1-1-2014

Efficient External-Memory Graph Search for Model Checking

Peter C. Lamborn

Efficient external-memory graph search for model checking

By

Peter C. Lamborn

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

May 2014

Efficient external-memory graph search for model checking

By

Peter C. Lamborn

Approved:

Eric Hansen
(Major Professor)

Donna S. Reese
(Committee Member)

Edward B. Allen
(Committee Member and Graduate Coordinator)

Susan M. Bridges
(Committee Member)

Sherif Abdelwahed
(Committee Member)

Achille Messac
Dean
College of Engineering

Name: Peter C. Lamborn

Date of Degree: May 16, 2014

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Eric Hansen

Title of Study: Efficient external-memory graph search for model checking

Pages of Study: 120

Candidate for Degree of Doctor of Philosophy

Model checking problems suffer from state space explosion. State space explosion is the number of states in the graph increases exponentially with the number of variables in the state description. Searching the large graphs required in model checking requires an efficient algorithm. This dissertation explores several methods to improve an external-memory search algorithm for model checking problems. A tool implementing these methods is built on top of the Murphi model checker. One improvement is a state cache for immediate detection leveraging the properties of state locality. A novel type of locality, intralayer locality is explained and shown to exist in a variety of search spaces. Another improvement, partial delayed duplicate detection, exploits interlayer locality to reduce search times. An automatic partitioning function is described that allows hash-based delayed duplicate detection to be used without domain knowledge of the state space. A phased delayed duplicate detection algorithm combining features of hash-based delayed duplicate

detection and sorting-based delayed duplicate detection is explained and compared to the other methods.

Key words: Model Checking, Search, External Memory, Delayed Duplicate Detection, Sorting-based Delayed Duplicate Detection, Hash-based Delayed Duplicate Detection, Interlayer Locality, Intralayer Locality, State Cache, Immediate Duplicate Detection, Phased Delayed Duplicate Detection, Partial Delayed Duplicate Detection, Automatic Hash Function

DEDICATION

To Kristi for her unconditional love and support.

To Eliza for her vivacity.

To Max for his determination.

To Hyrum for his energy.

To Clara for her joy.

To my father and grandfather for their advice.

To Aaron, Ben, Charolette, Denali, Doug, Emily, Emma, Gwen, Isaac, Heidi, Kade, Katie, Maggie, Peg, Reed, Ridge, Ryan, Ruth, Sara, Sarah, Steven, and Trek for their prayers.

To Martha and Jud for opening up their home and their hearts.

ACKNOWLEDGEMENTS

The author would like to acknowledge the financial support received from many sources, including a NASA Mississippi Space Grant Consortium Fellowship, a Bagley Fellowship, and a Barrier Engineering Graduate Fellowship.

The author would like to acknowledge the influence of his advisor on the ideas described in this dissertation.

The author thanks all those who have contributed to the Murϕ model checker.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

VOCABULARY

Bucket A set of states with the same value by a given hash function

Cache Distance Number of unique nodes generated between two states

Candidate Set All candidate states in a graph

Candidate State Generated state that may be duplicate

Child State Destination of transition in state space

Closed Set Set of all previously expanded states

Closed State Previously expanded state

Counter Example Path to error state

DDD Delayed duplicate detection

Depth States distance from start state

Duplicate Generated state that already exists in closed set

Error Violation of protocol being verified

Error State Vertex in graph exhibiting error behavior

Expand Process to generate children states for given parent state

FSM Finite State Machine

Hash-based DDD DDD the removes duplicates with a hash table

IDD Immediate Duplicate Detection

Interlayer Duplicate Duplicate state reached by a interlayer edge

Interlayer Edge Transition resulting in duplicate state of a previous layer

Interlayer Edge Distance Number of layers separating a duplicate from the original state

Interlayer Locality The principle that short interlayer edge distances are more common than long interlayer edge distances

Intralayer Duplicate A duplicate of another state in the same layer

Intralayer Locality The principle that within a single layer, duplicate states exhibit short cache distances

Layer Set of all states at same depth

Locality Maximum Interlayer Edge Distance in the Graph

LRU Least Recently Used

Open Set Set of open states

Open State Unique state that has not yet been expanded

Parent State Source of transition in state space

Partial DDD DDD that does not use the all the layers in the closed set

Predecessor Scope Abstract states that are parents of given abstract state

Phase States that sort between a given start and end state

Phased DDD A sorting-based DDD that eliminates duplicates in phases

SDD Structured duplicate detection

State Bit-vector encoding configuration of system at particular instant

State Space Set of all reachable configurations of system

State-Space Explosion The number of reachable states in the graph tends to increase exponentially with the number of variables in the state description

Sorting-based DDD DDD that removes duplicates from sorted sets

Successor Scope Abstract states that are children of given abstract state

CHAPTER 1

INTRODUCTION

This dissertation describes an external-memory breadth-first graph search algorithm that can be applied to model checking. The fundamental algorithm, breadth-first graph search has many applications, but this dissertation focuses on model checking for several reasons. First, model checking typically requires exhaustive searching of very large graphs. Second, implementation of a search algorithm in a model checker requires development of an automatic and domain-independent approach to graph search because the model checker accepts different models as input. This model-checking application allows the search algorithm to be tested on graphs of different sizes and properties. Third, the problem of model checking has considerable scientific and commercial importance.

Model checking is a methodology for verifying that property X is satisfied, or modeled, by transition system M. The transition system is often a protocol such as for wireless communication or an embedded hardware device. A violation of the property creates an error in the protocol that needs to be detected and corrected. For example, in a wireless protocol, an error could mean the irrecoverable loss of a packet. It is advantageous to use model checking to find certain errors during the design phase that are difficult to detect through testing and simulation, particularly to find errors that occur after long or unusual

sequences of events. Unlike simulation and testing, model checking can also guarantee the absence of an error. Model checking has proven to be an effective and practical tool widely used to test and verify safety-critical systems.

The external-memory breadth-first search algorithm described in this dissertation is implemented in Murϕ [15, 16], a well-known model checker that takes the specification of a model and a property as input and verifies that the model satisfies the property by exhaustively searching the state-space graph representing all possible behaviors of the model. The original Murϕ model checker includes an implementation of an internal-memory breadth-first search algorithm that is limited in the size of models it can verify by the size of RAM. It also includes implementation of a primitive, inefficient external-memory breadth-first search developed more than fifteen years ago [16]. Many improved methods for external-memory graph search have been developed over the past few years, and they provide a starting point for the implementation described in this dissertation.

Breadth-first graph search is a memory-intensive algorithm, since all nodes of the graph typically need to be stored in memory in order to detect duplicate nodes. On a typical desktop computer, a breadth-first search algorithm can fill RAM within minutes. For large graphs, an external-memory search algorithm is needed to store visited nodes of the graph on disk. The largest models tested for this dissertation require more than two hundred gigabytes of disk storage and days of computer time to search the entire graph.

Over the past few years, developers have shown great interest in building external-memory graph-search algorithms, not only for model checking but in the fields of artificial intelligence search algorithms, high-performance computing, and even computational

group theory. One goal of this dissertation is to show that new techniques for scalable external-memory graph search developed in the artificial intelligence search community can be applied to model checking. In addition, this dissertation describes several original techniques for improving the efficiency of external-memory graph search.

1.1 Organization

The dissertation is organized as follows. Chapter 2 describes relevant background information about the relationship between model checking and graph search. Chapter 3 describes improvements of a breadth-first search algorithm for external-memory model checking that uses a sorting-based approach to duplicate detection, including use of a state cache in the form of a hash table in RAM that stores frequently accessed states that would otherwise be stored on disk. It also describes a more efficient method for merging sorted files and explains an approach to partial checking of the set of closed nodes on disk that leverages the storage of closed nodes in breadth-first layers. Chapter 4 describes a novel approach to automatic partitioning that allows hash-based delayed duplicate detection (DDD) to be implemented in a model checker. Chapter 4 also introduces phased delayed duplicate detection, which combines some aspects of sorting-based and hash-based DDD. Chapter 5 concludes the dissertation, summarizes the contributions of this work, and proposed future work.

CHAPTER 2

BACKGROUND

This background chapter provides an overview of model checking, the relationship between model checking and graph search, and previous work on search algorithms for external-memory model checking. Much of this information can be found in model checking textbooks [12, 37]. This chapter also reviews the Murϕ model checker [16] that serves as a platform for implementing the search algorithms developed in this dissertation and describes the models used as test cases. This information provides a starting point from which to explain the significance of this dissertation. Additional relevant background that provides context for specific contributions will be reviewed in later chapters.

2.1 Model Checking

Model checking provides an automated approach to analysis and verification of reactive systems that have a finite state space or finite-state abstractions. The term *reactive system* refers to a system that maintains an ongoing interaction with its environment and typically does not terminate, in contrast to a traditional program that produces an output and terminates. Examples of reactive systems include process controllers, operating systems, and communication networks. In practice, model checking has been successfully

used to verify reactive systems including sequential circuit designs and communication protocols.

A reactive system typically consists of several components operating in parallel and communicating via messages (in the case of distributed systems) or shared variables (in the case of concurrent systems). Concurrency makes reactive systems especially difficult to analyze and debug. For applications where errors are expensive or potentially catastrophic, it is often cost effective to formally analyze the hardware or software components of a reactive system in order to detect errors in the design phase before deployment.

Model checking and theorem proving are the two main approaches to formal verification. Theorem proving involves showing that a system meets its specification by constructing a proof based on axioms and inference rules in a deductive system, such as temporal logic. Due to the complexity of mechanical theorem proving, however, theorem-proving software is typically interactive and requires human expertise and help at many steps along the way, and proof construction can take days or months to complete.

Where theorem proving is based on proof-theoretic reasoning, model checking adopts model-theoretic reasoning as an approach to formal verification. A model checker takes as input a finite-state description of the analyzed system's behavior and the specification of one or more properties expected to hold for the system. (Specifications are often written as formulas in temporal logic, which is particularly useful for expressing concurrency properties.) To show that a system meets its specification, a model checker shows that the system is a model, in the semantic sense, of the temporal logic formulae. An advantage of model checking over theorem proving is that it is much more easily automated, espe-

cially for finite-state concurrent systems. A model checker can determine whether a model meets its specification simply by performing an exhaustive search of all execution paths in the model. If the model does not meet the specification, the model checker produces a counterexample execution trace that can be used to help discover why the specification does not hold.

To understand model checking, one must understand how to model a reactive system as a transition system, how to specify the properties the reactive system should satisfy, and how to algorithmically check whether these properties are satisfied using reachability analysis in a state-space graph. The following section reviews each of these aspects of model checking in turn.

2.1.1 Transition Systems

A reactive system can be formally represented as a transition system, which is a tuple $M = (S, S_0, R, AP, L)$ with the following components:

- S is a nonempty, finite set of states (the state space);
- $S_0 \subseteq S$ is a nonempty subset of initial states;
- $R \subseteq S \times S$ is a transition relation;
- AP is a set of atomic propositions, i.e., Boolean expressions over variables, constants, and predicate symbols; and
- $L : S \to 2^{AP}$ is an *interpretation*, which is a function that maps each state in S to the set of atomic propositions that are true in that state. (The notation 2^{AP} represents the powerset of AP.)

Intuitively, a transition system defines the possible behaviors of a system. Starting from an initial state, the system evolves by making a sequence of transitions from one state to

another. When more than one transition can be made in a given state, the transition is selected nondeterministically. If the set S_0 contains more than one initial state, the initial state is also selected nondeterministically. Given a total transition relation, the system is nonterminating.

A transition system corresponds to a directed graph, where the nodes represent the system states and the arcs represent possible transitions between states. As a result, a transition system is sometimes called a *state transition graph* or, more precisely, a *labeled state transition graph*. It differs from a simple graph in that the nodes are labeled with propositions that represent atomic properties that hold in the corresponding state. The interpretation function is denoted L because it assigns these labels. The state transition graph is equal to a finite automaton, and because a node can have multiple outgoing edges that correspond to multiple possible transitions from a state, it is a nondeterministic automaton. A transition system can also sometimes be called a *Kripke structure*, after the logician Saul Kripke, since properties of the system are expressed as formulae in temporal logic of which the state transition system is to be a model, and temporal logics are traditionally interpreted in terms of Kripke structures.

Transition systems serve as simple, low-level representations with none of the semantic complications of high-level programming languages. But, of course, constructs such as branches, loops, and even procedure calls can be modeled within a transition system using explicit control variables. In practice, reactive systems are usually described in a higher-level modeling language specific to a model checker, which, in the case of Murϕ, is a pseudo-programming language described in Section 2.2. Even though the model is

described in a pseudo-programming language, the operational semantics of the modeling language are defined in terms of transition systems. Note that the transition system that corresponds to the model's description in such a language typically has a size exponential to the length of the description.

2.1.2 Temporal Properties

The interpretation L in a finite-state transition system defines local properties of states. Often one is also interested in global properties of the transitional behavior. For example, one might be examining reachability properties, such as, "Can we reach from an initial state, a state where the atomic proposition P holds?" These more complicated global properties are called *temporal properties*, where a temporal property is defined semantically as a set of behaviors (infinite sequences of states) that represent those executions of the system that satisfy the property. In this definition, note that a behavior is understood to be a sequence of states, and a temporal property is a predicate on behaviors.

Temporal logics are logical formalisms designed to express such properties. Several variants of propositional temporal logic have been used to express temporal properties - including linear temporal logic and branching time temporal logic - and different model checking tools have been developed to support different variants of temporal logic. In this dissertation, details of different temporal logics and how they can be used to express temporal properties are avoided in favor of a high-level summary of different types of temporal properties.

Safety and liveness properties are the most important temporal properties, and although this dissertation focuses on safety properties, both will be briefly discussed. A safety property is defined as a property in which some error state never occurs. *Deadlock* is an important example of an error state in protocols, especially mutual exclusion protocols, and refers to a protocol that reaches a situation with no possible egress. Protocols often share resources, and mutual exclusion protocols attempt to avoid deadlock in accessing those shared resources.

In a famous example of mutual exclusion, dining philosophers sit at a table eating or thinking deep thoughts. To commence eating, a philosopher must pick up first one fork and then another. When finished eating, the philosopher replaces the forks one at a time as well. To make this a mutual exclusion example, there must be too few forks for all philosophers to dine at the same time, so the philosophers could reach a deadlock condition. If all philosophers have one fork and desire a second fork, and there is no provision for one philosopher preempting another philosopher's right to the fork, the situation is deadlocked. There is no possibility to reach another state, so an error condition has been reached.

Model checking can be used to determine whether a system is deadlock free, as well as to check other safety properties. Other examples of error states that can be checked as safety properties include buffer overflow or file corruption. (Note that if a reactive system fails to satisfy a safety property, a finite execution trace reveals this fact.)

Where a safety property can be described as a property for which "something bad never happens," a *liveness property* can be described as a property for which "something good eventually happens." For example, momentarily being unable to send a token because of

network conflicts is fine, as long as the token can eventually be sent. If the token can never be sent, that is a liveness error. Other examples of liveness properties include livelock and message denial.

The dining philosophers can provide an example of livelock as well. If the philosophers arrive at a point where they each hold a single fork and no other forks are available, each can place his or her fork back on the table - a reasonable way to avoid the deadlock described previously. Let us consider just two philosophers, both holding one fork in the previous deadlock state. Our new rule causes both philosophers to put down the forks they have. Now there are two free forks. The philosophers each pick one up, and we reach the previous error state of no available forks, so the philosophers again replace their forks. This process can repeat indefinitely; as defined, the rule will not prevent this cycle. A cycle where neither philosopher ever eats is livelock, but remember, a liveness property is defined as "something good eventually happens." In this case, the philosophers being able to eat is good, but we have found an infinite path where neither philosopher ever eats, violating the liveness property.

A later section of this dissertation will demonstrate that checking liveness properties is more complex than checking safety properties, partially because, if a system fails to satisfy a liveness property, the counterexample found by a model checker does not take the form of a simple execution trace.

In some cases, it is important for liveness properties to satisfy *fairness conditions*, but although this concept is important in model checking, it is not relevant to this dissertation.

2.1.3 Reachability Analysis and Graph Search

A model checker decides whether transition system M is a model of a temporal logic formula ϕ, i.e., whether M satisfies ϕ, abbreviated $M \models \phi$. If so, the transition system satisfies the property specified by the temporal logic formula.

This decision procedure can be performed by doing a *reachability analysis*, which involves using a search algorithm to traverse the state-space graph of the transition system, beginning from an initial state. As mentioned earlier, a *temporal property* is defined as a set of behaviors (where a behavior is an infinite sequence of states) that represent those executions of the system that satisfy the property. A *path* in transition system M is a nonempty sequence of states $\pi = (s_0, s_1, ...)$, such that $s_0 \in S_0$ and $\forall i \geq 0, (s_i, s_{i+1}) \in R$. A state $t \in S$ is said to be *reachable* in M if there is a path in M from some initial state to t. The *reachable state space*, the set of states reachable from an initial state, is a subset of the state space and can only be determined by search.

The temporal properties checked by a model checker can be defined in terms of reachability. Thus, by exploring the states of a transition system reachable from an initial state, one can verify both safety properties and liveness properties.

Safety properties can be checked by traversing the reachable state space and showing that an error state that represents a violation of the safety property can never be reached. For example, in the graph of a transition system, deadlock corresponds to a node with either no outgoing edges or with all outgoing edges pointing back to itself. Reachability analysis can establish whether or not such a state exists in the state transition graph and is reachable from an initial state. Reachability analysis can be performed using depth-first,

breadth-first, or best-first search, and all have been used in model checking. If the search algorithm finds an error state, it terminates and returns the path from an initial state to the error state. The path (or execution trace) can be used to help diagnose the error. When a best-first search algorithm is used to traverse the state space, it employs a heuristic to guide the search toward an error state; in this approach, error states can be viewed as goal states for the search algorithm [21].

The state-space graph of the transition system is not given explicitly to the model checker. Instead, it is defined implicitly by a set of transition rules and a nonempty set of initial states. Given the transition rules, the model checker can generate the successor states of any state. In model checking, the search algorithm does this on the fly, which means the model checker generates the state-space graph in the course of traversing the graph. The search algorithm begins with a queue that contains the start state. The algorithm removes states from the queue in an order that depends on the search strategy (for example, depth-first, breadth-first, or best-first) and generates the successors of each state. In a graph, a successor state could be a duplicate of a state that has already been generated, since the same state can be reached by different paths in a graph. To detect duplicate states, the search algorithm stores previously generated states. In standard, internal-memory search, the algorithm stores previously generated states in a hash table and compares each newly generated successor state to the set of previously generated states to determine if it is a duplicate. A successor state is considered a *candidate* until it is proven to be unique, then it is added to both the queue and the hash table. The process of checking for duplicate states is called *duplicate detection*.

This dissertation employs breadth-first search to verify safety properties. Unlike depth-first search, breadth-first search is better able to detect duplicate states when searching a graph, an important advantage since this research focuses on how best to detect duplicate states when a graph is so large the graph-traversal algorithm must use disk memory as well as RAM. Other advantages of breadth-first search include easier parallelization and simpler algorithms for using disk memory, when compared to depth-first search or best-first search. Breadth-first search is also guaranteed to return the shortest counterexample.

Simple breadth-first traversal of a state transition graph can verify safety properties, but liveness properties usually require a more complex search strategy. A liveness property specifies that something good eventually happens. Thus, it is not enough to check for a single reachable state in an error condition. Instead, one must check for a cycle of states that includes failure of the property.

