the page-fault rate and thus can have significant overall performance.

[4.1] **# of ops:** the number of operations recursively generated by the BDD computations. This metric directly affects the hardware-cache miss rate and the TLB miss rate.

This graph indicates that even though the page-fault rate, hardware-cache miss rate, and TLB miss rate are good measures of memory locality, one must also take into account the memory usage and the amount of work performed to properly assess the impact of different algorithms on memory locality.

3.1.6 Metrics for Memory Usage

Figure 3.7 shows the dependence graph of the metrics for memory usage. The metrics are:

6.1 **max memory usage:** the maximum amount of memory used.

6.2 **max # of BDD nodes:** the maximum number of BDD nodes. In BDD computations, BDD nodes are typically the largest source of memory consumption.

6.3 **max # of live BDD nodes:** the maximum number of live BDD nodes. When variable reordering is disabled, the maximum number of live BDD nodes is generally fixed for given BDD computations. Thus this is a machine- and implementation-independent metric for measuring the minimum memory required to solve BDD computations.

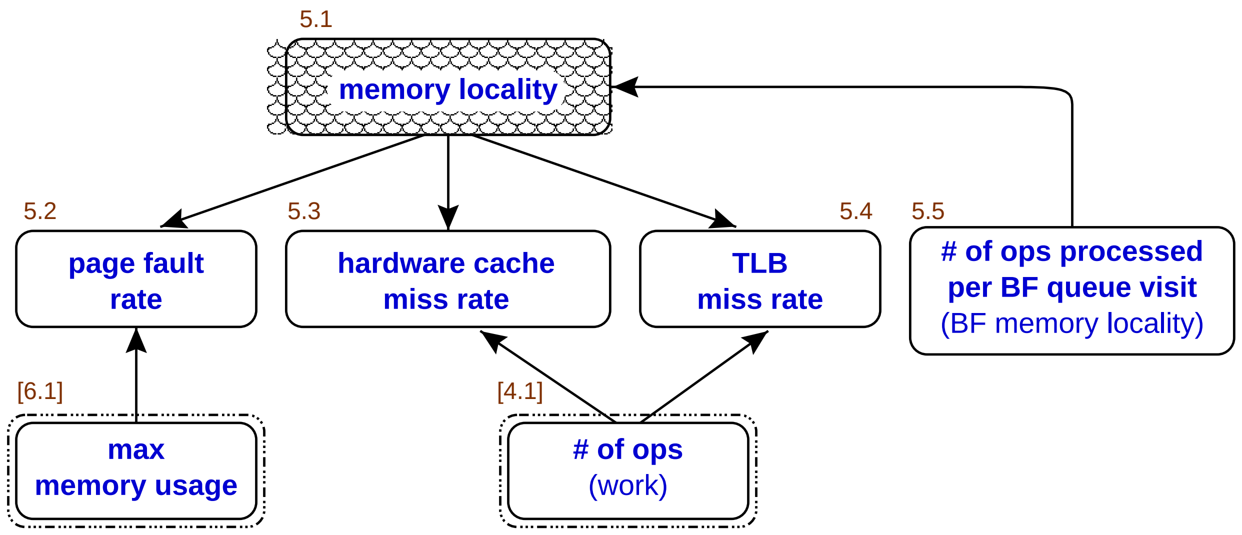


Figure 3.6: Metrics for studying memory locality.

The one exception where this metric fails to measure the absolute minimum memory requirement is when the BDD operations have more than one level of recursion, such as the relational-product operation which generates new OR operations as a second-level of recursion to quantify out specified variables (line 7 in Figure 2.5). For these operations, a computed-cache miss can mean generating temporary BDD results from the first level of recursion (for relational product, this is recursive decomposition of the relational-product operation), and then these temporary BDDs are used to construct the result of this operation through a second level of recursion (for relational product, this is the OR operation to perform quantification). These temporary BDDs not only can increase the maximum number of BDD nodes, they also can increase the maximum number of live BDD nodes in the following way. Suppose a relational-product operation was previously computed and the result BDD already exists in memory, but this operation is no longer cached. Then recomputing this relational product may generate many new temporary BDDs due to two levels of recursion and thus increases the number of live BDD nodes. This increase would not have happened if the result of this operation were kept in the cache. Since both the frequency of garbage collection and the effectiveness of computed-cache hashing both can attribute to cache misses, the maximum number of live BDD nodes depends on these two metrics.

Despite this exception, the maximum number of live BDD nodes still serves as a good measure for the minimum amount of memory required for BDD computations.

[2.6] # of GCs: the number of times garbage collection is invoked. The frequency of garbage collection affects the percentage of the dead BDD nodes in the system and thus affects the overall memory usage.

[3.1] reordering effectiveness: the effectiveness of dynamic variable reordering. Dynamic variable reordering directly affects the BDD graph sizes and thus can be a major contributing factor in memory usage.

[1.8] max computed cache size: the maximum number of total entries of the computed caches. The size of the computed cache is a major source of memory usage in BDD computations.

[1.5] **cache-node hashing:** the quality of the hashing algorithm used for the computed cache. For operations that have two levels of recursion, a cache miss may result in generating additional temporary BDDs and thus it can affect both the maximum number of BDD nodes and the maximum number of live BDD nodes. For more detail, please refer to the description for the metric "*max # of live BDD nodes*".

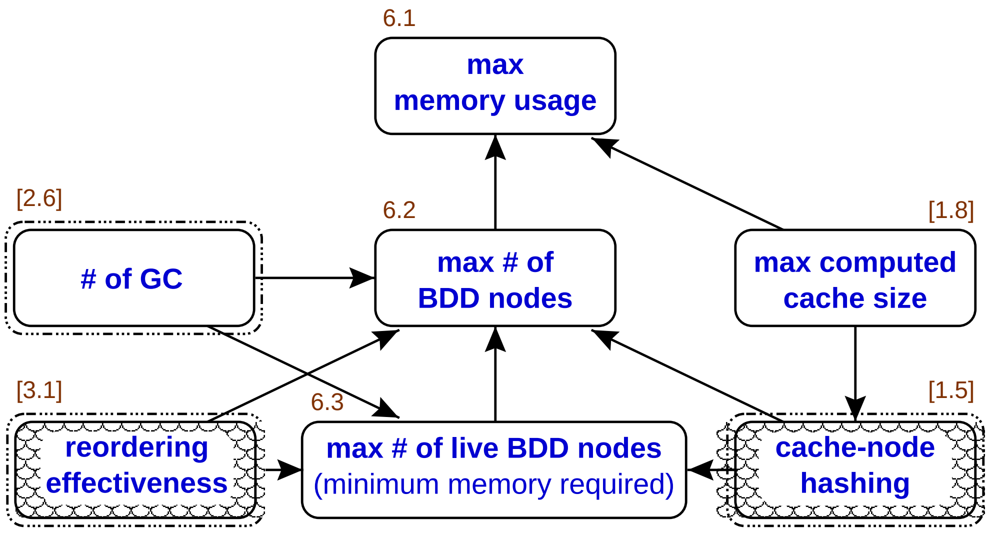


Figure 3.7: Metrics for studying memory usage.

3.1.7 Metrics for Time

The ultimate performance measure is running time. Figure 3.8 shows the dependence graph of the metrics related to time. The metrics are:

7.1 **elapsed time:** the total amount of time used. This is also known as the wall-clock time or real time.

7.2 **CPU time:** the amount of time used by the CPU.

[2.5] **GC time:** time spent on performing garbage collection. This is measured in CPU seconds. Garbage collection is one of the three major parts of a typical BDD package.

[3.2] **reordering time:** the time spent in dynamic variable reordering. This is measured in CPU seconds. Dynamic variable reordering is another major part of a typical BDD package.

7.3 **BDD alg time:** the time spent in computing BDD operations. This is measured in CPU seconds. This part is the third major part of a typical BDD package.

[4.1] **# of ops:** the number of operations recursively generated by the BDD computations. This is a machine-independent metric for measuring the amount of work performed by the BDD algorithms.

[1.1] **BDD-node hashing:** the quality of the hashing algorithm used for the BDD unique table. Hashing BDD nodes is necessary to guarantee the uniqueness of BDD nodes in both dynamic variable reordering and in generating the resulting BDDs for BDD operations. For garbage collection, depending on the algorithm used, rehashing BDD nodes may or may not be necessary.

[5.2] **page-fault rate:** the number of page faults / elapsed time. This measures the locality of memory accesses at page level. If the number of page-faults is high, the elapsed time can be over an order of magnitude larger than the CPU time.

[6.1] **max memory usage:** the maximum amount of memory used. If the page fault rate is high, then this metric can provide an indication of its causes. In particular, if the maximum memory usage exceeds the available physical memory, the page-fault rate will increase dramatically.

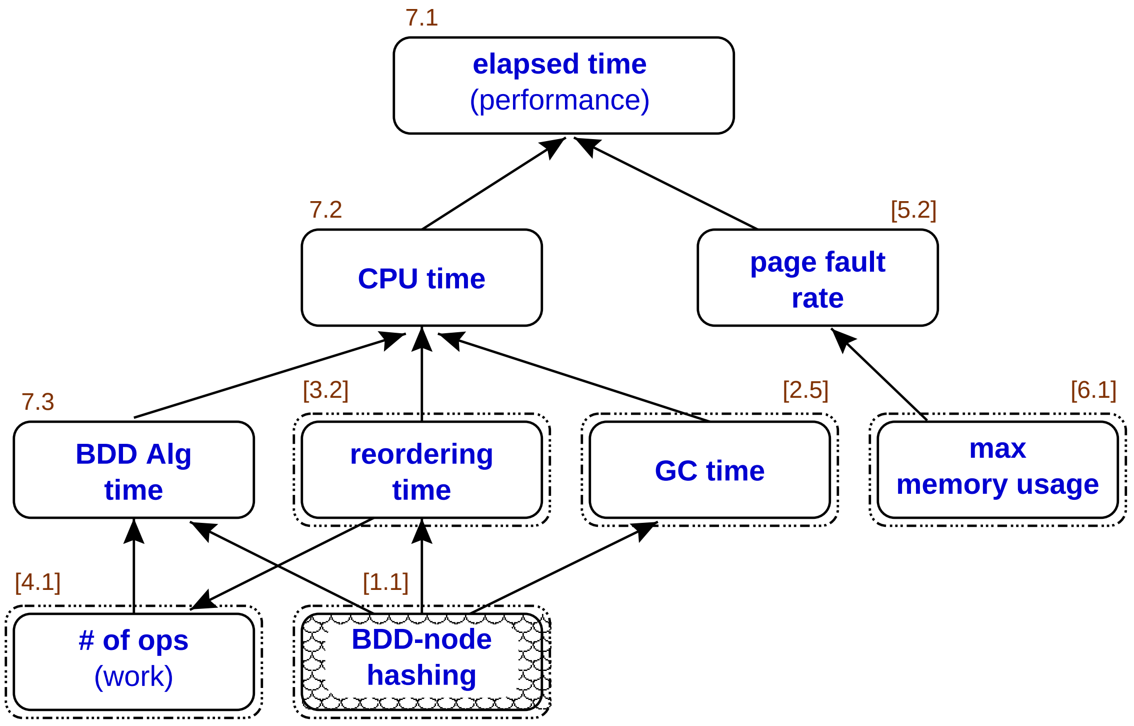


Figure 3.8: Metrics for studying overall performance.

3.1.8 Metrics for the Size of BDD Computations

In this section, we proposes three machine- and implementation-independent metrics for quantifying the size of BDD computations. Figure 3.9 shows the dependence graph of the metrics for this. The metrics are:

8.1 **size of BDD computations:** a measure on the difficulty of the given BDD computations. This qualitative metric cannot be measured directly but can be characterized by the metrics described next.

8.2 **# of BDD vars:** the total number of BDD variables used. This metric gives a very rough estimate on the difficulty of the computations involved.

8.3 **min # of ops:** the minimum number of recursive operations with dynamic variable reordering disabled. This metric provides the minimum amount of work necessary to compute the given BDD computations. This metric can be measured by (1) disabling the dynamic variable reordering, (2) maintaining a complete computed cache, and (3) disabling garbage collection. Step (2) and (3) are necessary to ensure the subproblems are never computed more than once. Note that due (2) and (3), the amount of memory required can be very large. For long running BDD computations, these two metrics can be approximated by garbage collecting only when necessary and by keeping the computed cache size as large as possible.

[6.3] **max # of live BDD nodes:** the maximum number of live BDD nodes with dynamic variable reordering disabled. With a fixed variable order, the maximum number of live BDD nodes are generally fixed for given BDD computations. Thus, this is a machine- and implementation-independent metric for measuring the minimum memory required to solve BDD computations. Note that there is an exception where this metric overestimates the absolute minimum memory requirement. This exception is discussed in the definition of this metric in Section 3.1.6.

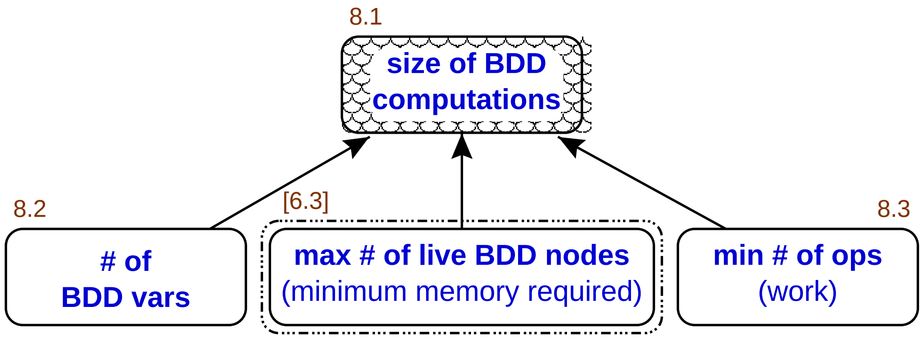


Figure 3.9: Metrics for quantifying the size of BDD computations. With dynamic variable reordering disabled, these metrics are both machine- and implementation-independent.

3.2 Trace-Based Evaluation Platform

This section describes a framework for studying BDD computations using realistic workloads. This framework is based on recording BDD execution traces from BDD-based tools and then replaying these traces on BDD packages. This framework allows the evaluation to focus on selected operations. Thus, we can study the impact of new BDD techniques without implementing all the auxiliary functions needed by BDD-based tools. For example, some model-checking tools such as SMV [58] use non-Boolean variants of BDDs (e.g., MTBDDs [28]) to represent non-Boolean state variables or to specify non-deterministic transitions. Implementing all these functions to study new BDD techniques can be time consuming. In the case of SMV, because the overall computation time is generally dominated by Boolean operations, we can focus the study on these Boolean operations by recording only these operations. In general, this framework not only allows evaluation based on a realistic workload, but it also facilitates the evaluation of new BDD techniques by eliminating the need to fully integrate with various BDD-based tools. Similar techniques have been used effectively for years by the processor and the cache design community.

Our evaluation framework consists of the following three parts:

Common Decision Diagram Interface

To facilitate the trace generation and replay, we define an interface standard called *Decision Diagram Interface* (DDI). Figure 3.10 shows a BDD-based tool (Figure 3.10(a)) instrumented with DDI adaptors (Figure 3.10(b)). In this figure, P_i is the tool's original interface to its own BDD package *i*. The adaptor $P_i \rightarrow$

DDI translates the tool's BDD function calls to conform with the DDI standard. The adaptor *DDI* $\rightarrow P_i$ reverses this process and translates the DDI calls back to the BDD package i's interface. Implementation-wise, each adaptor is a collection of wrapper functions that translate the BDD calls between the interfaces.

Other than facilitating trace generation and replay, this common interface has the additional advantage of allowing BDD-based tools to use other BDD packages (Figure 3.11). Note that as in the design of other software such as compilers, a common interface reduces the complexity of porting effort [29]. For example, to port n BDD packages to m BDD-based tools would traditionally require an $O(nm)$ amount of effort. With a common interface, this porting effort is reduced to $O(n+m)$.

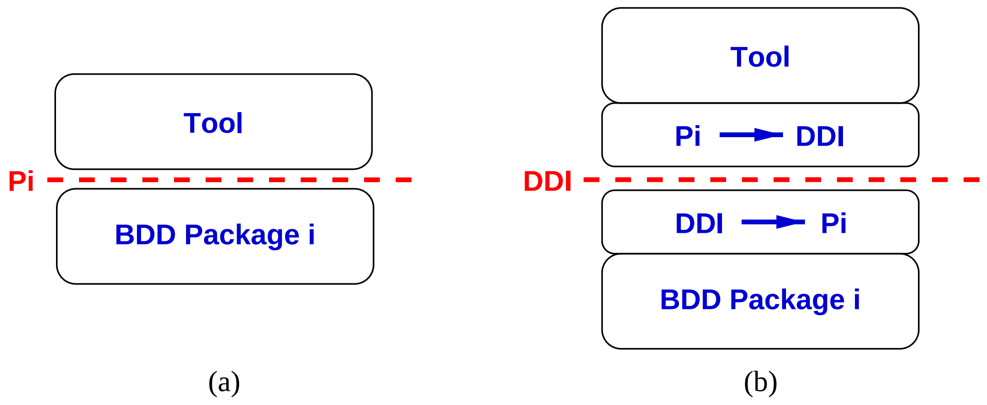


Figure 3.10: Inserting DDI adaptors to a BDD-based tool. (a) the tool, where P_i is the interface between the tool and its own BDD package i. (b) the tool with the DDI adaptors inserted.

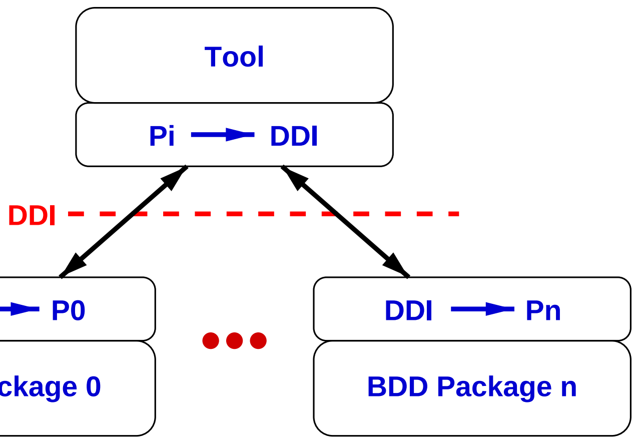


Figure 3.11: Using a common interface, a BDD-based tool can then interface with various BDD packages.

Benchmark Generation

The *trace recorder* is inserted into a BDD-based tool to record the BDD-function calls. The trace recorder can be used to record a subset of BDD function calls. The advantage of this feature is to facilitate the study of various BDD packages (algorithms) without requiring each package to implement the full set of BDD operations.

A side effect of recording only a subset of BDD operations is that the construction process of some

BDDs is skipped, and these BDDs might be needed later by some of the selected operations. Thus, before an operation is recorded in the trace, if the constructions of some of its arguments have not been recorded, the *trace recorder* must generate a description on how the BDDs of the missing arguments can be reconstructed. In the current implementation, we generate the reconstruction description of the missing BDD arguments by traversing the BDDs bottom-up and use the *if-then-else* operator to build each BDD node. This process is based on the property that each BDD node (v_i, $child_0$, $child_1$) essentially represents the Boolean function "if v_i then $child_1$ else $child_0$".

Another feature incorporated into the trace recorder is to record the size of result BDDs and the output of the equality comparison. This feature can be used to generate traces for debugging purposes during development of a BDD package.

Similar to the DDI adaptors, the trace recorder is mainly a collection of wrapper functions that records the BDD operations. Additional support is needed to perform the aforementioned BDD reconstruction where the *if-then-else* operations are added to the trace to reconstruct the necessary BDD arguments. Another implementation detail is that we need to establish a mapping between the BDDs and the corresponding names in the trace file. During garbage collection, there needs to be a call-back mechanism to remove dead BDDs from this mapping.

Figure 3.12 shows incorporation of the trace recorder into the tool. Here, using a common DDI interface has the advantage that the same trace recorder can be reused to gather traces from other BDD-based tools.

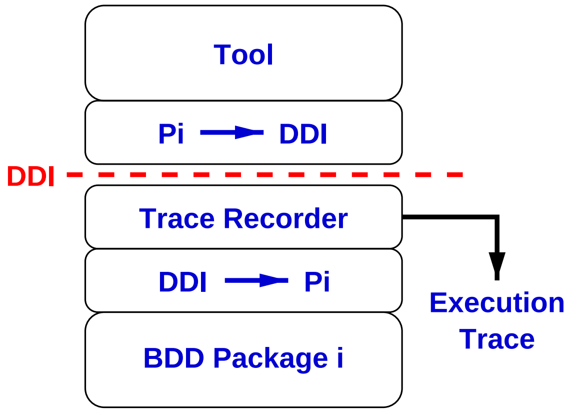


Figure 3.12: Benchmark generation.

Trace Replay

The *trace player* is a platform that provides a simple interface to BDD packages and replays the traces on these packages (Figure 3.13). This platform can then be used to systematically study various BDD packages using traces from real workloads.

The trace player also serves as a platform for developing new BDD packages. With the traces recording the attributes of result BDDs of each computation, we can use the trace player to check if a BDD package is producing the expected result and thus it serves as a debugging platform. Furthermore, it also serves as a performance-tuning platform. We can use the trace player to gather the results for the evaluation metrics for both the BDD package we are developing and for a state-of-the-art BDD packages. By comparing the results, we can identify the performance bottlenecks and try to come up with solutions.

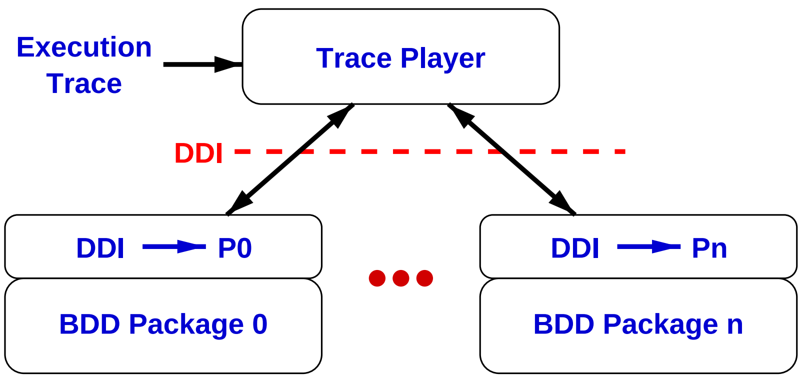


Figure 3.13: Trace replay.

3.3 Summary

This chapter described the first evaluation methodology to study general BDD computations. The dependence graphs for the evaluation metrics serve as a road map for systematically identifying performance bottlenecks. One powerful way to use this dependence graph is to incorporate it in a performance visualization tool that automatically identifies the main components for performance bottlenecks. Our trace recorder and the trace player software serve as an evaluation platform for generating realistic benchmarks and for studying BDD computations. We hope that this work will help transform this field's future performance evaluation to a more systematic quantitative approach.

Chapter 4

BDD Performance Study

Even though BDDs are the core technology for symbolic model checking, there has been little work on studying the computational aspects of BDDs in model checking. Historically, characterizations and comparisons of new BDD-based algorithms have been based on two sets of benchmark circuits: ISCAS85 [13] and ISCAS89 [12]. However, results based on these combinational-circuit benchmarks may not accurately characterize BDD computations in a completely different class of problems such as model checking. Furthermore, there are two qualitative differences between building BDD representations for combinational circuits versus model checking. The first difference is that for combinational circuits, the *output BDDs* (BDD representations for the circuit outputs) are built and then are only used for constant-time equivalence checking. In contrast, a model checker first builds the BDD representations for the system's transition relation, and then performs a series of fixed point computations analyzing the state space of the system. In doing so, it is solving PSPACE-complete problems. Another difference is that BDD construction algorithms for combinational-circuit operations have polynomial complexity in the graph sizes of the input arguments [17], while the key operations in model checking are NP-hard [58]. These differences are our main motivations for organizing a BDD study on model-checking computations.

Using the evaluation methodology described in Chapter 3, we have organized and conducted the first systematic BDD performance study. This study is a collaboration of six BDD designers with their own BDD packages. The model-checking benchmark consists of 16 execution traces from the Symbolic Model Verifier (SMV) [58]. For comparison with combinational circuits, we also studied 4 circuits derived from the ISCAS85 benchmark.

This systematic and collaborative evaluation methodology has led to better understanding of the effects of cache size and garbage collection frequency and has also resulted in significant performance improvements (some up to 2 orders of magnitude) across all the BDD packages. Systematic evaluation also uncovered vast differences in the computational characteristics of model checking and combinational circuits. These differences include the effects of the cache size, the garbage collection frequency, the complement edge representation [3], and the memory locality of the breadth-first BDD packages. For the difficult issue of dynamic variable reordering, we introduce some methodologies for studying the effects of variable reordering algorithms and initial variable orders.