Traditional search for liveness properties includes two stages: one, to find error states, and two, to search for a cycle that includes an error state. One way to do this is with *nested depth-first search algorithm* [14], which launches a secondary search at each detected error state. If the secondary search finds a cycle that returns to the error state, a violation of the liveness property has been found. Schuppan and Biere [70, 71] found a way to reduce checking for liveness properties to checking for safety properties. The Schuppan and Biere algorithm works by keeping track of two states for every state. The second state, which is blank for the start state, represents an ancestor error state of the current state. This representation reduces checking for liveness properties to simple graph traversal, without any secondary search. When a state and its error ancestor match, a cycle containing the error

state has been discovered. The Schuppan and Biere reduction allows breadth-first search to be used to check for liveness properties, whereas traditionally, checking for liveness properties requires depth-first search. Breadth-first search lends itself much more easily to duplicate detection than depth-first search, so it is often more efficient. But storing a double state for each state in the graph doubles the memory requirements of the search algorithm. Moreover, states may have multiple error state ancestors, and each additional ancestor creates an additional double state to search and store. Thus, Schuppan and Biere's algorithm can suffer from a dramatic increase in space requirements compared to simply checking for safety properties.

The Murϕ model checker used in this research only checks for safety properties. However, it could also be used to check for liveness properties if the reduction described by Schuppan and Biere were implemented in the Murϕ software. Because Schuppan and Biere's algorithm more than doubles the storage requirements of the search, the space scalability provided by the algorithm presented in this dissertation would be especially helpful.

2.1.4 Purposes of Model Checking

Model checking is used to eliminate error conditions in a transition system. After a transition system is designed, model checking would be used to find error states. As each error state is found, the transition system would be redesigned to avoid the detected error. The goal is to create a transition system without error states. This is an iterative process where many models with error states are searched before a final model with no errors is verified.

The purpose of a model checking tool is two fold. First, it should detect error states. Second, it should verify that models are error free. The most useful model checking tools can detect errors and verify models. As will be explained further in Section 2.2.5, models with error states exhibit a different graph structure than models without error states. A model checking tool should be tested on models with and without error states, to ensure it works on both types of graphs. Tools that only work on models with errors (or only on models without errors) are not as universal or as useful as tools that work on both kinds of graphs.

2.1.5 State Explosion Problem

The combinatorial explosion of the reachable state space - even in moderately sized applications - presents an important problem pervasive in all model-checking tools. The exact size of the reachable state space is not known in advance, since it is a subset of the full state space that can only be determined by search. The number of reachable states in the graph tends to increase exponentially with the number of variables in the state description - called the state-space explosion problem. Since all reachable states must be generated and stored by the search algorithm, the problem of state-space explosion is the primary limiting factor of model checking and a major research challenge.

One approach developed to combat state-space explosion, called *symbolic model checking* [12], represents a transition system and a verification property as Boolean characteristic functions using a data structure called a binary decision diagram (BDD). It searches the reachable state space more efficiently using this compact representation. However,

symbolic model checking is not effective for all problems, including the protocol verification problems for which Murϕ was specially developed. Murϕ belongs to the class of *explicit-state model checkers*. In contrast to symbolic model checkers, explicit-state model checkers individually store each generated state; thus, limited memory is a more serious concern, and the motivation for external-memory model checking is stronger.

A variety of other methods have been used to reduce the state-space explosion problem. *Symmetry reduction* removes sets of states that are nontrivial permutations of other sets [12]. *Partial order reduction* exploits the commutativity property on state transitions to reduce the number of states explored [12]. *Abstraction* removes some details from the state's representation, thus reducing the required state space [12]. Not all of these techniques can be applied in every situation, though, and the problems model checking attempts to address continue to grow.

2.2 Murϕ Model Checker

The search algorithms developed in this dissertation research are implemented in the Murϕ model checker [15]. (Murϕ is pronounced Murphy and is sometimes spelled Murphi.) The Murϕ model checker was developed at Stanford University in the 1990s [16] as an academic tool, so the software is free to use and modify. In addition to its role in academic research, it has been widely used in industry. Murϕ was originally built to verify the design for the Stanford DASH multiprocessor [56]. Since then, it has been used in the analysis and verification of many finite-state and concurrent protocols, especially multiprocessor cache coherence protocols, link-level protocols, and cryptographic and security-

related protocols [24, 44, 55, 56, 62, 73, 69]. Indeed, it is often referred to as a protocol verification tool.

The Murϕ model checker consists of two components: (i) a language for describing models and specifications, and (ii) a compiler that uses lex and yacc to translate a model and specification written in the Murϕ description language into C++ code for a special-purpose verifier that automatically checks, by explicit-state enumeration, if all reachable states of the model satisfy the given specification. If not, Murϕ generates a counterexample in the form of an error trace that can be used for debugging.

Murϕ is able to detect deadlocks and to check invariants, which are Boolean expressions that have to be true in every state of a system. These are specified by the user in the input language of Murϕ. Additionally, Murϕ offers error and assert statements to detect design errors. However, it does not allow the expression of behavoirs directly in temporal logic. In the following section, different aspects of Murϕ will be considered in more detail.

2.2.1 High-Level Description Language

Murϕ models are specified in the Murϕ language, which is relatively simple and similar to C code. Four components are specified: variables in the state description, transitions between states, start states, and error conditions.

Each variable has a unique name. To specify a variable, users must specify the possible values that can be assigned to the variable. The data range can include enumerated sets, and variables can be arrays and complex objects containing multiple other variables. Murϕ represents all variables internally as a single bit vector, called the state vector, with each

variable mapped to portions of the bit vector. The first variable specified is placed furthest left, and the last variable specified is represented in the right-most bits. All other variables appear in between in the order specified. Each distinct combination of variable values is a unique state.

State transitions have two components, and the first is a Boolean formula on the state variables. If the Boolean statement for rule r is true for the variable values of state s, then rule r is a valid transition for state s. The second part of the transition appears in C-like code that specifies changes to one or more state variables to generate a child state. This code can include if statements and loops. Murϕ also supports functions that can be called by transitions, either in the Boolean formula or the C-like code. Transitions can be clustered into rule sets that differ only by the elements of an array they act upon. Multiple rules can be valid transitions for a single state, and when this happens in a reactive system, one transition is chosen nondeterministically. When breadth first search (BFS) is used, all valid transitions are applied to each state. Models must have at least one transition, and there is no maximum number of transitions.

Start states are specified with C-like code that initializes the variables in the state space. This code supports if statements and loops, and function calls can be made from start-state specifications. At least one start state must be included in every model, but multiple start states can be specified. In a reactive system, any start state is a valid entry point. When BFS is applied, all start states are included in layer zero.

Error conditions are represented as a Boolean formula on the state variables called invariants. Any combination of state variable values that makes the formula false represents

an error state. Multiple error conditions can be included in a single model, and a single error condition can be true in multiple states.

2.2.2 Dining Philosophers Example

We will now walk through an example Murϕ model of dining philosophers split across Figures 2.1, 2.2, 2.3, and 2.4. Figure 2.1 describes how variables are declared, Figure 2.2 covers state transitions, Figure 2.3 specifies start states, and the invariants in Figure 2.4 determine which states represent an error condition. This mutual exclusion algorithm for dining philosophers was created as a simple example. It was designed to be readable, demonstrate the more common uses of Murϕ, and have both a safety and a liveness error.

```
1.  Const
2.    N: 5;
3.
4.  Type
5.    p: 0..(N-1); – number of philosophers/forks
6.    f: 0..2; – maximum number of forks that can be held
7.
8.  Var
9.    forkTaken : Array [p] of boolean – list of the forks
10.   eating: Array [p] of boolean; – philosopher's state is eating or not
11.   forksInHand: Array[p] of f; – number of forks current held by philosopher
```

Figure 2.1

Variables for Dining Philosophers in Murϕ Language.

Figure 2.1 shows the variables in the model. This section of code is divided into constants, types, and variables. First, we declare the constants: in this model, the only constant

is the number of philosophers. According to this model's design, any number of philosophers can be modeled by changing just this single constant - a common design methodology in Murϕ models. In this example, 5 philosophers are used. Next, we declare types, which amount to data ranges and can be used as the number of elements in an array or the possible data values of a variable. One of the data ranges in this model refers to the number of forks a philosopher can be holding: 0, 1, or 2. The other range is for the number of philosophers. Smaller data ranges take less space to store, so it is worthwhile to specify the minimal range that will represent the behavior expressed. Finally, we declare the variables that will be tracked in this model: all variables are arrays over the range of p. There are some built-in data types; for example, the built-in Boolean data type is used in this model. Additionally, you can use custom data ranges that are defined in the types section, as we use for the forksInHand variable. All variables are globally visible within the state. The remaining portions of the model act on the variables described here.

For the rule declarations in Figure 2.2, rules specify state transitions. They have two parts, with the first determining if the transition is possible from a given state and the second specifying what effects the state transition has. In this model, all rules are part of rule sets wherein rules differ only by the value of a single element; in this example, it is element i. The rule sets are used here to apply the same rule to every element of an array; the variable i is used as an array index. The first part of the rule is a Boolean statement that evaluates to true or false, indicating whether the rule is enabled for this state. The first rule is enabled for i if forkTaken[i] is false. The second part of the rule specifies portions of the state to change when generating the child state. The first rule increments the number

```
1.  Ruleset i: p Do
2.    Rule “fork on right”
3.      !forkTaken[i] – if the fork on the right is not already taken
4.      ==>
5.    Begin
6.      forkTaken[i]:=true; – take the fork
7.      forksInHand[i]:=forksInHand[i]+1; – increment the number of forks being held
8.    End;
9.
10.   Rule “fork on left”
11.     !forkTaken[(i+1)%N] – if the fork on the left is not already taken
12.     ==>
13.   Begin
14.     forkTaken[(i+1)%N]:=true; – take the fork
15.     forksInHand[i]:=forksInHand[i]+1; – increment the number of forks being held
16.   End;
17.
18.   Rule “eat”
19.     forksInHand[i]=2 – two forks are required for eating
20.      ==>
21.   Begin
22.     eating[i]:=true;
23.   End;
24.
25.   Rule “finish eating”
26.     eating[i]
27.     ==>
28.   Begin
29.     forkTaken[i]:=false;
30.     forkTaken[(i+1)%N]:=false;
31.     eating[i]:=false;
32.     forksInHand[i]:=0;
33.   End;
34. End; –Ruleset
```

Figure 2.2

Rules for Dining Philosophers in Murϕ Language.

of forksInHand[i] and sets forkTaken[i] to true. The body of a rule will be treated as an atomic action, and all other portions of the state will remain unchanged. The modified state will be a child of the original state, and most states will have several active rules.

```
1. Startstate
2. Begin
3.   For i:p Do
4.     forkTaken[i]:=false; – all forks start on the table
5.     eating[i]:=false; – no one is eating in the start state
6.     forksInHand[i]:=0; – all hands are empty at start
7.   End;
8. End;
```

Figure 2.3

Start State for Dining Philosophers in Murϕ Language.

Figure 2.3 shows how start states are declared. Though not shown in this model, multiple start states are supported. The start state initializes the variables in the state. In this case, we initialize every variable, although it is possible to leave the value of some variables as undefined. A variable being undefined can even be referenced in rules and invariants. Variables are initialized with C-like code that matches in syntax the body of the rule declarations, and start states are the entry point of the reactive system.

Finally, Figure 2.4 shows how invariants are declared. Each invariant is a Boolean statement, and an error state is any state that causes an invariant to evaluate to false. The first invariant is a safety property. In this model, if we ever reach a state where every philosopher holds one fork, a deadlock has been reached with no possible egress from the

```
1.  Invariant ”Deadlock (Safety)”
2.  !(forall i: p do
3.    forksInHand[i]=1
4.    end);
5.
6.  Invariant “Never get to eat (liveness)”
7.  exits i: p do
8.    !eating[i]
9.    end;
```

Figure 2.4

Invariants for Dining Philosophers in Murϕ Language.

state. The first invariant tests for this deadlocked condition. The second invariant is for a liveness property and is true for every state that has at least one philosopher not eating, which is every state in this model. But the model only has a liveness error if a loop through the error state can be found.

2.2.3 State Vector Representation

This section describes how Murϕ internally represents a state. For our dining philosopher model, which is a toy problem, each variable described in Figure 2.1 has a value for every state in the model. The state vector records these values. The state vector is implemented as an array of unsigned characters. Despite the storage type, state variables are represented as binary values. Depending on the range of possible values, variables can span multiple character in the state vector.

Most of this dissertation focuses on similarities between model checking and other applications of search. But model checking has uniquely large state vectors. The dining

philosopher example in Figure 2.1 has 15 variables, 10 of these require a single bit of storage, and the other 5 require two bits. This example is not representative of the size of state vectors in model checking. Most models have hundreds of variables per state and those variables usually have much larger data ranges. Model checking problems have much larger state vectors than most search problems.

Comparing state vectors is expensive. State vectors are compared in many ways in the algorithms described in this dissertation. For example, states are compared to determine sorted order and states are compared to establish uniqueness. To compare two states means to compare the state vectors of those states. Since these state vectors are large, state comparisons are computationally expensive.

The size of the state vector affects state storage. In the algorithms in this dissertation, the states are stored in RAM and on disk. Because model checking states have large representations, storage is also a problem.

The default state representation of Murϕ uses more bits per variable than is strictly necessary, largely because it requires that each variable start on a byte boundary. But Murϕ includes an option for bit compaction, which is a form of lossless compression of the state representation that has several advantages. The bit compacted representation of our philosophers model does away with the requirement of starting variables on byte boundaries. Bit compaction, as implemented in Murϕ, stores variables in the minimal number of bits that is lossless. For the dining philosophers example, the standard representation requires 32 bytes. By contrast, when bit compaction is used the state vector requires four bytes. For this model bit compaction reduces the state vector by a factor of eight. The exact

amount of reduction provided by bit compaction varies by model, but is usually closer to a factor of four. The algorithm studied in this dissertation shows three positive effects of bit compaction: smaller states result in less I/O time, less disk space used, and more states that fit into the state cache, which results in more duplicates being eliminated by immediate duplicate detection (IDD).

2.2.4 Search Algorithms and Techniques for Improving Scalability

Many algorithms have improved the scalability of model checking, with some concentrating on reducing the number of states that need to be searched. Abstraction in search [11] groups states into less-specific abstract states and explores a smaller state space. Two methods, symmetry reduction [7] and partial order reduction [7], use logic to reduce the size of the searchable state space. Other methods compress the state representation (sometimes with lossy methods) to allow more states to be explored, including bit-state hashing [16] and hash compaction [16]. Heuristic search [72] attempts to find error states sooner in the search process, reducing the effort to find error states. Parallel search [42, 50, 74] harnesses multiple processors to one task. External-memory search [3] stores all or portions of the state space on disk. Partial memory search [76] only maintains portions of the state space, while still attempting to verify the model.

2.2.5 Test Models

Table 2.1

Basic search space information on the test models

Model	Size of State Space States	Edges	Largest Layer
Models Without Errors			
newlist	80,109,979	555,579,029	2,867,128
kerb	49,844,072	77,053,133	8,530,085
mcslock	666,254,196	2,665,016,784	18,982,750
eadash	145,106,401	2,614,892,276	10,878,186
directory	1,071,401,468	1,629,512,836	52,824,772
Models With Errors			
arbiter1	71,850,195	485,787,586	14,827,649
sci1	16,436,226	24,448,193	15,418,195
ns	363,581,415	485,367,304	184,955,334
adashe	4,962,986	134,925,462	2,554,262
arbiter2	530,024,702	7,960,997,549	272,440,347
ldash	119,768,101	1,508,732,174	55,698,041
sci2	351,937,017	440,449,323	321,644,671

We collected basic information about all the models used in this dissertation using the algorithm described in Section 3.3 (without partial DDD). Tables 2.1 and 2.2 report some of the basic statistics of the models used in this dissertation, divided into two types: models without error states and models with error states. These numbers were collected by the sorting-based search algorithm described in the next chapter. Since these numbers are produced by a search algorithm, they represent the reachable state space. The models with error states are searched incompletely, since the search terminates when an error state is found, and the numbers reported reflect a search that stops at the error state. If the

Table 2.2

Basic characteristics of the test models

Model	Variables	State Vector Bytes	GB RAM	GB Disk	Depth	Locality
Models Without Errors						
newlist	90	40	4	4	110	41
kerb	247	96	5	5	28	1
mcslock	25	40	34	32	154	86
eadash	541	543	75	75	63	63
directory	87	83	98	95	114	8
Models With Errors						
arbiter1	94	24	3	3	31	15
sci1	265	120	4	4	8	5
ns	77	28	8	7	15	2
adashe	2,330	2,332	34	34	16	5
arbiter2	171	44	58	58	15	10
ldash	503	512	155	155	21	15
sci2	301	352	221	217	8	5

search were allowed to continue, larger numbers would be reported for every category. Table 2.1 lists the number of unique states found during the search, the number of edges encountered in the model, and the number of unique states in the largest layer seen in the model. Table 2.2 lists the number of variables defined in the Murϕ model, the number of bytes required to store the state vector with bit compaction, the number of gigabytes of external memory required to store the model in RAM, the number of gigabytes of external memory required to store the model in external memory, the number of layers in the model, and the locality [83] observed. Since the same state vector representation is used in both RAM and disk, the size of the state space is nearly the same in each medium. However, some information in addition to the state vector is stored with each state. In external memory we store enough information with each state to locate the parent of that state in external memory. This extra data can be used for solution reconstruction and amounts to 12 bytes per state in external memory. The states in RAM also have some additional information including: parent pointers, pointers for organizing them into linked lists, and information that helps us build the tables and graphs reported in this document. All of this information amounts to 15 bytes per state in RAM. The difference in the size of the state representation for RAM states and disk states is 3 bytes per state. The size of the state vector usually dominates the total state size. When the total size of the state space is rounded to the nearest GB it is usually the same for both RAM and external memory. Since all of our algorithms use a maximum of 3,000 MB to store states in RAM, all of these models require external memory to be searched completely.

Model checking state representations are usually larger than those in most search problems. In Section 2.2.3, we said a model of 5 dining philosophers took 15 variables and 4 bytes. Our implementation of the *Sliding Tile Puzzle*, in a 4 by 4 arrangement, takes 16 variables and 12 bytes. The *Towers of Hanoi* with 4 pegs and 12 disks has been implemented with states of 28 bytes. In contrast, as reported by Table 2.2, the average size of state representations in the models used in this dissertation is 382 bytes. The larger size of the state vector means comparison operations (equals, less than, greater than, etc.) are more expensive in model checking, which affects the performance of the algorithms.

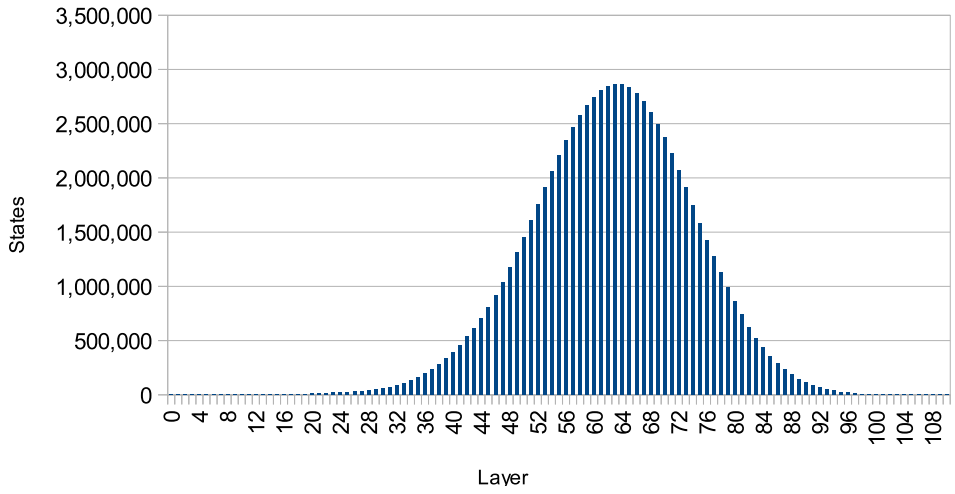


Figure 2.5

Number of unique states in each layer of the *newlist* model.

Graphs showing the number of states per layer in a model usually approximate a bell-shaped curve. This means the majority of the states are near the waist of the graph. Figures

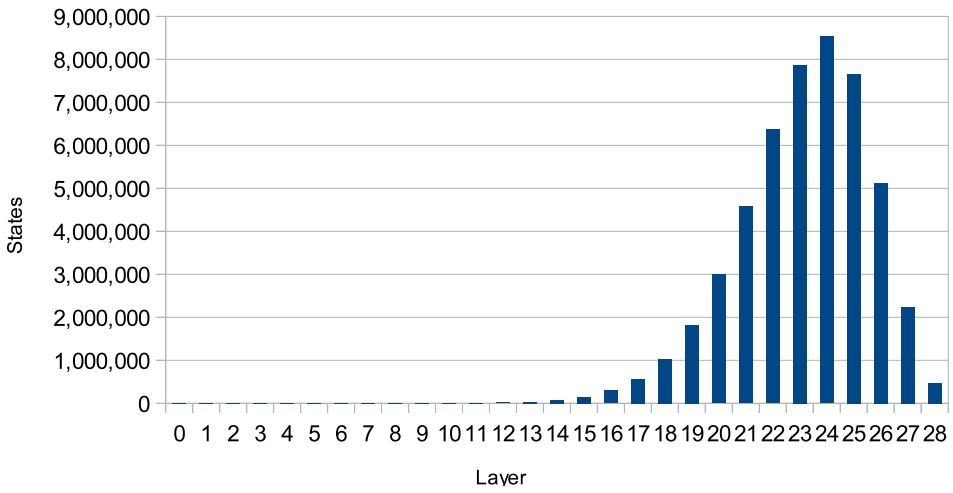


Figure 2.6

Number of unique states in each layer of the *kerb* model.

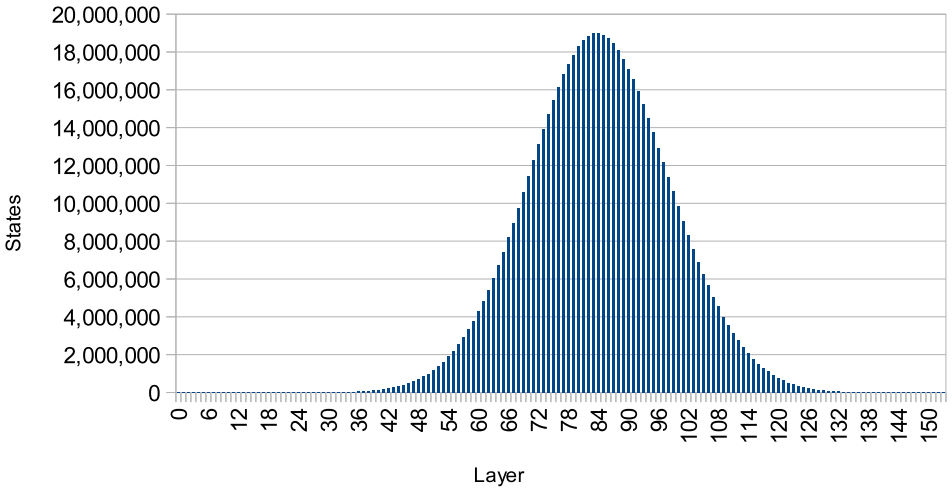


Figure 2.7

Number of unique states in each layer of the *mcslock* model.

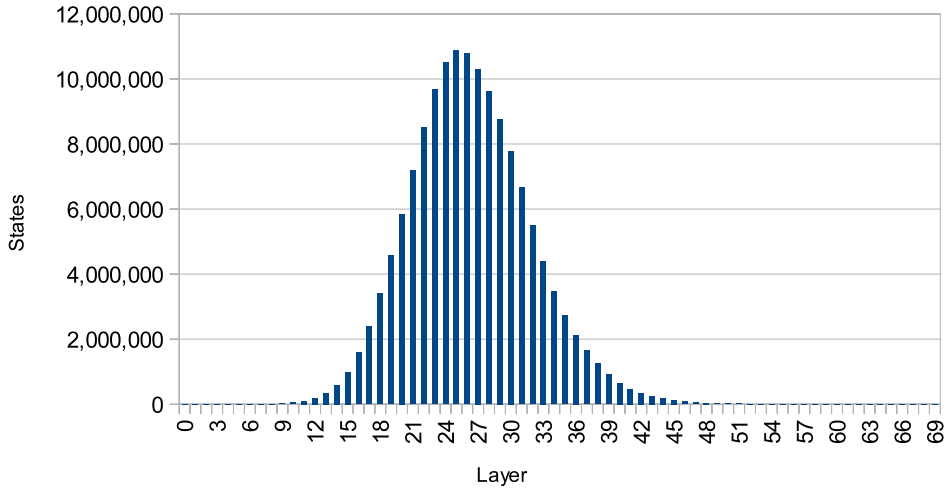


Figure 2.8

Number of unique states in each layer of the *eadash* model.

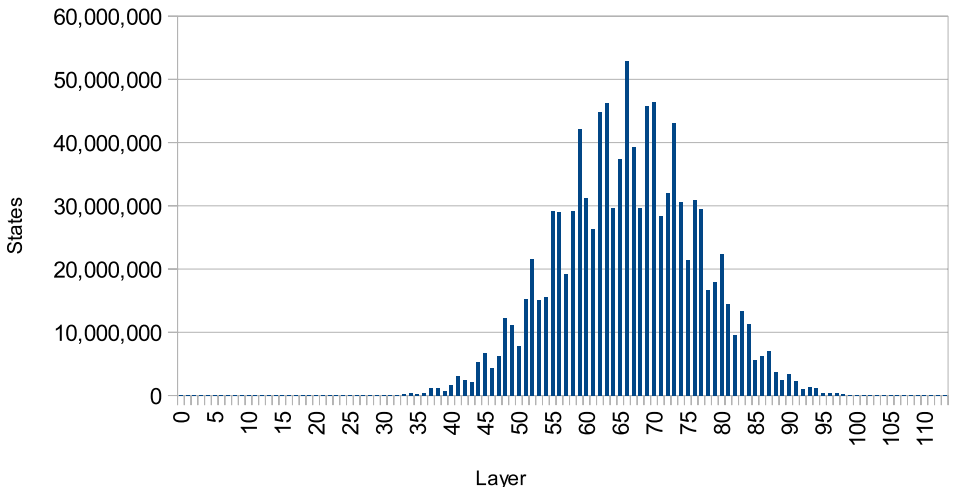


Figure 2.9

Number of unique states in each layer of the *directory* model.

2.5, 2.6, 2.7, 2.8, and 2.9 depict the number of unique states per layer in the models without error states. These graphs are not perfectly bell-shaped; the *kerb* model has one very short tail and the *directory* model spikes overlayed on the bell-shape. But all of these models approximate a bell curve. Having a large number of states in the middle layer of the graph strongly affects how algorithms behave on this kind of graph. Models with error states exhibit the same bell-shaped properties, although they do not continue past the layer in which the error state is found. When the error is found before the waist of the graph, the largest layer is usually the last layer. This different layer structure will affect algorithm behaviour differently than having the largest layers near the middle of the search. Each model has unique properties, while tending to have a bell-shape overall.

All of the models are based on complex protocols. All of our models are parametrized by one or more values. Variations of these models can be produced by altering the parameters. The parameters used for the test models are given below.

1 *arbiter*: A mutual exclusion protocol with an arbiter that allocates resources [50]. Two variants of *arbiter* were tested:

arbiter1 : a model with 13 resources that contains an error state

arbiter2 : a model with 24 resources that contains an error state

2 *dash*: The Stanford Dash Communication Protocol [56] - this protocol has the following three variant models:

adashe: An abstract model of the dash protocol - this variant has been changed to deliberately include error states; our *adashe* model had 1 cluster with memory, 4

clusters without memory, 1 address in the memory cluster, and 3 value types to be saved

eadashe: An elementary abstraction of the dash protocol - we tested a model with 1 cluster with memory, 2 clusters without memory, 2 addresses for each memory cluster, and 2 value types to be saved; this variant also allowed the cluster with memory to act as a client

ldash: A literal model of the dash protocol - we tested a model with 2 clusters with memory where 2 addresses per cluster could be locked and 2 clusters without memory; all clusters can buffer 1 message, and this model has an error state

3 *directory*: A directory-based cache protocol [24], a model created by IBM and used for evaluating the protocol in servers - we tested a model with 14 clients

4 *kerb*: A model of the Kerberos authentication service for distributed open systems [44] that deals with several clients, servers, and intruders; session keys are maintained and messages are passed - our model included 2 clients, 1 server, 1 key distribution center, 1 ticket-granting server, and 1 intruder and allowed 1 outstanding message; intruders could remember 10 messages and could send 1 message at a time; there were 10 authenticators and 10 encrypted keys; the model had 2 encrypted key tickets, 1 authenticator ticket, 1 authenticator ticketed id, and 10 session keys

5 *mcslock*: A protocol for distributed list-based queuing lock without atomic compare and swap operations [59, 68] - we tested a model that had 4 threads sharing the lock

6 *newlist*: A protocol for sorting a distributed linked list [60, 68] - we used a system with 8 distributed nodes

7 *ns*: A model of the Needham-Schroeder protocol for mutually authenticating parties in a system with intruders [62, 73]; a security protocol used to ensure parties are communicating with legitimate other parties; environment is hostile, and messages can be overheard, deleted, and created by intruders - we used a model with 2 message initiators, 2 message responders, and 1 intruder; 3 outstanding messages were allowed at a time, and the intruder could remember 5 messages (this model has an error state)

8 *sci*: A model of the IEEE/ANSI Standard for a Scalable Coherent Interface [38, 73]; defines several cache coherence protocols, each a subset of the last; provides a shared-memory architecture at the software level, while the actual hardware is distributed memory. Two variants of *sci* were tested:

sci1: This model had 4 processors, 4 caches, and 2 memory locations; each memory location could hold 2 values, and each value was one of 4 types (this model has an error state)

sci2: This model had 4 processors, 4 caches, and 4 memory locations; each memory location could hold 4 values, and each value was one of 4 types (this model has an error state)

2.3 External-Memory Graph Search for Model Checking

This section reviews approaches to external-memory search, particularly focusing on model checking. It begins with early approaches to delayed duplicate detection (DDD), covers sorting-based DDD and hash-based DDD, and then explains structured duplicate detection.

2.3.1 Delayed Duplicate Detection

External-memory algorithms usually require delayed duplicate detection. In a RAM-only search, duplicate elimination can be performed immediately on individual states. In the large state spaces searched by external-memory algorithms, the whole state space cannot be held in RAM and duplicate elimination is delayed and performed periodically on a batch of states. Many external-memory search algorithms have two modes. First, state expansion where open states are expanded and the generated states are saved in a candidate set. Second, periodically the entire candidate set is refined to eliminate duplicates. These two modes repeat in cycles until the search is completed. There are various methods for that duplicate elimination which will be explained in the following sections.

2.3.2 Early Approaches to Delayed Duplicate Detection

Two early attempts at external-memory search involve comparing all candidate states to all closed states to determine if the candidate states are unique: first, the algorithm by Stern and Dill and then, the Della Penna et al. method.

2.3.2.1 Stern and Dill

In an early implementation of external-memory search created by Stern and Dill [75] and implemented in Murϕ, if layers were smaller than RAM, DDD was performed once per layer. When a layer of states exceeded the size of RAM, duplicate detection happened immediately when new states were generated. In this approach, all states were kept in a single external-memory file in the original insertion order. Since the file was not ordered and was larger than RAM, each candidate state was compared to every state in external memory during duplicate elimination, an inefficient implementation that has been improved in many ways.

2.3.2.2 Della Penna et al.

Della Penna et al. created an external-memory search algorithm that exploits transition locality [66, 68], a property of state spaces for which duplicate states are likely to be explored close to each other rather than far apart. In this method, all open states are expanded before DDD occurs. Most approaches require DDD to compare the candidate states to the entire closed set, but, in this approach, candidate states are compared during DDD to a subset of the closed states, which are stored in files of a constant size. When a file fills, another file is created. During DDD, a randomized heuristic is used to determine how many closed files will be used to eliminate duplicates in the candidate set. To exploit transition locality, selected files always start at the most recently generated, and only when the heuristic selects all closed files will the least recently generated file be used. The selected closed states are read from disk one at a time and compared to the candidate states

stored in a hash table; when duplicates are discovered, they are removed from the hash table. The candidates that remain after the partial DDD are treated as open states, even though some of them are duplicates. (Because of transition locality, few duplicate states are treated as open states.) The algorithm then repeats by expanding all of the open states.

2.3.3 Sorting-Based Delayed Duplicate Detection

This section describes three different algorithms that use sorting-based DDD. In sorting-based methods, the candidate, closed and open sets are kept in sorted conditions. Duplicates within the sorted candidate set will be adjacent and can be recognized by comparing a candidate to the next state in the set. Candidate states will also be compared to closed states in sorted order until a closed state is found that equals or exceeds the candidate state. If an identical closed state is found, the duplicate candidate is removed. If a closed state that exceeds the candidate state is found, the candidate is shown to be unique and added to the open set. Because both sets are in sorted order, a single pass through each set is all that is required to eliminate all the duplicate states in the candidate set.

2.3.3.1 Roscoe

Roscoe [46, 69] used sorting based delayed duplicate detection, but let virtual memory handle all accesses to external memory. The use of virtual memory limited the effectiveness of the approach.

2.3.3.2 Korf

Korf [45, 46] used breadth-first search with sorting-based DDD and frontier search. He used input files containing the open states and the most recent layer of closed states. As the file is read, the open states are expanded and the closed states are discarded. The parent states are retained in a closed file containing just the most recent layer. The candidate states are put into a separate file. When the input file is exhausted, the candidate states are sorted and merged with the closed states into a single file, during the file merging all duplicates are eliminated, resulting in a file of closed states from the most recent layer and the open states, which is the starting condition of the algorithm. The process then repeats. The algorithm only maintains two layers, making it a frontier search, but not all duplicates will be eliminated by two layers, even in undirected graphs, so Korf added "used" bits to each state. The used bits record which neighbors of a state have been previously expanded, which keeps duplicate states from even being generated. Used bits, however, are difficult to apply to all problem domains.

2.3.3.3 Edelkamp, Jabbar, and Schrödl

External A* [17] extends the A* algorithm for external memory. In this approach, all states are assigned to buckets, with a unique bucket for each unique combination of g and h values. As states are generated, they are written to individual files that correspond to their buckets. When all candidate states for bucket b have been added, bucket b goes through DDD. First, the bucket is sorted. Duplicates of states in a bucket labeled g+h can only be in other buckets with the same h value, which reduces the number of closed states required

during DDD. The duplicate detection is performed by linear passes through all files with the same h value. Because this is a heuristic search, the buckets are expanded from smallest f value to largest. If no heuristic is used, this approach is equivalent to breadth-first search.

2.3.4 Hash-Based Delayed Duplicate Detection

Hash-based DDD can perform DDD without sorting, dividing the state space into buckets using a hash function. Duplicates can be eliminated by examining the states of a single bucket in a hash table, as found in the following examination of algorithms created by Korf and Schultze, Bao and Jones, and Evangelista and Kristensen.

2.3.4.1 Korf and Schultze

In hash-based DDD, as described by Korf and Schultze [47, 48], the candidate set is divided into buckets by a hash function. All open states for layer d are expanded, with the children states placed into files according to the buckets they hash to. (The hash function ensures a small maximum size for all buckets.) When all the open states are expanded, each bucket is put through DDD separately. When bucket b is put through DDD, all candidates belonging to bucket b are read from disk and put into a hash table in RAM. Because the bucket is small, all candidate states in the bucket will fit completely into a RAM hash table. Duplicate states will be recognized and not inserted twice into the hash table. Each closed state belonging to bucket b is also read from disk, and if the closed state duplicates a candidate state in the hash table, the redundant state is removed from the hash table. When all closed states have been reviewed, the remaining candidate states in the hash table are written to a disk file as open states.

2.3.4.2 Bao and Jones

The Bao and Jones [3, 26] algorithm is a variant on hash-based DDD. Instead of expanding states in layer order, this algorithm expands states belonging to a single bucket until there are no more open states currently available for that bucket. Expanded states that belong to the current bucket are added to a hash table, with duplicates discarded, and then added to the expansion queue. When a expanded state does not belong to the current bucket, it is written to a candidate file for the bucket it hashes to. When all the currently available states for the current bucket have been expanded, the algorithm then switches to another bucket and repeats the process. Every time buckets are switched, the Bao and Jones algorithm writes the entire contents of the current bucket to disk and reads the entire contents of another bucket from disk. When the algorithm starts using another bucket, the candidate file for the new bucket is also read into the hash table. Duplicates are discarded as they are encountered and the remaining unique states expanded. The process repeats until no more states are available to be expanded.

2.3.4.3 Evangelista and Kristensen

Evangelista and Kristensen [26] based their external-memory search on the algorithm by Bao and Jones [3] but changed the base algorithm by modifying the bucketing scheme as the search progresses to ensure that bucket sizes remain small.

2.3.5 Structured Duplicate Detection

Zhou and Hansen [81] created a structured duplicate detection (SDD) algorithm, a method for always eliminating duplicates immediately in external-memory search. Burns

and Zhou created a SDD algorithm for model checking [10]. The Zhou and Hansen algorithm divides the open states into buckets based on a hash function. When generating a child state s' of state s from bucket b, the new state s' can belong only to a subset of the buckets called the detection scope. When expanding states from bucket b, the entire detection scope of the bucket is read into a hash table. Generated states are immediately compared against this hash table, and those that do not exist in the table are unique and are written to a file of the bucket they hash to. When all states in the current open bucket have been expanded, a new open file is selected. When the open file changes, the detection scope also changes, so the states in the hash table are switched to match the current detection scope, and the process repeats. This approach always finds duplicates immediately, but swapping states in and out of memory can be computationally expensive.

CHAPTER 3

SORTING-BASED EXTERNAL-MEMORY SEARCH

This chapter describes an approach to external-memory graph search for model checking that uses sorting-based delayed duplicate detection (DDD). The sorting-based approach to duplicate detection is well-known and was reviewed in Section 2.3.3. This chapter proposes and evaluates several improvements of the basic algorithm. In addition, two forms of local structure in external-memory breadth-first graph search are identified, intralayer and interlayer locality, and techniques are described for leveraging both forms of locality to improve the efficiency of duplicate detection.

3.1 Basic Algorithm

This section describes a basic approach to external-memory search with sorting-based delayed duplicate detection (DDD). Pseudocode for the basic algorithm is shown in Figure 3.1. This algorithm proceeds one layer at a time, generating all unique states. There are two distinct phases for each layer: state generation and DDD. States can be classified three ways: candidate states, open states, and closed states. Open states are unique states that have not yet been expanded to generate children states. Candidate states are generated states that have not yet been proven unique by DDD. Closed states are unique states that have been expanded to generate children states. During state generation, the algorithm will

read open states from the current layer and expand each open state to generate candidate states for the next layer. The candidate states are saved in files, sorted, and later refined by DDD. DDD refines the candidate set by removing all duplicate states by comparison to other candidate and closed states that are read from files. After DDD, the open states of the next layer are written to files. These two steps of state generation and DDD are repeated for every layer of the breadth-first graph. The algorithm stops on two conditions: if an error state is encountered or if no more open states remain to expand.