It is important to note that the results in this study are obtained based on a very small sample of all possible BDD-based model-checking computations. Thus, in this chapter, the results are presented as hypotheses along with their supporting evidence. These results are not conclusive. Instead, they raise a number of interesting issues and suggest new research directions.

The rest of this chapter is organized as follows. We first present the experimental setup for the study (Section 4.1). This is followed by three sections of experimental results. First, we report the findings without dynamic variable reordering (Section 4.2). Then, we present the results on dynamic variable reordering algorithms and the effects of initial variable orders (Section 4.3). Third, we present results that may be generally helpful in studying or improving BDD packages (Section 4.4). After these result sections, we wrap up with a discussion on some unresolved issues (Section 4.5). The work described in this chapter can also be found in [82].

4.1 Setup of the Study

4.1.1 Benchmark

The benchmark used in this study is a set of execution traces gathered from the Symbolic Model Verifier (SMV) [58] from Carnegie Mellon University. The traces were gathered by recording BDD function calls made during the execution of SMV. To facilitate the porting process for different packages, we recorded only a set of the key Boolean operations and discarded all word-level operations. The coverage of the selected set of BDD operations is greater than 95% of the total SMV execution time for all but one case (*abp11*) which spends 21% of CPU time in the word-level functions constructing the transition relation.

For this study, we have selected 16 SMV models to generate the traces. The following is a brief description of these models along with their sources.

abp11: alternating bit protocol.
Source: Armin Biere, Universität Karlsruhe.

dartes: communication protocol of an Ada program.

dpd75: dining philosophers protocol.

ftp3: file transfer protocol.

furnace17: remote furnace program.

key10: keyboard/screen interaction protocol in a window manager.

mmgt20: distributed memory manager protocol.

over12: automated highway system overtake protocol.
Source: James Corbett, University of Hawaii.

dme2-16: distributed mutual exclusion protocol.
Source: SMV distribution, Carnegie Mellon University.

futurebus: FutureBus cache coherence protocol.
Source: Somesh Jha, Carnegie Mellon University.

motor-stuck: batch-reactor system model.

valves-gates: batch-reactor system model.
Source: Adam Turk, Carnegie Mellon University.

phone-async: asynchronous model of a simple telephone system.

phone-sync-CW: synchronous model of a telephone system with call waiting.
Source: Malte Plath and Mark Ryan, University of Birmingham, Great Britain.

tcas: traffic alert and collision system for airplanes.
Source: William Chan, University of Washington.

tomasulo: a buggy model of the Tomasulo algorithm for instruction scheduling in superscalar processors.
Source: Yunshan Zhu, Carnegie Mellon University.

As we studied and improved on the model-checking computations during the course of the study, we compared their computational characteristics with the BDD construction of combinational-circuit outputs. For this comparison, we used the ISCAS85 benchmark circuits as the representative circuits. We chose ISCAS85 because it is perhaps the most popular benchmark suite for BDD performance evaluations. The ISCAS85 circuits were converted into the same format as the model checking traces (courtesy of Yirng-An Chen). The variable orders used were generated by the *order-dfs* in SIS [75]. We excluded cases that were either too small ($<$ 5 CPU seconds) or too large ($>$ 1 GByte memory requirement). Based on this criterion, we were left with two circuits—C2670 and C3540. To obtain more circuits, we derived 13-bit and 14-bit integer multipliers, based on the C6288, which we refer to as C6288-13 and C6288-14. For the multipliers, the variable order is $a_{n-1} \prec a_{n-2} \prec ... \prec a_0 \prec b_{n-1} \prec b_{n-2} \prec ... \prec b_0$, where $A = \sum_{i=0}^{n-1} 2^i a_i$ and $B = \sum_{i=0}^{n-1} 2^i b_i$ are the two n-bit input operands to the multiplier.

Figure 4.1 quantifies the sizes of the traces based on the metrics in Section 3.1.8. The metric "*# of BDD vars*" is the number of BDD variables used. The metric "*min # of ops*" is the minimum number of sub-operations (or subproblems) needed for the computation. This metric characterizes the minimum amount of work for each trace. It was gathered using a BDD package with a complete cache and no garbage collection. Thus, this metric represents the minimum number of sub-operations needed for a typical BDD package. Due to insufficient memory, there are 4 cases (*futurebus*, *phone-sync-CW*, *tcas*, *tomasulo*) for which we were not able to obtain the results for this metric. For these cases, the results shown are the minimum across all the packages used in the study. These results are marked with the "$<$" symbol. The third metric, "*max # of live BDD nodes*", represents the peak number of reachable BDD nodes during the execution. It provides a lower bound on the memory required to execute the corresponding trace. Note that neither "*min # of ops*" nor "*max # of live BDD nodes*" reflects the effects of the dynamic variable reordering process.

4.1.2 BDD Packages

The following is a list of the BDD packages used in the study. For each BDD package, we note how it differs from the common implementation described in Section 2.4. Although many of these BDD packages contain a wide variety of useful features, only those pertinent to the study are described in this section.

ABCD (Author: Armin Biere)
ABCD [9] is an experimental BDD package based on the classical depth-first traversal. Interesting features include mark-and-sweep based garbage collection, the integration of BDD nodes with the BDD unique table by using open addressing, and index-based (instead of pointer-based) references to BDD nodes. These techniques reduce the BDD node size by half (2 machine words instead of 4). In addition, to avoid clustering in open addressing, ABCD uses a quadratic probe sequence for the hashing collision resolution.

CAL (Authors: Rajeev Ranjan and Jagesh Sanghavi)
CAL [70] is a publicly available BDD package based on breadth-first traversal to exploit memory locality. The garbage collection algorithm is based on reference-counting with memory compaction. To increase locality of reference, each BDD node contains the indices of its cofactor nodes. To keep

Trace	# of BDD vars	min # of ops ($\times 10^6$)	max # of live BDD nodes ($\times 10^3$)
abp11	122	116	53
dartes	198	6	468
dme2-16	586	106	905
dpd75	600	41	1719
ftp3	100	132	763
furnace17	184	30	2109
futurebus	348	$<$ 10270	4473
key10	140	91	626
mmgt20	264	35	1113
motors-stuck	172	29	325
over12	174	58	3008
phone-async	86	329	1446
phone-sync-CW	88	$<$ 3803	22829
tcas	292	$<$ 1323	19921
tomasulo	212	$<$ 1497	26944
valves-gates	172	44	433
c2670	233	15	4363
c3540	50	57	7775
c6288-13	26	60	3378
c6288-14	28	178	9662

Figure 4.1: Sizes of the benchmark traces. "*# of BDD vars*" is the number of BDD variables. "*min # of ops*" is the minimum number of sub-operations which characterizes work. The entries marked with the "$<$" symbol indicate the cases where we were not able to obtain the results for this metric. The results shown are the minimum across all the packages used in the study. "*max # of live BDD nodes*" is the maximum number of reachable BDD nodes, which characterizes the minimum memory requirement.

the node size to 4 machine words, bit tagging is used to store and retrieve the value of the reference count of a node. For this study, the relational product operation is based on the depth-first traversal with the quantification step (line 7 in Figure 2.5) computed using the breadth-first traversal.

CUDD (Author: Fabio Somenzi)
CUDD [78] is a publicly available BDD package based on depth-first traversal. In CUDD, the reference counts of the nodes were kept up-to-date throughout the computation. To counter the impact on performance of these updates when many nodes are freed and reclaimed, CUDD enqueues the requests for updates and performs them only if they are still valid when they are extracted from the queue. The growth of the tables in CUDD is determined by a reward policy. For instance, the cache grows if the hit rate is high. CUDD partially sorts the free list during garbage collection to improve memory locality. Another distinguishing feature is that CUDD contains a suite of heuristics for dynamic variable reordering.

EHV (Author: Geert Janssen)
EHV [49] is a publicly available BDD package based on depth-first traversal. The main differences from the common implementation features are additional support for inverted inputs [60] and provisions for user data to be attached to a BDD node. The latter feature allows intermediate results to be stored in the BDD nodes, which in turn removes the need to use separate computed caches for some special BDD operations. This feature incurs a memory overhead of 2 extra machine words per BDD node.

PBF (Authors: Bwolen Yang and Yirng-An Chen)
PBF [83] is an experimental BDD package based on partial breadth-first traversal. The partial breadth-first traversal along with per-variable memory managers and the memory-compacting mark-and-sweep garbage collector are used to exploit memory locality. The partial breadth-first traversal also bounds the breadth-first expansion to avoid the potential excessive memory overhead of a full breadth-first expansion.

TiGeR (Authors: Olivier Coudert, Jean C. Madre and Herve Touati)
TiGeR [32] is a commercial BDD package based on the depth-first approach. Interesting features include the segmentation of the computed caches and the garbage collection algorithm. In TiGeR, each operation type has its own cache. This allows the caches to be tuned independently. For this study, the caches for the non-polynomial operations such as relational product are set to be about four times as sparse as the caches for the polynomial operations. TiGeR's garbage collection algorithm is different from typical garbage collection algorithms in two ways: the free-list is sorted to maintain memory locality, and the memory compaction is performed when memory resources become critical.

4.1.3 Evaluation Process

The performance study was carried out in two phases. The first phase studied performance issues in BDD construction without variable reordering. The second phase focused on the impact of dynamic variable reordering. The evaluation process was iterative, with the study evolving dynamically as new issues were raised and new insights gained. Based on the results from each iteration, we collaboratively tried to identify the performance issues and possible improvements. Each BDD package designer then incorporated and validated the suggested improvements. During this iterative process, we also tried to hypothesize the characteristics of the computation and design new experiments to test these hypotheses.

4.2 Phase 1 Results: No Variable Reordering

Figure 4.2 presents the overall performance improvements for Phase 1 with dynamic variable reordering disabled. There are 6 packages and 16 model-checking traces, for a total of 96 cases. Figure 4.2(a) categorizes the results for these cases based on speedups. Note that the speedups are plotted in a cumulative fashion; i.e., the $> x$ column represents the total number of cases with speedups greater than x. Figure 4.2(b) presents a comparison between the initial timing results (when we first started the study) and the final timing results (after the authors made changes to their packages based on insights gained from previous iterations). The *n/a* results represent cases where results could not be obtained.

Initially, 19 cases did not complete because of implementation bugs or memory limits. 13 of these 19 cases were completed at the end of the study (the *new* cases in the charts). The other 6 cases still did not complete within the resource limit of 8 hours and 900 MByte (the *failed* cases in the charts). There is one case (the *bad* case in the charts) that initially completed, but did not complete at the end of the study. This *bad* case occurred because the particular BDD package involved increased its overall memory usage to improve its performance across all the traces. However, for one trace, this package used too much memory and exceeded the memory limit.

Figure 4.2(a) shows that significant speedups have been obtained for many cases. Most notably, 22 cases have speedups greater than an order of magnitude (the > 10 column), and 6 out of these 22 cases actually achieve speedups greater than two orders of magnitude (the > 100 column)!

Figure 4.2(b) shows that significant speedups have been obtained mostly from the small to medium traces, although some of the larger traces have achieved speedups greater than 3. Another interesting point is that the *new* cases (those that initially failed but are doable at the end of the study) range across small to large traces.

Overall, for the 76 cases where the comparison could be made, the total CPU time was reduced from 554,949 seconds to 127,786 seconds—a speedup of 4.34. Another interesting overall statistic is that initially none of the 6 BDD packages could complete all 16 traces. At the end of the study, 3 BDD packages can complete all of them.

The remainder of this section presents results on a series of experiments that characterize the computational aspects of the BDD traces. We first present results on two aspects with significant performance impact—computed cache size and garbage collection frequency. Then we present results on the effects of the complement edge representation. Finally, we give results on memory locality issues for the breadth-first based traversal.

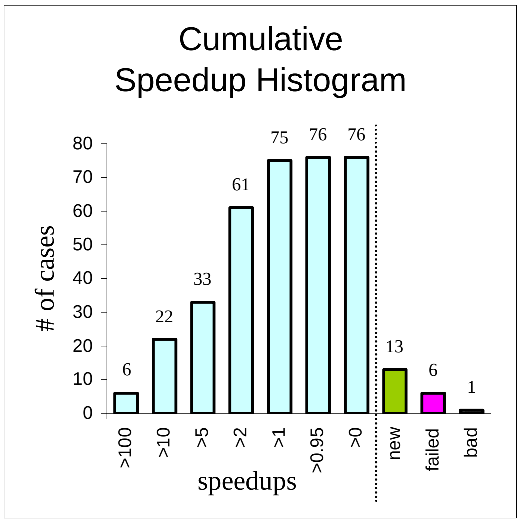
4.2.1 Computed Cache Size

We have found that dramatic performance improvements are possible by using a larger computed cache. To study the impact of the computed cache, we performed some experiments and arrived at the following two hypotheses.

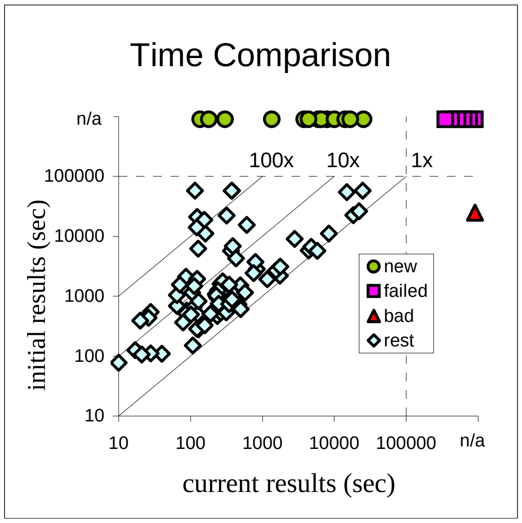
Hypothesis 1 *Model-checking computations have a large number of repeated subproblems across the top-level operations. On the other hand, combinational-circuit computations generally have far fewer such repeated subproblems.*

Experiment: Measure the minimum number of subproblems needed by using a complete cache (denoted CC-NO-GC). Compare this with the same setup but with the cache flushed between top-level operations (denoted CC-GC). For both cases, BDD-node garbage collection is disabled.

Result: Figure 4.3 shows the results of this experiment. Note that the results for the four largest model checking traces are not available due to insufficient memory.



(a)



(b)

Figure 4.2: Overall performance improvements without dynamic variable ordering. The *new* cases represent the cases that failed initially and are doable at the end of the study. The *failed* cases represent those that still exceed the limits of 8 CPU hours and 900 MByte at the end of the study. The *bad* shows the case that finished initially, but cannot complete at the end of the study. The *rest* are the remaining cases. (a) Results shown as histograms. For the 76 cases where both the initial and the final results are available, the speedup results are shown in a cumulative fashion; i.e., the $> x$ column represents the total number of cases with speedups greater than x. (b) Time comparison (in seconds) between the initial and the final results. *n/a* represents results that are not available due to exceeding resource limits.

These results show that for model-checking traces, there are indeed many subproblems repeated across the top-level operations. For 8 traces, the ratio of the number of operations in CC-GC over the number of operations in CC-NO-GC is greater than 10. In contrast, this ratio is less than 2 for building output BDDs for the ISCAS85 circuits. For model-checking computations, since subproblems can be repeated further apart in time, a larger cache is crucial.

Hypothesis 2 *The computed cache is more important for model checking than for combinational circuits.*

Experiment: Vary the cache size as a percentage of the number of BDD nodes and collect the results on the number of subproblems generated to measure the effect of the cache size. In this experiment, the cache sizes vary from 10% to 80% of the number of BDD nodes. The cache replacement policy used is FIFO (first-in-first-out).

Results: Figure 4.4 plots the results of this experiment. Each curve represents the result for a trace with varying cache sizes. The "*# of ops*" metric is normalized over the minimum number of operations necessary (i.e., the CC-NO-GC results). Note that for the four largest model-checking traces, the results are not available due to insufficient memory.

These results clearly show that the cache size can have much more significant effects on the model-checking computations than on building BDDs for the ISCAS85 circuit outputs. In particular, the traditional wisdom of keeping the cache size to be a small percentage of the total memory usage is not valid for model-checking computations.

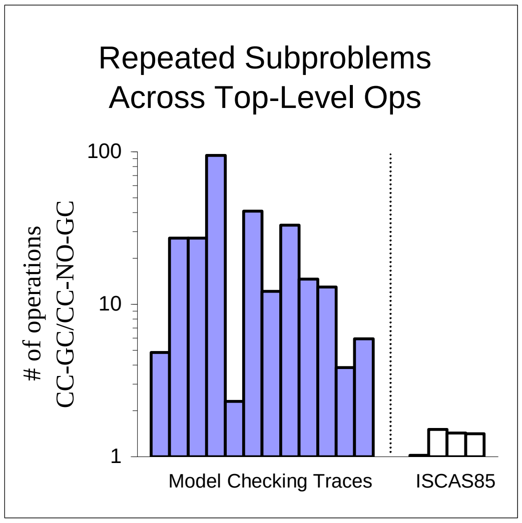


Figure 4.3: Performance measurement on the frequency of repeated subproblems across the top-level operations. CC-GC denotes the case in which the cache is flushed between the top-level operations. CC-NO-GC denotes the case in which the cache is never flushed. In both cases, a complete cache is maintained within a top-level operation and BDD-node garbage collection is disabled. For four model-checking traces, the results are not available (and are not shown) due to insufficient memory.

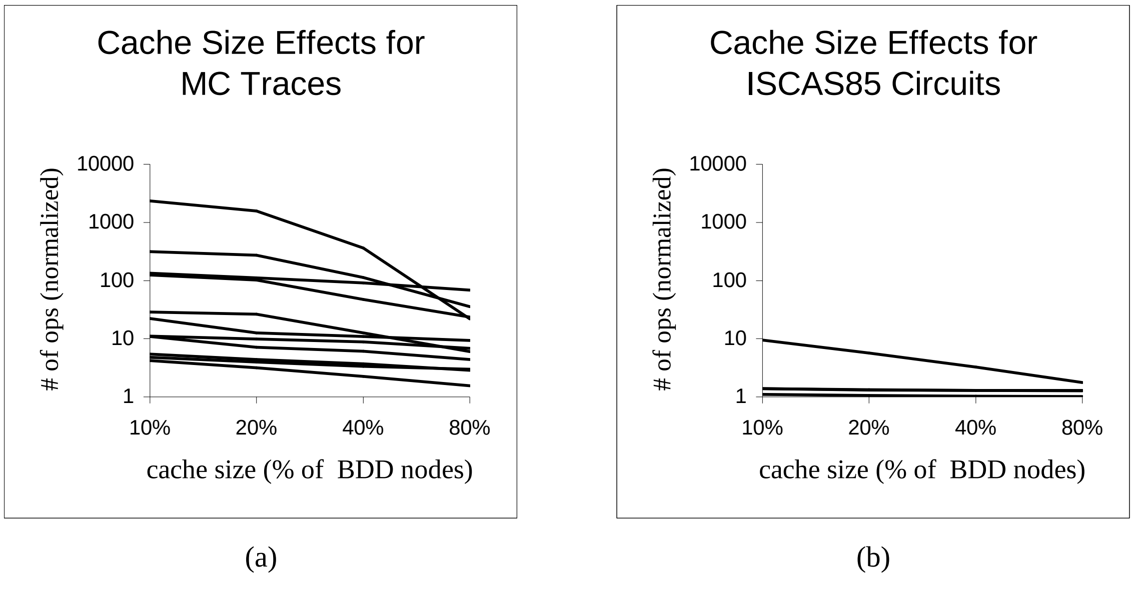


Figure 4.4: Effects of cache size on overall performance for (a) the model-checking traces and (b) the ISCAS85 circuits. The cache size is set to be a percentage of the number of BDD nodes. The number of operations (subproblems) is normalized to the minimum number of subproblems necessary (i.e., the CC-NO-GC results).

4.2.2 Garbage Collection Frequency

The other source of significant performance improvement is the reduction of the garbage collection frequency. We have found that for the model-checking traces, the rate at which reachable BDD nodes become unreachable (death rate) and the rate at which unreachable BDD nodes become reachable (rebirth rate) can be quite high. This leads to the following conclusions:

- Garbage collection should occur less frequently.
- Garbage collection should not be triggered solely based on the percentage of the unreachable nodes.
- For reference-counting based garbage collection algorithms, maintaining accurate reference counts all the time may incur non-negligible overhead.

Hypothesis 3 *Model-checking computations can have very high death and rebirth rates, whereas combinational-circuit computations have very low death and rebirth rates.*

Experiment: Measure the death and rebirth rates for the model-checking traces and the ISCAS85 circuits.

Results: Figure 4.5(a) plots the ratio of the total number of deaths over the total number of sub-operations. The number of sub-operations is used to represent *time*. This chart shows that the death rates for the model-checking traces can vary considerably. In 5 cases, the number of deaths is higher than the number of sub-operations (i.e., death rate is greater than 1). In contrast, the death rates of the ISCAS85 circuits are all less than 0.3.

The death rates of greater than 1 are quite unexpected. To explain the significance of this result, we digress briefly to describe the process of BDD nodes becoming unreachable (death) and then becoming reachable again (rebirth). When a BDD node becomes unreachable, its children can also become unreachable if this BDD node is its children's only reference. Thus, it is possible that when a BDD node becomes unreachable, a large number of its descendants also become unreachable. Similarly, if an unreachable BDD node becomes reachable again, its unreachable descendants, which can be large in numbers, can also become reachable. Other than rebirth, the only way the number of reachable nodes can increase is when a sub-operation creates a new BDD node as its result. Because each sub-operation can produce at most one new BDD node, a death rate of greater than 1 can only occur when the corresponding rebirth rate is also very high. In general, high death rate coupled with high rebirth rate indicates that many nodes are toggling between being reachable and being unreachable. Thus, for reference-counting based garbage collection algorithms, maintaining accurate reference counts all the time may incur significant overhead. This problem can be addressed by using a bounded-size queue (known as the *death row*[1]) to delay the reference-count updates until the queue overflows.

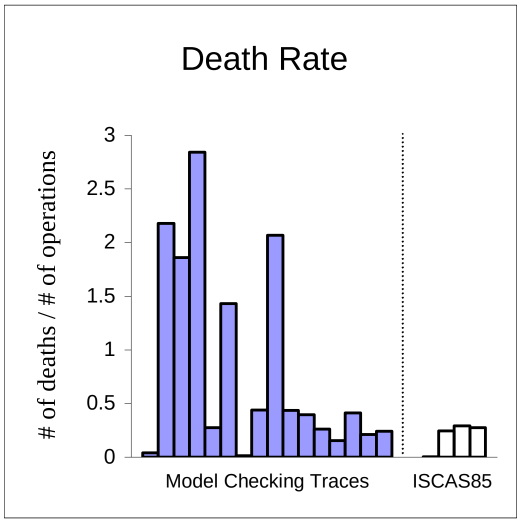
Figure 4.5(b) plots the ratio of the total number of rebirths over the total number of deaths. Since garbage collection is enabled in these runs and does reclaim unreachable nodes, the rebirth rates shown may be lower than without garbage collection. This figure shows that the rebirth rates for the model-checking traces are generally very high—8 out of 16 cases have rebirth rates greater than 80%. In comparison, the rebirth rate for the ISCAS85 circuits are all less than 30%.

The high rebirth rates indicate that garbage collection for the model checking traces should be delayed as long as possible. There are two reasons for this: first, because a large number of unreachable nodes become reachable again, garbage collection will not be very effective in reducing the memory usage. Second, the high rebirth rate may result in repeated subproblems involving the currently unreachable nodes. By garbage collecting these unreachable nodes, their corresponding computed-cache entries

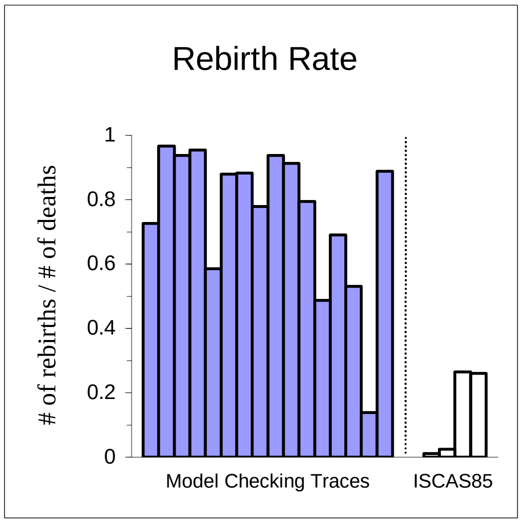
[1]The term *death row* is coined by Fabio Somenzi from the University of Colorado.

must also be cleared. Thus, garbage collection may greatly increase the number of re-computations of identical subproblems.

The high rebirth rates and the potentially high death rates also suggest that the garbage collection algorithm should not be triggered based solely on the percentage of the dead nodes, as with the classical BDD packages.



(a)



(b)

Figure 4.5: (a) Rate at which BDD nodes become unreachable (death). (b) Rate at which unreachable BDD nodes become reachable again (rebirth).

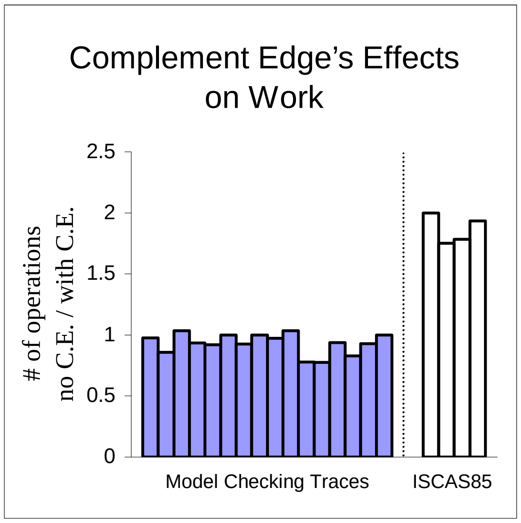
4.2.3 Effects of the Complement Edge

The complement-edge representation [3] has been found to be somewhat useful in reducing both the space and time required to build the output BDDs for the ISCAS85 circuits [60]. In the following experiments, we study the effects of the complement edge on the model-checking traces and compare them with the results for the ISCAS85 circuits.

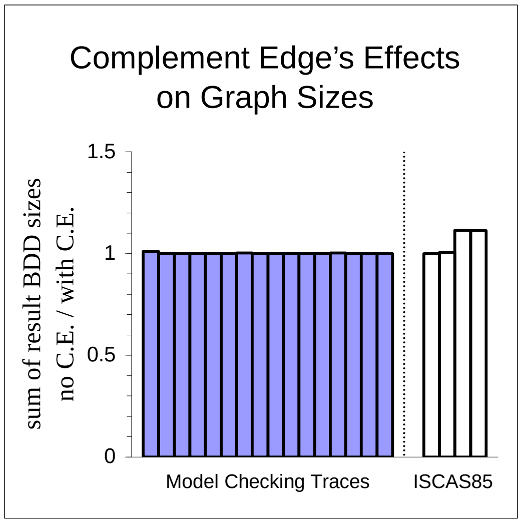
Hypothesis 4 *The complement-edge representation can significantly reduce the amount of work for combinational-circuit computations, but not for model-checking computations. However, in general, it has little impact on memory usage.*

Experiment: Measure and compare the number of subproblems (amount of work) and the resulting graph sizes (memory usage) generated from two BDD packages—one with and the other without the complement-edge feature. For the graph size measurements, sum the resulting BDD graph sizes of all top-level operations. Note that because two packages are used, minor differences in the number of operations can occur due to different garbage collection and caching algorithms.

Results: Figure 4.6(a) shows that the complement edges have no significant effect for model-checking traces. In contrast, for the ISCAS85 circuits, the ratio of the no-complement-edge results over with-complement-edge results ranges from 1.75 to 2.00. Figure 4.6(b) shows that the complement edges have no significant effect on the BDD graph sizes in any of the benchmark traces.



(a)



(b)

Figure 4.6: Effects of the complement-edge representation on (a) number of the operations and (b) graph sizes.

4.2.4 Memory Locality for Breadth-First BDD Construction

In recent years, a number of researchers have proposed breadth-first BDD construction to exploit memory locality [4, 63, 64, 71, 83]. The basic idea is that for each expansion phase, all sub-operations of the same variable are processed together. Similarly, for each reduction phase, all BDD nodes of the same variable are produced together. Based on this structured access pattern, we can exploit memory locality by using per-variable memory managers and per-variable breadth-first queues to cluster the nodes of the same variable together. This clustering is beneficial only if many nodes are processed for each breadth-first queue during each expansion and reduction phase.

In this study, we have not found any evidence that the breadth-first based packages are better than the depth-first based packages when the computation fits in main memory. Our conjecture is that since the relational-product algorithm (Figure 2.5) can have exponential complexity, the graph sizes of the BDD arguments do not have to be very large to incur a long running time. As a result, the number of nodes processed each time can be very small. The following experiment tests this conjecture.

Hypothesis 5 *For our test cases, few nodes are processed each time a breadth-first queue is visited. For the same amount of total work, combinational-circuit computations have much better* "breadth-first" *locality than model checking computations.*

Experiment: Measure the number of sub-operations processed each time a breadth-first queue is visited. Then compute the maximum, mean, and standard deviation of the results. Note that these calculations do not include the cases where the queues are empty since they have no impact on the memory locality issue.

Result: Figure 4.7 shows the results for this experiment. The top part of the table shows the results for the model-checking traces. The bottom part shows the results for the ISCAS85 circuits. We have also included the *"average / total # of ops"* column to show the results for the average number of sub-operations processed per pass, normalized against the total amount of work performed.

The results show that on average, 10 out of 16 model-checking traces processed less than 300 sub-operations (less than one 8-KByte memory page) in each pass. Overall, the average number of sub-operations in a breadth-first queue is at most 4685, which is less than 16 memory pages (128 KByte). This number is quite small given that hundreds of MByte of total memory are used. This shows that for these traces, the breadth-first approaches are not very effective in clustering accesses.

Trace	# of ops processed per BF queue visit			average / total # of ops
	average	max.	std. dev.	($\times 10^{-6}$)
abp11	228	41108	86.43	1.86
dartes	27	969	12.53	3.56
dme2-16	34	8122	17.22	0.31
dpd75	15	186	4.75	0.32
ftp3	1562	149792	63.11	8.80
furnace17	75	131071	42.40	2.38
futurebus	2176	207797	76.50	0.23
key10	155	31594	48.23	1.70
mmgt20	66	4741	21.67	1.73
motors-stuck	11	41712	50.14	0.39
over12	282	28582	55.60	3.32
phone-async	1497	175532	87.95	3.53
phone-sync-CW	1176	186937	80.83	0.19
tcas	1566	228907	69.86	1.16
tomasulo	2719	182582	71.20	1.95
valves-gates	25	51039	70.41	0.55
c2670	3816	147488	71.18	204.65
c3540	1971	219090	45.49	34.87
c6288-13	4594	229902	24.92	69.52
c6288-14	4685	237494	42.29	23.59

Figure 4.7: Results on memory locality of the breadth-first approach.

Another interesting result is that the maximum number of nodes in the queues is quite large and is generally more than 100 standard deviations away from the average. This result suggests that some depth-first and breadth-first hybrid (perhaps as an extension to what is done in the CAL package) may obtain further performance improvements.

The results for *"average / total # of ops"* clearly show that for the same amount of work, the ISCAS85 computations have much better locality for the breadth-first approaches. Thus, for a comparable level of "*breadth-first*" locality, model-checking applications might need to be much larger than the combinational circuit applications.

We have also studied the effects of the breadth-first approach's memory locality when the computations do not fit in the main memory. This experiment was performed by varying the size of the physical memory. The results show that the breadth-first based packages are significantly better only for the three largest cases (largest in terms of memory usage). The results are not very conclusive because as an artifact of this study, the participating BDD packages tend to use a lot more memory than they did before the study began, and furthermore, because these BDD packages generally do not adjust memory usage based on the actual physical memory sizes and page fault rates, the results are heavily influenced by excessive memory usage. Thus, they may not accurately reflect the effects of the memory locality of the breadth-first approach.

4.3 Phase 2 Results: Dynamic Variable Reordering

Evaluating dynamic variable reordering is inherently difficult for many reasons. First, there is a tradeoff between time spent in variable reordering and the total elapsed time. Second, small changes in the triggering and termination criteria may have significant impact in both the space and time requirements. Another difficulty is that because the space of possible variable orders is so huge and variable reordering algorithms tend to be very expensive, many machines are required to perform a comprehensive study. Due to these inherent difficulties and lack of resources, we were only able to obtain very preliminary results and have performed only one round of evaluation.

For this phase, only the CAL, CUDD, EHV, and TiGeR BDD packages were used, because the ABCD and PBF packages have no support for dynamic variable reordering. There are 4 packages and 16 traces, for a total of 64 cases. Figure 4.8 presents the timing results for these 64 cases. In this figure, the cases that did not complete within the resource limits are marked with *n/a*. The speedup lines ranging from 0.01x to 100x are included to help classify the performance results.

Figure 4.8(a) compares running time with and without dynamic variable reordering. With dynamic variable reordering enabled, 19 cases do not finish within the resource limits. Six of these 19 cases also cannot finish without variable reordering (the *failed* cases in Figure 4.8(a)). Thirteen of these 19 cases are doable without dynamic variable reordering enabled (the *bad* cases in Figure 4.8(a)). There is one case that does not finish without dynamic variable reordering, but finishes with dynamic variable reordering enabled (the *new* in Figure 4.8(a)). The remaining 45 cases are marked as the *rest* in Figure 4.8(a). These results show that given reasonably good initial orders (e.g., those provided by the original authors of these SMV models), dynamic variable reordering generally slows down the computation. This slowdown may be partially caused by the cache flushing in the dynamic variable reordering phase; i.e., given the importance of the computed cache, cache flushing can increase the number of repeated subproblems.

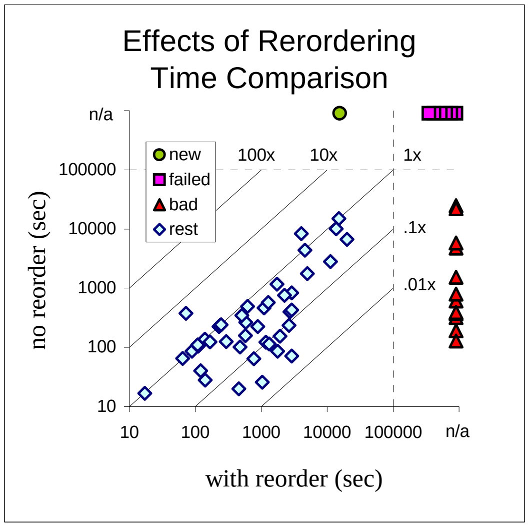
To evaluate the quality of the orders produced, we used the final orders produced by the dynamic variable reordering algorithms as new initial orders and reran the traces without dynamic variable reordering. Then we compared these results with the results obtained using the original initial order and also without dynamic variable reordering. This comparison is one good way of evaluating the quality of the variable reordering algorithms because in practice, good initial variable orders are often obtained by iteratively feeding back the resulting variable orders from the previous variable reordering runs.

Figure 4.8(b) plots the results for this experiment. The y-axis represents the cases using the original initial variable orders. The x-axis represents the cases where the final variable orders produced by the dynamic variable reordering algorithms are used as the initial variable orders. In this figure, the cases that finished using the original initial orders but failed using the new initial orders are marked as the *bad* and the remaining cases are marked as the *rest*. The results show that improvements can still be made from the original variable orders. A few cases even achieved speedups of over 10.

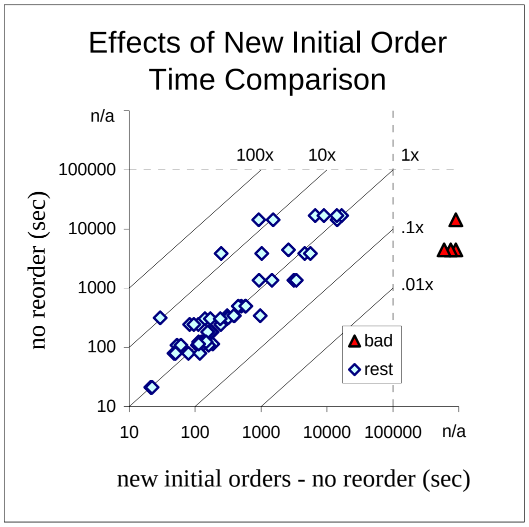
The remainder of this section presents results of a limited set of experiments for characterizing dynamic variable reordering. We first present the results on two heuristics for dynamic variable reordering. Then we present results on sensitivity of dynamic variable reordering to the initial variable orders. For these experiments, only the CUDD package is used. Note that the results in this section are very limited in scope and are far from being conclusive. Our intent is to suggest new research directions for dynamic variable reordering.

4.3.1 Present- and Next-State Variable Grouping

We set up an experiment to study the effects of *variable grouping*, where the grouped variables are always kept adjacent to each other.



(a)



(b)

Figure 4.8: Overall results for variable reordering. The *new* case represents the case that failed without the dynamic variable reordering, but can be completed with the dynamic variable reordering. The *failed* cases represent those that always exceed the resource limits. The *bad* cases represent those that are originally doable but failed with the new setup. The *rest* represent the remaining cases. (a) Timing comparison between with and without dynamic variable reordering. (b) Timing comparison between original initial variable orders and new initial variable orders. The new initial variable orders are obtained from the final variable orders produced by the dynamic variable reordering algorithms. For results in (b), dynamic variable reordering is disabled.

Hypothesis 6 *Pairwise grouping of present-state variables with their corresponding next-state variables is generally beneficial for dynamic variable reordering.*

Experiment: Measure the effects of this grouping on the number of subproblems (work), maximum number of live BDD nodes (space), and number of nodes sifted down during dynamic variable reordering (reorder cost).

Results: Figure 4.9 plots the effects of grouping on work (Figure 4.9(a)), space (Figure 4.9(b)), and reorder cost (Figure 4.9(c)). Note that the results for two traces are not available. One trace (*tomasulo*) exceeded the memory limit, while the other (*abp11*) is too small to trigger variable reordering.

These results show that pairwise grouping of the present- and the next-state variables is a good heuristic in general. However, there are a couple of exceptions. A better solution might be to use the grouping initially and relax the grouping criteria somewhat as the reordering process progresses.

4.3.2 Reordering the Transition Relations

Since the BDDs for the transition relations are used repeatedly in model-checking computations, we set up an experiment to study the effects of reordering the BDDs for the transition relations.

Hypothesis 7 *Finding a good variable order for the transition relation is an effective heuristic for improving overall performance.*

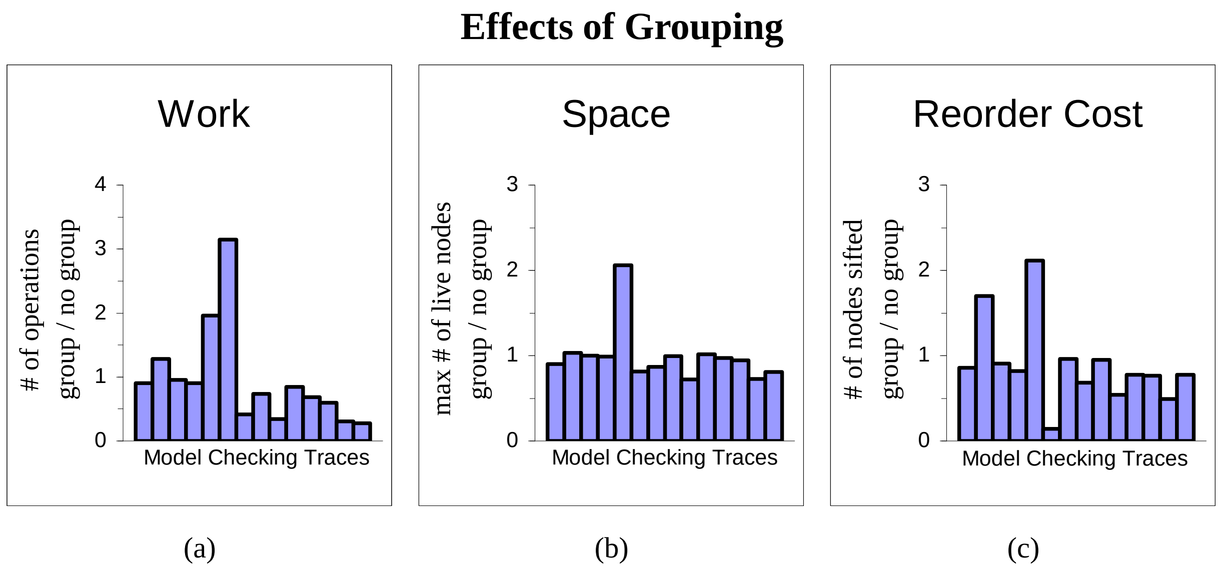


Figure 4.9: Effects of pairwise grouping of the present- and next-state variables on (a) the number of subproblems, (b) the number of maximum live BDD nodes, and (c) the amount of work in performing dynamic variable reordering.

Experiment: Reorder variables once, immediately after the BDDs for the transition relations are built, and measure the effect on the number of subproblems (work), maximum number of live BDD nodes (space), and number of nodes sifted down during dynamic variable reordering (reorder cost).

Results: Figure 4.10 plots the results of this experiment on work (Figure 4.10(a)), space (Figure 4.10(b)), and reorder cost (Figure 4.10(c)). The results are normalized against the results from automatic dynamic variable reordering for comparison purposes. Note that the results for two traces are not available. With automatic dynamic variable reordering, one trace (*tomasulo*) exceeded the memory limit, while the other (*abp11*) is too small to trigger variable reordering.

The results show that reordering once, immediately after the construction of transition relations' BDDs, generally works well in reducing the number of subproblems (Figure 4.10(a)). This heuristic's effects on the maximum number of live BDD nodes is mixed (Figure 4.10(b)). Figure 4.10(c) shows that this heuristic's reordering cost is generally much lower than automatic dynamic variable reordering. Overall, the variable-reordering frequency for automatic dynamic variable reordering is 5.75 times the variable-reordering frequency using this heuristic. These results are not strong enough to support our hypothesis as cache flushing may be the main factor for the effects on the number of subproblems. However, it does provide an indication that the automatic dynamic variable reordering algorithm may be invoking the variable reordering process too frequently.

4.3.3 Effects of Initial Variable Orders

In this section, we study the effects of initial variable orders on BDD construction with and without dynamic variable reordering. We generate a suite of initial variable orders by perturbing a set of good initial orders. In the following, we describe this experimental setup in detail and then present some hypotheses along with supporting evidence.

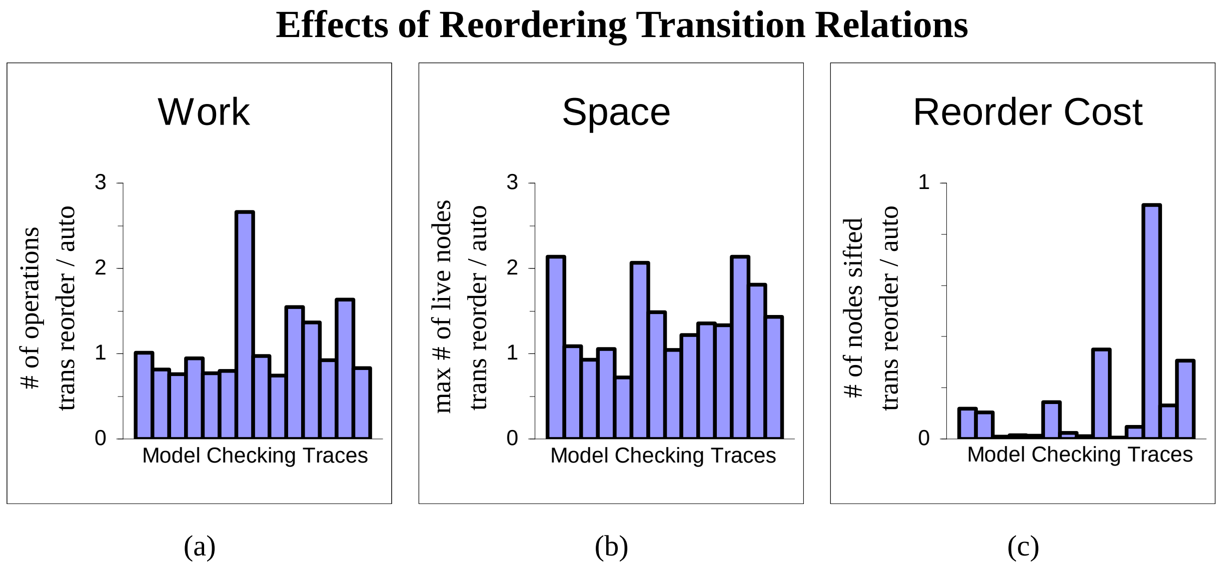


Figure 4.10: Effects of variable reordering the transition relations on (a) the number of subproblems, (b) the number of maximum live BDD nodes, and (c) the amount of work in performing variable reordering. For comparison purposes, all results are normalized against the results for automatic dynamic variable reordering.

Experimental Setup

The first step is the selection of good initial variable orders—one for each model-checking trace. The quality of an initial variable order is evaluated by the running time using this order without dynamic variable reordering.

Once the best initial variable order is selected, we perturb it based on two perturbation parameters: the probability (p), which is the probability that a variable will be moved, and the distance (d), which controls how far a variable may move. The perturbation algorithm used is shown in Figure 4.11. Initially, each variable is assigned a weight corresponding to its variable order (line 1). If this variable is chosen (with the probability of p) to be perturbed (by the distance parameter d), then we change its weight by δw, where δw is chosen randomly from the range $[-d, d]$ (lines 3–5). At the end, the perturbed variable order is determined by sorting the variables based on their final weights (line 6). This algorithm has the property that on average, p fraction of the BDD variables are perturbed and each variable's final variable order is at most $2d$ away from its initial order. Another property is that the perturbation pair $(p = 1, d = \infty)$ essentially produces a completely random variable order.

Since randomness is involved in the perturbation algorithm, to gain better statistical significance, we generate multiple initial variable orders for each pair of perturbation parameters (p, d). For each trace, if we study n_p different perturbation probabilities, n_d different perturbation distances, and k initial orders for each perturbation pair, we will generate a total of kn_pn_d different initial variable orders. For each initial variable order, we compare the results with and without dynamic variable reordering enabled. Thus, for each trace, there will be $2kn_pn_d$ runs. Due to lack of time and machine resources, we were able to complete this experiment only for the smallest trace—*abp11*.

The perturbed initial variable orders were generated from the best initial variable ordering we found for *abp11*. Using this order, the *abp11* trace can be executed (with the dynamic variable reordering disabled) using 12.69 seconds of CPU time and 127 MByte of memory on a 248 MHz UltraSparc II. This initial order and its results are used as the base case for this experiment. Using this base case, we set the time limit of

```
   perturb_order($v[n]$, $p$, $d$)
      /* perturb the variable order with probability $p$ and distance $d$.
         $v[\,]$ is an array of $n$ variables sorted based on decreasing
         variable order precedence. */
1     for $(0 \leq i < n)$ $w[i] \leftarrow i$ /* initialize weight */
2     for $(0 \leq i < n)$ /* for each variable, with probability $p$, perturb its weight. */
3        with probability $p$ do
4           $\delta w \leftarrow$ randomly choose an integer from the range $[-d, d]$
5           $w[i] \leftarrow w[i] + \delta w$
6     sort variables in array $v[\,]$ based on increasing weight $w[\,]$
7     return $v[\,]$
```

Figure 4.11: Variable-order perturbation algorithm.

each run to 1624.32 seconds (128 times the base case) and 500 MByte of memory.

For the perturbation parameters, we let p range from 0.1 to 1.0 with an increment of 0.1. Since *abp11* has 122 BDD variables, we let d range from 10 to 100 with an increment of 10 and added the $d = \infty$ case. These choices result in 110 perturbations pairs (with $n_p = 10$ and $n_d = 11$). For each perturbation pair, we generate 10 initial variable orders ($k = 10$). Thus, there are a total of 1100 initial variable orders and 2200 runs.

Results for *abp11*

Hypothesis 8 *Dynamic variable reordering improves the performance of model checking computations.*

Supporting Results: Figure 4.12 plots the number of cases that did not complete within various time limits for runs with and without dynamic variable reordering. For these runs, the memory limit is fixed at 500 MByte. The time limits in this plot are normalized to the base case of 12.69 seconds and are plotted in log scale.

The results clearly show that given enough time, the cases with dynamic variable reordering perform better. Overall, with a time limit of 128 times the base case, only 10.1% of cases with dynamic variable reordering exceeded the resource limits. In comparison, 67.6% of cases without dynamic variable reordering failed to complete.

Note that for the time limit of 2 times the base case (the $>$ 2x case in the chart), the results with dynamic variable reordering is worse. This reflects the fact that dynamic variable reordering can be expensive. As the time limit increases, the number of unfinished cases for with dynamic variable reordering drops more quickly until at about 32 times the base case. After this point, the number of unfinished cases for both with and without dynamic variable reordering appear to be decreasing at about the same rate.

Another interesting result is that none of the cases takes less time to complete than the base case of 12.69 seconds (i.e., the $>$ 1x results are both 1100). This result indicates that the initial variable order of our base case is indeed a very good variable order.

To better understand the impact of the perturbations on running time, we analyzed the distribution of these results (in Figure 4.12) across the perturbation space and formed the following hypothesis.