The DDD portion of the algorithm starts on Line 18. Both the candidate set and the closed set are read in sorted order from files. Since candidates are sorted, duplicate candidates will be adjacent. The candidates are considered one at a time. If the current candidate is identical to the next candidate in sorted order, it is discarded as a duplicate. In this way all duplicates of other candidate states are eliminated. To prove a candidate state is unique, it must be shown to not duplicate a closed state. Since the closed set is in sorted order, closed states are compared to states in the candidate set until a closed state is found that equals or exceeds the candidate state in sorted order. If a duplicate is found in the closed set, the candidate is discarded. Otherwise the candidate has been shown to be unique and is added to the open set. Because both the closed set and candidate set are in sorted order, the next candidate considered cannot possibly duplicate closed states already considered, so each set only needs to be examined once. This makes the complexity of DDD linear in the size of the closed and candidate sets.

```
 1. externalBFS(S_0,ErrorStates)
 2.   i := 0   % index of layer
 3.   Layer(i) := S_0   % set of initial states
 4.   Candidates := ∅
 5.   while Layer(i) ≠ ∅ do
    % Generate successors of all states in current layer
 6.     for each s ∈ Layer(i) do
 7.       for each successor s' of s do
 8.         if (s' ∈ ErrorStates) return false
 9.         else Candidates.enqueue(s')
10.     sort(Candidates)
11.     i := i + 1
12.     DDD(Candidates,i)
13.   return true
14.
15.
16.
17. % Remove duplicates among generated states
18. DDD(Candidates,candidate_layer)
19.   for each c ∈ Candidates do
20.     if c ≠ Candidates.top()
21.       while c > Closed.top() do
22.         Closed.pop()
23.       if c ≠ Closed.top()
24.         Layer(candidate_layer)].enqueue(c)
```

Figure 3.1

External-memory BFS with sorting-based DDD.

3.2 Related Work

Jabbar and Edelkamp [17, 40] implemented sorting-based external-memory search for model checking in SPIN [8]. Their approach is called External A* and is based on a best-first search. Their method employs a heuristic to compute a h value for each state, records the number of steps each state is from the start state, and records the g value. Jabbar and Edelkamp have one bucket for each unique combination of h and g values observed. Each state is bucketed by the associated h and g values.

In Jabbar and Edelkamp's algorithm, buckets can contain duplicate states. This algorithm employs DDD to refine the candidate sets to include only unique open states. In this approach, the system waits until there are no more possible states to add to a particular bucket, then the entire bucket is sorted. In a sorted list, duplicates are adjacent, and the system traverses the list and removes them. States can also duplicate states in other buckets that share the same h value. Comparison to previous buckets is also done in sorted order to eliminate duplicates efficiently. The candidate set is compared to every closed state with the same h value to ensure it is not a duplicate.

Jabbar and Edelkamp's algorithm expands the buckets in a heuristic order. The sum of the h and g values is referred to as f, and the existing bucket with the lowest f value is expanded first. In the case of multiple buckets with the same f value, the bucket with the smallest g value is selected. Heuristic expansion can find error states earlier but depends on the strength of the heuristic and provides no value in graphs that do not exhibit an error state.

Expanding a bucket means generating the child states of every state belonging to the bucket. The newly generated states are then bucketed as described above. The cycle continues until no new states are generated (meaning the model is verified) or an error state is discovered.

3.2.1 Differences

While the algorithm that is the focus of this chapter is very similar to Jabbar and Edelkamp's, two details do not match. First, there was no useful heuristic for a best-first search, so a breadth-first search was used instead. Second, without the heuristic, h values were not available to bucket by, so all states are treated as if they have the same h value.

It is unclear how Jabbar and Edelkamp stored the closed set. It may have been combined into one file per bucket or one file per layer per bucket. Since it is ambiguous, both approaches were tested, but ultimately separate files for each layer were used, since this method provides the best performance. Results for these tests will be presented later.

3.3 Algorithm Improvements

This section describes four improvements made to the basic algorithm. Improvements include sorting in RAM, immediate duplicate detection (IDD), efficient merging of files, and partial delayed duplicate detection (DDD).

3.3.1 RAM Sorting

There are a variety of external-memory sorting algorithms, but it is computationally cheaper to sort in RAM. In the algorithm presented in this chapter, the states are buffered

before being written to disk. Each time the buffer is written to disk, the states in RAM are first sorted. Each time a buffer of states is written to disk, a new file is created. The result is many individually sorted candidate files. The files are merged with a priority queue during DDD. This approach takes less time than a pure disk sort algorithm.

The sort algorithm used is Three-way Radix Quicksort as described by Bentley and Sedgewick [5, 6]. In this algorithm a list of items to be sorted is split into three sublists that are sorted recursively. The sort algorithm compares the strings one character at a time, starting with the first character. In our sorting algorithm, the state representation of unsigned characters is used as our string. The sublists are created by classifying the strings according to the value of the current character with respect to a pivot value. The pivot is selected by choosing the median element of three median elements. Strings with a value less than the pivot for the current character go into the first sublist. Strings that equal the pivot for the current character go into the middle sublist. Strings that are greater than the pivot for the current character go to the final sublist. Each sublist is passed recursively to the same function, where the lists are further subdivided. Additionally, since every string in the middle sublist has identical values for the target character, that character is incremented by one for the middle sublist. In this sort algorithm, the recursion terminates when the sublist contains ten elements or less and uses insertion sort on the remaining elements. Switching to an insertion sort on short sublists avoids the worst case performance possible in the final levels of recursion of quicksort [61].

The Bentley and Sedgewick algorithm [5, 6] is also an index sort. Index sort [1] saves time by moving pointers to states rather than moving the states themselves. Since the state

pointers are much smaller than the states, this saves time. Index sort works well with buffered I/O since the buffer is already stored as pointers to states.

This approach results in many sorted candidate files. For sorting-based DDD, the candidates have to be in a globally sorted order. The individual files are merged using a priority queue, which is implemented as a heap. Merging these individual files can be thought of as the merge step of an external-memory merge sort [43]. Ten states from each file are added to the queue. The priority queue ensures that states are dequeued in sorted order. Each time a state is dequeued from the priority queue, a state from the same file is added to the queue. The priority queue efficiently organizes the states into one merged order.

When RAM is full, states are flushed to disk. As a consequence of this approach, more candidate files are created than are strictly necessary, since RAM is smaller than the maximum file size. The extra files increase the amount of file I/O required but make sorting much faster - an acceptable tradeoff.

Usually, the sorted candidate files can be merged in a single pass during DDD. A single pass is necessary to keep the I/O complexity of the merge step at order $O(n)$ rather than order $O(n \, log \, n)$ of a multi-pass merge. All candidate files must be opened during a single-pass merge, but operating systems impose a limit on the maximum number of open files (for our research that number was 1,000), so the single-pass merge is not possible when too many candidate files exist.

3.3.2 Immediate Duplicate Detection

The approach taken in this dissertation is to translate the model checking problem into a graph search problem. Each node in the graph represents a potential state of the model. Each edge in the graph represents an event occurring in the model. When the model is translated into a graph, any technique for graph search can be applied. The graphs derived from model checking have common structural characteristics. When these characteristics are known, they can be exploited to eliminate duplicate states. This section focuses on two types of locality in a breadth-first graph: interlayer locality [54] and intralayer locality. Interlayer locality is covered in Section 3.3.4, but intralayer locality is a novel concept explained below.

Intralayer locality refers to duplicate states being clustered within a layer when an open list is expanded in a specific order. The ordering of the open list affects the amount of intralayer locality observed. The open list can cluster states in such a way that similar states are expanded contiguously. There are many ways to group similar states and produce intralayer locality, but in this chapter a sorted open list is used to produce intralayer locality. Chapter 4 details an alternative way to order the open list that also produces intralayer locality in the generated states.

Since the open list is sorted, states that are similar are grouped together. In general, child states are only slightly different from their parent state, and the children of two similar parent states are more likely to duplicate each other than the children of two randomly selected states. States are expanded in the order of the open list. Because the open list is sorted the open states are grouped by the state variables. The sorted order also creates

intralayer locality in the candidate set. It takes no additional processing time to produce a sorted open list, since such lists are a natural consequence of using sorting-based DDD.

Two alternative approaches to exploiting intralayer locality - eliminating duplicates in the buffer or using an least recently used (LRU) cache to eliminate duplicate states - are explained here before a later section experimentally examines the differences.

3.3.2.1 Duplicates in I/O Buffer

Since candidate states are buffered and sorted before being written to disk, it is easy to eliminate duplicates within a buffer. As states are written to disk, one can check to see if the current state is the same as the previous state in the buffer, which means it will not be written to disk. Immediate duplicate detection (IDD) does not require a hash table; the size of the buffer is simply increased to provide the space required. It is not clear if previous approaches used this method, but any search algorithm that sorts in RAM could do so. Eliminating duplicates in this way is very cheap computationally and reduces the number of candidate states, which saves file I/O and intralayer DDD time.

3.3.2.2 Least Recently Used Cache Replacement

The RAM cache for the algorithm described here was designed to exploit intralayer locality. Since intralayer states are more likely to duplicate states that have been generated recently, a least recently used (LRU) cache [78] was employed. This LRU cache is a hash table that records how recently each element has been accessed and bounds the number of duplicate states that are not detected.

An LRU cache keeps states in order of usage and keeps the most recently generated states in the cache at all times. In order to stay within the limits of RAM size, the cache must sometimes remove the least recently used states. When states are inserted into the state cache, they are placed in the hash table without regard to LRU, but pointers to the states are kept in a linked list that maintains the LRU ordering. States are added to the head of the list, so the tail of the list becomes the least recently used part. Sometimes a state is added to the cache that duplicates a state already in the cache; in this case, the state's position in the hash table remains unchanged. However, that state is removed from its current position in the linked list and is reassigned to the head of the list. Reordering the linked list when duplicate states are encountered maintains the LRU aspect of the list. When the cache is full, the oldest half of the cache is removed. The states are removed by starting with the tail of the list and working toward the front until half of the states have been removed. Removing the oldest states follows the LRU cache replacement policy.

3.3.3 Efficient File Merging

The improvement described in this section saves time by combining intralayer and interlayer DDD. Candidate states are selected one at a time in sorted order from the candidate files, checked against the closed set, and compared to the most recent candidate state before being written to the open file. By detecting duplicates within the candidate set at the same time as detecting duplicates of closed set, both sets only need to be traversed once, saving time.

3.3.4 Partial Delayed Duplicate Detection

Another improvement of the algorithm is to perform DDD by checking only portions of the closed set. Reading all of what is typically a very large closed set every time DDD is performed takes a lot of time, especially in the largest models. Instead, this algorithm reads just those parts of the closed set most likely to eliminate duplicates, eliminating most duplicates and saving time. Because the closed list is split into one file per layer, as described in Section 3.2.1, partial DDD is easier to implement.

Duplicate states are not randomly distributed around the state space; typical properties of state spaces define where duplicates are most likely to be found, including interlayer locality [76]. In breadth-first search, layer g refers to all states that are g steps from the start state. When a duplicate state is generated in layer g, the original version of that state must be in layers 0 to g of the graph. These duplicate states are created by interlayer edges with a distance equal to the number of layers in between the original version of the state and the duplicate version.

Interlayer locality, a property of most graphs searched with the breadth-first method [54, 64, 68, 76], dictates that interlayer edges (called back edges in the cited papers) are most likely to be short and thus local to the most recent layers. All reachable states are some number of steps distant from the start state, and duplicate states occur when there are multiple paths to the same state. These paths tend to be nearly the same length because they often share some of the same steps, in a slightly different order. A common form of interlayer locality is graph symmetry [39], which results in interlayer edges with a distance of zero. Other types of duplicates lead to interlayer edges of different distances, but the

principle of interlayer locality states that short interlayer edges are more common than long interlayer edges.

An interesting case of interlayer locality is an undirected graph which has interlayer edges with a maximum distance of two. This means that states in layer g can only duplicate states in layer g, layer $g-1$, or layer $g-2$. In undirected graphs, all duplicate states are copies of states in the three most recent layers. This property of undirected graphs is used to great effect in frontier search [46].

When searched breadth-first, directed graphs also exhibit interlayer locality, but the maximum locality is usually a small value. In graphs of model-checking problems, the average interlayer edge distance is usually short.

Figures 3.16, 3.10, 3.11, 3.12, 3.15, 3.13, and 3.14 show the distribution of interlayer edge distance on several models, indicating that the models used for this dissertation exhibit interlayer locality. A few other external-memory search algorithms use partial DDD, such as hash-collision partial DDD [66, 67, 68], frontier search [46], selective-layer DDD [25], and periodic DDD [25]. When the typical interlayer edge distance is known, static partial DDD [54] can be used. By examining the distance of interlayer edges (Figures 3.16, 3.10, 3.11, 3.12, 3.15, 3.13, and 3.14), the appropriate number of layers to check during DDD can be chosen to eliminate most duplicates but save time over complete DDD. The algorithm uses the most recent L layers during DDD, where L is a user-defined parameter. By using the most recent layers, the static partial DDD algorithm leverages interlayer locality [35, 48, 53, 54, 64, 66, 67, 68, 76, 82, 83] and saves time because few duplicates exist beyond the L layers. (This algorithm was previously published [54].) Using a heuris-

tic or principled approach to dynamically alter the number of layers checked is, of course, possible but will be left to future work.

3.4 Experimental Evaluation

This section describes the results of experiments that test the various algorithm improvements described in the previous section. In all of these experiments the number of states stored in RAM is limited to what could fit in 3,000 MB. Tests were run on a machine with an Intel i7 CPU with 4 cores running at 3.07 GHz with 8 GB of RAM.

3.4.1 Impact of Storing the Closed List in Many Files

Table 3.1 shows the time required for using a single file for the closed set versus storing the closed list in one file per layer. When the closed list is in one file, that file must be rewritten at every layer. Alternatively, if the closed list is in separate files, the file I/O is mildly increased, as well as time for a priority queue used to merge the states from the various closed files.

Table 3.1

Comparison of time for storing the closed list in a single file or multiple files.

Model	Single Closed File *(d:hh:mm:ss)*	Many Closed Files *(hh:mm:ss)*	Layers
sci1	24:52	22:28	8
ns	43:09	35:31	15
kerb	4:06:18	2:40:12	28
arbiter1	1:15:55	55:49	31
newlist	>1:11:40:28	3:58:58	110

When comparing the required time between the two methods of closed-list storage, it is clear from Table 3.1 that storing the list in many files is more efficient, and the advantage becomes more apparent in models with more layers, such as *newlist*.

3.4.2 Immediate Duplicate Detection

This section details the results of an experiment related to IDD, first showing the existence of intralayer locality in multiple models and then comparing methods for exploiting intralayer locality - using the LRU cache and eliminating duplicates in sorted buffers.

3.4.2.1 Intralayer Locality

To demonstrate intralayer locality, we measured the number of unique states generated between duplicates of the same state. We call this measure the *cache distance* of the duplicates. We produced samplings of the cache distance distribution for the models in Figures 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, and 3.9. These graphs show the distance in the state cache on the x-axis and the number of duplicates at this distance on the y-axis. Note that the y-axis is logarithmically scaled. The distributions show a large degree of intralayer locality because short cache distances are far more common than long distances.

3.4.2.2 Performance of LRU State Cache

We compared external-memory search without immediate duplicate detection to external-memory search with an LRU state cache, as described in Section 3.3.2, with the results presented in Table 3.2. The columns include the name of the model tested, which algorithm was used (LRU state cache or no state cache), and the time spent on state generation, IDD,

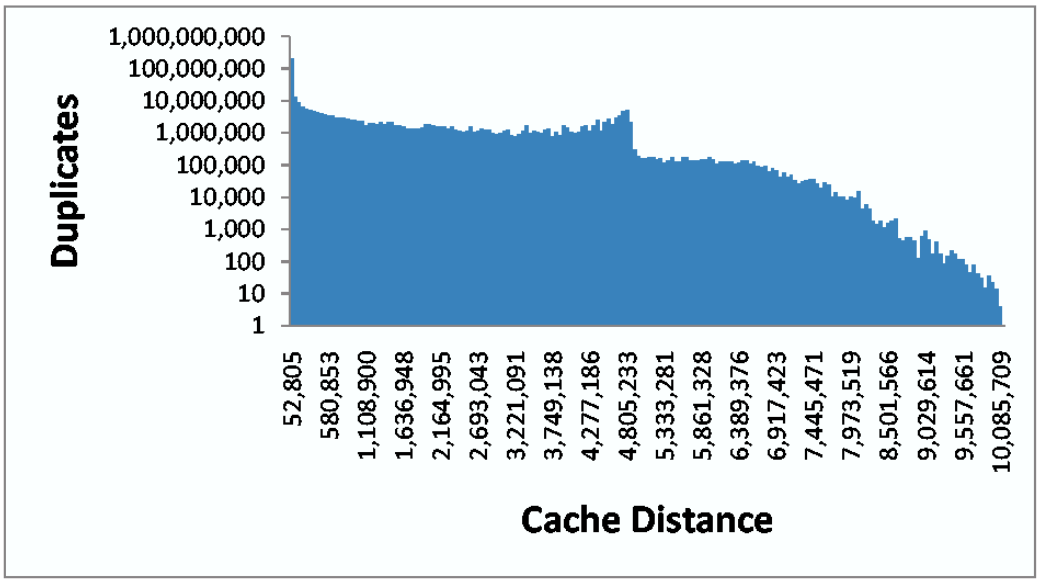


Figure 3.2

Intralayer locality of *newlist* model.

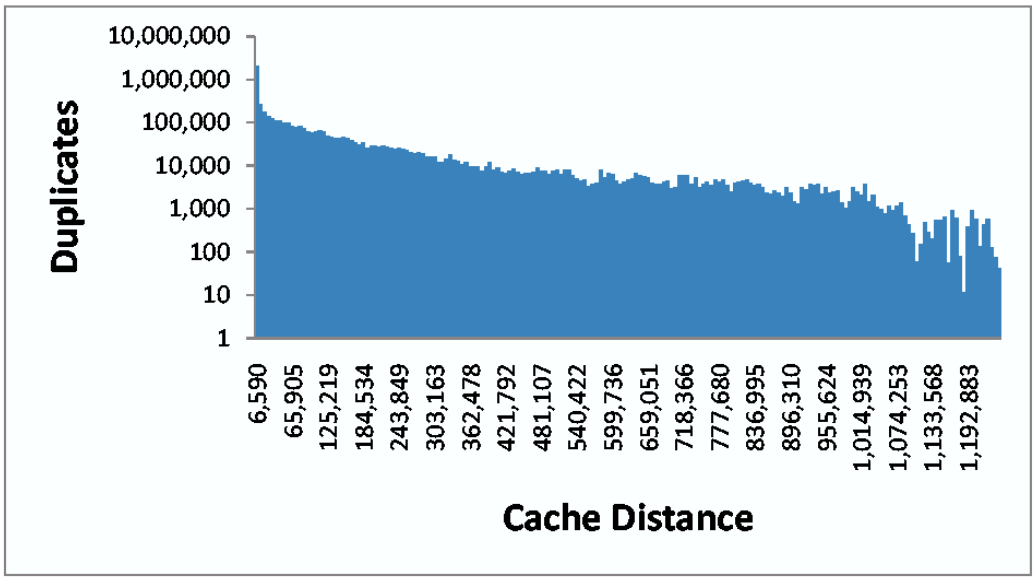


Figure 3.3

Intralayer locality of *arbiter1* model.