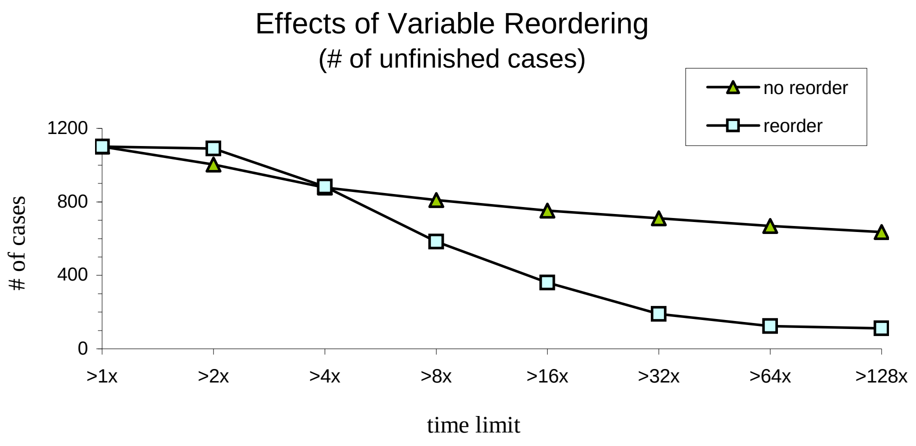


Figure 4.12: Effects of variable reordering on *abp11*. This chart plots the number of unfinished cases for various time limits. The time limits are normalized to the base case of 12.69 seconds. The memory limit is set at 500 MByte.

Hypothesis 9 *The dynamic variable reordering algorithm performs "unnecessary" work when it is already dealing with reasonably good variable orders. Overall, given enough time, dynamic variable reordering is effective in recovering from poor initial variable order.*

Supporting Results: Figure 4.13(a) shows the results with a time limit of 4 times the base case of 12.69 seconds. These plots show that when the perturbation level is low ($p = 0.1$ or $d = 10$), we are better off without dynamic variable reordering. However, for higher levels of perturbations, the cases with dynamic variable reordering usually does a little better.

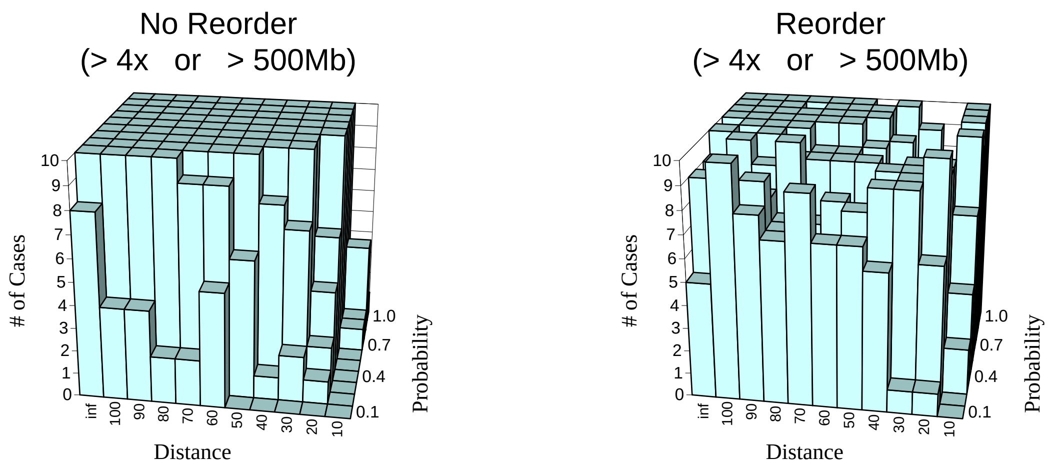
Figures 4.13(b) and 4.13(c) show the results with time limits of 32 and 128 times, respectively, the base case. Note that because 128 times is the maximum time limit we studied, Figure 4.13(c) also represents the distribution of the cases that did not complete at all for this study. These results clearly show that given enough time, the cases with dynamic variable reordering perform much better.

Hypothesis 10 *The quality of initial variable order affects the space and time requirements, with or without dynamic variable reordering.*

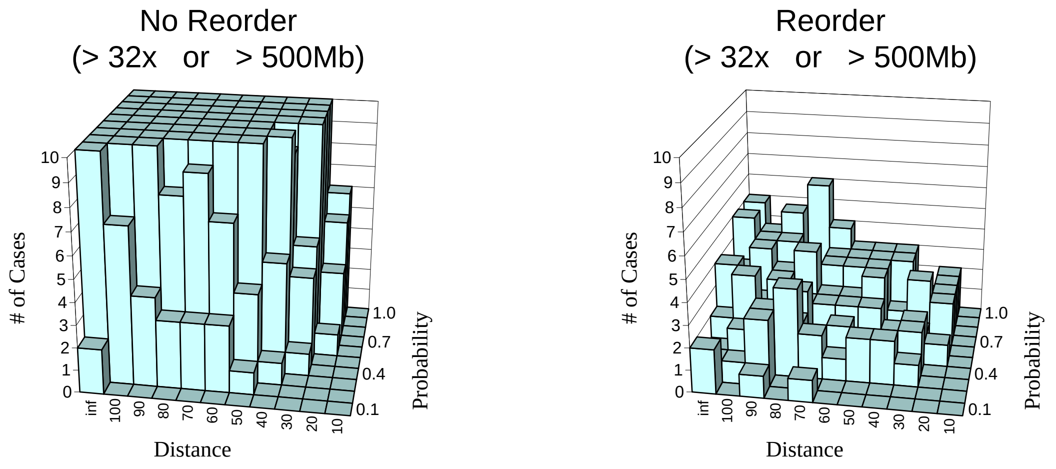
Supporting Results: Figure 4.14 classifies the unfinished cases into memory-out (Figure 4.14(a)) or timed-out (Figure 4.14(b)). For clarity, we repeated the plots for the total number of unfinished cases (memory-out plus timed-out results) in Figure 4.14(c). It is important to note that because the BDD packages used in this study still do not adapt very well upon exceeding memory limits, memory-out cases should be interpreted as indications of high memory pressure instead of that these cases inherently do not fit within the memory limit.

The results show that levels of perturbation directly influence the time and memory requirement. With a very high level of perturbation, most of the unfinished cases are due to exceeding the memory limit of 500 MByte (the upper-left triangular regions in Figure 4.14(a)). For a moderate level of perturbation, most of the unfinished cases are due to the time limit (the diagonal bands from the lower-left to the upper-right in Figure 4.14(b)).

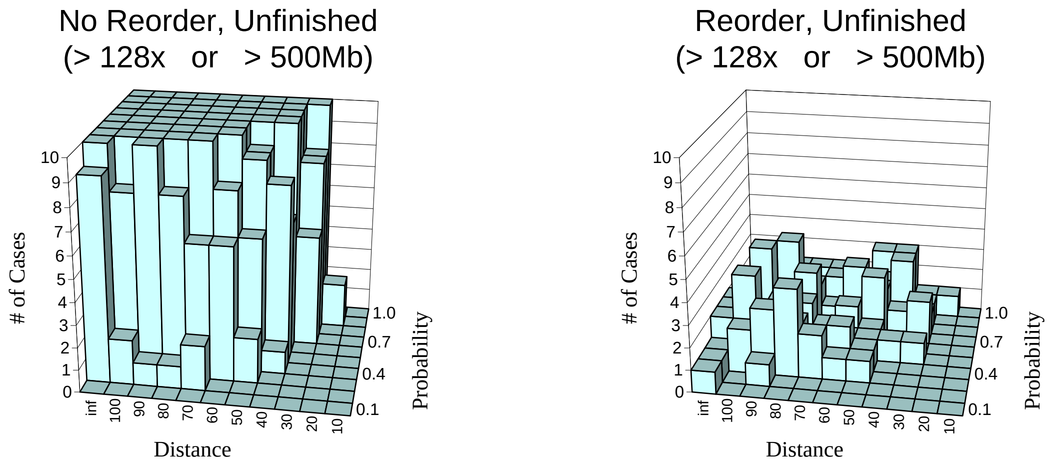
Note that the results in Figure 4.14 are not very monotonic; i.e., the results are not necessarily worse with a larger degree of perturbation. This leads to the next hypothesis.



(a)

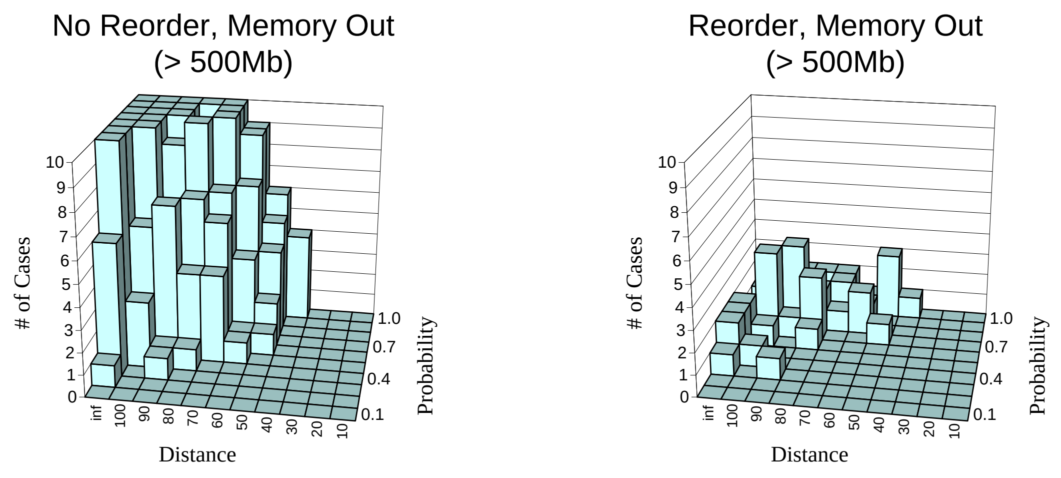


(b)

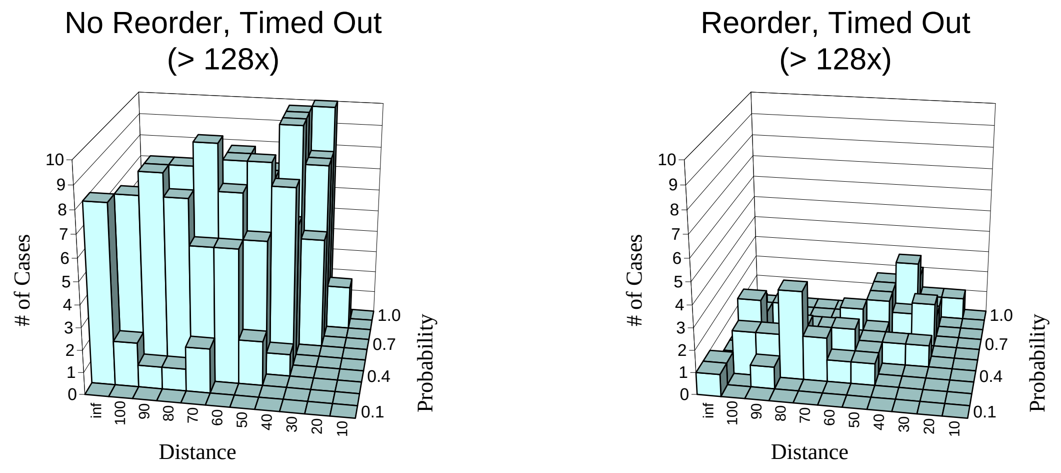


(c)

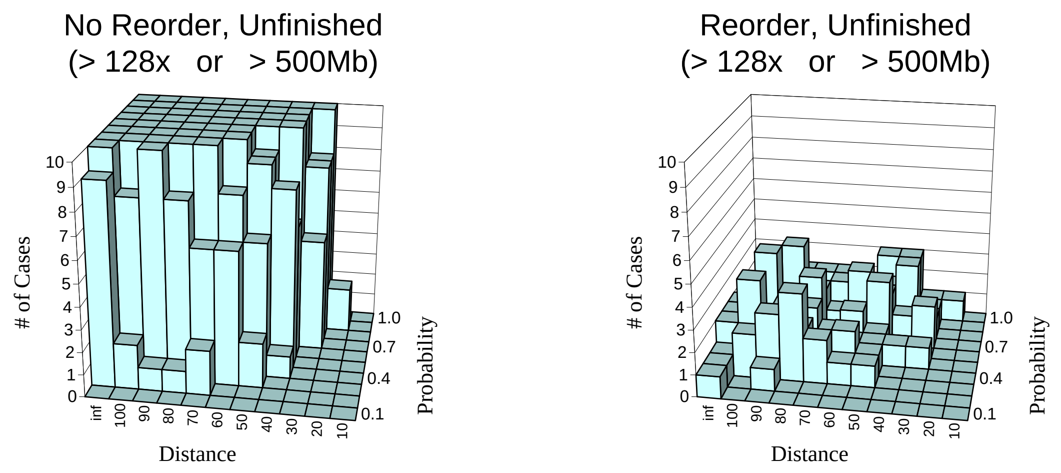
Figure 4.13: Histograms on the number of cases that cannot be finished within the specified resource limits. For all cases, the memory limit is set at 500 MByte. The time limit varies from (a) 4 times, (b) 32 times, to (c) 128 times the base case of 12.69 seconds.



(a)



(b)



(c)

Figure 4.14: Breakdown on the cases that cannot be finished. (a) memory-out cases, (b) timed-out cases, (c) total number of unfinished cases.

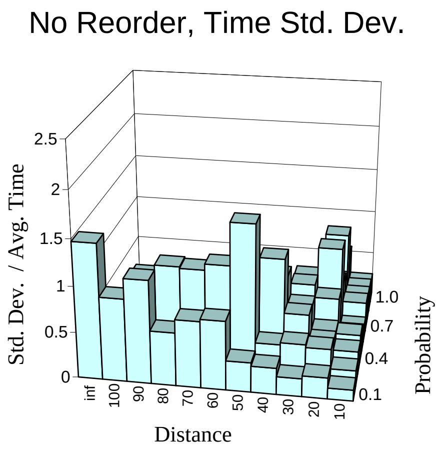
Hypothesis 11 *The effects of the dynamic variable reordering algorithm and the initial variable orders are very chaotic.*

Supporting Results: Figure 4.15 plots the standard deviation of running time normalized against average running time. For the cases that cannot complete within the resource limits, they are included as if they use exactly the time limit. Note that as an artifact of this calculation, when all 10 variants of a perturbation pair exceed the resource limits, the standard deviation is 0. In particular, without variable reordering, none of the cases can be completed in the highly perturbed region (upper-left triangular region in Fig 4.14(c)) and thus these results are all shown as 0 in the chart.

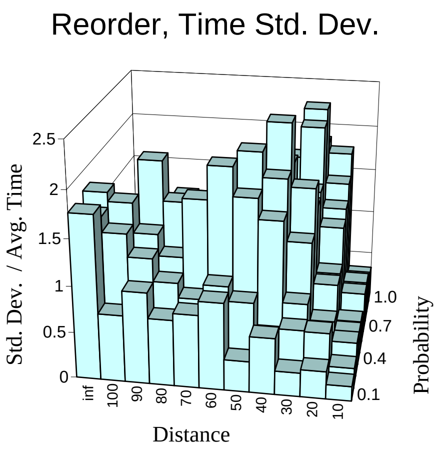
The results show that the standard deviations are generally greater than the average time (i.e., with the normalized result of > 1). This finding partially confirms our hypothesis. It also indicates that 10 initial variable orders per perturbation pair (p, d) is probably too small for some perturbation pairs.

The results also show that with very low level of perturbation (lower-right triangular region), the normalized standard deviation is generally smaller. This gives an indication that higher perturbation level may result in more unpredictable performance behavior.

Furthermore, the normalized standard deviation for without dynamic variable reordering is generally smaller than the same measure for with dynamic variable reordering. This result provides an indication that dynamic variable reordering may also have very unpredictable effects.



(a)



(b)

Figure 4.15: Standard deviation of the running time for *abp11* with perturbed initial variable orders (a) without dynamic variable reordering, and (b) with dynamic variable reordering. Each results are normalized to the average running time.

4.4 General Results

This section presents results which may be generally helpful in studying or improving BDD packages.

Hash Function

Hashing is a vital part of BDD construction since both the uniqueness of BDD nodes and the cache accesses are based on hashing. Currently, we have not found any theoretically good hash functions for handling multiple hash keys. In this study, we have empirically found that the hash function used by the TiGeR BDD package worked well in distributing the nodes. This hash function is of the form

$$H(k_1, k_2) = ((k_1 p_1 + k_2) p_2)/2^{w-n}$$

where k's are the hash keys, p's are sufficiently large primes, w is the number of bits in an integer, and 2^n is the size of the hash table. Note that division by 2^{w-n} is used to extract the n most significant bits and is implemented by right shifting $(w - n)$ bits.

The basic idea is to distribute and combine the bits in the hash keys to the higher order bits by using integer multiplications, and then to extract the result from the high order bits. The power-of-2 hash table size is used to avoid the more expensive *modulus* operation. Some small speedups have been observed using this hash function. One pitfall is that for backward compatibility reasons, some compilers might generate a function call to compute integer multiplication, which can cause significant performance degradation (up to a factor of 2). In these cases, architecture-specific compiler flags can be used to ensure the integer-multiplier hardware is used instead.

Caching Strategy

Given the importance of cache, a natural question is: *Can we cache more intelligently?* One heuristic, used in CUDD, is that the cache is accessed only if at least one of the arguments has a reference count greater than 1. This technique is based on the fact that if all arguments have reference counts of 1, then this subproblem is not likely to be repeated within the current top-level operation. In fact, if a complete cache is used, this subproblem will not be repeated within the same top-level operation. Using this technique, CUDD is able to reduce the number of cache lookups by up to half, with a total time reduction of up to 40%.

Relational Product Algorithm

The relational product algorithm in Figure 2.5 can be further improved. The new optimizations are based on the following derivations. Let r_0 be the result of the 0-cofactors (line 4 in Figure 2.5), $\vec{v}$ be the set of variables to be quantified, and h be any Boolean function, then

$$r_0 \vee (\exists \vec{v}. r_0 \wedge h) = r_0 \vee (r_0 \wedge \exists \vec{v}. h) = r_0$$

and

$$r_0 \vee (\exists \vec{v}. (\neg\, r_0) \wedge h) = r_0 \vee ((\neg\, r_0) \wedge \exists \vec{v}. h) = r_0 \vee \exists \vec{v}. h$$

The validity comes from the fact that r_0 does not depend on the variables in $\vec{v}$. Based on these equations, we can add the following optimizations (between line 7 and line 8 in Figure 2.5) to the relational product algorithm:

```
7.1    else if ($r_0 == f|_{\tau \leftarrow 1}$) or ($r_0 == g|_{\tau \leftarrow 1}$)
7.2        $r \leftarrow r_0$
7.3    else if ($r_0 == \neg f|_{\tau \leftarrow 1}$)
7.4        $r \leftarrow r_0 \vee (\exists \vec{v}. g|_{\tau \leftarrow 1})$
7.5    else if ($r_0 == \neg g|_{\tau \leftarrow 1}$)
7.6        $r \leftarrow r_0 \vee (\exists \vec{v}. f|_{\tau \leftarrow 1})$
```

In general, these optimizations only slightly reduce the number of subproblems, with the exception of the *futurebus* trace, where the number of subproblems is reduced by over 20%.

BDD Package Comparisons

In comparing BDD packages, one fairness question is often raised: *Is it fair to compare the performance of a bare-bones experimental BDD package with a more complete public domain BDD package?* This question arises particularly when one package supports dynamic variable reordering, while the other does not. This is an issue because supporting dynamic variable reordering requires additional data structures and indirection overheads to the computation for BDD construction. To partially answer this question, we studied a package with and without its support for variable reordering in place. Our preliminary results show that the additional overhead to support dynamic variable reordering has no measurable performance impact. This may be due to the fact that BDD computation is so memory intensive, a couple additional non-memory intensive operations can be scheduled either by the hardware or the compiler without any measurable performance penalty.

Cache Hit Rate

The computed-cache hit rate is not a reliable measure of overall performance. In Section 3.1.1, we have shown that when the cache hit rate is less than 49%, a cache miss can actually result in a higher hit rate. This is because a cache miss generates more subproblems and these subproblems' results could have already been computed and are still in cache.

Platform Independent Metrics

Throughout this study, we have found several useful machine-independent metrics for characterizing the BDD computations. These metrics are:

- the *number of subproblems* as a measure for work,
- the *maximum number of live nodes* as a measure for the lower bound on memory requirement,
- the *number of subproblems processed for each breadth-first queue visit* to reflect the possibility of exploiting memory locality using the breadth-first traversal, and
- the *number of nodes swapped with their children during dynamic variable reordering* as a measure of the amount of work performed in dynamic variable reordering.

These metrics are incorporated in the BDD evaluation methodology as described in Chapter 3.

4.5 Issues and Open Questions

Cache Size Management

In this study, we have found that the size of the compute cache can have a significant impact on model-checking computations. Given that BDD computations are very memory intensive, there is an inherent conflict between using a larger cache for better performance and using a smaller cache to conserve memory usage. For BDD packages that maintain multiple compute caches, there are additional conflicts as these caches will compete with each other for the memory resources. As the problem sizes get larger, finding a good dynamic cache management algorithm will become more and more important for building an efficient BDD package.

Garbage Collection Triggering Algorithm

Another dynamic memory management issue is the frequency of garbage collection. The results in Figure 4.5(b) clearly suggest that delaying garbage collection can be very beneficial. Again, this is a space and time tradeoff issue. One possibility is to invoke garbage collection when the percentage of unreachable nodes is high and the rebirth rate is low. Note that for BDD packages that do not maintain reference counts, the rebirth rate is not readily available and thus a different strategy is needed.

Resource Awareness

Given the importance of space and time tradeoff, a commercial strength BDD package not only needs to know when to gobble up the memory to reduce computation time, it should also be able to free up space under resource contention. This contention could come from different parts of the same tool chain or from a completely different application. One way to deal with this issue is for BDD packages to become more aware of the environment, in particular, the available physical memory, various memory limits, and the page fault rate. This information is readily available to the users of modern operating systems. Several of the BDD packages used in this study already have some limited form of resource awareness. However, this problem is still not well understood and probably cannot be easily studied using the trace-driven framework.

Cross Top-Level Sharing

For the model-checking traces, why are there so many subproblems repeated across the top-level operations? This question is particularly important because this may be the main reason for the necessity of both much bigger cache and infrequent garbage collection. We have two conjectures. First, there is quite a bit of symmetry in some of these SMV models. These inherent symmetries are *somehow* captured by the BDD representation. If so, it might be more effective to use higher level algorithms to exploit the symmetries in the models. The other conjecture is that the same BDDs for the transition relations are used repeatedly throughout model checking in the fixed-point computations. This repeated use of the same set of BDDs increases the likelihood of the same subproblems being repeated across top-level operations. At this point, we do not know how to validate these conjectures. To better understand this property, one starting point would be to identify how far apart are these cross top-level repeated subproblems; i.e., is it within one state transition, within one fixed-point computation, within one temporal logic operator, or across different temporal logic operators?

Breadth-First's Memory Locality

In this study, we have found no evidence that breadth-first based techniques have any advantage when the computation fits in the main memory. An interesting question would be: *As the BDD graph sizes get much larger, is there going to be a crossover point where the breadth-first packages will be significantly better?* If so, another issue would be finding a good depth-first and breadth-first hybrid to get the best of both worlds.

Inconsistent Cross Platform Results

Inconsistency in timing results across machines is yet another unresolved issue in this study. More specifically, for some BDD packages, the CPU-time results on a UltraSparc II machine are up to twice as long as the corresponding results on a PentiumPro, while for other BDD packages, the differences are not so significant. Similar inconsistencies are also observed in the Sentovich study [73]. A related performance discrepancy is that for the depth-first based packages, the garbage collection cost for UltraSparc II is generally twice as high as that of PentiumPro. However, for the breadth-first based packages, the garbage collection performances between these two machines are much closer. In particular, for one breadth-first based package, the ratio is very close to 1. This discrepancy may be a reflection of the memory locality of these BDD packages. To test this conjecture, we have performed a set of simple experiments using synthetic workloads. Unfortunately, the results did not confirm this hypothesis. However, these results do indicate that our PentiumPro machine appears to have a better memory hierarchy than our UltraSparc II machine. A better understanding of this issue can probably shed some light on how to improve memory locality for BDD computations.

Pointer- vs. Index-Based References

Another issue is that within the next ten years, machines with memory sizes greater than 4 GByte are going to become common. Thus the size of a pointer (i.e., memory address) will increase from 32 to 64 bits. Since most BDD packages today are pointer-based, the memory usage will double on 64-bit machines. One way to reduce this extra memory overhead is to use integer indices instead of pointers to reference BDDs as in the case of the ABCD package. One possible drawback of an index-based technique is that an extra level of indirection is introduced for each reference. However, since ABCD's results are generally among the best in this study, this provides a positive indication that the index-based approach may be a feasible solution to this impending memory overhead problem.

Computed Cache Flushing in Dynamic Variable Reordering

In Section 4.3, we showed that dynamic variable reordering can generally slow down the entire computation when given a reasonably good initial variable order. Since the computed cache is typically flushed when dynamic variable reordering takes place, it would be interesting to study what percentage of the slowdown is caused by an increase in the amount of work (number of subproblems) due to cache flushing. If this percentage is high, then another interesting issue would be in finding a good way to incorporate the cache performance as a parameter for controlling dynamic variable reordering frequency.

4.6 Related Work

In [73], Sentovich presented a BDD study comparing the performance of several BDD packages. Her study covered building output BDDs for combinational circuits, computing reachability of sequential circuits, and variable reordering.

In [57], Manne *et al.* performed a BDD study examining the memory locality issues for several BDD packages. This work compares the hardware cache miss rates, TLB miss rates, and page fault rates in building the output BDDs for combinational circuits.

In contrast to the Sentovich study, our study focuses in characterizing the BDD computations instead of doing a performance comparison of BDD packages. In contrast to the Manne study, our work uses platform independent metrics for performance evaluation instead of hardware specific metrics. Both types of metrics are equally valid and complementary. Our study also differs from these two prior studies in that our performance evaluation is based on the execution of a model checker instead of benchmark circuits.

4.7 Summary

In this BDD performance study, we have significantly improved the overall performance of various BDD packages and have also learned a number interesting characteristics of BDD computations in the context of model checking. In particular, we have confirmed that model checking and combinational circuit computations have fundamentally different performance characteristics. These differences include the effects of the cache size, the garbage collection frequency, the complement edge representation, and the memory locality for the breadth-first BDD packages. For dynamic variable reordering, we have introduced some new methodologies for studying the effects of variable reordering algorithms and initial variable orders. Furthermore, from these experiments, we have uncovered a number of open problems and future research directions.

As this study is fairly limited in scope, especially for the dynamic variable reordering phase, further validations of the hypotheses are still necessary. It would be especially interesting to repeat the same experiments on execution traces from other BDD-based tools.

Even though the study is limited in scope, the results of this study have clearly demonstrated the power of our evaluation methodology and the importance of our collaborative efforts in advancing the knowledge of BDD computations.

Chapter 5

BDD Optimizations for Conjunctive Partitioning

Conjunctive partitioning is used to represent the transition relation when the monolithic BDD representation is too big to build. The ordering and merging algorithms of conjunctive partitions play an important role in avoiding the BDD explosion of intermediate results. In commonly used heuristics [40, 69], the ordering step greedily schedules these partitions to quantify out more variables as soon as possible, while introducing fewer new variables. The merging step then tentatively merges each of the ordered partitions with its neighbors to reduce the number of conjuncts, which in turn reduces the number intermediate results during image and pre-image computations for the symbolic state traversal. Each merged result is kept only if the result graph size is less than a pre-determined limit, which we will refer to as the *partition-size limit.* Please refer to Section 2.5.3 for details of the conjunctive partitioning and the early quantification optimization.

One problem we have empirically observed is that the performance impact of conjunctive partitioning algorithms tends to be unstable; i.e., small perturbations to adjustable parameters, such as the partition-size limit, in the algorithm can result in big performance differences. In particular, different model-checking computations often require very different partition-size limits, where a wrong choice results in either complete failure or significant performance degradation in both space and time usage.

In this chapter, we describe new heuristics that utilize information encoded in BDDs to help stabilize conjunctive partitioning. Our improvements are based on the observation that a potential problem with the previous approaches is that the ordering phase does not account for its impact on the subsequent merging phase. Our approach for stabilizing conjunctive-partitioning algorithm is to use BDD characteristics, such as the support variables and graph sizes, to define the strength of *interaction* between each pair of partitions and merge strongly interacting partitions as a preprocessing step, and to apply a slightly modified version of the conventional greedy algorithm for ordering and merging the conjunctive partitions. In Section 5.1, we describe this work. Then in Section 5.2, we present experimental results on the impact of our improvements on performance stability.

5.1 Our Conjunctive-Partitioning Algorithm

The main idea is to group *strongly interacting* partitions together first. In our algorithm, the underlying hypothesis is that two partitions are *strongly interacting* if

1. they share a large portion of *support variables* (variables whose values have direct influence on the function), and

2. merging these two partitions results in a BDD that is not too large.

Another key idea of our algorithm is the use of *tentative BDD operations* to merge partitions. A *tentative BDD operation* is a BDD operation that is restricted to some resource limitations. When a tentative BDD operation exceeds the specified resource limit, the operation aborts as soon as possible and returns a token to indicate the computation has failed. Our use of tentative BDD operations is inspired by Chen and Bryant's *short-circuiting* technique in the verification of floating point adders [20], where the tentative operations are used to automatically remove the unnecessary part of the circuit in conditional symbolic simulation. In our approach, we use tentative BDD operations to bound the cost of applying potentially expensive optimizations.

In the implementation of tentative BDD operations, we use the number of sub-operations generated as the time limit, and use both the resulting graph size and the number of newly generated BDD nodes as the space limit. Note that this definition of the space limit bounds both the additional memory usage and the result graph size. To implement the time limit, we use a counter to track the number of sub-operations generated so far and abort when the count exceeds the specified limit. For the space limit, we keep another counter to track the number of new BDD nodes generated and abort when the count exceeds the specified space limit. To properly implement the space limit as defined, when the computation completes successfully, we need to check the size of the result graph since this graph may contain nodes that existed previously and thus are not accounted for by the counter for the number of new nodes. Note that whether or not a tentative operation will be able to finish successfully is somewhat unpredictable because it depends on what sub-operations are cached and what BDD nodes already exist in the system.

Based on the above concept of *strongly interacting* partitions and tentative operations, our conjunctive partitioning algorithm has three phases: pre-merging, ordering, and post-merging. Note that our algorithm is based on the Ranjan et al.'s work [69]. The main difference is that we have added the pre-merging phase and that the ordering phase is based on different heuristics (as described below). The phases in our algorithm are:

Pre-Merging

In this phase, strongly interacting partitions are merged. The strength of interaction is defined by the percentage of shared support variables and the growth in the graph size of the merged result. Our pre-merging algorithm is an iterative algorithm with each subsequent iteration relaxing the minimum interaction-strength requirement for merging partitions. The purpose of this iterative approach is to merge more strongly interacting partitions first. An alternative would be to use a priority queue that prioritizes the merging based on the interaction strength.

The core pre-merging algorithm that we iterate over with different interaction strength requirements is shown in Figure 5.1. The interaction strength requirements are characterized by two input parameters: *minShareVarsRatio*, which specifies the minimum threshold for the number of support variables shared by two partitions, and *growthLimit*, which specifies the maximum growth in the graph size of the merged result in comparison to the graph sizes of the two partitions. This algorithm also takes two additional parameters *sizeLimit* and *timeLimit* to limit the space and time usage of each pair of partitions we tried to merge.

The core pre-merging algorithm works as follows. For each pair of partitions (line 1), we determine if this pair satisfies the *minShareVarsRatio* threshold (lines 2–5). If so, we tentatively tried to merge these two partitions (line 6). If the merged result can be successfully computed, then we check to see if the result BDD's graph size satisfies the *growthLimit* constraint on the maximum growth limit (lines 9–11). If so, then this pair of partitions is replaced by the merged result (line 12) and this process repeats for the next pair of partitions.

```
  premerge_cp_tr($T$, minShareVarsRatio, growthLimit, sizeLimit, timeLimit)
    /* pre-merging for conjunctive partitioning of transition relation.
     * $T$: the set of partitions as candidates to be merged.
     * minShareVarsRatio: minimum variable-sharing threshold before two partitions are to be merged.
     * growthLimit: limit on the growth of the merged graph size.
     * sizeLimit: limit on the merged graph size.
     * timeLimit: time limit set for each merge operation.
     */
1   for each pair $t_i$ and $t_j \in T$  /* try to merge $t_i$ with $t_j$ */
2       numShareVars $\leftarrow ||\text{support}(t_i) \cap \text{support}(t_j)||$
3       minNumVars $\leftarrow \min(||\text{support}(t_i)||, ||\text{support}(t_j)||)$
4       if (numShareVars / minNumVars) $<$ minShareVarsRatio  /* not enough shared variables */
5           go to beginning of the loop (line 1) and try next pair
6       (merged, isSuccessful) $\leftarrow$ tentative_and($t_i$, $t_j$, sizeLimit, timeLimit)
7       if ($\neg$ isSuccessful) /* exceeded limits */
8           go to beginning of the loop (line 1) and try next pair
9       growth $\leftarrow$ graph_size(merged) / (graph_size($t_i$) + graph_size($t_j$))
10      if growth $>$ growthLimit  /* exceeded growth limit */
11          go to beginning of the loop (line 1) and try next pair
        /* at this point, keep the merged result */
12      $T \leftarrow (T \setminus \{t_i, t_j\}) \cup \{merged\}$
13  return $T$
```

Figure 5.1: The core pre-merging algorithm for merging *strongly interacting* components for conjunctive partitioning of transition relation. In this subroutine, the *support()* function returns the set of support variables, the *tentative_and()* function is the tentative operation of Boolean AND, and the *graph_size()* function returns the size of the BDD.

Ordering

In this phase, a linear order of the partitions is computed based on the number of early quantification variables, the total number of new variables introduced, the graph size of the partition, and the growth ratio if a partition is merged with its predecessor in the linear order. Based on these BDD characteristics, the partitions are ordered greedily.

Our ordering algorithm is shown in Figure 5.2. This is a greedy algorithm that selects the partition that *scores* the highest to be the next partition in the order. The scoring function is shown in Figure 5.3.

```
   order_cp_tr($T$, *quanVars*)
      /* ordering partitions of transition relation.
       * $T$: the set of partitions to be ordered.
       * *quanVars*: the set of quantifying variables. used to determine benefits of early quantification.
       */
1     $L \leftarrow$ empty list  /* initialize the result ordered list */
2     while $T$ is not empty  /* select the next partition */
3         for each $t \in T$
4             *score*[$t$] $\leftarrow$ cp_compute_score($t$, $L$, $T \setminus \{t\}$, $L$, *quanVars*)
5         $t_{\max} \leftarrow$ partition $t \in T$ with the highest *score*[$t$].
6         $T \leftarrow T \setminus \{t_{\max}\}$
7         $L \leftarrow$ append$(L, t_{\max})$
8     return $L$
```

Figure 5.2: Algorithm for ordering partitions of transition relation. The ordering is perform by greedily ordering the "best" partitions based on a scoring function.

The scoring function is based on the following six BDD characteristics of a partition:

1. the number of support variables in the set of variables to be quantified (line 1),
2. the number of variables that can be quantified out if the partition is added next to the ordered list (line 2),
3. the number of variables that are not in the set of variables to be quantified (line 3),
4. the number of new variables introduced if the partition is added next (line 4),
5. the BDD graph size of the partition (line5), and
6. the growth in the BDD graph size if this partition were to be merged with the last partition in the currently ordered list L (lines 6–7).

These characteristics are chosen because the cost of a relational-product operation depends strongly on the number of variables, the number of variables to be quantified, and the graph sizes. We added the growth-factor parameter (line 6) to account for the next phase (post-merging) that merges neighboring partitions in the linear order together. The value of these characteristics are then weighted with constant factors (W's) to compute the score of this partition (line 8).

The ordering algorithm is a variant of the conjunctive partitioning by Ranjan et al. [69]. The only differences are in the heuristic scoring function "cp_compute_score". Other than the addition of the growth-factor parameter (lines 6–7), another difference is that we use absolute numbers, such as the number of variables to be quantified, instead of relative numbers, such as the percentage of the number of variables to

```
cp_compute_score($t$, $L$, $R$, quanVars)
    /* compute the score if $t$ were the next partition to be add to list $L$.
     * $t$: the partition to be scored.
     * $L$: list of partitions ordered so far.
     * $R$: set of partitions not yet ordered, excluding $t$.
     * quanVars: the set of quantifying variables. used to determine benefits of early quantification.
     */
    /* compute the number of $t$'s support variables that are in quanVars */
1   nInQuanVars $\leftarrow ||\text{support}(t) \cap quanVars||$
    /* compute the number of $t$'s support variables that are in quanVars but not in any of partitions in $R$ */
2   nEarlyQuanVars $\leftarrow ||(\text{support}(t) \cap quanVars) \setminus \bigcup_{r \in R} \text{support}(r)||$
    /* compute the number of $t$'s support variable that are not in quanVars */
3   nNonQuanVars $\leftarrow ||\text{support(t)} \setminus quanVars||$
    /* compute the number of new variables that $t$ would introduce */
4   nNewNonQuanVars $\leftarrow ||(\text{support}(t) \setminus quanVars) \setminus \bigcup_{l \in L} \text{support}(l)||$
    /* compute the graph size $t$ */
5   nNodes $\leftarrow$ graph_size($t$)
    /* compute the growth factor if $t$ is merged with the last partition in $L$ */
6   $t_{\text{last}} \leftarrow$ last partition in list $L$
7   growth $\leftarrow$ graph_size($t \wedge t_{\text{last}}$) / (graph_size($t$) + graph_size($t_{\text{last}}$))
    /* compute the overall score of adding $t$ next. $W$'s are pre-selected constant weights. */
8   score $\leftarrow W_1 \cdot nInQuanVars + W_2 \cdot nEarlyQuanVars + W_3 \cdot nNonQuanVars +$
        $W_4 \cdot nNewNonQuanVars + W_5 \cdot nNodes + W_6 \cdot growth$
9   return score
```

Figure 5.3: Compute the scoring function for choosing the next partition to be added. In this subroutine, the *support()* function returns the set of support variables, and the *graph_size()* function returns the size of the BDD.

be quantified. We believe that the cost of computing the relational-product operation is more directly related to absolute numbers instead of relative numbers,

Post-Merging

In this phase, the ordered partitions are merged with their neighbors in linear order. The purpose of this step is to reduce the number of partitions and thus reduce the number of intermediate results in computing the image and the pre-image computation. Our post-merging algorithm is an iterative algorithm with the merging constraints (graph size and graph growth) successively relaxed from one iteration to the next. Similar to the pre-merging algorithm, the purpose of this iterative approach is to merge the strongly interacting partitions first. However, since the partitions are already linearly ordered, the merge is only performed between neighboring partitions in the order.

The core post-merging algorithm (Figure 5.4) works as follows. Given a linearly ordered list of partitions L, remove the first partition (t) in the list (lines 3–4) and try to merge with the next partition (u) in the list (lines 6–7). If the merge does not exceed the size limit, then check that the graph size of the merged result also satisfies the growth-factor constraint (lines 10–12). If the graph size satisfies the growth-factor constraint, then continue to merge with the next partition in the list (lines 13–14). If either the graph-size constraint or the growth-factor constraint fails (lines 8–9, 11–12), then keep the merged result (line 15) and repeat this entire process until the list L is empty (line 2).

```
   postmerge_cp_tr($L$, growthLimit, sizeLimit)
      /* post-merging step to merge partitions adjacent to each other in $L$ with
       * the following constraints:
       *  growthLimit: limit on the growth of the merged graph size.
       *  sizeLimit: limit on the merged graph size.
       */
1     $P \leftarrow$ empty list  /* initialize the result ordered list */
2     while $L$ is not empty
3         $t \leftarrow$ first element in $L$
4         remove $t$ from $L$
5         while $L$ is not empty  /* merge as many neighbors to $t$ as possible */
6             $u \leftarrow$ first element in $L$
7             (merged, isSuccessful) $\leftarrow$ tentative_and($t$, $u$, sizeLimit)
8             if ($\neg$ isSuccessful) /* exceeded size limit */
9                 goto line 15  /* done with this merging. */
10            growth $\leftarrow$ graph_size(merged) / (graph_size($t$) + graph_size($u$))
11            if growth $>$ growthLimit  /* exceeded growth limit */
12                goto line 15  /* done with this merging */
              /* merge is successful */
13            remove $u$ from $L$
14            $t \leftarrow$ merged
15        $P \leftarrow$ append$(P, t)$
16    return $P$
```

Figure 5.4: The core post-merging algorithm for conjunctive partitioning. Merge adjacent partitions with the specified size and growth constraints. In this subroutine, the *tentative_and()* function is the tentative operation of Boolean AND, and the *graph_size()* function returns the size of the BDD.

5.2 Evaluation

5.2.1 Experimental Setup

The benchmark suite used is a collection of 58 SMV models gathered from a wide variety of sources, including the 16 models used in the BDD performance study described in Chapter 4. In 10 models, there is only one transition relation for the whole system (instead of one transition relation for each state variable) and thus conjunctive partitioning has no effect on these models. Out of the remaining 48 models, 21 very small models ($<$ 10 seconds) are eliminated. There is one model that is too large for SMV to verify and may require other classes of optimizations, such as cone-of-influence analysis, to reduce the model first. We also excluded this model from the experiments. In the following, the experimental results reported are based on the remaining 27 models.

The results reported in this section are labeled with the following keys to indicate which optimizations are enabled:

BASE: the base case using the algorithm by Ranjan et al. [69]; i.e., their ordering heuristic plus the post-merging phase.

PreMerge: the base case with the addition of the pre-merging preprocessing step.

ORDER: the base case with the ordering phase modified to use our scoring heuristics instead.

OURS: PreMerge plus **ORDER** case; i.e., with all the techniques described in this chapter turned on.

The partition-size limits used in this experiments are: 1, 1000, 10,000, 100,000, 1 million, and 10 million BDD nodes. The partition-size limit of 1 means that only the ordering phase is active because merging can only occur when the BDD size is less than the partition-size limit. Note that the partition-size limit of 10 million is quite large (i.e., each BDD can use up to 160 MByte of memory). The usual practice is to use a limit of no greater than 10,000 nodes. We studied the results up to the partition-size limit of 10 million node because that's when our algorithm started to behave badly.

We performed the evaluation using the Symbolic Model Verifier (SMV) model checker [58] from Carnegie Mellon University. For the ordering phase using our scoring function, the values of the constant weights (W_i's) are as follows:

$$\begin{array}{rclcrcl} W_1 & = & 0, & \quad\quad & W_4 & = & 0, \\ W_2 & = & 8, & & W_5 & = & 0, \\ W_3 & = & -4, & & W_6 & = & -1. \end{array}$$

These values are *not* determined very systematically. They were chosen based on experimental results on a set of medium-size models (roughly 30–100 seconds running time) in the benchmark suite. We adjusted the weights until these examples' running time for partition-size limit of 5000 nodes is at most a factor of 3 slower than without conjunctive partitioning (i.e., using a monolithic BDD for the overall transition relation). Another general criteria we used is to have as few non-zero weights as possible. Using these criteria, only three out of the six parameters are chosen: (1) W_2, the weight for the number of early quantification variables, (2) W_3, the weight for the number of non-quantifying variables, and (3) W_6, the growth factor if merged with the last partition in currently ordered list.

The evaluation was performed on a 200MHz Pentium-Pro with 1 GB of memory running Linux. Each run was limited to 3 hours of CPU time and 900 MByte of memory. All the timing results shown later in this section were normalized against the best running time across all 24 combinations—6 different partition-size limits over 4 different setups (**BASE**, **PreMerge**, **ORDER**, and **OURS**).

5.2.2 Overall Results

Figure 5.5 shows the overall impact of our improvements. For the two boundary cases, partition-size limit of 1 and 10 million, our optimizations (**OURS**) perform as poorly as the base case (**BASE**). For partition-size limit of 1000, **OURS** begins to perform better than **BASE**. For the remaining partition-size limits (10,000 to 1 million), **OURS** performs much better than **BASE**. In particular, within this range, **OURS** can finish all the models. This result shows that our improvements have greatly stabilized the conjunctive-partition algorithm because it now works well for a wide range of partition-size limits (10,000 to 1 million) with the worst case penalty to be around a factor of 15 slowdown in comparison the best of all combinations we tested. In the following sections, we examine the impact of our improvements in more detail.

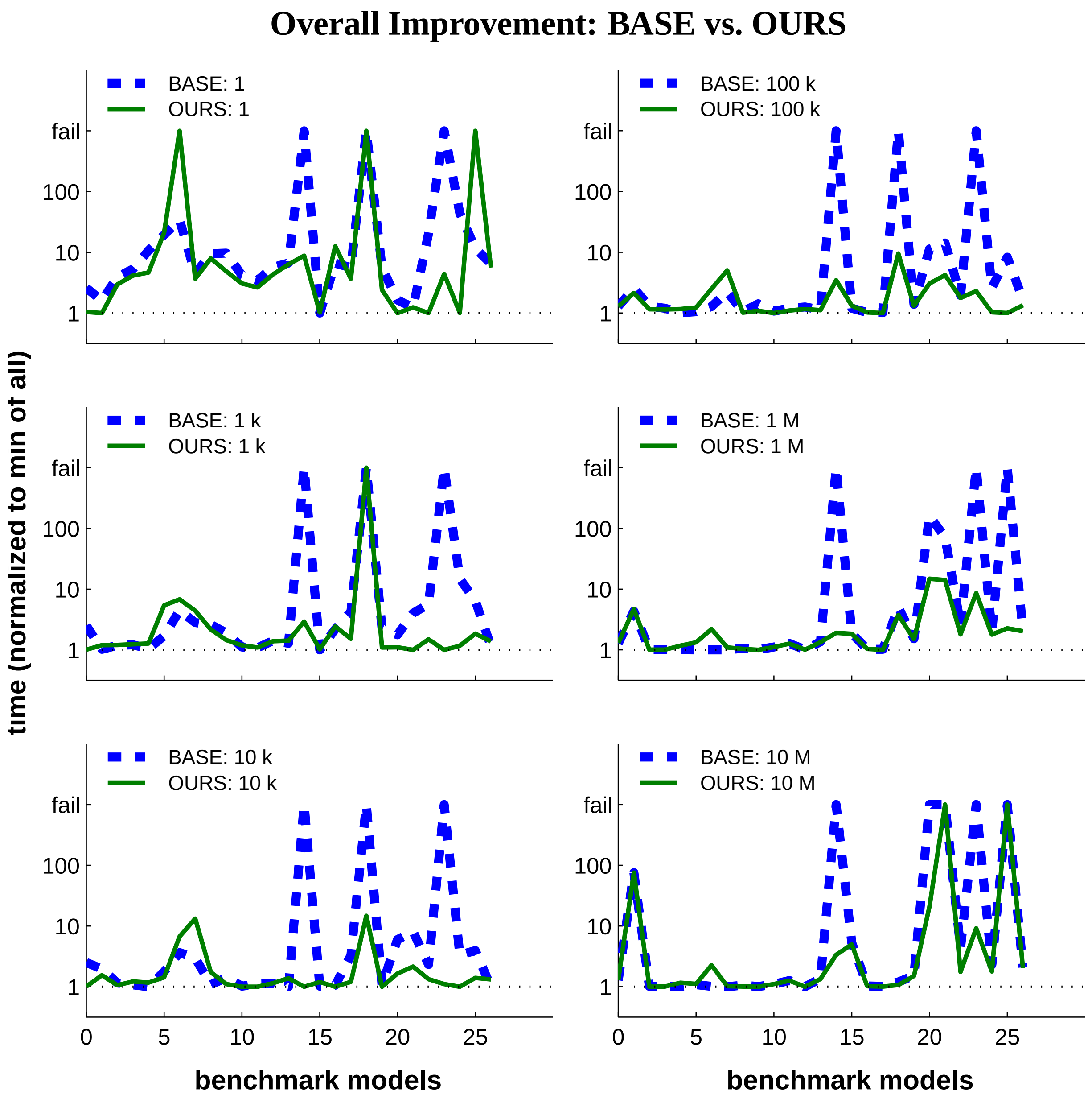


Figure 5.5: Overall impact of our improvements. The result for each model is normalized to the best running time across all runs.

5.2.3 Impact of Pre-Merging

To study the impact of our pre-merging heuristic, we compare its effects both with and without our changes to the ordering phase. Figure 5.6 compares the results without our changes to the ordering phase; i.e., **BASE** vs. **PreMerge**. The results show that overall, adding the pre-merging phase decreases the number of failed cases and generally improves performance. One notable exception is for the benchmark model 25, **PreMerge** fails to complete for both the partition-size limit of 10,000 and 100,000 nodes.