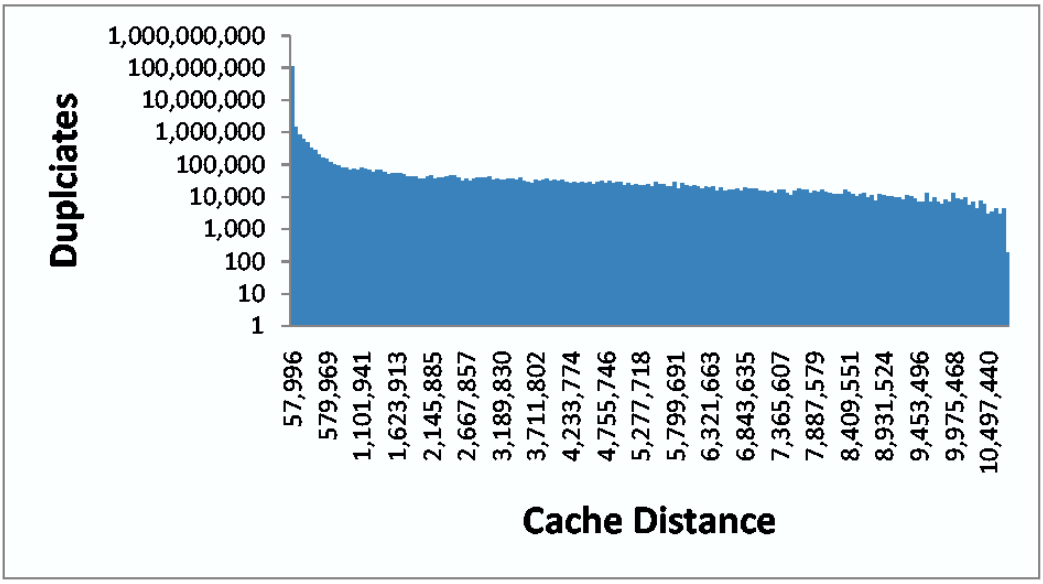


Figure 3.4

Intralayer locality of *ns* model.

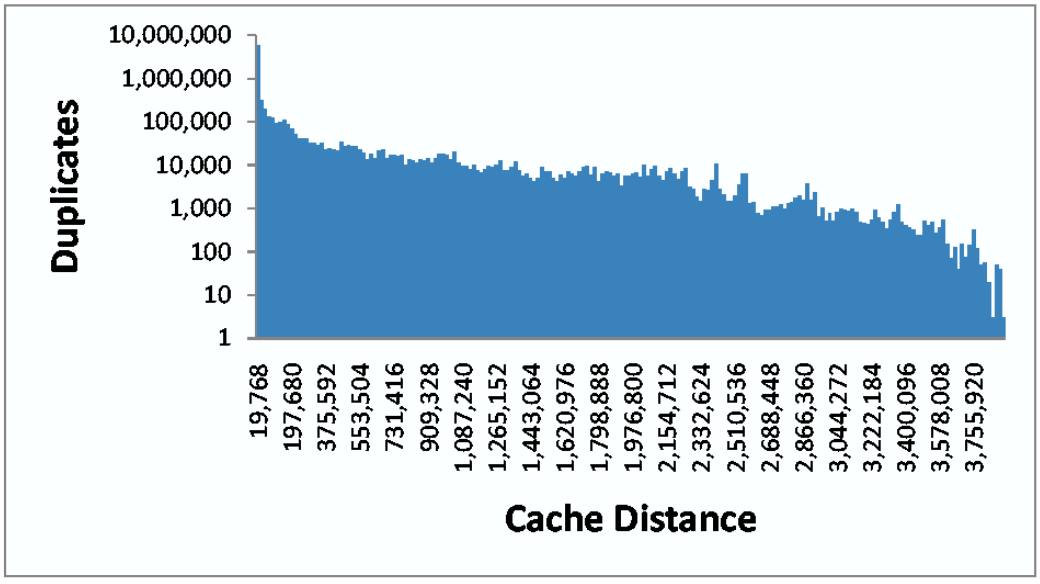


Figure 3.5

Intralayer locality of *sci1* model.

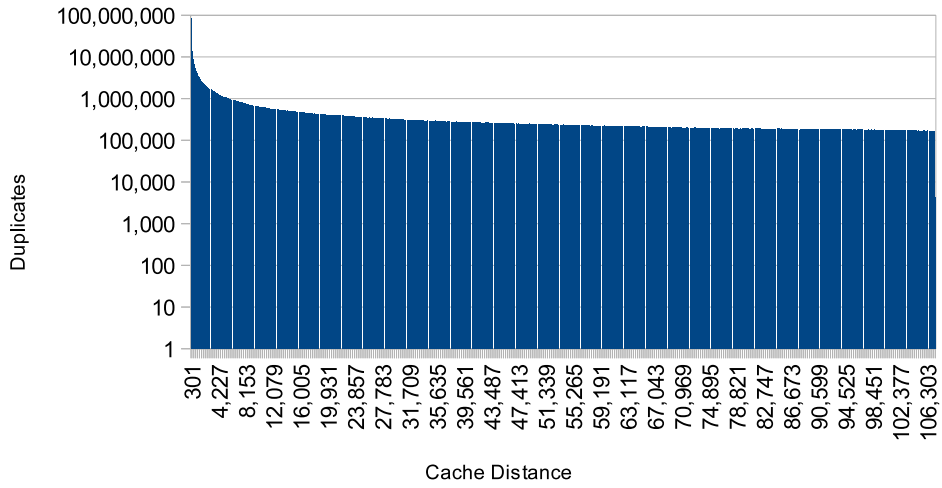


Figure 3.6

Intralayer locality of *ldash* model.

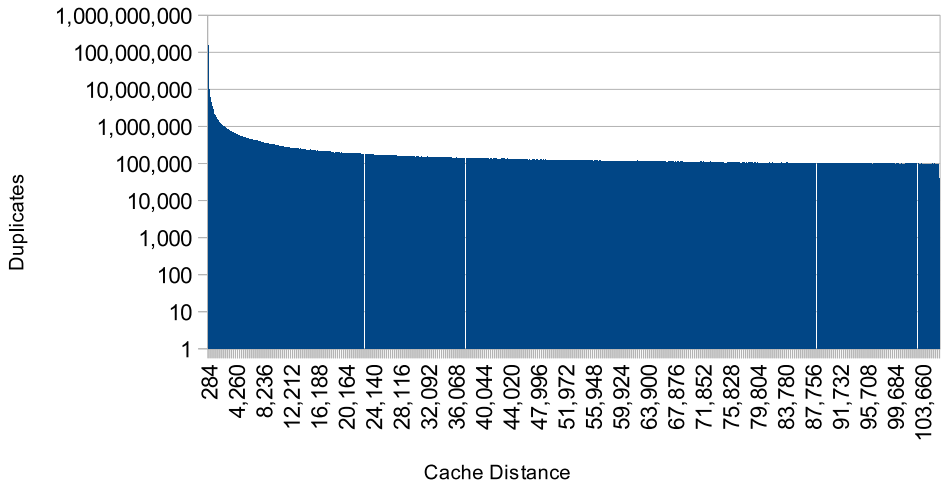


Figure 3.7

Intralayer locality of *eadash* model.

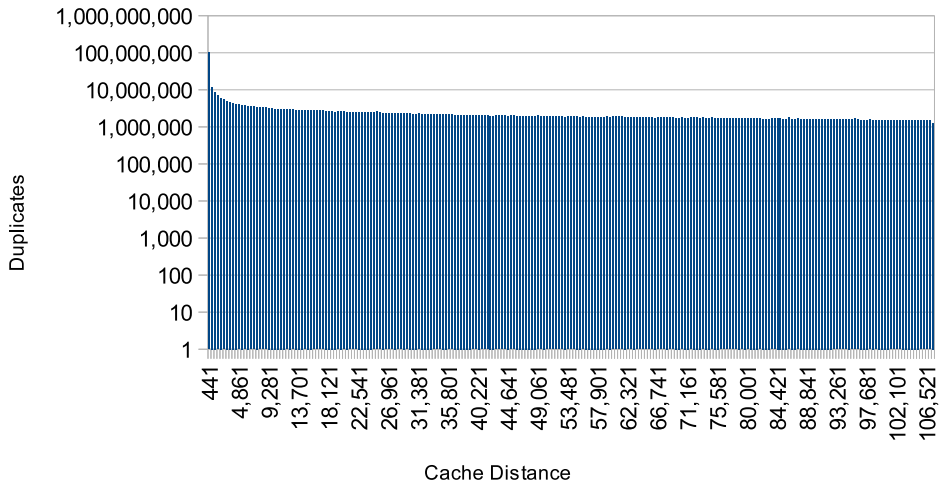


Figure 3.8

Intralayer locality of *mcslock* model.

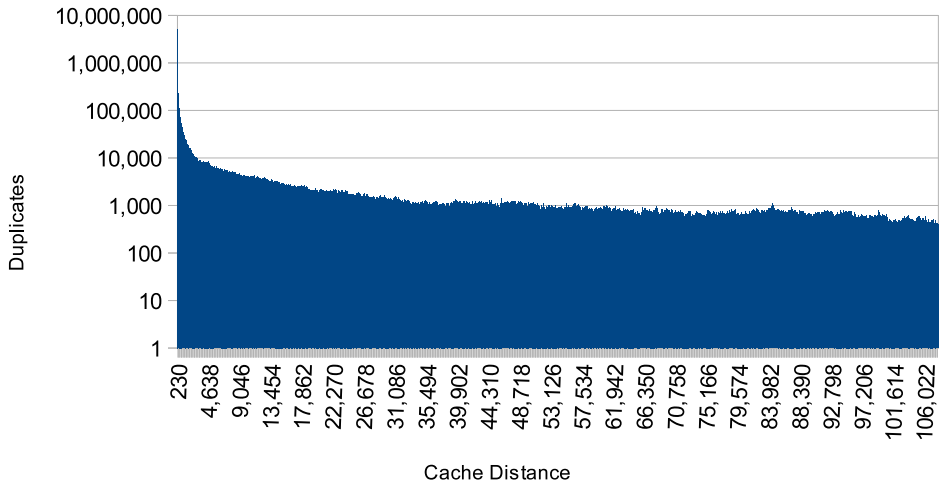


Figure 3.9

Intralayer locality of *adashe* model.

intralayer DDD, and interlayer DDD. Duplicates eliminated in IDD and DDD are shown, and the number of candidate states remaining after intralayer DDD was completed is also included. Because the state cache and state generation times are tightly interrelated, they are reported in a single column.

Table 3.2

LRU state cache compared to search without a state cache and partial DDD

		State Gen. and IDD		DDD			
				Intralayer		Interlayer	
Model	LRU	Time *(hh: mm:ss)*	Duplicates *(states)*	Time *(hh:) mm:ss)*	Size *(states)*	Time *(hh: mm:ss)*	Duplicates *(states)*
sci1	yes	14:33	9,627,651	4:37	17,060,192	4:54	594,344
sci1	no	14:07	7,006,523	4:59	17,081,100	6:58	1,960,155
ns	yes	25:26	119,004,140	5:10	63,206,623	5:49	2,781,749
ns	no	21:45	111,859,697	7:60	121,785,889	7:53	63,235,139
kerb	yes	2:17:20	22,054,616	9:09	50,468,174	15:38	624,237
kerb	no	2:14:50	22,487,138	10:41	50,951,945	20:42	2,637,301
arbiter1	yes	35:17	384,407,742	7:47	92,851,697	14:15	414,474,985
arbiter1	no	31:48	228,501,067	14:52	97,447,917	17:52	97,447,917
newlist	yes	1:46:26	423,820,281	10:08	131,725,271	2:04:24	51,644,413
newlist	no	1:39:28	233,970,984	20:49	228,535,236	2:54:14	241,493,694
adashe	yes	26:12	15,165,740	7:30	5,618,619	13:36	3,301,600
adashe	no	32:31	14,600,611	8:27	5,688,664	17:09	3,866,729
directory	yes	4:17:35	518,695,570	1:49:20	1,115,463,981	6:25:32	45,585,435
directory	no	5:58:46	469,706,413	3:22:24	1,111,769,297	11:41:56	94,574,592

In all cases, the majority of duplicate states were eliminated with an LRU state cache. Also, in every case, the extra time spent on IDD was offset by the time saved in other places. The performance of a state cache was closely related to the number of duplicates in a graph - few duplicate states meant not as much room for improvement. But as the

Table 3.3

Duplicates eliminated in LRU state cache and state buffer

	State Cache		State Buffer		
Model	Time *(hh:mm:ss)*	Duplicates in Cache	Time *(hh:mm:ss)*	Duplicates in Buffer	Total Duplicates
sci1	22:28	9,627,651	25:17	7,006,523	9,221,985
ns	35:31	119,004,140	36:21	111,859,697	121,785,889
arbiter1	55:49	384,407,742	2:02:51	228,501,067	414,474,985
newlist	3:58:58	423,820,281	4:52:23	233,970,984	475,464,694
adashe	1:19:48	15,165,740	1:27:25	14,600,611	18,467,340
directory	12:14:29	518,695,570	20:28:12	469,706,413	564,281,005
sci2	11:30:19	95,176,964	12:30:29	85,903,601	113,634,786

number of duplicate states in the graph increased, the amount of time savings for IDD likewise increased.

Table 3.3 compares the performance of the state cache to the version that checks for duplicates in the sorted buffer - as described in Section 3.3.2.1 - and shows the total number of duplicates in the model, the time in required for both algorithms, and how many of the duplicate states were eliminated with IDD by that algorithm. Searches with an LRU state cache were always faster than searches eliminating duplicates within a sorted buffer (although not remarkably so) due to the number of candidate states. Candidate states may or may not be unique but are written to disk, sorted, and put through DDD. Reducing the number of candidate states reduces the amount of disk I/O, the time spent on sorting, and the time spent in DDD. Having fewer candidate states always saves time in those three areas. Alternatively, IDD did not impact the times for reading open states or interlayer DDD.

Eliminating duplicates in the I/O buffer is computationally cheaper than using a state cache and nearly as effective. States in the buffer are also in order of expansion, which means a state is more likely be a to duplicate of a state in the same buffer than in some other buffer. Not as many duplicates are eliminated in this way as with a state cache, but the difference is not high, as can be seen in Table 3.3.

The reduction in candidate states comes at the price of time spent checking for the presence of a state in the state cache, and an LRU cache must maintain itself. The LRU cache must always know which items are the least recently used and must occasionally discard some of the states it currently caches. However, Table 3.3 shows that, overall, time is saved by adding an LRU state cache to an external-memory search because it eliminates more duplicates than a sorted buffer and has fewer candidate states.

3.4.3 Impact of Efficient File Merging

Time saved when combining intralayer and interlayer DDD into one step (as described in Section 3.3.3) is illustrated in Table 3.4, with time for several different portions of the search. It is clear that combining the steps saves a significant amount of time, some due to less required file I/O, but most because of less required merging time.

Table 3.4

Combining intralayer DDD with Interlayer DDD

Model	Combined	State Generation and IDD *(hh:mm:ss)*	DDD Read Can. *(mm:ss)*	DDD Write Open *(mm:ss)*	DDD Read Closed *(hh:mm:ss)*	DDD Other IO *(mm:ss)*	DDD Merge *(hh:mm:ss)*
sci1	yes	20:54	1:49	27	1	N/A	15:55
sci1	no	21:06	2:01	24	6	48	41:55
ns	yes	1:01:32	14	41	1	N/A	16:12
ns	no	1:06:28	18	42	2	57	22:28
kerb	yes	1:44:37	18	59	18	N/A	2:14
kerb	no	1:45:56	17	4	16	57	6:01
arbiter1	yes	17:49	11	44	8	N/A	4:59
arbiter1	no	20:25	10	1:02	10	47	11:07
directory	yes	3:21:08	12:05	27:23	5:27:42	N/A	3:31:49
directory	no	3:20:46	23:14	22:58	4:00:37	38:59	4:49:50
sci2	yes	7:07:53	1:31:29	28:50	1:29	N/A	2:20:37
sci2	no	7:19:42	1:37:30	21:12	2:17	1:16:52	3:02:25

3.4.4 Partial Delayed Duplicate Detection

The following figures show that model-checking graphs exhibit interlayer locality. Figures 3.10, 3.11, 3.12, 3.13, 3.14, 3.15, 3.16, 3.17, and 3.18 show the locality on the x-axis and the number of duplicates at that distance on the y-axis. These graphs indicate that the models tested for this dissertation exhibit interlayer locality. Notice that short localities are the most common.

This section compares the performance of the complete DDD algorithm to the static partial DDD algorithm. Table 3.5 shows the time for state expansion and DDD, as well as the number of layers in the model, the number of layers used during DDD, and the number of states expanded. Since partial DDD can expand duplicate states, this is a good measure of the redundant work being done by the partial DDD algorithm. This redundant work

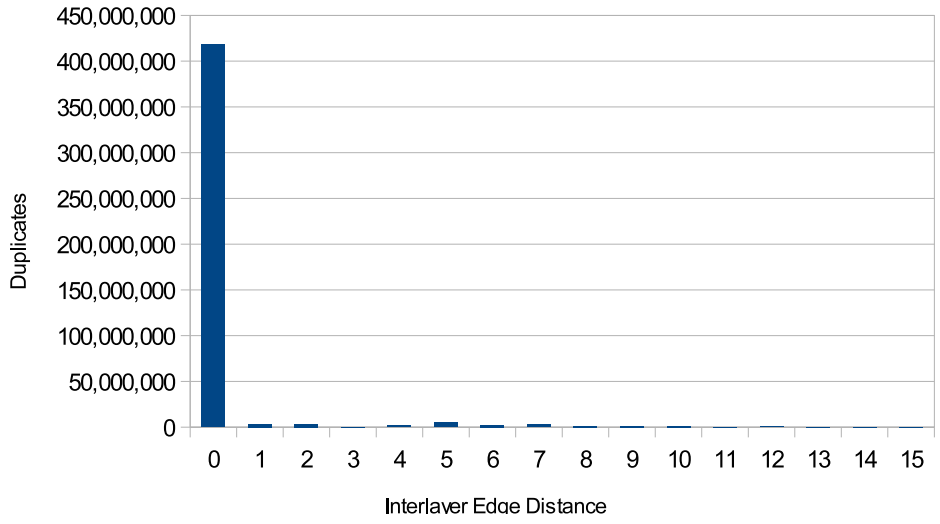


Figure 3.10

Interlayer locality in the *arbiter1* model.

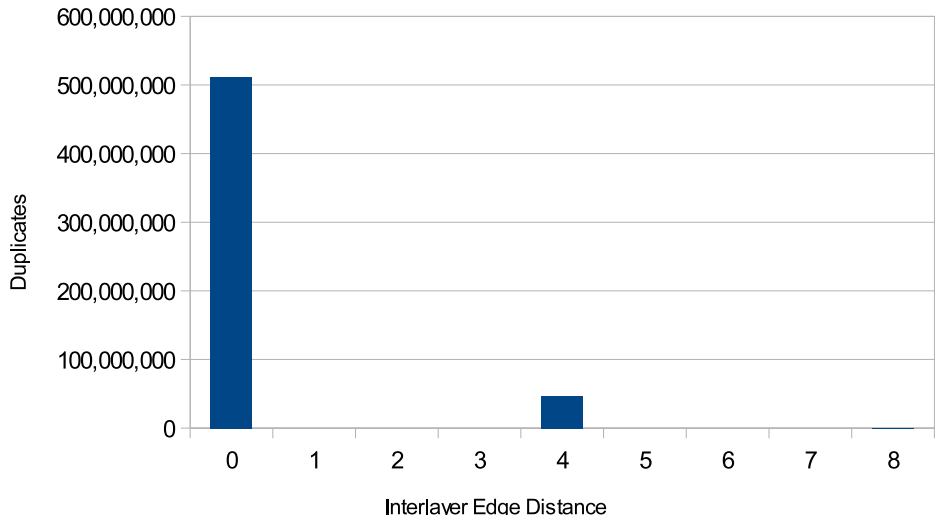


Figure 3.11

Interlayer locality in the *directory* model.

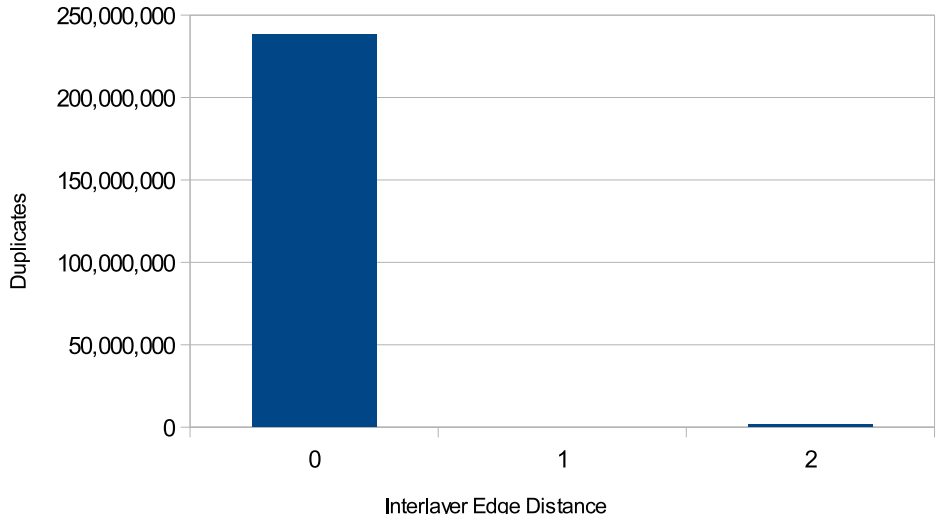


Figure 3.12

Interlayer locality in the *ns* model.

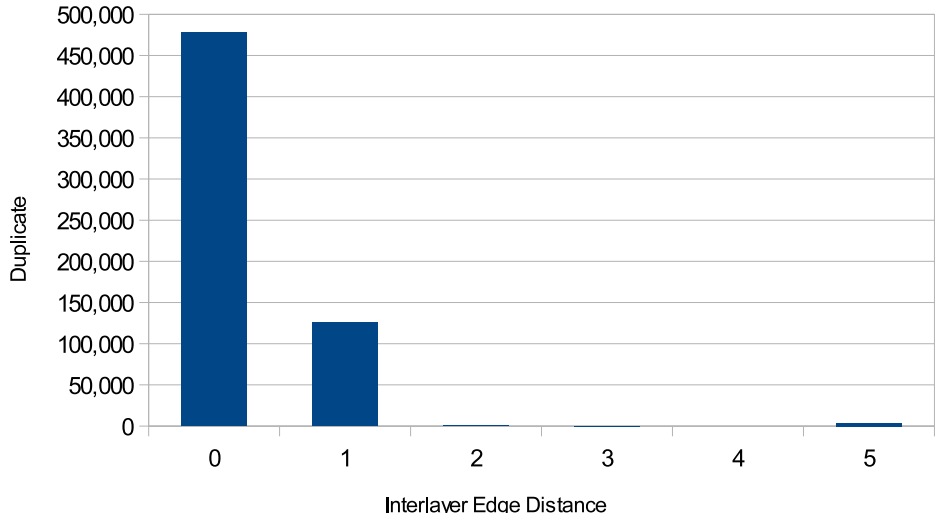


Figure 3.13

Interlayer locality in the *sci1* model.

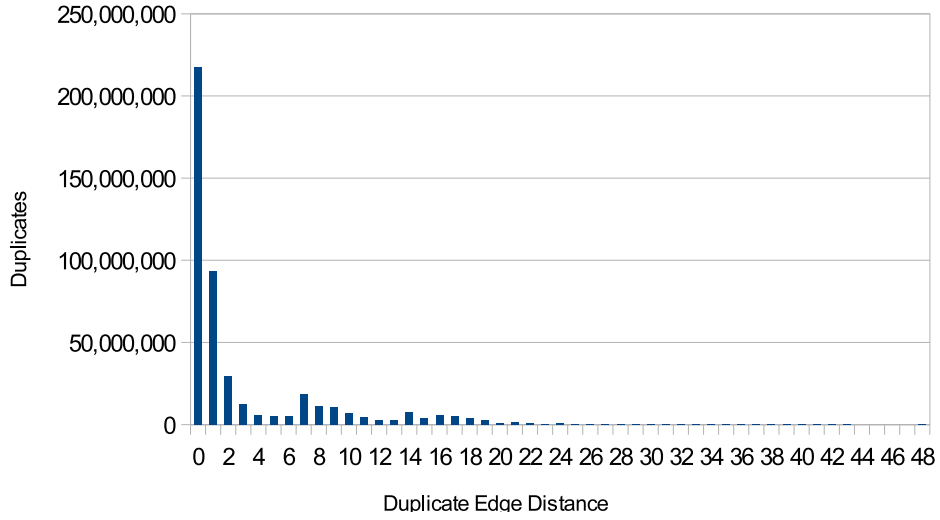


Figure 3.14

Interlayer locality in the *newlist* model.

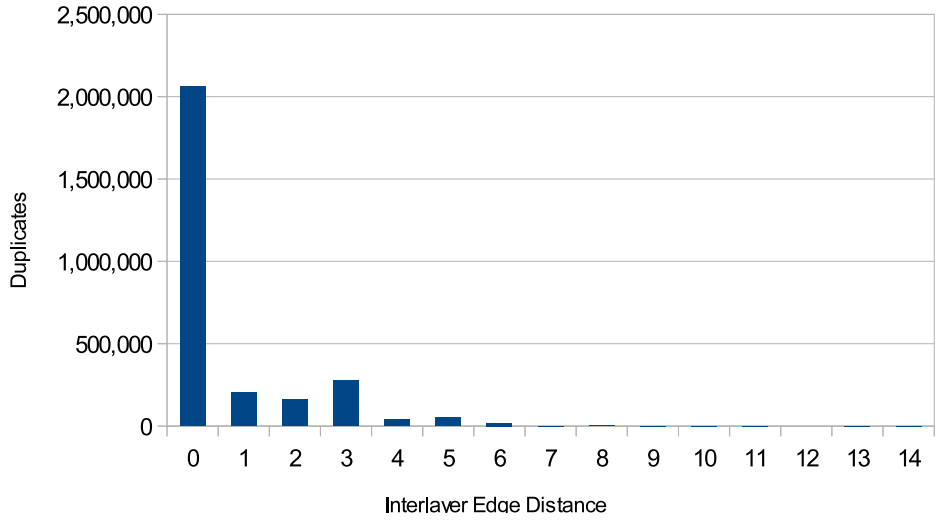


Figure 3.15

Interlayer locality in the *adashe* model.

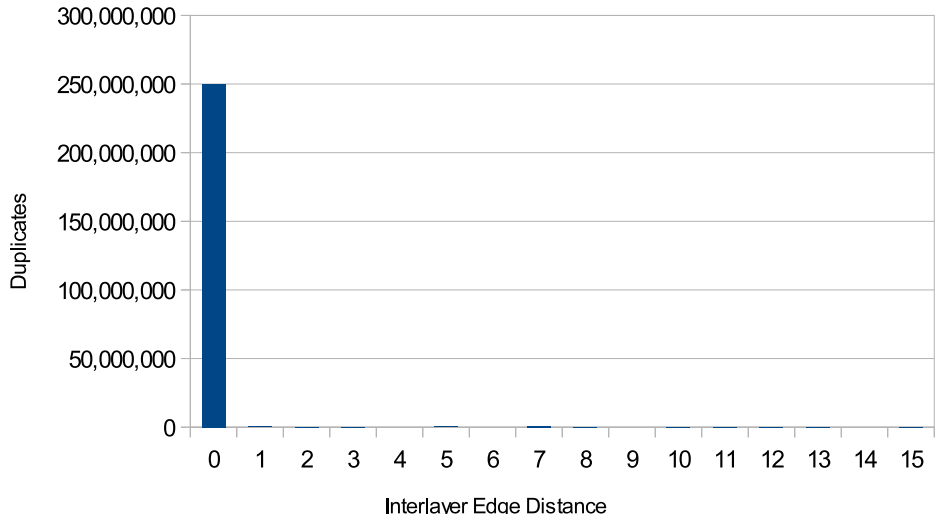


Figure 3.16

Interlayer locality in the *ldash* model.

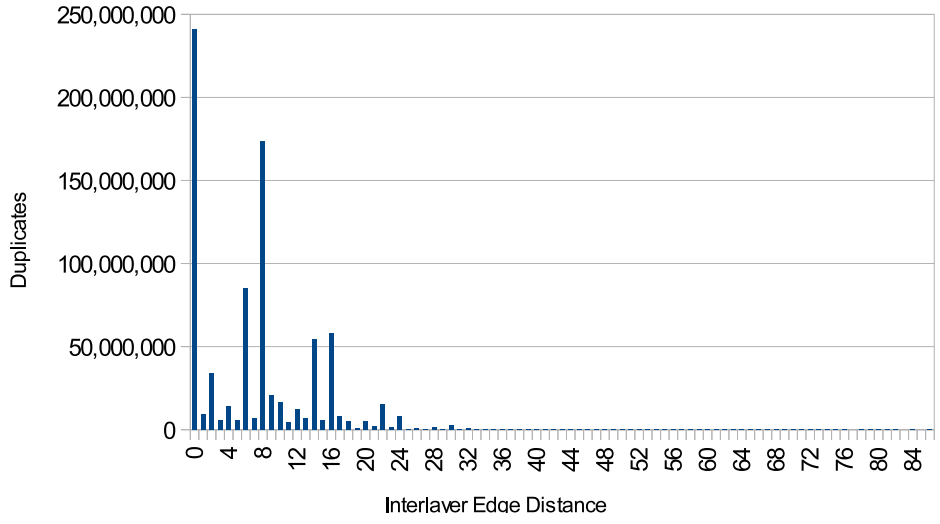


Figure 3.17

Interlayer locality in the *mcslock* model.

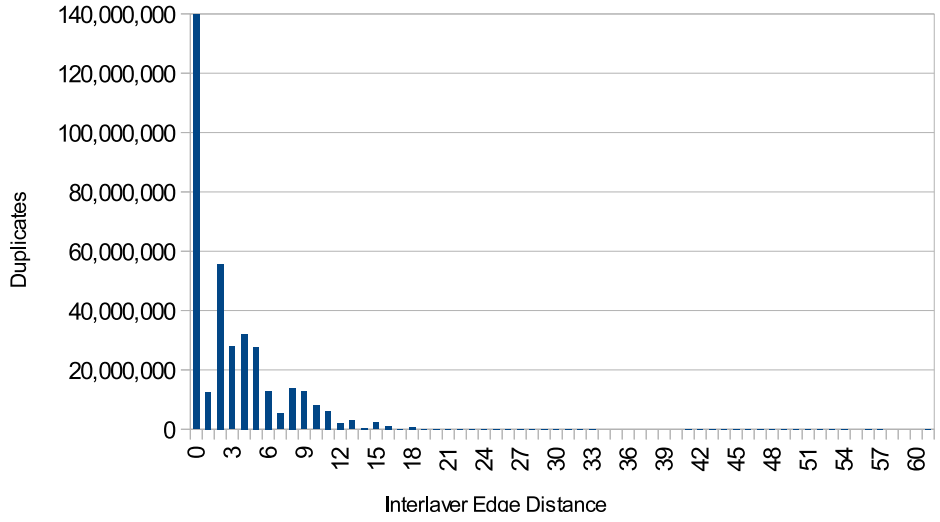


Figure 3.18

Interlayer locality in the *eadash* model.

Table 3.5

Comparison of time in and efficiency of partial DDD with full DDD

Model	Type DDD	Expansion Time *(hh:mm:ss)*	DDD Time *(dd:hh:mm:ss)*	States Expanded	Total Layers	Layers Checked
newlist	Partial	1:36:39	50:24	565,574,056	110	55
newlist	Full	1:25:04	1:37:44	555,579,029	110	all
mcslock	Partial	4:58:23	11:02:15	2,665,016,676	154	86
mcslock	Full	4:49:29	1:06:53:14	2,665,016,676	154	all
eadash	Partial	13:46:11	2:08:18:20	2,614,892,143	63	34
eadash	Full	13:49:10	2:22:07:30	2,614,891,276	63	all
directory	Partial	3:21:08	9:38:59	1,629,512,836	113	8
directory	Full	2:49:46	2:04:31:44	1,629,512,836	113	all
ldash	Partial	19:15:26	40:24	1,508,732,174	21	16
ldash	Full	19:30:19	2:41:30	1,508,732,174	21	all
sci2	Partial	6:22:37	6:01:03	5,173,413	7	2
sci2	Full	6:10:16	7:29:41	5,173,406	7	all

does add time overhead, but not as much as the time saved, in the cases shown here. These results also show search with partial DDD is much faster than search with full DDD, and the increase in the number of expanded states (which is a risk for partial DDD) is very small. This indicates that intralayer locality can be exploited effectively in the partial DDD algorithm.

The static bounds used for these results were selected manually. They were chosen to minimize DDD time. Static bounds were selected after information on interlayer locality was gathered through a complete search. The first approximation for the static bound was the distance of the locality. To be effective, the bound needs to be longer than the majority of interlayer edges. Using the information in Figures 3.10, 3.11, 3.12, 3.13, 3.14, 3.15, 3.16, 3.17, and 3.18, a bound that would eliminate most interlayer duplicates was selected and experimentally verified. Selecting bounds in this manner is only helpful for repeated searches. A heuristic for choosing the static bound or a dynamic bound are possibilities for future work.

Tables 3.6 and 3.7 show how many states were generated per layer, how many of those states were found to be duplicates by the state cache, how many intralayer and interlayer duplicates were found during DDD, and how many unique states belong to each layer. IDD is most challenging on the largest layers of the graph, which exceed the size of the state cache, so duplicates might be missed by IDD. These models have many more duplicate states than unique states: for *eadash* the ratio is 18:1, and for *mcslock* the ratio is 4:1. In *eadash*, the state cache holds 3,196,681 states; for *mcslock* 9,825,701 states fit into the state cache. On the largest layers, the state cache for *mcslock* can only hold 51% of the

Table 3.6

Eadash layer by layer.

Layer	Generated nodes	Duplicates in cache	Duplicates on disk		Unique nodes
			Intralayer	Interlayer	
0	0	0	0	0	1
1	12	8	0	0	4
2	52	28	0	0	24
3	330	241	0	0	89
4	1,278	992	0	0	286
5	4,259	3,505	0	0	754
6	11,583	9,673	0	0	1,910
7	30,070	25,409	0	0	4,661
8	74,726	63,635	0	0	11,091
9	180,244	155,579	0	0	24,665
10	406,038	354,366	0	0	51,672
11	860,695	759,746	0	0	100,949
12	1,702,714	1,514,015	0	0	188,699
13	3,214,447	2,874,131	0	0	340,316
14	5,837,074	5,243,279	0	0	593,795
15	10,228,608	9,232,901	0	0	995,707
16	17,192,692	15,348,151	0	252,870	1,591,671
17	27,529,252	23,914,321	91,760	1,126,607	2,396,564
18	41,573,436	34,961,957	1,128,364	2,083,981	3,399,134
19	59,214,476	49,147,649	2,264,534	3,239,579	4,562,714
20	79,888,775	65,023,752	4,325,233	4,695,805	5,843,985
21	102,875,958	82,592,063	6,675,751	6,423,203	7,184,941
22	127,149,765	101,489,369	8,786,845	8,364,149	8,509,402
23	151,279,997	120,339,741	10,824,885	10,441,170	9,674,201
24	172,731,376	137,110,032	12,564,767	12,549,671	10,506,906
25	188,409,934	149,614,269	13,438,513	14,478,966	10,878,186
26	195,873,421	155,651,949	13,505,095	15,935,682	10,780,695
27	194,830,593	154,916,360	12,889,396	16,713,473	10,311,364
28	186,938,005	148,723,702	11,814,016	16,791,843	9,608,444
29	174,668,574	139,191,867	10,420,387	16,290,903	8,765,417
30	159,738,582	127,672,888	8,889,279	15,393,064	7,783,351
31	142,201,832	114,034,922	7,267,393	14,232,587	6,666,930
32	122,045,168	98,206,816	5,540,660	12,796,947	5,500,745
33	100,777,494	81,373,243	3,911,829	11,089,676	4,402,746
34	80,699,296	65,479,091	2,504,034	9,250,938	3,465,233
35	63,627,677	51,914,879	1,481,180	7,511,376	2,720,242
36	50,135,520	41,118,580	836,088	6,048,443	2,132,409
37	39,483,776	32,454,169	489,634	4,889,269	1,650,704
38	30,648,919	25,360,953	107,302	3,933,896	1,246,768
39	23,139,186	19,117,146	19,427	3,089,285	913,328
40	16,926,150	13,944,187	245	2,328,079	653,639
41	12,119,156	10,022,310	0	1,629,182	467,664
42	8,696,164	7,287,177	0	1,068,550	340,437
43	6,352,147	5,337,079	0	763,147	251,921
44	4,706,338	4,089,835	0	429,876	186,627
45	3,480,378	3,020,328	0	326,283	133,767
46	2,486,235	2,179,413	0	217,355	89,467
47	1,656,147	1,473,196	0	126,384	56,567
48	1,041,263	950,569	0	54,315	36,379
49	665,934	615,420	0	25,053	25,461
50	464,948	434,200	0	11,118	19,630
51	356,396	335,449	0	5,429	15,518
52	277,655	264,572	0	2,301	10,782
53	189,967	182,994	0	1,083	5,890
54	103,005	99,507	0	653	2,845
55	49,459	47,546	0	84	1,829
56	30,829	29,089	0	29	1,711
57	27,819	26,163	0	7	1,649
58	26,369	25,242	0	6	1,121
59	18,033	17,611	0	5	417
60	7,089	7,004	0	4	81
61	1,635	1,615	0	3	17
62	355	350	0	4	1
63	19	15	0	3	1
64	20	15	0	4	1
65	18	15	0	2	1
66	18	14	0	3	1
67	20	18	0	1	1
68	20	16	0	3	1
69	20	16	0	3	1

Table 3.7

Mcslock layer by layer.