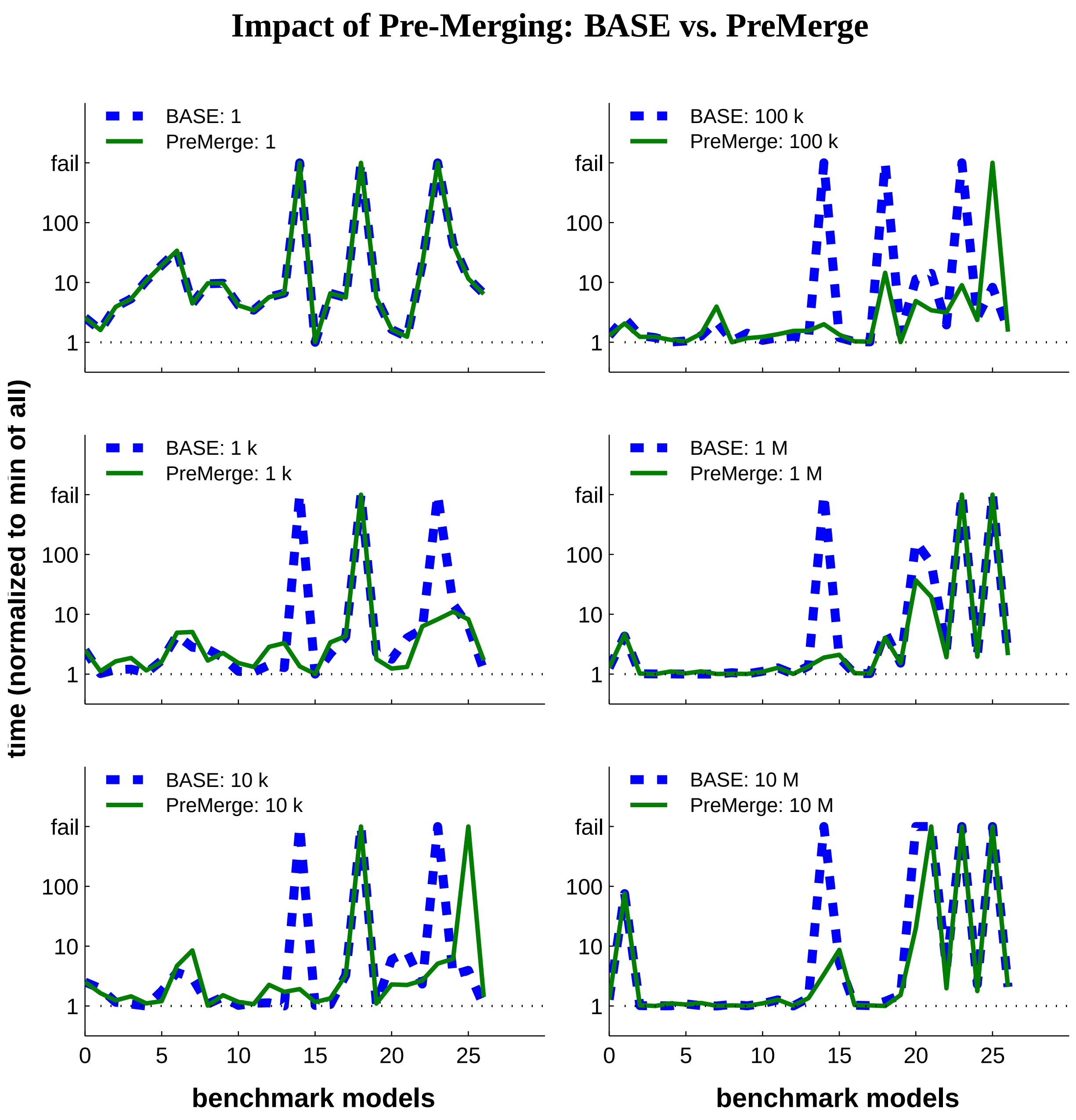


Figure 5.6: Impact of pre-merging phase: without our improved ordering algorithm. The result for each model is normalized to the best running time across all runs. Note that the partition-size limit of 1, the merging does not occur and thus the results are the same.

Figure 5.7 shows the impact of the pre-merge phase in presence of our ordering heuristic by comparing the results between **ORDER** and **OURS**. Similar to the previous set of results, these results show that the pre-merging phase helps to decrease the number of failed cases and to improve the overall performance. This and the previous set of results provide a strong evidence that the pre-merging phase, independent of our ordering phase, is effective in stabilizing the conjunctive-partition algorithm.

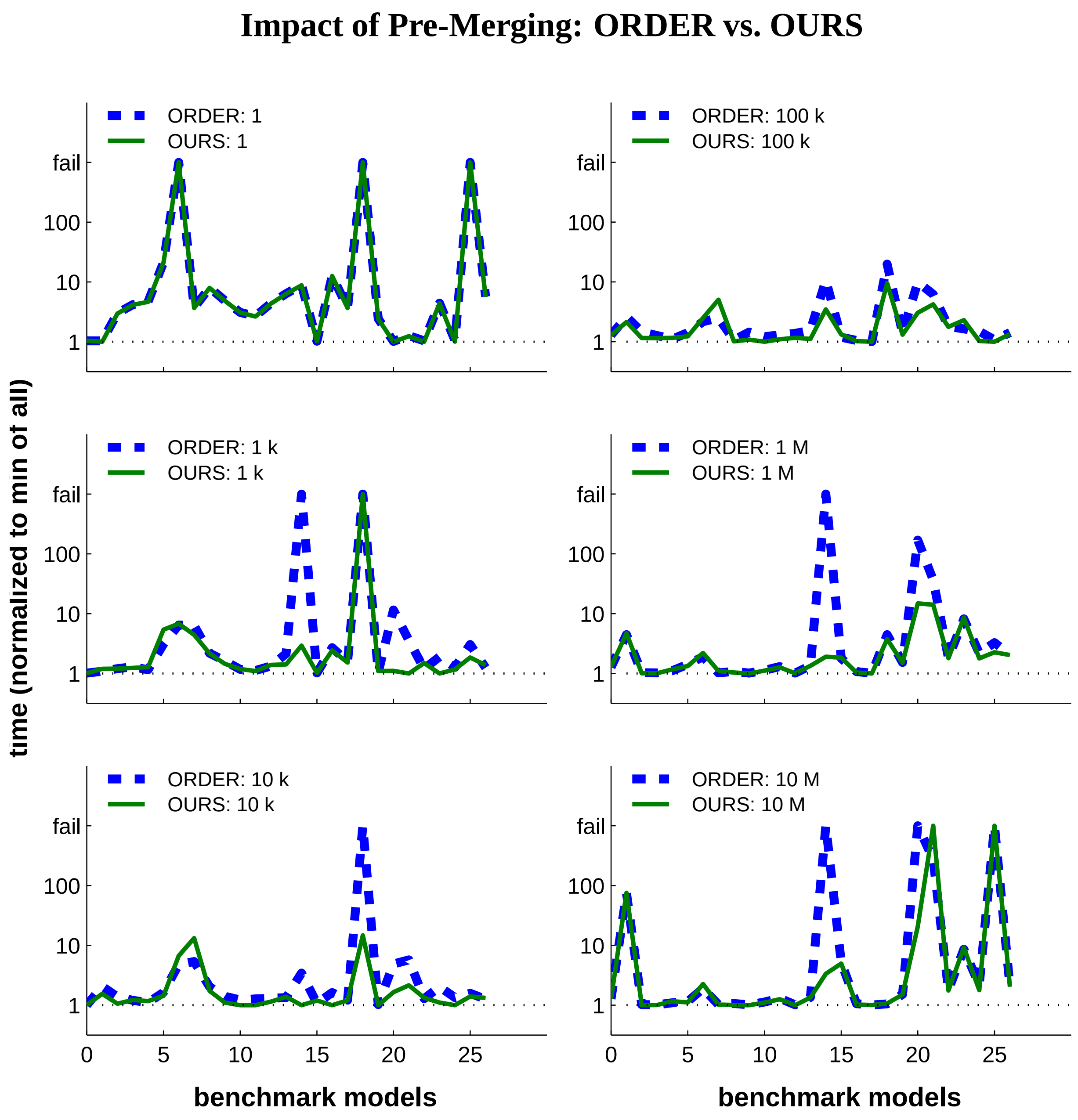


Figure 5.7: Impact of pre-merging phase: with our improved ordering algorithm. The result for each model is normalized to the best running time across all runs. Note that the partition-size limit of 1, the merging does not occur and thus the results are the same.

5.2.4 Impact of Our Ordering Heuristic

To study the impact of our changes to the ordering phase, we compare its effects both with and without the pre-merging phase. Figure 5.8 compares the results without the pre-merging: **BASE** vs. **ORDER**. The results show that our ordering heuristic helps to reduce the number of failed cases. One notable case is that for the limit of 100,000 nodes, using our ordering heuristic, we are able to finish all the cases within at most a factor of 15 slower than the best.

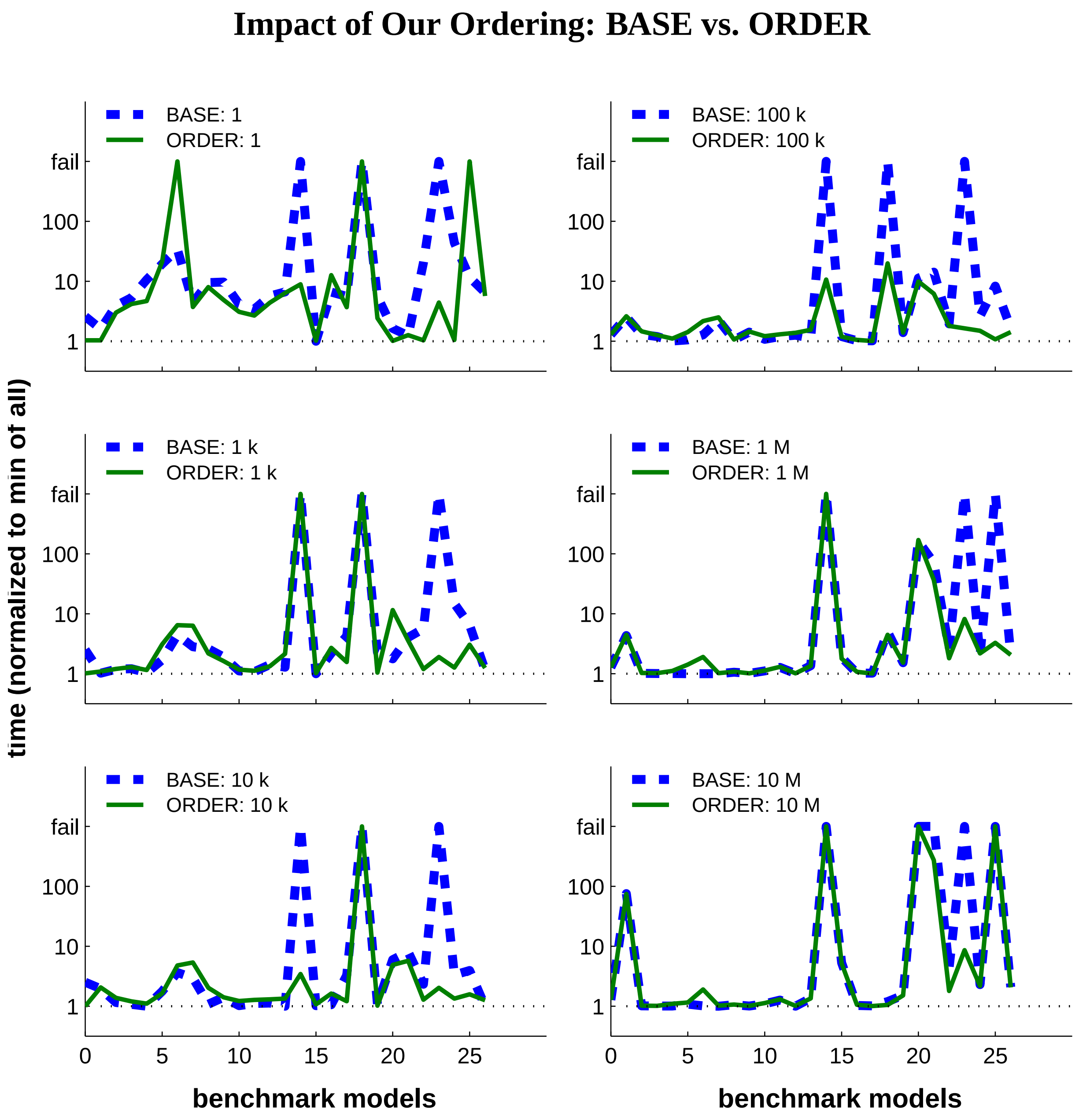


Figure 5.8: Impact of our improved ordering algorithm: without pre-merging. The result for each model is normalized to the best running time across all runs.

Figure 5.9 shows the impact of our ordering heuristic in presence of the pre-merging phase by comparing the results between **ORDER** and **OURS**. Similar to the results without pre-merging phase, our ordering heuristic improves the overall performance in general.

Combining the results from this section and from the previous section, we conclude that the pre-merging phase and our ordering heuristic complement each other in producing a much more stable conjunctive-partition algorithm.

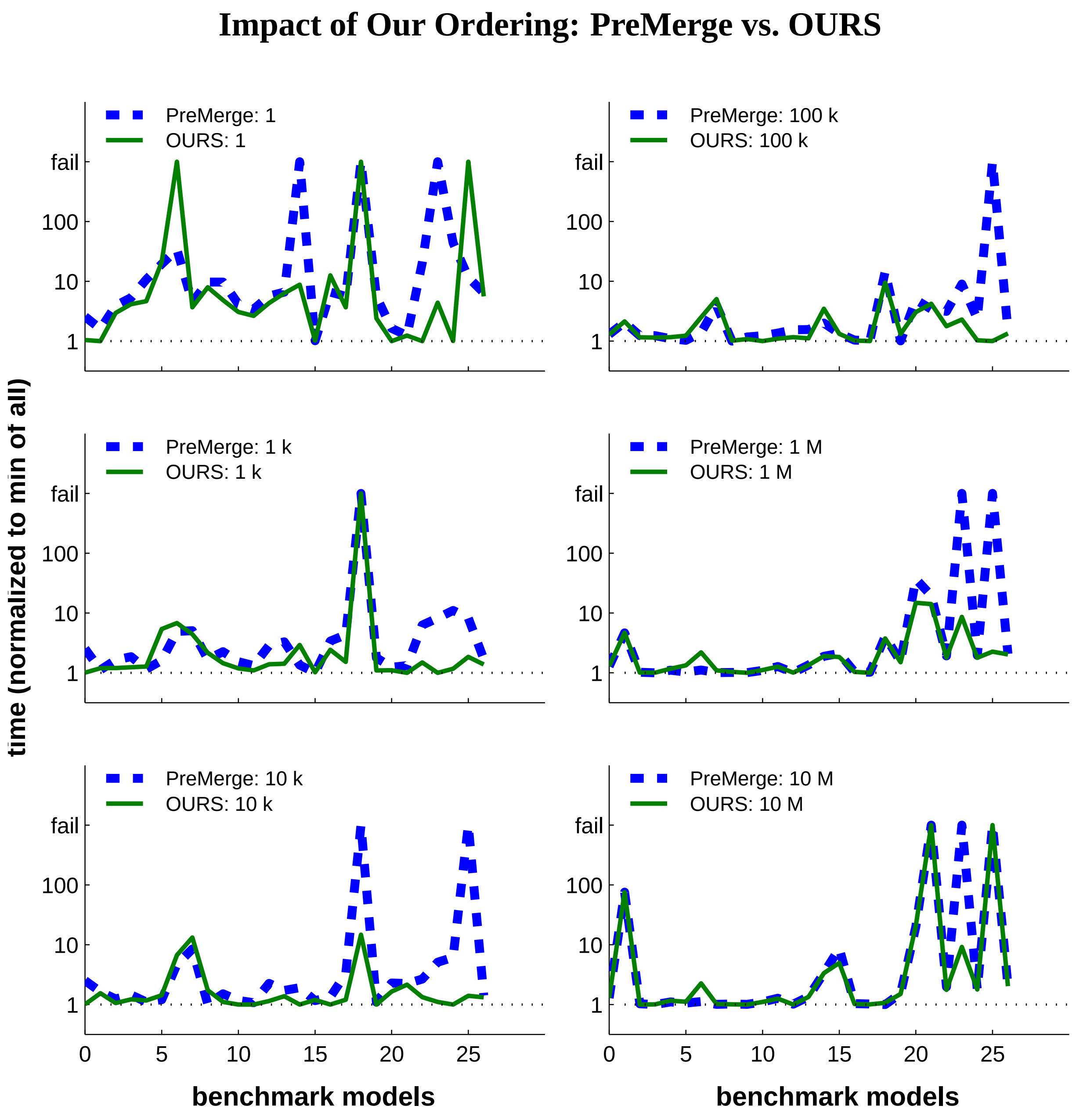


Figure 5.9: Impact of our improved ordering algorithm: with pre-merging. The result for each model is normalized to the best running time across all runs.

5.3 Summary

The main idea in our improvements is to use BDD characteristics to cluster partitions that would merge well together. The experimental results show that our optimizations have successfully improved the stability of the conjunctive-partitioning algorithm. In particular, for a wide range of partition-size limits (10,000 to 1 million), the improved algorithm has the worst-case penalty of around a factor of 15 slowdown in comparison to manually finding the best parameters. This stability result implies less manual intervention is necessary because the penalty for choosing the wrong parameter setting is now much less severe. Furthermore, because the range of good parameter settings is so large, the likelihood of making a *wrong* choice is also significantly decreased.

However, a factor of 15 slower than manually finding the best set of parameters still leaves a lot of room for further improvement. One fundamental problem that we did not solve in this thesis is that we do not have a good objective function for quickly and accurately evaluating the goodness of the results produced by different conjunctive-partitioning algorithms. From optimization point of view, an accurate objective function, once found, can be used to design new heuristics. In particular, given such an objective function, there are powerful optimization techniques from AI domains that will help us find a good heuristic for solving these NP-hard problems. In general, the lack of good objective functions is a fundamental reason why many model-checking optimizations are unstable. In Chapter 7, we will discuss our ongoing research to address this issue.

Chapter 6

BDD Optimizations for Constraint-Rich Models

This chapter combines the improvements made in the previous chapters with a new set of BDD-based optimizations and applies them to enable the verification of a class of real-world applications that have complex time-invariant constraints. An example of constraint-rich systems is the symbolic models developed by NASA for on-line fault diagnosis [81]. These models describe the operation of components in complex electro-mechanical systems, such as autonomous spacecraft or robot explorers. The models consist of interconnected components (e.g., thrusters, sensors, motors, computers, and valves) and describe how the *mode*[1] of each component changes over time. Based on these models, the Livingstone diagnostic engine [81] monitors sensor values and detects, diagnoses, and tries to recover from inconsistencies between the observed sensor values and the predicted modes of the components. The relationships between the modes and sensor values are encoded using symbolic constraints. Constraints between state variables are also used to encode interconnections between components. We have developed an automatic translator from such fault models to SMV (Symbolic Model Verifier) [58], where mode transitions are encoded as transition relations and state-variable constraints are translated into sets of time-invariant constraints.

To verify constraint-rich systems, we introduce two new optimizations. The first optimization is a simple extension of the conjunctive-partitioning algorithm. The other is a collection of BDD-based macro-extraction and macro-expansion algorithms to remove redundant state variables. We show that these two optimizations are essential in verifying constraint-rich problems. In particular, these optimizations have enabled the verification of fault diagnosis models for the Nomad robot (an Antarctic meteorite explorer) [5] and the NASA Deep Space One (DS1) spacecraft [7]. These models can be quite large, with up to 1200 state bits.

In this chapter, we first briefly describe how time-invariant constraints arise naturally from modeling (Section 6.1). We then present our new optimizations: an extension to conjunctive partitioning (Section 6.2), and BDD-based algorithms for eliminating redundant state variables (Section 6.3). We then show the results of a performance evaluation on the effects of each optimization (Section 6.4). Finally, we present a comparison to prior work (Section 6.5). The work described in this chapter can also be found in [85, 86].

[1]A mode of a component qualitatively describes the state of a component. For example, the mode of a thruster's force is one of the following: *low*, *nominal*, or *high*.

6.1 Time-Invariant Constraints and Their Common Usages

In symbolic model checking, time-invariant constraints specify the conditions that must always hold. More formally, let $C_1, \ldots, C_l$ be the time-invariant constraints and let $C := C_1 \wedge C_2 \wedge \ldots \wedge C_l$. Then, in symbolic state traversal, we consider only states where C is true. We refer to C as the *constrained space*. More specifically, the *image* and the *pre-image* of the transition relation T on a state set S, while restricting the computations to the constrained space C, are computed as follows:

$$\begin{array}{rcl} \textit{image}_{V'}(S) & := & C(V') \wedge \exists V.[T(V,V') \wedge (S(V) \wedge C(V))], \\ \textit{pre-image}_{V}(S) & := & C(V) \wedge \exists V'.[T(V,V') \wedge (S(V') \wedge C(V'))]. \end{array}$$

To motivate how time-invariant constraints arise naturally in modeling complex systems, we describe three common usages. One common usage is to make the same non-deterministic choice across multiple expressions in transition relations. For example, in a master-slave model, the master can non-deterministically choose which set of idle slaves to assign the pending jobs, and the slaves' next-state values will depend on the choice made. To model this, let **f** be the function representing how the master makes its non-deterministic choice. Note that we are using the bold-face font to emphasize that the function **f** returns a *set* (of possible choices). If the slaves' transition relations are defined using **f** directly, then each use of **f** makes its own non-deterministic choice independent of other uses. Thus, to ensure that all the slaves see the same non-deterministic choice, a separate state variable u is used to record the choice made, and u is then used (in its present state form) to define the slaves' transition relations. This recording process is expressed as the time-invariant constraint $u \in \mathbf{f}$.

Another common usage is for establishing the interface between different components in a system. For example, suppose two components are connected with a pipe of a fixed capacity. Then, the input of one component is the minimum of the pipe's capacity and the output of the other component. This relationship is described as a time-invariant constraint between the input and the output of these two components.

The third common usage is specific uses of generic parts. For example, a bi-directional fuel pipe may be used to connect two components. If we want to make sure the fuel flows only one way, we need to constrain the valves in the fuel pipe. These constraints are specified as time-invariant constraints. In general, specific uses of generic parts arise naturally in both the software and the hardware domain as we often use generic building blocks in constructing a complex system.

In the examples above, the use of time-invariant constraints is not always necessary because some these constraints can be directly expressed as a part of the transition relation and the associated state variables can be removed. For example, a specific uni-directional fuel pipe can be used in the above example instead of a generic bi-directional fuel pipe and thus would eliminate the extra state variable needed to specify the direction of the valves. However, these constraints are used to facilitate the description of the system or to reflect the way complex systems are built. Without these constraints, multiple expressions will need to be combined into possibly a very complicated expression. Performing this transformation manually can be labor intensive and error-prone. Thus it is up to the verification tool to automatically perform these transformations and remove unnecessary state variables. Our optimizations for constraint-rich models are to automatically eliminate redundant state variables (Section 6.3) and partition the remaining constraints (Section 6.2).

6.2 Extended Conjunctive Partitioning

The first optimization is the application of the conjunctive-partitioning algorithm on the time-invariant constraints. This extension is derived based on two observations. First, as with the transition relations, the BDD representation for time-invariant constraints can be too large to be represented as a monolithic graph. Thus, it is crucial to represent the constraints as a set of conjuncts rather than a monolithic graph.

Second, in constraint-rich models, many *quantifying variables* (variables being quantified) do not appear in the transition relation. There are two common causes for this. First, when time-invariant constraints are used to make the same non-deterministic choices, variables are introduced to record these choices (described as the first example in Section 6.1). In the transition relation, these variables are used only in their present-state form. Thus, their corresponding next-state variables do not appear in the transition relation, and for the pre-image computation, these next-state variables are parts of the quantifying variables. The other cause is that many state variables are used only to establish time-invariant constraints. Thus, both the present- and the next-state version of these variables do not appear in the transition relations.

Based on this observation, we can improve the early-quantification optimization by pulling out the quantifying variables (V_0') that do not appear in any of the transition relations. Then, these quantifying variables (V_0') can be used for early quantification in conjunctive partitioning of the constrained space (C) where the time-invariant constraints hold. Formally, let $P_1 \wedge P_2 \wedge ... \wedge P_k$ be the conjunctive partitioning of the transition relation with V_i, a subset of V, being the set of variables that do not appear in any of the subsequent V_j's, where $1 \leq i \leq k$ and $i < j \leq k$. Let V_0, a subset of V, be the set of quantifying variables that do not appear in any of the P_i's. Let $Q_1, Q_2, ..., Q_m$ be the partitions produced by the conjunctive partitioning of the constrained space C, where $C = Q_1 \wedge Q_2 \wedge ... \wedge Q_m$. Let W_i, a subset of V_0, be the set of variables that do not appear in any of the subsequent Q_j's, for $1 \leq i \leq m$ and $i < j \leq m$. Then, the conjunctive partitioning of the image computation from Section 2.5.3 is extended to be

$$\begin{array}{rcl}
q_1 & := & \exists W_1.[Q_1(V) \wedge S(V)] \\
q_2 & := & \exists W_2.[Q_2(V) \wedge q_1] \\
 & \vdots & \\
q_m & := & \exists W_m.[Q_m(V) \wedge q_{m-1}] \\
 & & \\
p_1 & := & \exists V_1.[P_1(V, V') \wedge q_m] \\
p_2 & := & \exists V_2.[P_2(V, V') \wedge p_1] \\
 & \vdots & \\
p_k & := & \exists V_k.[P_k(V, V') \wedge p_{k-1}] \\
\textit{image}_{V'}(S) & := & C(V') \wedge p_k
\end{array}$$

Similarly, this extension also applies to the pre-image computation.