Layer	Generated nodes	Duplicates in cache	Duplicates on disk		Unique nodes	Layer	Generated nodes	Duplicates in cache	Duplicates on disk		Unique nodes
			Intralayer	Interlayer					Intralayer	Interlayer	
0	0	0	0	0	1	77	64,681,844	34,094,989	4,181,280	9,589,184	16,816,391
1	4	3	0	0	1	78	67,265,564	35,163,984	4,362,643	10,353,063	17,385,874
2	4	2	0	0	2	79	69,543,496	35,589,142	4,991,244	11,099,230	17,863,880
3	8	5	0	0	3	80	71,455,520	35,891,620	5,441,539	11,823,394	18,298,967
4	12	7	0	0	5	81	73,195,868	35,992,088	6,028,454	12,547,326	18,628,000
5	20	13	0	0	7	82	74,512,000	36,036,066	6,377,572	13,243,244	18,855,118
6	28	17	0	0	11	83	75,420,472	35,995,428	6,511,614	13,930,680	18,982,750
7	44	26	0	0	18	84	75,931,000	35,901,940	6,516,274	14,534,813	18,977,973
8	72	44	0	0	28	85	75,911,892	35,584,541	6,367,539	15,062,522	18,897,290
9	112	67	0	0	45	86	75,589,160	34,835,076	6,455,568	15,548,703	18,749,813
10	180	112	0	0	68	87	74,999,252	34,324,544	6,234,445	15,984,751	18,455,512
11	272	171	0	0	101	88	73,822,048	33,079,092	6,337,161	16,315,524	18,090,271
12	404	258	0	0	146	89	72,361,084	32,290,752	5,854,927	16,562,805	17,652,600
13	584	379	0	0	205	90	70,610,400	31,496,053	5,284,018	16,710,912	17,119,417
14	820	538	0	0	282	91	68,477,668	30,086,014	5,037,051	16,778,041	16,576,562
15	1,128	749	0	0	379	92	66,306,248	29,046,229	4,534,059	16,788,881	15,937,079
16	1,516	1,015	0	0	501	93	63,748,316	27,358,147	4,483,297	16,665,273	15,241,599
17	2,004	1,344	0	0	660	94	60,966,396	26,114,733	3,843,013	16,485,083	14,523,567
18	2,640	1,778	0	0	862	95	58,094,268	24,927,947	3,197,092	16,218,968	13,750,261
19	3,448	2,320	0	0	1,128	96	55,001,044	23,176,776	3,011,883	15,872,557	12,939,828
20	4,512	3,044	0	0	1,468	97	51,759,312	21,512,444	2,654,967	15,404,980	12,186,921
21	5,872	3,970	0	0	1,902	98	48,747,684	20,051,349	2,388,737	14,923,845	11,383,753
22	7,608	5,157	0	0	2,451	99	45,535,012	18,442,810	2,119,442	14,341,401	10,631,359
23	9,804	6,659	0	0	3,145	100	42,525,436	16,933,051	1,931,267	13,795,262	9,865,856
24	12,580	8,577	0	0	4,003	101	39,463,424	15,529,732	1,722,046	13,150,132	9,061,514
25	16,012	10,923	0	0	5,089	102	36,246,056	13,921,779	1,554,924	12,439,776	8,329,577
26	20,356	13,925	0	0	6,431	103	33,318,308	12,664,560	1,320,817	11,732,584	7,600,347
27	25,724	17,653	0	0	8,071	104	30,401,388	11,416,213	1,075,439	10,995,654	6,914,082
28	32,284	22,163	0	0	10,121	105	27,656,328	10,093,794	996,193	10,287,049	6,279,292
29	40,484	27,798	0	0	12,686	106	25,117,168	9,287,802	596,390	9,572,285	5,660,691
30	50,744	34,803	0	0	15,941	107	22,642,764	8,241,341	488,794	8,846,404	5,066,225
31	63,764	43,661	0	0	20,103	108	20,264,900	7,259,101	346,797	8,113,861	4,545,141
32	80,412	55,071	0	0	25,341	109	18,180,564	6,573,660	176,036	7,424,768	4,006,100
33	101,364	69,571	0	0	31,793	110	16,024,400	5,595,236	153,714	6,700,915	3,574,535
34	127,172	87,255	0	0	39,917	111	14,298,140	4,965,145	77,756	6,115,312	3,139,927
35	159,668	109,797	0	0	49,871	112	12,559,708	4,325,714	0	5,476,303	2,757,691
36	199,484	137,336	0	0	62,148	113	11,030,764	3,708,581	0	4,926,135	2,396,048
37	248,592	171,079	0	0	77,513	114	9,584,192	3,192,394	0	4,328,103	2,063,695
38	310,052	213,680	0	0	96,372	115	8,254,780	2,664,218	0	3,819,390	1,771,172
39	385,488	266,093	0	0	119,395	116	7,084,688	2,251,907	0	3,308,068	1,524,713
40	477,580	329,822	0	0	147,758	117	6,098,852	1,896,810	0	2,894,884	1,307,158
41	591,032	409,100	0	0	181,932	118	5,228,632	1,632,298	0	2,479,220	1,117,114
42	727,728	504,177	0	0	223,551	119	4,468,456	1,573,788	0	1,948,954	945,714
43	894,204	619,966	0	0	274,238	120	3,782,856	1,362,141	0	1,639,878	780,837
44	1,096,952	762,500	0	0	334,452	121	3,123,348	1,081,215	0	1,389,124	653,009
45	1,337,808	931,885	0	0	405,923	122	2,612,036	912,214	0	1,173,927	525,895
46	1,623,692	1,132,703	0	0	490,989	123	2,103,580	695,689	0	961,937	445,954
47	1,963,956	1,373,481	0	0	590,475	124	1,783,816	609,198	0	815,076	359,542
48	2,361,900	1,654,753	0	0	707,147	125	1,438,168	477,638	0	656,516	304,014
49	2,828,588	1,984,262	0	0	844,326	126	1,216,056	403,876	0	573,469	238,711
50	3,377,304	2,375,181	0	0	1,002,123	127	954,844	346,271	0	415,737	192,836
51	4,008,492	2,823,114	0	0	1,185,378	128	771,344	329,159	0	296,769	145,416
52	4,741,512	3,346,071	0	0	1,395,441	129	581,664	238,700	0	225,694	117,270
53	5,581,764	3,898,188	0	50,888	1,632,688	130	469,080	271,028	0	110,689	87,363
54	6,530,752	4,381,507	0	246,766	1,902,479	131	349,452	213,073	0	61,758	74,621
55	7,609,916	5,054,821	0	347,117	2,207,978	132	298,484	193,646	0	49,308	55,530
56	8,831,912	5,835,908	0	449,408	2,546,596	133	222,120	139,046	0	37,709	45,365
57	10,186,384	6,718,841	0	542,503	2,925,040	134	181,460	119,206	0	30,530	31,724
58	11,700,160	7,669,321	0	689,157	3,341,682	135	126,896	81,954	0	22,705	22,237
59	13,366,728	8,663,223	0	905,741	3,797,764	136	88,948	61,100	0	12,872	14,976
60	15,191,056	9,781,310	0	1,110,007	4,299,739	137	59,904	42,060	0	6,714	11,130
61	17,198,956	11,039,128	0	1,314,685	4,845,143	138	44,520	32,377	0	3,958	8,185
62	19,380,572	12,387,825	0	1,561,971	5,430,776	139	32,740	23,449	0	2,377	6,914
63	21,723,104	13,792,069	0	1,863,088	6,067,947	140	27,656	20,854	0	1,798	5,004
64	24,271,788	15,267,112	67,387	2,195,885	6,741,404	141	20,016	15,216	0	1,262	3,538
65	26,965,616	16,773,358	194,307	2,548,626	7,449,325	142	14,152	11,122	0	894	2,136
66	29,797,300	17,979,657	682,859	2,936,851	8,197,933	143	8,544	6,802	0	610	1,132
67	32,791,732	19,378,759	1,088,421	3,352,346	8,972,206	144	4,528	3,578	0	296	654
68	35,888,824	20,806,857	1,492,800	3,813,166	9,776,001	145	2,616	2,192	0	111	313
69	39,104,004	22,393,163	1,788,884	4,315,456	10,606,501	146	1,252	965	0	68	219
70	42,426,004	24,064,426	2,067,986	4,858,762	11,434,830	147	876	688	0	42	146
71	45,739,320	25,654,998	2,378,310	5,435,588	12,270,424	148	584	481	0	2	101
72	49,081,696	27,166,123	2,744,449	6,059,004	13,112,120	149	404	343	0	3	58
73	52,448,480	28,699,147	3,115,855	6,706,365	13,927,113	150	232	192	0	4	36
74	55,708,452	30,184,389	3,404,498	7,392,126	14,727,439	151	144	132	0	0	12
75	58,909,756	31,676,793	3,640,667	8,109,897	15,482,399	152	48	43	0	0	5
76	61,929,596	32,987,483	3,931,984	8,839,668	16,170,461	153	20	19	0	0	1

largest layer; on *edash*, the state cache can only hold 29% of the largest layer. On *eadash*, more than 84% of the duplicates are eliminated with the state cache on each layer; on *mcslock*, more than 63% of the duplicates are eliminated during IDD on the largest layer in the graph. These two tables show that the LRU state cache is effective even on layers that are much larger than the state cache size.

CHAPTER 4

HASH-BASED DDD WITH AUTOMATIC PARTITIONING

This chapter describes an approach to external-memory search starting with the concept of hash-based DDD, and compares it to the approach described in the previous chapter, which uses sorting-based DDD. A hybrid approach that borrows from hash-based and sorting-based DDD is described and evaluated as well.

4.1 Basic Algorithm

External-memory breadth-first search with hash-based DDD [47, 48] generates all unique states in each layer of a breadth-first search graph in two phases: state generation and delayed duplicate detection (DDD). All states belonging to layer l are generated before moving on to layer $l + 1$. The hash-based DDD algorithm divides each layer of the state space into buckets using a hash function. There are three kinds of states: open states, candidate states, and closed states. Open states are unique states that have not yet been expanded to generate children states. Candidate states are states that have not yet been proven unique by DDD. Closed states are unique states that have been expanded to generate children states. During state generation, the algorithm expands each open state in the current layer to generate candidate states for the next layer. The candidate states are saved in files and later refined by DDD. DDD refines a candidate set by removing all duplicate

states by comparison to other candidate and closed states that are read from files belonging to the same bucket. The result of DDD is an open set that contains the open states of the next layer. The open states are written to files, one for each bucket. These two steps of state generation and DDD are repeated for every layer of the graph. The algorithm stops on one of two conditions: an error state is encountered or no more open states remain to expand.

In hash-based DDD, all states are assigned a particular bucket by a hash function. Candidate states are separated into files, with one file per bucket. Each bucket is put through DDD separately. When bucket b is put through DDD, all candidates belonging to bucket b are read from disk and put into a hash table. If the hash function is designed appropriately, all candidate states in the bucket will fit completely into a RAM hash table. Duplicate states will be recognized and not inserted twice into the hash table. Each closed state belonging to bucket b is also read from disk, and if it is a duplicate of a candidate state in the hash table, the redundant candidate state is removed. When all closed states have been reviewed, the remaining candidate states are written to disk as open states. There is one open file per bucket per layer.

The original motivation for the hash-based approach to DDD was to eliminate the overhead for sorting in sorting-based DDD [48]. But the first hash-based approaches to DDD were tested for frontier search algorithms that do not save and check previous layers of the graph for duplicates. This dissertation compares sorting-based and hash-based DDD for external-memory graph search algorithms that keep and check previous layers of the search graph for duplicates.

```
1.  externalBFS($S_0$,ErrorStates)
2.    i := 0   % index of layer
3.    OpenBucket[numBuckets] := {∅}
4.    for each $s \in S_0$   % set of initial states
5.      OpenBucket[hash_of($s$)].add($s$)
6.    CandidateBucket[numBuckets] := {∅}
7.    while $\exists_b$ OpenBucket[$b$][$i$] $\neq \emptyset$ do
      % Generate successors of all states in current layer
8.      for each $b \in$ numBuckets
9.        for each $s \in$ OpenBucket[$b$][$i$] do
10.         for each successor $s'$ of $s$
11.           if ($s' \in$ ErrorStates) return false
12.           else CandidateBucket[hash_of($s'$)].enqueue($s'$)
      % Remove duplicates among generated states
13.     i := i + 1
14.     for each $b \in$ numBuckets
15.       for each $c \in$ CandidateBucket[$b$]
16.         hash->add_if_unique($c$)
17.       for each $s \in$ ClosedBucket[$b$]
18.         hash->remove_if_present($s$)
19.       hash->write_remaining_states_to(OpenBucket[$b$][$i$])
20.       hash->clear()
21.   return true
```

Figure 4.1

External-memory BFS with hash-based DDD.

4.2 Algorithm Improvements

A challenge in using hash-based DDD is selecting a proper hash function. The hash function must create buckets that are smaller than the size of the hash table. Also, since there is overhead for creating buckets, the hash function needs to minimize the total number of buckets. This section describes an automatic partitioning algorithm for domain-independent hash-based DDD. The hash function described in this section also creates temporal locality, which is exploited by immediate duplicate detection (IDD).

4.2.1 Immediate Duplicate Detection

Immediate duplicate detection (IDD) is a method for removing many duplicate candidate states immediately, rather than waiting for DDD. IDD was described in detail in Section 3.3.2.2. Intralayer locality must be present in the open sets for the LRU state cache to be effective. Because of intralayer locality, the states in the state cache are more likely to eliminate duplicates than a randomly selected set of states from the state space. Hash-based DDD creates intralayer locality. In hash-based DDD, the open set is split into buckets by the hash function. For this research, the function is an abstraction of the state, taking into account a subset of the state variables. Under this hashing method, the states belonging to the same bucket are the same on all criteria considered by the hash function. As described in Section 3.3.2, states that are similar produce more duplicate children than state that are different. Thus, the hash bucket itself provides intralayer locality in this hash-based DDD implementation.

The algorithm detailed for this section has two hash tables, each of size 3,000 MB, one for IDD and one for DDD. If IDD and DDD shared a single hash table, the hash table would have to be wiped clean every time a new candidate set was put through DDD. If the hash table was cleaned out once per layer, it would reduce the number of intralayer duplicates eliminated. Additionally it takes time to clear the hash table. By employing two separate hash tables, the algorithm can maintain the state cache from layer to layer, which allows interlayer duplicates to be eliminated by the state cache.

4.2.2 Automatic Partition

The hash-based DDD search created for this chapter uses an automatic partition that dynamically alters the hash function to ensure candidate buckets stay smaller than the size of RAM, a feature required to do duplicate elimination using a RAM hash table. Creating such a hash function is difficult without a detailed knowledge of the state space. Instead, an automatic hash function was employed that changes based on the states encountered, where buckets that become too large are split.

The ideal hash function would have buckets of equal size for each layer, and the maximum number of candidates for a single bucket would be smaller than the size of the hash table. Ideally, the partition would never need to be altered because changing the partition can be expensive. Also the hash function would need to have the minimum number of buckets possible, since a bucket requires some overhead in time and memory. All of these ideals cannot be satisfied because they are, in some ways, contradictory. Instead, the algorithm created for this chapter focuses on a balance between reducing the number of times

repartitioning is necessary and avoiding the need for an excessive number of buckets. The next section describes the hash function developed for this algorithm: first, how the initial partition is determined, and second, how the partition is maintained throughout the search.

4.2.2.1 Creation of Initial Partition

This algorithm starts with no knowledge of the state space. A modified BFS is used to sample the state space and employs the set of states explored to create an initial hash function. In Murϕ models, rules are grouped into rule sets. The sampling search only expands the first five rules in any given rule set, ignoring other children states, which means it encounters deeper states than a pure BFS would. This algorithm samples 100 MB of the state space using this method. The sampling search also finds a wider variety of states than the first few layers of a breadth-first search, a variety that makes the initial partition less likely to need modification.

The hash function developed here is an abstraction of the state. Each state represents values for several variables, and a few selected variables are used as the hash function. A heuristic selects variables for the hash function that have a wide range of observed values - the theory is that variables with a wide variety of values will lead to more active buckets. Around a dozen variables are used for the initial partitioning function, and new variables are added until the number of potential buckets exceeds 1,000. In practice, many of those potential buckets remain empty and are not tracked by this algorithm.

If more variables are added to the initial partition, more buckets result. More buckets generally means smaller buckets and less changing of the hash function on the fly, but more

buckets does not always guarantee smaller buckets. Buckets also come with an overhead, and an excessive number of buckets is more expensive than occasionally altering the hash function.

4.2.2.2 Dynamic Repartitioning

The initial partition can sometimes be used without modification. However, sometimes the candidate states in a single bucket of the partition can grow larger than the hash table can hold. In order to ensure candidate buckets remain small, the hash function is dynamically altered. When a large bucket is encountered, states are redistributed by splitting the largest bucket and leaving the remaining buckets unchanged. The advantage of splitting a single bucket is that all states in all other buckets can remain in the files as they stand. Only states in the split bucket is rearranged. Since moving the states is expensive, there is a huge benefit to splitting only a single bucket at a time.

The hash function can be represented as a tree, as illustrated in the example in Figure 4.2. The interior nodes, drawn as rectangles, are labeled with a variable name from the state representation. The outgoing edges are labeled with possible values of that variable. The leaf nodes, drawn as circles, are labeled with a bucket number. To find the proper bucket for a state, we traverse the tree until we reach a leaf node. At each interior node, we select the branch with a label that matches this state's value for the variable of the interior node. The tree on the top left represents the initial partition, with a single interior node. The initial partition is always a balanced tree as depicted here. The tree on the top right shows the situation after bucket 2 is split. A new interior node replaces the leaf labeled

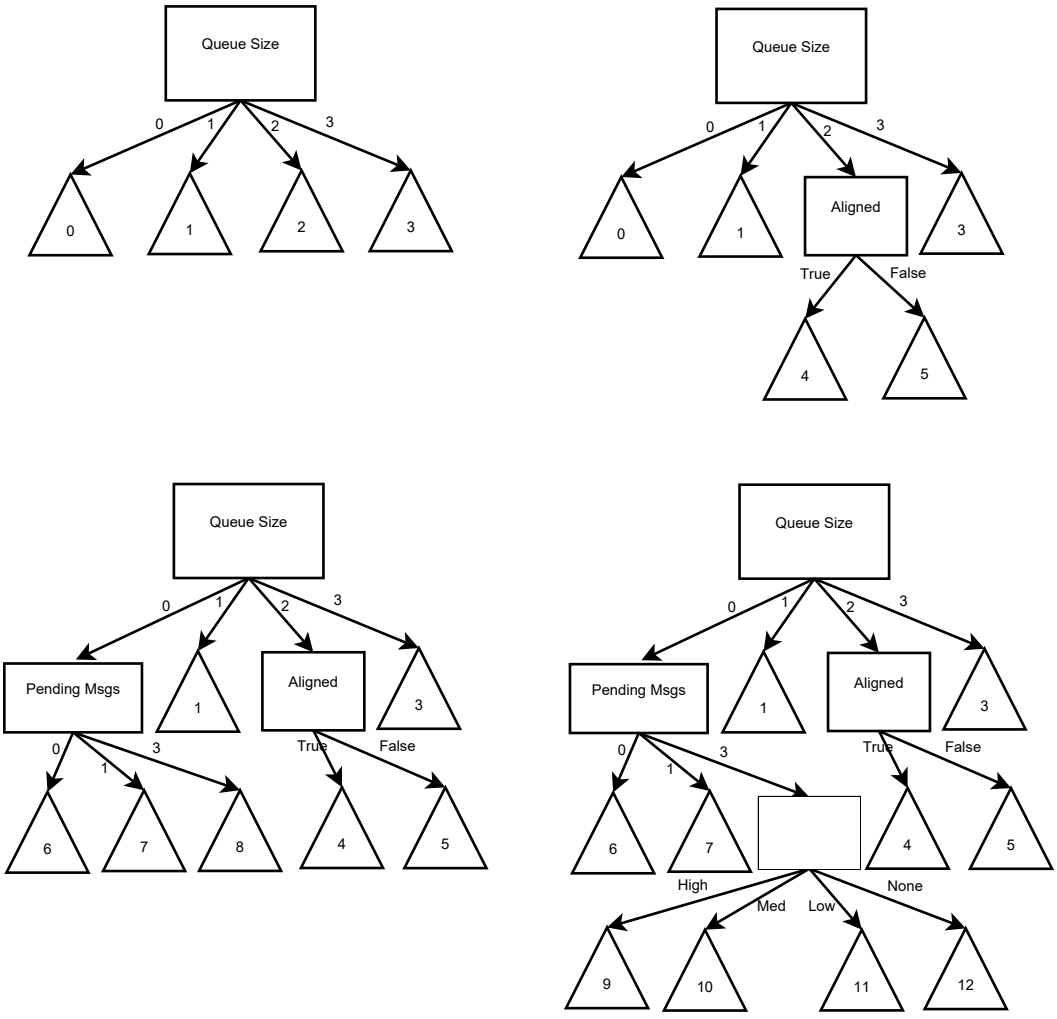


Figure 4.2

Tree Representation of Hash Function.