6.3 Elimination of Redundant State Variables

Our second optimization for constraint-rich models is targeted at reducing the state space by removing unnecessary state variables. This optimization is a set of BDD-based algorithms that compute an equivalent expression for each variable used in the time-invariant constraints (*macro extraction*) and then globally replace a suitable subset of variables with their equivalent expressions (*macro expansion*) to reduce the total number of variables.

The use of macros is traditionally supported by language constructs (e.g., DEFINE in the SMV language [58]) and by simple syntactic analyses such as detecting deterministic assignments (e.g., $a == f$ where a is a state variable and f is an expression) in the specifications. However, in constraint-rich models, the time-invariant constraints are often specified in a more complex manner such as *conditional* dependencies on other state variables; e.g., $p \Rightarrow (a == f)$ as the conditional assignment of expression f to variable

a when p is true. To identify the set of valid macros in such models, we need to combine the effects of multiple constraints. One drawback of syntactic analysis is that, for each type of expression, syntactic analysis will need to add a template to pattern match these expressions. Another more severe drawback is that it is difficult for syntactic analysis to estimate the actual cost of instantiating a macro. Estimating this cost is important because reducing the number of variables by macro expansion can sometimes result in significant performance degradation caused by large increases in other BDDs' sizes. These two drawbacks make the syntactic approach unsuitable for models with complex time-invariant constraints.

Our approach uses BDD-based algorithms to analyze time-invariant constraints and to derive the set of possible macros. The core algorithm is a new assignment-extraction algorithm that extracts assignments from arbitrary Boolean expressions (Section 6.3.1). For each variable, by extracting its assignment form, we can determine the variable's corresponding equivalent expression, and when appropriate, globally replace the variable with its equivalent expression (Section 6.3.3). The strength of this algorithm is that by using BDDs, the cost of macro expansion can be better characterized because the model-checking computation is performed using BDDs.

Note that there have been a number of research efforts on BDD-based redundant state-variable removal. To better compare our approach to these previous research efforts, we postpone the discussion of this prior work until Section 6.5, after describing our algorithms and the performance evaluation.

6.3.1 BDD-Based Assignment Extraction

The assignment-extraction problem can be stated as follows: given an arbitrary Boolean function f and a variable v (where v can be non-Boolean), find $\mathbf{g}$ and h such that

- $f = (v \in \mathbf{g}) \wedge h$,
- $\mathbf{g}$ does not depend on v, and
- h is a Boolean function and does not depend on v.

We refer to the expression $(v \in \mathbf{g})$ as a *non-deterministic assignment* to the variable v from the values in the set returned by the function $\mathbf{g}$. This terminology is derived from the fact that this expression is structurally similar to the assignment of the next-state values of state variables as described in Section 2.5.1.

In the case that $\mathbf{g}$ always evaluates to a singleton set, the assignment $(v \in \mathbf{g})$ is deterministic. A solution to this assignment-extraction problem is as follows:

$$\begin{aligned} h &= \exists v.f \\ \mathbf{t} &= \bigcup_{k \in K_v} \mathit{ITE}(f|_{v \leftarrow k}, \{k\}, \emptyset) \\ \mathbf{g} &= \mathit{restrict}(\mathbf{t}, h) \end{aligned}$$

where K_v is the set of all possible values of variable v, and *restrict* (described in Section 2.3.2) is a care-space optimization algorithm that tries to reduce the BDD size (of $\mathbf{t}$) by collapsing the don't-care space $(\neg h)$. The BDD algorithm for the $\bigcup_{k \in K_v}$ operator is similar to the BDD algorithm for the existential quantification with the $\vee$ operator replaced by the $\cup$ operator for variable quantification. A correctness proof of this algorithm is shown in the next subsection.

Note that in the assignment-extraction algorithm, the use of the *restrict* algorithm is not necessary. In fact, any care-space optimization algorithms can be used instead of the *restrict* algorithm. We choose to use the *restrict* algorithm because it works well in practice for other part of model-checking computations.

6.3.2 Correctness Proof for Assignment-Extraction Algorithm

In this section, we present a correctness proof for the assignment-extraction algorithm in Section 6.3.1. Before presenting the main result, we first state and prove two supporting lemmas.

Lemma 1 *Given any care-space optimization care-opt. Then, for any function* $\mathbf{t}$*, Boolean function* h*, and variable* v*,*

$$(v \in \textit{care-opt}(\mathbf{t}, h)) \wedge h = \textit{care-opt}(v \in \mathbf{t}, h) \wedge h.$$

Proof

By the definition of the care-space optimization, we have the following properties:

$$\begin{aligned} h &\Rightarrow (\textit{care-opt}(\mathbf{t}, h) == \mathbf{t}), \\ h &\Rightarrow [\textit{care-opt}(v \in \mathbf{t}, h) == (v \in \mathbf{t})]. \end{aligned}$$

Therefore,

$$\begin{aligned} (v \in \textit{care-opt}(\mathbf{t}, h)) \wedge h &= (v \in \mathbf{t}) \wedge h \\ &= \textit{care-opt}(v \in \mathbf{t}, h) \wedge h. \end{aligned}$$

□

Lemma 2 *Given an arbitrary Boolean function* f *and a variable* v*. Let*

$$\mathbf{t} = \bigcup_{k \in K_v} \textit{ITE}(f|_{v \leftarrow k}, \{k\}, \emptyset),$$

where K_v *is the set of all possible values of variable* v*. Then,*

$$(v \in \mathbf{t}) = f.$$

Proof

$$\begin{aligned} v \in \mathbf{t} &= \bigvee_{k' \in K_v} (v == k') \wedge (k' \in \mathbf{t}) \\ &= \bigvee_{k' \in K_v} (v == k') \wedge [k' \in \bigcup_{k \in K_v} \textit{ITE}(f|_{v \leftarrow k}, \{k\}, \emptyset)] \\ &= \bigvee_{k' \in K_v} (v == k') \wedge \bigvee_{k \in K_v} [k' \in \textit{ITE}(f|_{v \leftarrow k}, \{k\}, \emptyset)] \\ &= \bigvee_{k' \in K_v} (v == k') \wedge \bigvee_{k \in K_v} \textit{ITE}(f|_{v \leftarrow k}, k' \in \{k\}, k' \in \emptyset) \\ &= \bigvee_{k' \in K_v} (v == k') \wedge \bigvee_{k \in K_v} \textit{ITE}(f|_{v \leftarrow k}, k' \in \{k\}, 0) \\ &= \bigvee_{k' \in K_v} (v == k') \wedge \textit{ITE}(f|_{v \leftarrow k'}, k' \in \{k'\}, 0) \\ &= \bigvee_{k' \in K_v} (v == k') \wedge \textit{ITE}(f|_{v \leftarrow k'}, 1, 0) \\ &= \bigvee_{k' \in K_v} (v == k') \wedge f|_{v \leftarrow k'} \\ &= f. \end{aligned}$$

□

Using the two lemmas above, we can now prove the correctness of the assignment-extraction algorithm.

Theorem 1 *Given an arbitrary Boolean function f and a variable v. Let*

$$\begin{aligned} h &= \exists v.f, \\ \mathbf{t} &= \bigcup_{k \in K_v} \mathit{ITE}(f|_{v \leftarrow k}, \{k\}, \emptyset), \\ \mathbf{g} &= \mathit{care\text{-}opt}(\mathbf{t}, h), \end{aligned}$$

where care-opt is any care-space optimization algorithm and care-opt$(\mathbf{t}, h)$ does not depend on any new variables (other than those already in **t** *and h). Then, the following conditions are true.*

1. $f = (v \in \mathbf{g}) \wedge h$,
2. **g** *does not depend on v, and*
3. *h is a Boolean function and does not depend on v.*

Proof

To prove Condition 1, we apply both Lemma 1 and Lemma 2 in the following derivation:

$$\begin{aligned} (v \in \mathbf{g}) \wedge h &= (v \in \mathit{care\text{-}opt}(\mathbf{t}, h)) \wedge h \\ &= \mathit{care\text{-}opt}(v \in \mathbf{t}, h) \wedge h \\ &= \mathit{care\text{-}opt}(f, h) \wedge h \\ &= f \wedge h \\ &= f \wedge \exists v.f \\ &= f \end{aligned}$$

Condition 2 is true because **t** does not depend on v (by construction) and the *care-opt* algorithm does not introduce new variable dependencies (given). Condition 3 is true because f is a Boolean function and $h = \exists v.f$ is a Boolean function that does not depend on v.

□

6.3.3 Macro Extraction and Expansion

In this section, we describe the elimination of state variables based on macro extraction and macro expansion. The first step is to extract macros with the algorithm shown in Figure 6.1. This algorithm extracts macros from the constrained space (C), which is represented as a set of conjuncts. It first uses the assignment-extraction algorithm to extract assignment expressions (line 5). It then identifies the deterministic assignments as candidate macros (line 6). For each candidate, the algorithm determines if instantiating the macro may be beneficial (line 7). This test is based on the heuristic that if the BDD size of a macro is not too large and its instantiation does not cause excessive increase in other BDDs' sizes, then instantiating this macro may be beneficial. If the resulting right-hand-side **g** is not a singleton set, it is kept separately (line 9). These **g**'s are combined later (line 10) to determine if their intersection would result in a macro (lines 11–13). Finally, this algorithm returns the set of selected macros (line 14).

After the macros are extracted, the next step is to determine the instantiation order. The main purpose of this algorithm (in Figure 6.2) is to remove circular dependencies. For example, if one macro defines variable v_1 to be $(v_2 \wedge v_3)$ and a second macro defines v_2 to be $(v_1 \vee v_4)$, then instantiating the first macro results in a circular definition in the second macro ($v_2 = (v_2 \wedge v_3) \vee v_4$) and thus invalidates this second

```
    extract_macros($C, V$)
      /* Extract macros for variables in $V$ from
         the set $C$ of conjuncts representing the constrained space */
1     $M \leftarrow \emptyset$    /* initialize the set of macros found so far */
2     for each $v \in V$
3        $N \leftarrow \emptyset$    /* initialize the set of non-singletons found so far */
4        for each $f \in C$ such that $f$ depends on $v$
5           $(\mathbf{g}, h) \leftarrow$ assignment-extraction $(f, v)$    /* $f = (v \in \mathbf{g}) \wedge h$ */
6           if ($\mathbf{g}$ always returns a singleton set)    /* macro found */
7              if (is-this-result-good($\mathbf{g}$))
8                 $M \leftarrow \{(v, \mathbf{g})\} \cup M$
9           else $N \leftarrow \{\mathbf{g}\} \cup N$
10       $\mathbf{g'} \leftarrow \bigcap_{\mathbf{g} \in N} \mathbf{g}$
11       if ($\mathbf{g'}$ always returns a singleton set)    /* macro found */
12          if ((is-this-result-good($\mathbf{g'}$))
13             $M \leftarrow \{(v, \mathbf{g'})\} \cup M$
14    return $M$
```

Figure 6.1: Macro-extraction algorithm. In lines 7 and 12, "is-this-result-good" uses BDD properties (such as graph sizes) to determine if the result should be kept.

macro. Similarly, the reverse is also true. To determine the set of macros to remove, the algorithm builds a dependence graph (line 1) and breaks circular dependencies based on graph sizes (lines 2–4). It then determines the ordering of the remaining macros based on the topological order (line 4) of the dependence graph.

Finally, in the topological order, each macro $(v, \mathbf{g})$ is instantiated in the remaining macros and in all other expressions (represented by BDDs) in the system, by substituting the variable v with its equivalent expression $\mathbf{g}$.

```
    order_macros($M$)
      /* Determine the instantiation order of the macros in set $M$ */
      /* first build the dependence graph $G = (M, E)$ */
1     $E = \{(x, y) | x = (v_x, \mathbf{g}_x) \in M, y = (v_y, \mathbf{g}_y) \in M, \mathbf{g}_y$ depends on $v_x\}$
      /* then remove circular dependences */
2     while there are cycles in $G$,
3        $M_C \leftarrow$ set of macros that are in some cycle
4        remove the macro with largest BDD size in $M_C$
5     return a topological ordering of the remaining macros in $G$
```

Figure 6.2: Macro-ordering algorithm.

6.4 Evaluation

6.4.1 Experimental Setup

The benchmark suite used is a collection of 58 SMV models gathered from a wide variety of sources, including the 16 models used in the BDD performance study described in Chapter 4. Out of these 58 models, 37 models have no time-invariant constraints, and thus our optimizations are not triggered and have no influence on the overall verification time. Out of the remaining 21 models, 10 very small models ($<$ 10 seconds) are eliminated. On the remaining 11 models, our optimizations have made non-negligible performance improvements on 7 models, where the results changed by more than 10 CPU seconds and 10% from the base case where no optimizations are enabled. In Figure 6.3, we briefly describe these 7 models. Note that some of these models are quite large, with up to 1200 state bits.

Model	# of State Bits	Description
acs	497	the altitude-control module of NASA's DS1 spacecraft
ds1-b	657	a buggy fault diagnosis model for NASA's DS1 spacecraft
ds1	657	corrected version of *ds1-b*
f-bus	174	FutureBus cache coherency protocol
nomad	1273	fault diagnosis model for an Antarctic meteorite explorer
v-gate	86	reactor-system model
xavier	100	fault diagnosis model for CMU's Xavier robot

Figure 6.3: Description of models whose performance results are affected by our optimizations.

The results reported in this section are labeled with the following keys to indicate which optimizations are enabled:

None: no optimizations.

Quan: the "early quantification on the constrained space" optimization (Section 6.2).

SynM: syntactic analysis for macro-extraction and macro-expansion. This algorithm pattern matches deterministic assignment expressions $v == f$, where v is a state variable and f is an expression, as macros and expands these macros.

BDDM: the BDD-based macro extraction and macro expansion (Section 6.3).

Q+SynM: both **Quan** and **SynM** optimizations.

Q+BDDM: both **Quan** and **BDDM** optimizations.

We performed the evaluation using the Symbolic Model Verifier (SMV) model checker [58] from Carnegie Mellon University. Conjunctive partitioning was used only when it was necessary to complete the verification. In these cases (including *acs, ds1-b, ds1* and *nomad*), the size limit for each partition was set to 10,000 BDD nodes. For the remaining cases, the transition relations were represented as monolithic BDDs. The constrained space C was represented as a conjunction with each conjunct's BDD size limited to 10,000 nodes. Without partitioning, we could not construct the BDD representation for the constrained space for 4 models. The evaluation was performed on a 200MHz Pentium-Pro running Linux. Each run was limited to 6 hours of CPU time and 900 MByte of the main memory.

In Figure 6.4, we show the running time of different optimizations. Note that for all benchmarks, the time spent by our optimizations is very small ($<$ 5 seconds or $<$ 5% of total time) and is included in the running time shown. In the rest of this section, we analyze these results in the following order: the overall impact of our optimizations (Section 6.4.2), the impact of early quantification on the constraint space (Section 6.4.3), and the impact of macro optimization (Section 6.4.4). We then finish with a brief study on the impact of different size limits for conjunctive partitioning (Section 6.4.5).

6.4.2 Overall Results

The results in Figure 6.5 show the overall performance impact of our optimizations. These results demonstrate that our optimizations have significantly improved the performance for 2 models (with speedups up

Model	**None** (sec)	**Quan** (sec)	**SynM** (sec)	**BDDM** (sec)	**Q+SynM** (sec)	**Q+BDDM** (sec)
acs	*m.o.*	32	*m.o.*	1059	76	7
ds1-b	*m.o.*	321	*t.o.*	*m.o.*	138	54
ds1	*m.o.*	*m.o.*	*m.o.*	*t.o.*	*t.o.*	37
f-bus	1410	53	78	37	35	19
nomad	*m.o.*	*t.o.*	*m.o.*	*t.o.*	7801	633
v-gates	36	35	51	50	53	50
xavier	16	5	6	5	1	2

Figure 6.4: Running time with different optimizations enabled. The *m.o.*'s and *t.o.*'s are the results that exceeded the 900-MByte memory limit and the 6-hour time limit, respectively.

to 74) and have enabled the verification of 4 models. For the *v-gates* model, the performance degradation (speedup = 0.7) is in the computation of the reachable states from the initial states. Upon further investigation, we believe that it is caused by the macro optimization, which increases the graph size of the transition relation from 122-thousand to 476-thousand nodes. This case demonstrates that reducing the number of state variables does not always improve performance.

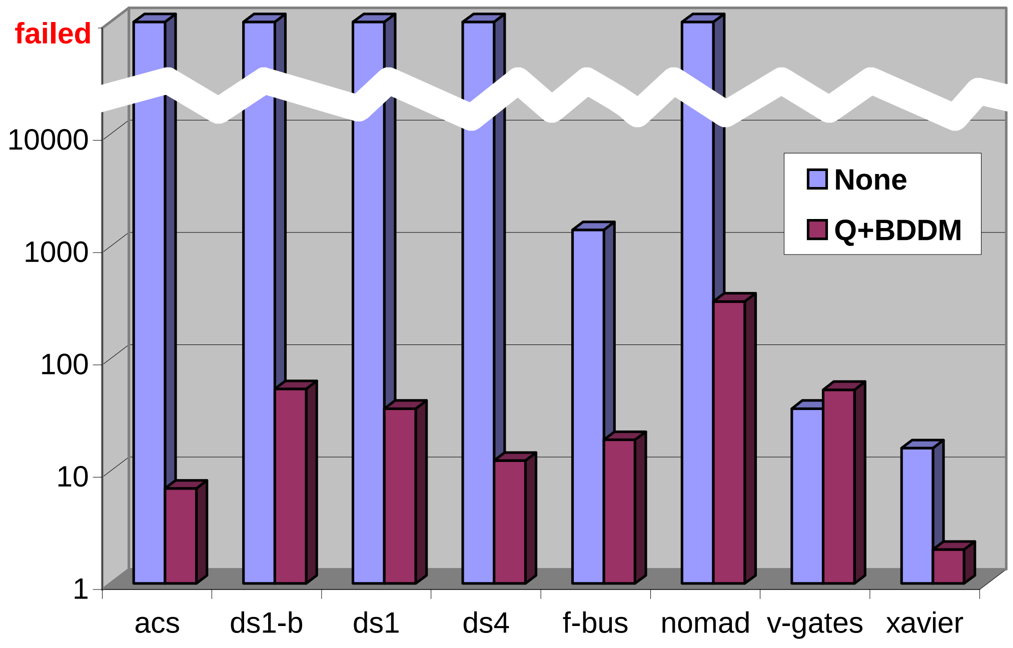


Figure 6.5: Overall impact of our optimizations. The *failed*'s are the results that exceeded the 900-MByte memory limit.

6.4.3 Impact of Early Quantification

The results in Figure 6.6 show the impact of applying early quantification on time-invariant constraints. The impact is measured both in the number of quantifying BDD variables extracted from the transition relations and in the performance speedups. The speedup results for **None / Quan** show that adding this optimization has enabled the verification of *acs* and *ds1-b*, and achieved significant performance improvement on *f-bus* (speedup of 26). The results in the **Quan** columns show that this improvement is mostly due to the fact that a large number of variables can be pulled out of the transition relations and applied to conjunctive partitioning and early quantification of the time-invariant constraints.

From the **Q+BDDM** and **BDDM / Q+BDDM** columns, we observe similar results in presence of BDD-based macro optimization. Note that for the **Q+BDDM** columns, the "# of BDD vars extracted" results also include the number of BDD variables that are removed by the macro optimization. This is done to make the comparison between **Quan**'s and **Q+BDDM**'s effects on "# of BDD vars extracted" easier.

The results in Figure 6.6 show two additional interesting points. First, the number of variables extracted for pre-image computation is more than that extracted for image computation. This is because some variables are only used in their present-state form in the transition relation (see the first example in Section 6.1). Second, comparing the results between **Quan** and **Q+BDDM** columns indicates that the macro optimization generally does not interfere with the early-quantification optimization. The one exception is the *nomad* model, where the macro optimization introduced 114 additional BDD variables ((1121 - 1067) present-state variables plus (1174 - 1114) next-state variables) to the overall transition relation.

	Total	**Effects of CP Optimization**					
	# of	# of BDD vars extracted				performance speedup	
	BDD	**Quan**		**Q+BDDM**		**None /**	**BDDM /**
Model	**Vars**	img	p-img	img	p-img	**Quan**	**Q+BDDM**
acs	994	439	449	437	449	*enabled*	151.0
ds1-b	1314	550	566	546	566	*enabled*	*enabled*
ds1	1314	550	566	546	566	n/a	*enabled*
f-bus	348	58	110	54	110	26.6	1.9
nomad	2546	1121	1174	1067	1114	*n/a*	*enabled*
v-gates	172	0	17	8	17	1.0	1.0
xavier	200	69	86	69	86	3.2	2.5

Figure 6.6: Effectiveness of the extended conjunctive-partitioning optimization. The effectiveness measures are (1) the number of quantifying BDD variables that are pulled out of the transition relation for the early quantification of the time-invariant constraints, and (2) the impact on overall running time as performance speedups. For both measures, we present results both with (**+BDDM**) and without the BDD-based macro optimization. The *n/a* indicates that the speedup can not be computed because both cases failed to finish within the resource limits. **Note:** the number of BDD variables is twice the number of state variables—one copy for the present state and one copy for the next state.

6.4.4 Impact of Macro Extraction and Macro Expansion

The results in Figure 6.7 show the impact of the BDD-based macro optimization. This impact is measured both in the number of BDD variables removed and in the performance speedups. The performance results in the **None / BDDM** column show that adding this optimization has enabled the verification of *acs* and achieved significant performance improvement on *f-bus* (speedup of 38). The results are similar in presence of the early-quantification optimization (the **Quan / Q+BDDM** column). The results for "# of BDD vars removed" show that these performance improvements are due to the effectiveness of BDD-based macro optimization in removing variables; in particular, over a third of variables are removed for 4 models.

To evaluate the effectiveness of syntactic-based vs. BDD-based macro extraction, we compare the impact of these two approaches using both the number of BDD variables removed and the running time (Figure 6.8). The comparison is done for both with and without the early-quantification optimization. Note that the early-quantification optimization does not affect the number of BDD variables removed. Thus, the "# of BDD

Model	**Total # of BDD Variables**	**Effects of BDD-based Macro Optimization** # of BDD variables removed	performance speedup **None / BDDM**	**Quan / Q+BDDM**
acs	994	352	*enabled*	4.5
ds1-b	1314	492	*n/a*	5.9
ds1	1314	496	*n/a*	*enabled*
f-bus	348	18	38.1	2.7
nomad	2546	844	*n/a*	*enabled*
v-gates	172	16	0.7	0.7
xavier	200	116	3.2	2.5

Figure 6.7: Effectiveness of macro optimizations. The effectiveness measures are (1) the number of BDD variables removed by the macro optimization, and (2) the impact on overall running time as performance speedups. For both measures, we present results both with and without the early-quantification optimization. The *n/a* indicates that the speedup can not be computed because both cases failed to finish within the resource limits. **Note:** the number of BDD variables is twice the number of state variables—one copy for the present state and one copy for the next state.

vars removed" results are the same for both with and without the early-quantification optimization.

Without the early-quantification optimization (the **SynM / BDDM** column), the results show that the BDD-based approach is better with the verification of *acs* enabled. With the early-quantification optimization (the **Q+SynM / Q+BDDM** column), the results show that the BDD-based approach has enabled the verification of *ds1* and generally has better performance, with speedups of over 10 in *acs* and *nomad.* In the *xavier* case, the slowdown is 2 (speedup of 0.5) because the **Q+BDDM** used one extra CPU second in macro extraction. The overall performance improvements are due to the fact that the BDD-based approach is more effective in reducing the number of variables ("# of BDD vars removed" columns). In particular, for the *acs, ds1-b, ds1* and *nomad* models, > 150 additional BDD variables (i.e., > 75 additional state bits) are removed in comparison to using syntactic analysis.

Model	**Total # of BDD Vars**	**Syntax vs. BDD-based Macro Optimization** # of BDD vars removed **SynM** or **Q+SynM**	**BDDM** or **Q+BDDM**	performance speedup **SynM / BDDM**	**Q+SynM / Q+BDDM**
acs	994	82	352	*enabled*	10.8
ds1-b	1314	148	492	*n/a*	2.5
ds1	1314	220	496	*n/a*	*enabled*
f-bus	348	12	18	2.1	1.8
nomad	2546	688	844	*n/a*	12.3
v-gates	172	16	16	1.0	1.0
xavier	200	64	116	1.2	1 / 2 = 0.5

Figure 6.8: Syntactic-based vs. BDD-based macro optimization. The effectiveness measures are (1) the number of BDD variables removed, and (2) the impact on overall running time as performance speedups. For both measures, we present results both with and without the early-quantification optimization. The *n/a* indicates that the speedup can not be computed because both cases failed to finish within the resource limits. **Note:** the number of BDD variables is twice the number of state variables—one copy for the present state and one copy for the next state.

6.4.5 Impact of Conjunctive-Partitioning Size Limit

Because the conjunctive-partitioning algorithm often produces significantly different performance results with different partition-size limits, we have also re-evaluated the above results using a partition-size limit of 100,000 nodes. The new results generally follow the same trend as before with the exception of *ds1-b* and *ds1*. For these two models, the results (Figure 6.9) show that if we choose the right partition-size limit for each case, we do not need to perform the macro optimization to verify them. (Note that this is not always true; e.g., the *nomad* model cannot be verified without the macro optimization.) However, with the BDD-based macro optimization (the **Q+BDDM** column), the performance results are more stable and are generally much better.

Model	Partition Size Limit (# of nodes)	**None** (sec)	**Quan** (sec)	**SynM** (sec)	**BDDM** (sec)	**Q+SynM** (sec)	**Q+BDDM** (sec)
ds1-b	10,000	*m.o.*	321	*t.o.*	*m.o.*	138	54
ds1-b	100,000	*t.o.*	309	*t.o.*	*m.o.*	*t.o.*	101
ds1	10,000	*m.o.*	*m.o.*	*m.o.*	*t.o.*	*t.o.*	37
ds1	100,000	*m.o.*	255	*t.o.*	*m.o.*	92	74

Figure 6.9: Effects of the partition-size limit. The *m.o.*'s and *t.o.*'s are the results that exceeded the 900-MByte memory limit and the 6-hour time limit, respectively.

6.5 Related Work

There have been many research efforts on BDD-based redundant state-variable removal in both logic synthesis and verification. These research efforts all use the reachable state space (set of states reachable from initial states) to determine functional dependencies for Boolean variables (macro extraction). The reachable state space effectively plays the same role as a time-invariant constraint, because the verification process only needs to check the correctness of specifications within the reachable state space.

Berthet et al. propose the first redundant state-variable removal algorithm in [8]. In [54], Lin and Newton describe a branch-and-bound algorithm to identify the maximum set of redundant state variables. In [74], Sentovich et al. propose new algorithms for latch removal and latch replacement in logic synthesis. There is also some work on detecting and removing redundant state variables while the reachable state space is being computed [47, 80].

From the algorithmic point of view, our approach is different from prior work in two ways. First, in determining the relationship between variables, the algorithms used to extract functional dependencies in previous work can be viewed as direct extraction of deterministic assignments to Boolean variables. In comparison, our assignment extraction algorithm is more general because it can also handle non-Boolean variables and extract non-deterministic assignments. Second, in performing the redundant state-variable removal, the approach used in the previous work would need to combine all the constraints first and then extract the macros directly from the combined result. However, for constraint-rich models, it may not be possible to combine all the constraints because the resulting BDD is too large to build. Our approach addresses this issue by first applying the assignment extraction algorithm to each constraint separately and then combining the results to determine if a macro can be extracted (see Figure 6.1).

Another difference is that in previous work, the goal is to remove as many variables as possible. However, we have empirically observed that in some cases, removing additional variables can result in significant performance degradation in overall verification time (slowdown over 4). To address this issue, we use simple heuristics (size of the macro and the growth in graph sizes) to choose the set of macros to expand. This simple heuristic works well in the test cases we tried. However, to fully evaluate the impact of different heuristics, we need to gather a larger set of constraint-rich models from a wider range of applications.

6.6 Summary

This chapter described a case study where we applied the same systematic approach to study the verification of constraint-rich applications. As a result, we developed new BDD-based optimizations targeting this new class of applications. Combining these optimizations with improvements to the BDD package (Chapter 4) and conjunctive partitioning (Chapter 5), we have enabled the verification of real-world applications such as the Nomad robot and the NASA Deep Space One spacecraft.

Chapter 7

Conclusions and Future Work

The complexity of model checking computations together with poor evaluation methodologies often results in unstable algorithms. In this thesis, I have demonstrated that rigorous quantitative analysis is a powerful tool in obtaining both performance stability and performance improvement.

This thesis' contribution to the field of model checking is at two levels: methodology and optimizations. In terms of methodology, I have proposed and validated a general BDD evaluation framework. This framework allows us to systematically study all types of BDD computations more easily. It not only helps BDD package designers in performance tuning, it can also helps designers understand the underlying computational characteristics. This methodology is gradually being adopted and is making an impact in this field. For example, researchers have used it to extend a BDD variant called MORE [46] to perform model-checking computations. Furthermore, the benchmark traces and the trace player serve as a platform for other researchers to study other aspects of BDD computations, e.g., in evaluating new dynamic variable reordering algorithms. Since its web site was setup 6 months ago, there have been more than 450 accesses from over 20 different countries.

In terms of new optimizations, I have proposed heuristics to stabilize the conjunctive-partitioning algorithm, which plays an important role of model checking computation, and have introduced a new assignment extraction algorithm that enabled the verification of real-world applications such as the Deep Space One spacecraft.

Combining the improvements obtained from systematic evaluation of underlying BDD packages with the improvements from the higher level optimizations plus incorporating existing algorithms such as better counter-example generation [26], I have dramatically improved the performance of the SMV model checker with speedups of over 10 for most cases (Figure 1.4 in Chapter 1). The key to these successes is a bottom-up approach where we systematically study BDD computations at every stage of the model checking process.

Another contribution of this thesis is that it introduces a number of interesting future research directions. The BDD performance study described in Chapter 4 raises a number of open questions. One issue in particular is the large number of repeated subproblems across top-level operations. This question is important because it may be the main reason why the computed cache size and the garbage collection frequency have such a significant impact on model-checking computations. Thus, understanding this issue may help us design even better BDD packages for model-checking computations.

Another interesting research direction is to apply our evaluation methodology to study other BDD-based tools—both in model checking and in other areas. In model checking, studying other BDD-based tools, such as VIS from UC Berkeley, is important to test whether or not the insights gained from studying SMV computations are particular to SMV or general characteristics of model-checking computations. As for other BDD-based applications, I am particularly interested in studying how BDDs can be used in terms of integer programming. In particular, exploiting how BDDs can be used in satisfying resource constraints in network optimization (Kosak, Carnegie Mellon) and in job scheduling (Zhu, Iowa State University).

Finally, an exciting future research direction is statistical modeling of BDD computations. This is an ongoing research to address a fundamental cause for the performance instability problem—the lack of an accurate objective function for optimizations. Previously in Section 5.3, I have briefly described this problem in the context of the conjunctive partitioning. In the rest of the chapter, I will describe how this problem is prevalent in model checking computations and our ongoing effort to address this issue.

7.1 In Search of an Accurate Objective Function

A fundamental reason for performance instability of optimization algorithms is the lack of an accurate objective function that can quickly and accurately estimate the impact of each optimization. For example, in macro expansion, we use very crude heuristics such as graph size to determine which macros may be beneficial to instantiate. However, we do not know how accurately this measure corresponds to its actual impact. Similarly, in dynamic variable reordering, all the algorithms are based on a simple objective function—minimize the total number of BDD nodes. However, having smaller BDDs now, do not necessarily mean cheaper computations in the future. For conjunctive-partitioning algorithms, again, we have the same problem. We cannot predict (not even roughly) the performance impact of one ordered set of conjuncts versus another. The lack of accurate objective functions to predict performance impacts means that we can only optimize based on crude heuristics, such as graph sizes and the set of variables. However, because we do not know how these metrics correlate to future performance, even if we find an optimal solution for these crude objective functions, the performance-instability problem will continue to persist.

How can an accurate objective function help? For starters, we can use various algorithms to produce solutions and use the objective function to choose the best solution. Furthermore, the objective function itself may shed some light on what parameters are important and thus can be used to design new algorithms for finding better solutions. Ultimately, given an accurate objective function, there are powerful techniques such as genetic algorithms that can be used to automatically find a good solution.

To find a good objective function, we currently focus on characterizing the relational-product operation. This is because the relational products of transition relations and set of states are the core of the symbolic state traversal and they are often the most time-consuming operations in model checking. Thus by finding a function that can accurately estimate the cost of relational-product operations, we can use it as the cost function for the conjunctive-partitioning of transition relations. For dynamic variable reordering, an accurate objective function for relational product can also be used to predict the impact of different variable orders on future relational-product operations.

The procedure we use to find the cost-estimator function for the relational-product operation has two phases. In the first phase, we gather data on the attributes of a large number of top-level relational-product operations. These attributes include the graph sizes of the BDD arguments and the result, the number of variables involved, and the cost of computing the operation in both space and time. In the second phase, we fit these data by constructing a cost-estimate function based on the attributes.

This is an ongoing research. At this point, we have completed the first phase and gathered a few hundred thousand data points. We are currently trying to fit the data using statistical techniques. Our first attempt based on least squares fit [67] failed to produce any meaningful results.

7.2 Summary

As hardware and software are both becoming an integral part of our daily life, the issue of correctness will be an essential part of the design process. Symbolic model checking has proven to be a powerful paradigm to automatically verify real-world applications. This thesis took a systems approach on studying and improving the underlying computations at every level of model checking computations. This bottom-up approach complements this field's current top-down approach which emphasizes algorithmic or theoretic improvements. Together, we hope an industrial strength verification tool will emerge in the very near future and model checking will become a yet-another-useful-tool in the designers' tool-box.

Bibliography

[1] ABADI, M., AND LAMPORT, L. Composing specifications. *ACM Transactions on Programming Languages and Systems 15*, 1 (Jan. 1993), 73–132.

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

[3] AKERS, S. B. Functional testing with binary decision diagrams. In *Proceedings of Eighth Annual International Conference on Fault-Tolerant Computing* (June 1978), pp. 75–82.

[4] ASHAR, P., AND CHEONG, M. Efficient breadth-first manipulation of binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design* (November 1994), pp. 622–627.

[5] BAPNA, D., ROLLINS, E., MURPHY, J., AND MAIMONE, M. The Atacama Desert trek - outcomes. In *Proceedings of the 1998 International Conference on Robotics and Automation* (May 1998), pp. 597–604.

[6] BENSALEM, S., BOUAJJANI, A., LOISEAUX, C., AND SIFAKIS, J. Property preserving simulations. In *Proceedings of the Computer Aided Verification* (June 1992), pp. 260–273.

[7] BERNARD, D. E., DORAIS, G. A., FRY, C., JR., E. B. G., KANEFSKY, B., KURIEN, J., MILLAR, W., MUSCETTOLA, N., NAYAK, P. P., PELL, B., RAJAN, K., ROUQUETT, N., SMITH, B., AND WILLIAMS, B. Design of the remote agent experiment for spacecraft autonomy. In *Proceedings of the 1998 IEEE Aerospace Conference* (March 1998), pp. 259–281.

[8] BERTHET, C., COUDERT, O., AND MADRE, J. C. New ideas on symbolic manipulations of finite state machines. In *1990 IEEE Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors* (September 1990), pp. 224–227.

[9] BIERE, A. ABCD: an experimental BDD library, 1998.
`http://iseran.ira.uka.de/~armin/abcd/`.

[10] BOSE, S., AND FISHER, A. L. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In *International Workshop on Applied Formal Methods for Correct VLSI Design* (November 1989).

[11] BRACE, K., RUDELL, R., AND BRYANT, R. E. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference* (June 1990), pp. 40–45.

[12] BRGLEZ, F., BRYAN, D., AND KOZMISKI, K. Combinational profiles of sequential benchmark circuits. In *1989 International Symposium on Circuits And Systems* (May 1989), pp. 1924–1934.

[13] BRGLEZ, F., AND FUJIWARA, H. A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran. In *1985 International Symposium on Circuits And Systems* (June 1985). Partially described in F. Brglez, P. Pownall, R. Hum. Accelerated ATPG and Fault Grading via Testability Analysis. In *1985 International Symposium on circuits and Systems*, pages 695-698, June 1985.

[14] BROCK, B., KAUFMANN, M., AND MOORE, J. S. ACL2 theorems about commercial microprocessors. In *Proceedings of the Formal Methods on Computer-Aided Design* (November 1996), pp. 275–293.

[15] BROWNE, M. C., CLARKE, E. M., AND DILL, D. L. Checking the correctness of sequential circuits. In *1985 IEEE Proceedings of the International Conference on Computer Design* (October 1985), pp. 545–548.

[16] BROWNE, M. C., CLARKE, E. M., DILL, D. L., AND MISHRA, B. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers C-35*, 12 (December 1986), 1035–1044.

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

[18] BRYANT, R. E. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *Proceedings of the International Conference on Computer-Aided Design* (November 1995), pp. 236–243.

[19] BURCH, J. R., CLARKE, E. M., LONG, D. E., MCMILLAN, K. L., AND DILL, D. L. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13*, 4 (April 1994), 401–424.

[20] CHEN, Y.-A., AND BRYANT, R. E. Verification of floating-point adders. In *Proceedings of the Computer Aided Verification* (June 1998), pp. 488–499.

[21] CLARKE, E., GRUMBERG, O., AND PELED, D. *Model Checking.* MIT Press, November 1999. To be published.

[22] CLARKE, E. M., AND EMERSON, E. A. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, vol. 131 of *Lecture Notes in Computer Science.* Springer Verlag, Yorktown Heights, NY, May 1981, pp. 244–263.

[23] CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems 1*, 2 (April 1986), 244–263.

[24] CLARKE, E. M., FILKORN, T., AND JHA, S. Exploiting symmetry in temporal logic model checking. In *Proceedings of the Computer Aided Verification* (June 1993), pp. 450–462.

[25] CLARKE, E. M., GRUMBERG, O., AND LONG, D. E. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems 16*, 5 (September 1994), 1512–1542.

[26] CLARKE, E. M., GRUMBERG, O., MCMILLAN, K. L., AND ZHAO, X. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference* (June 1995), pp. 427–432.

[27] CLARKE, E. M., LONG, D. E., AND MCMILLAN, K. L. A language for compositional specification and verification of finite state hardware controllers. In *9th International Symposium on Computer Hardware Description Languages and their Applications* (June 1989), pp. 281–295.

[28] CLARKE, E. M., MCMILLAN, K. L., ZHAO, X., FUJITA, M., AND YANG, J. C.-Y. Spectral transform for large Boolean functions with application to technology mapping. In *Proceedings of the 30th ACM/IEEE Design Automation Conference* (June 1993), pp. 54–60.

[29] CONWAY, M. Proposal for an UNCOL. *Communications of the ACM 1*, 10 (Oct. 1958), 5–8.

[30] COUDERT, O., BERTHET, C., AND MADRE, J. C. Verification of sequential machines using Boolean functional vectors. In *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design* (November 1989), pp. 179–196.

[31] COUDERT, O., AND MADRE, J. C. A unified framework for the formal verification of circuits. In *Proceedings of the International Conference on Computer-Aided Design* (Feb 1990), pp. 126–129.

[32] COUDERT, O., MADRE, J. C., AND TOUATI, H. *TiGeR Version 1.0 User Guide.* Digital Paris Research Lab, December 1993.

[33] DAMS, D., GERTH, R., AND GRUMBERG, O. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems 19*, 2 (March 1997), 253–291.

[34] DAMS, D., GRUMBERG, O., AND GERTH, R. Generation of reduced models for checking fragments of CTL. In *Proceedings of the Computer Aided Verification* (June 1993), pp. 479–490.

[35] DILL, D. L., AND CLARKE, E. M. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings E (Computers and Digital Techniques) 133*, 5 (September 1986), 267–282.

[36] EMERSON, E. A. *Branching Time Temporal Logic and the Design of Correct Concurrent Programs.* PhD thesis, Harvard University, 1981.

[37] EMERSON, E. A., AND SISTLA, A. P. Symmetry and model checking. *Formal Methods in System Design 9*, 1-2 (August 1996), 103–131.

[38] FOR BUILDING CUSTOMIZED PROGRAM ANALYSIS TOOLS, A. A. S. Verifying properties of large sets of processes with network invariants. In *ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (June 1994), pp. 196–205.

[39] FUJITA, M., MATSUNGA, Y., AND KAKUDA, T. On variable ordering of binary descision diagrams for the application of multi-level synthesis. In *Proceedings of the European Design Automation Conference* (1991), pp. 50–54.

[40] GEIST, D., AND BEER, I. Efficient model checking by automated ordering of transition relation partitions. In *Proceedings of the Computer Aided Verification* (June 1994), pp. 299–310.

[41] GODEFROID, P., AND PIROTTIN, D. Refining dependencies improves partial-order verification methods. In *Proceedings of the Computer Aided Verification* (July 1993), pp. 438–449.

[42] GODEFROID, P., AND PIROTTIN, D. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the Computer Aided Verification* (June 1994), pp. 377–390.

[43] GORDON, M. J. C. HOL: a proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis* (January 1986), pp. 73–128.

[44] GRAF, S., AND STEFFEN, B. Compositional minimization of finite state systems. In *Proceedings of the Computer Aided Verification* (June 1990), pp. 186–196.

[45] GRUMBERG, O., AND LONG, D. E. Model checking and modular verification. In *2nd International Conference on Concurrency Theory* (August 1991), pp. 250–265.

[46] HETT, A., DRECHSLER, R., AND BECKER, B. MORE: Alternative implementation of BDD-packages by multi-operand synthesis. In *Proceedings of the European Design Automation Conference* (September 1996), pp. 16–20.

[47] HU, A. J., AND DILL, D. L. Reducing BDD size by exploiting functional dependencies. In *Proceedings of the 30th ACM/IEEE Design Automation Conference* (June 1993), pp. 266–71.

[48] IP, C. N., AND DILL, D. L. Better verification through symmetry. *Formal Methods in System Design 9*, 1-2 (August 1996), 41–75.

[49] JANSSEN, G. *The Eindhoven BDD Package*. University of Eindhoven. Anonymous FTP address: `ftp://ftp.ics.ele.tue.nl/pub/users/geert/bdd.tar.gz`.

[50] JOKSKO, B. Verifying the correctness of AADL modules using model checking. In *REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness.* (May 1989), pp. 386–400.

[51] JONES, C. B. Specification and design of (parallel) programs. In *IFIP 9th World Computer Congress* (September 1983), pp. 321–332.

[52] KURSHAN, R. P. Analysis of discrete event coordination. In *REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness.* (May 1989), pp. 414–453.

[53] LARSEN, K. G. Modal specifications. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems* (June 1989), pp. 232–246.

[54] LIN, B., AND NEWTON, A. R. Exact redundant state registers removal based on binary decision diagrams. *IFIP Transactions A, Computer Science and Technology A*, 1 (August 1991), 277–86.

[55] LONG, D. E. ROBDD Package, November 1993.

[56] LONG, D. E. The design of a cache-friendly BDD library. In *Proceedings of the International Conference on Computer-Aided Design* (November 1998), pp. 639–645.

[57] MANNE, S., GRUNWALD, D., AND SOMENZI, F. Remembrance of things past: Locality and memory in BDDs. In *Proceedings of the 34th ACM/IEEE Design Automation Conference* (June 1997), pp. 196–201.

[58] MCMILLAN, K. L. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[59] MCMILLAN, K. L. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *Proceedings of the Computer Aided Verification* (June 1998), pp. 110–121.

[60] MINATO, S., ISHIURA, N., AND JAJIMA, S. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference* (June 1990), pp. 52–57.

[61] MISHRA, B., AND CLARKE, E. M. Hierarchical verification of asynchronous circuits using temporal logic. *Theoretical Computer Science 38*, 2-3 (June 1985), 269–291.

[62] MISRA, J., AND CHANDY, K. M. Proofs of networks of processes. *IEEE Transactions on Software Engineering SE-7*, 4 (July 1981), 417–426.

[63] OCHI, H., ISHIURA, N., AND YAJIMA, S. Breadth-first manipulation of SBDD of Boolean functions for vector processing. In *Proceedings of the 28th ACM/IEEE Design Automation Conference* (June 1991), pp. 413–416.

[64] OCHI, H., YASUOKA, K., AND YAJIMA, S. Breadth-first manipulation of very large binary-decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design* (November 1993), pp. 48–55.

[65] OWRE, S., RUSHBY, J. M., AND SHANKAR, N. PVS: a prototype verification system. In *11th International Conference on Automated Deduction* (June 1992), pp. 748–752.

[66] PIXLEY, C. Introduction to a computational theory and implementation of sequential hardware equivalence. In *Proceedings of the Computer Aided Verification* (June 1990), pp. 54–64.

[67] PRESS, W. H., TEUKOISKY, S. A., VETTERING, W. T., AND FLANNERY, B. P. *Numerical Recipes in C*, second ed. Cambridge University Press, 1992.

[68] RAJAN, S., SHANKAR, N., AND SRIVAS, M. K. An integration of model checking with automated proof checking. In *Proceedings of the Computer Aided Verification* (July 1995), pp. 84–97.

[69] RANJAN, R. K., AZIZ, A., BRAYTON, R. K., PLESSIER, B., AND PIXLEY, C. Efficient BDD algorithms for FSM synthesis and verification. Presented in the IEEE/ACM International Workshop on Logic Synthesis, May 1995.

[70] RANJAN, R. K., AND SANGHAVI, J. CAL-2.0: Breadth-first manipulation based BDD library. Public software. University of California, Berkeley, CA, June 1997. `http://www-cad.eecs.berkeley.edu/Research/cal_bdd/`.

[71] RANJAN, R. K., SANGHAVI, J. V., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. High performance BDD package based on exploiting memory hierarchy. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference* (June 1996), pp. 635–640.

[72] RUDELL, R. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design* (November 1993), pp. 139–144.

[73] SENTOVICH, E. M. A brief study of BDD package performance. In *Proceedings of the Formal Methods on Computer-Aided Design* (November 1996), pp. 389–403.

[74] SENTOVICH, E. M., AND HORIA TOMA, G. B. Latch optimization in circuits generated from high-level descriptions. In *Proceedings of the International Conference on Computer-Aided Design* (November 1996), pp. 428–35.

[75] SENTOVICH, E. M., SINGH, K. J., LAVAGNO, L., MOON, C., MURGAI, R., SALDANHA, A., SAVOJ, H., STEPHAN, P. R., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI., A. L. SIS: A system for sequential circuit synthesis. Tech. Rep. UCB/ERL M92/41, Electronics Research Lab, University of California, May 1992.

[76] SHIPLE, T. R., HOJATI, R., SANGIOVANNI-VINCENTELLI, A. L., AND BRAYTON, R. K. Heuristic minimization of BDDs using don't cares. In *Proceedings of the 31st ACM/IEEE Design Automation Conference* (June 1994), pp. 225–231.

[77] SHUREK, G., AND GRUMBERG, O. The modular framework of computer-aided verification. In *Proceedings of the Computer Aided Verification* (June 1990), pp. 214–223.

[78] SOMENZI, F. CUDD: CU decision diagram package. Public software. University of Colorado, Boulder, CO, April 1997. `http://vlsi.colorado.edu/~fabio/`.

[79] VALMARI, A. A stubborn attack on state explosion. In *Proceedings of the Computer Aided Verification* (June 1990), pp. 156–165.

[80] VAN EIJK, C. A. J., AND JESS, J. A. G. Exploiting functional dependencies in finite state machine verification. In *Proceedings of European Design and Test Conference* (March 1996), pp. 266–71.

[81] WILLIAMS, B. C., AND NAYAK, P. P. A model-based approach to reactive self-configuring systems. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference* (August 1996), pp. 971–978.

[82] YANG, B., BRYANT, R. E., O'HALLARON, D. R., BIERE, A., COUDERT, O., JANSSEN, G., RANJAN, R. K., AND SOMENZI, F. A performance study of BDD-based model checking. In *Proceedings of the Formal Methods on Computer-Aided Design* (November 1998), pp. 255–289.

[83] YANG, B., CHEN, Y.-A., BRYANT, R. E., AND O'HALLARON, D. R. Space- and time-efficient BDD construction via working set control. In *Proceedings of Asia and South Pacific Design Automation Conference* (February 1998), pp. 423–432.

[84] YANG, B., AND O'HALLARON, D. R. Parallel breadth-first BDD construction. In *Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (June 1997), pp. 145–156.

[85] YANG, B., SIMMONS, R., BRYANT, R. E., AND O'HALLARON, D. R. Optimizing symbolic model checking for constraint-rich models. Tech. Rep. CMU-CS-99-118, School of Computer Science, Carnegie Mellon University, March 1999.

[86] YANG, B., SIMMONS, R., BRYANT, R. E., AND O'HALLARON, D. R. Optimizing symbolic model checking for constraint-rich models. In *Proceedings of the Computer Aided Verification* (June 1999). To Appear.