2, and two new leaf nodes are added as children of the node labeled $aligned$, creating an unbalanced tree. The two trees on the bottom represent the hash function after two more buckets are split. No tree balance is enforced during repartitioning.

Any time the number of candidate states in a single bucket exceeded 95% of the size of the hash table, the bucket was split. Because intralayer duplicates contained in the candidate set do not need to be stored in the hash table, this method could sometimes split buckets unnecessarily. Alternatively, it is possible to start with DDD regardless of how many candidate states exist and only split the bucket when the hash table overflows. This algorithm was designed to split buckets preemptively for three reasons. First, a bucket that just barely fits in the hash table in layer l is likely to require splitting in layer $l + 1$. That is, even if buckets are split when it is not immediately required, it will often be required soon. Second, if DDD is under way before the decision to split is made, the effort spent on DDD up to that point is wasted. Those same candidates will have to be read in again under the new partitioning scheme. Finally, splitting the bucket earlier is computationally cheaper. Every state in the bucket must be moved to new files, which is less work when the bucket is preemptively split.

The new distribution is created with a heuristic computed based on a sample of 10,000 of the most recent states in the bucket to be split. Each unused variable is considered for addition to the hash function. For each variable, sample states are divided into potential buckets, one for each possible value of that variable. Variables are scored by the size of the largest potential new bucket, and the variable that receives the lowest score is used to split the bucket. With this heuristic, a small score indicates that the split will result in small

new buckets. Ensuring that the new buckets will be small reduces the number of times repartitioning must occur.

All states belonging to the split bucket are redistributed to the new buckets created. The closed states belonging to that bucket are read from disk and then written to files for the newly created buckets. Only the states in the split bucket are rearranged, and states are moved in a single pass. Rewriting files is expensive, which inspires two goals: to minimize the number of times buckets are split and to change only a single bucket during repartitioning. The algorithm detailed in this chapter satisfies these goals well.

This system of automatic partitioning guarantees small candidate buckets on any domain with limited overhead, and use of smaller buckets reduces the number of times repartitioning is required. Ideally, the heuristic would create new buckets that contain the same number of states and would reduce the number of new buckets created. But sometimes the newly generated buckets become larger when trying to fulfill both of the goals stated in the previous paragraph, and larger buckets mean more repartitioning. For this reason, the splitting heuristic is focused solely on minimizing the size of the generated buckets.

4.2.3 Saved Partition

Repartitioning would be unnecessary if the perfect hash function were known *a priori*. A perfect hash function is impossible to guarantee before the search completes, but when a good hash function was found, it is recorded for later use. A second search on the same model can then employ the previous hash function to avoid all repartitioning time. This approach is called using a saved partition.

4.2.4 Phased Delayed Duplicate Detection

An algorithm that combines attributes of sorting-based DDD and hash-based DDD was also implemented. Generally, hash-based DDD is faster than the implementation of sorting-based DDD described in the previous chapter due to fewer state comparisons. This observation led to development of an improved implementation of sorting-based DDD, called phased DDD, which only changes the DDD portion of the search.

In phased DDD, the sorted candidates are inserted into a hash table until the hash table is full. Intralayer duplicate states are removed as they are inserted into the hash table. All states in the hash table at the same time belong to a single phase. When all candidates in a phase are in the hash table, all closed states in the same phase are checked for membership in the hash table and removed if present. The remaining states in the hash table are written to disk in sorted order as open states. Because all duplicates of states in phase p must also belong to phase p, this method removes all duplicate states. When the phase is finished, the hash table is emptied and the process is repeated on the next phase until no more candidate states remain to refine.

Phase membership is determined as the hash table fills. If the first candidate state inserted into the hash table is s_F and the last candidate inserted into the hash table is s_L, then the phase can be defined as all states that sort between s_F and s_L. Since, candidate states are inserted into the hash table in sorted order until it fills, all of those candidate states belong to the phase and sort between s_F and s_L. Finding closed states in a particular phase means finding all closed states that sort between s_F and s_L. Closed states are read in batches and put into a buffer, filling slots b_0 through b_n. Because the closed files are in

sorted order, states b_{i+1} sorts higher than b_i, and b_n will always sort last of the buffered states. State b_n is compared to s_L, if b_n is before s_L in sorted order, it is known by the property of transitivity that the entire buffer belongs to the current phase. If not, a binary search [13] is used on the buffer, until the first state (b_j) that sorts after than s_L is found. States b_0 through b_{j-1} will be used during the current phase. States b_j through b_n will be saved for the next phase. A binary search takes $O(log\ n)$ state comparisons to determine phase membership [13], where n is the number of elements to be searched through.

A phase is like a temporary bucket from hash-based DDD. Each layer of the BFS will have a different set of phases, phase membership will be discovered as the hash table fills. Each state will belong to exactly one phase per layer, which is like a bucket for just that layer. Also like a bucket, all duplicates in a phase can be eliminated by only comparing to other states that belong to the same phase. This duplicate elimination is done through the hash table, which is similar to hash-based DDD.

Phased search reduces the number of state comparisons required to complete sorting-based DDD. A simple sorting-based DDD requires two heaps, one to merge candidate files and one to merge closed files. Phased DDD does not use a heap to merge closed files. But phased search does have two categories of state comparison that a simple sorting-based DDD does not. A binary search is used to establish phase membership in the closed states, which requires state comparisons. Additionally, phased search uses a chained hash table for duplicate elimination, so each hash collision results in more state comparisons. However, when compared to simple sorting-based DDD, phased DDD reduces the overall number of state comparisons.

Phased DDD still requires more state comparisons than hash-based DDD. Hash-based DDD does not have a binary search, or sorted candidate sets, or a heap to merge candidate states into a sorted list. For these reasons, a hash-based search where repartitioning is not required will usually be faster than a phased search. However, phased search does not require a partitioning function. Phase membership is determined very simply from which candidate states fit into the hash table. In situations where a satisfactory partition is difficult to obtain, phased DDD could be superior.

4.3 Related Work

Hash-based DDD requires a hash function to partition the state space, and the partition must guarantee that no bucket exceeds the size of available RAM. Two methods previously used to generate this partition include a user-defined hash function that requires knowledge of the state space and automatically generating the partition with heuristics and sampling. Both methods will be explained.

4.3.1 Hash-Based DDD with User-Defined Partition

A partition for hash-based DDD can be generated manually. Korf and Schultze [47, 48] implemented hash-based DDD for *Rubik's Cube*, *Sliding Tile Puzzle*, and *Towers of Hanoi*. In each case, Korf and Schultze created a hash function manually that ensured a small maximum size for all buckets. They were able to create these hash functions because these problem sets have regular properties that can be discovered in small versions of the problem such as looking extensively at *Towers of Hanoi* with just three disks. The properties can then be generalized to much larger problem sets. However, it is difficult to apply this work

to unrelated problem sets because extensive knowledge of the graph is required to create a user defined hash function that meets the necessary conditions. Also, many graphs in model checking are much more complicated and do not scale in such obvious ways.

4.3.2 Hash-Based DDD with Automatic Partition

Evangelista and Kristensen [26] based their external-memory search on the algorithm by Bao and Jones [3]. The Bao and Jones algorithm expands states belonging to a single bucket until there are no more open states currently available for that bucket. It then switches to another bucket and repeats the process. Every time buckets are switched, the Bao and Jones algorithm writes the entire contents of the current bucket to disk and reads the entire contents of another bucket from disk, a process called a context switch. Cross transitions occur when a state in bucket b has a child state in a bucket other than b. In the Bao and Jones algorithm, increasing cross transitions increases the amount of disk I/O and overall time to complete the search because self transitions do not result in context switches with disk - only cross transitions do so. In this algorithm, it is beneficial to reduce the number of cross transitions.

Evangelista and Kristensen [26] created an alternative method for dynamically changing the partitioning scheme as the search progresses. Using this method allows Evangelista and Kristensen to search spaces without manually creating a hash function for the space. Their algorithm only divides buckets that are larger than available RAM, which saves time by not changing the bucketing scheme for the entire state space. When a bucket is split, Evangelista and Kristensen use a heuristic to select how to alter the partition. Evangelista

and Kristensen evaluated six different heuristic functions, mostly focused on having a minimal set of buckets. Their theory seems to have been that fewer buckets equate to fewer cross transitions. Evangelista and Kristensen then evaluated which heuristics prevented cross transitions and were able to reduce cross transitions when compared to off-the-shelf hash functions. This reduced context switching and the time required for the search. The number of cross transitions does not affect the amount of disk I/O in the algorithm described in this dissertation because each bucket is encountered no more than two times per layer: once in state expansion of open states and once in DDD of candidate states. Also, in the algorithm detailed here, the entire bucket is read from disk only during DDD, so cross transitions have no effect on the amount of disk I/O. The heuristics employed here minimize the number of repartitions required and reduce the number of buckets created.

4.4 Experimental Evaluation

This section describes the experimental results associated with this newly developed hash-based DDD algorithm. First, it is demonstrated that intralayer locality is preserved in the hash-based algorithm. Then, the effectiveness of the hash function is examined. Finally, the hash-based algorithm is compared to the sorting-based algorithm from the last chapter.

These results come from continued use of the partial DDD algorithm described in Section 3.3.4. As described in Section 4.2.1, two hash tables of equal 3,000 MB size are used, as is the state cache described in Section 3.3.2.2. As described in Section 4.2.2.2, buckets are split when the size of the candidate set exceeds 95% of the size of the hash table. Tests

were run on a machine with an Intel i7 CPU with 4 cores running at 3.07 GHz with 8 GB of RAM.

4.4.1 Temporal Locality for Immediate Duplicate Detection

Intralayer locality is present in hash-based DDD and is necessary for the LRU state cache to work effectively. Figures 4.3 and 4.4 show graphs that compare the intralayer locality for a bucketed open set without sorting to a sorted open set. These graphs show the number of duplicates eliminated by various-sized state caches for both sorting-based DDD and hash-based DDD. Hash-based DDD does not exhibit as much intralayer locality as sorting-based DDD. Specifically, hash-based DDD does not have as many duplicates with very short locality. Looking at the larger cache sizes, hash-based DDD is only slightly worse than sorting-based DDD, but sorting-based DDD eliminates more duplicates overall. Therefore, considering all variables can be advantageous for IDD with a state cache.

Intralayer locality is present in hash-based DDD because every state in a bucket is identical on every variable considered by the hash function and states are expanded in bucket order. These two features of the algorithm group open states by similarity and create the intralayer locality. The intralayer locality in hash-based DDD is a side effect of the states being bucketed by a hash function, and this particular function was chosen for its effects on bucket size and the total number of buckets, not the state cache effects. Looking at the comparison graphs (Figures 4.3 and 4.4), it is clear that some bucketing schemes are much more successful at creating intralayer locality than others, so it might be possible to choose abstraction variables to maximize the performance of the IDD.

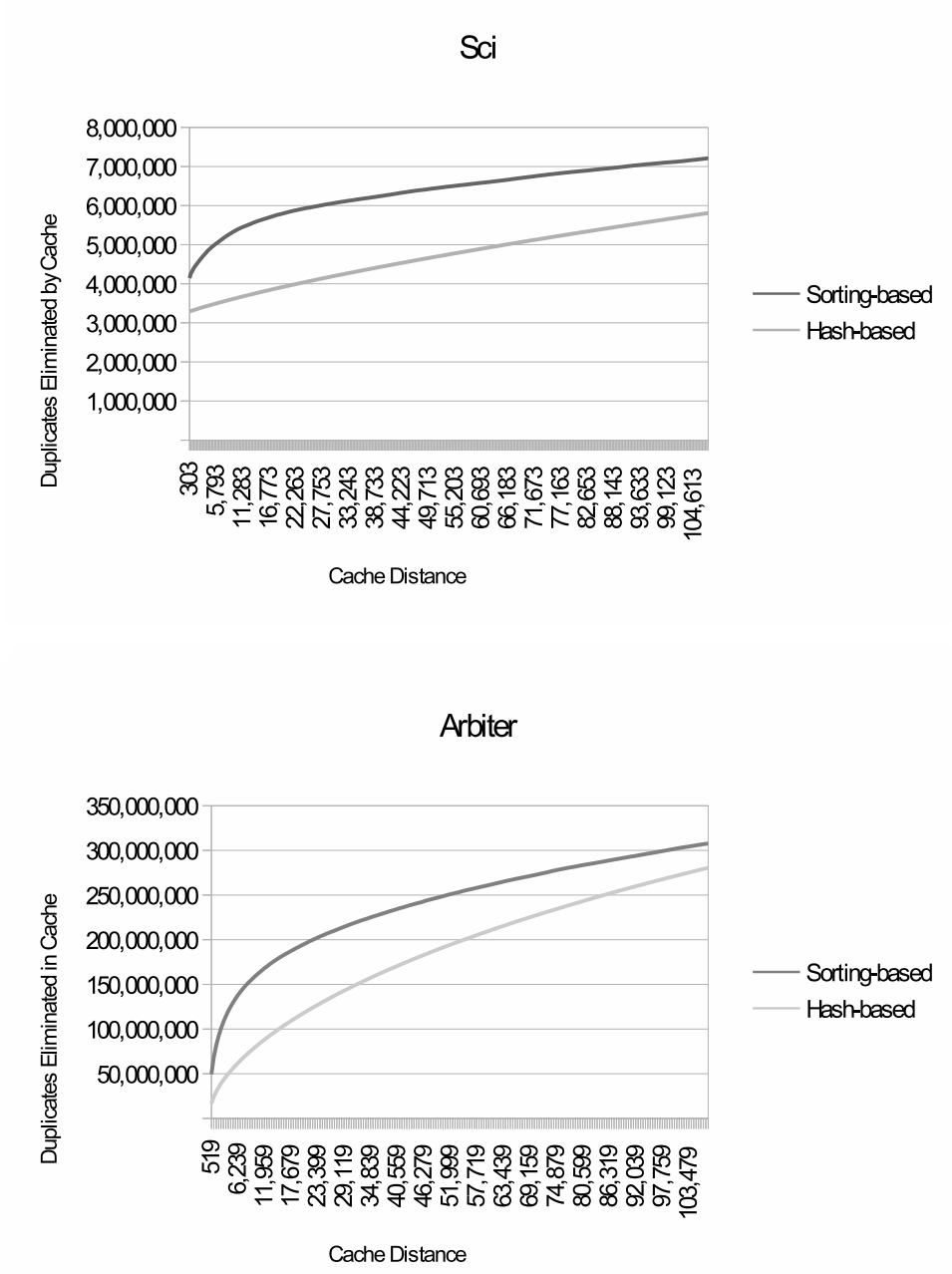
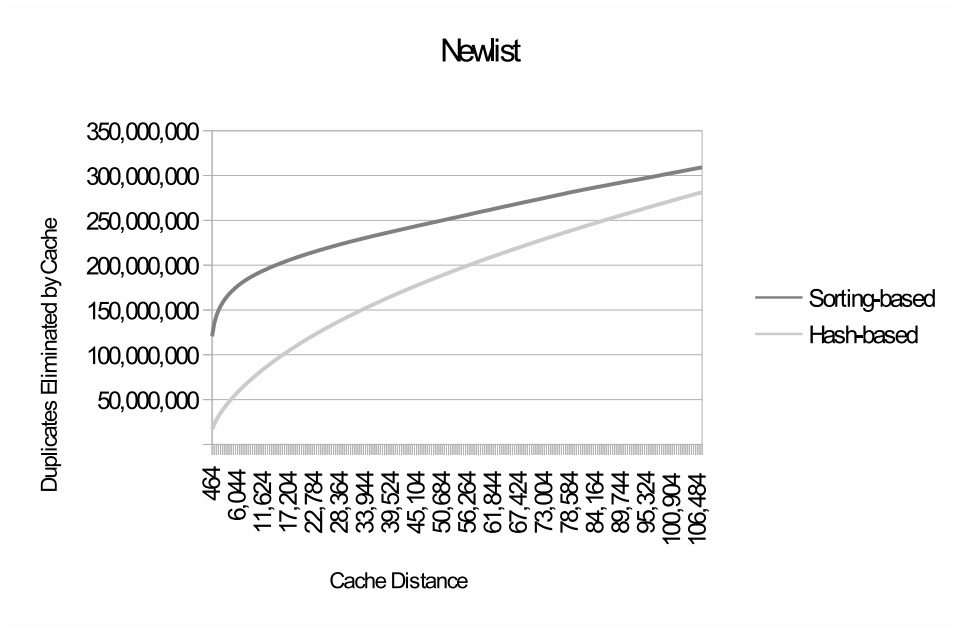


Figure 4.3

Duplicates eliminated immediately by cache size in *sci1* and *arbiter1* models with sorting and hash-based DDD.



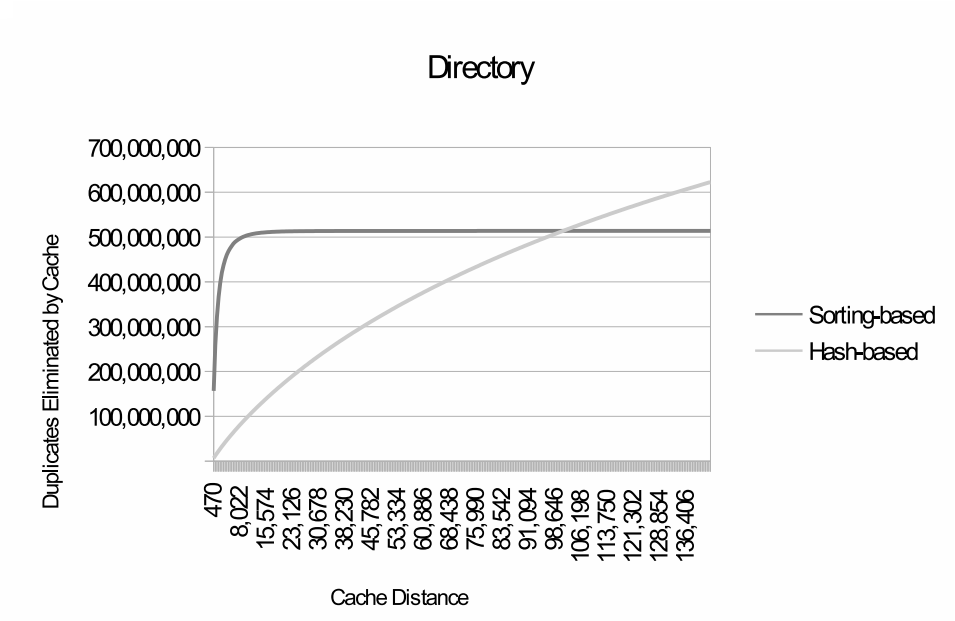


Figure 4.4

Duplicates eliminated immediately by cache size in *newlist* and *directory* models with sorting and hash-based DDD.

4.4.2 Effectiveness of the Hash Function

This section examines whether the hash function developed for this chapter outperforms randomized selections and considers several slight variations of the proposed automatic-partition function. These variant heuristics are of our own design. Comparing the algorithm to these alternatives helps explain the reasons the automatic-partitioning algorithm is successful.

4.4.2.1 Heuristics for Creating Initial Partition

This section examines several heuristics specifically for the initial partition. First an partition based on randomly selected variables. Then a partition derived from a BFS. Finally a partition that selects more initial variables.

Table 4.1

Hash function with randomly selected initial variables

	Sampling Initialization			Random Initialization		
Model	Total Time *(hh:mm:ss)*	Number Final Buckets	Number Split Buckets	Total Time *(hh:mm:ss)*	Number Final Buckets	Number Split Buckets
sci1	41:27	19	3	49:56	40	6
ns	41:24	68	1	1:06:27	93	4
kerb	1:57:58	12	1	2:47:38	50	2
arbiter1	41:48	137	0	2:14:19	512	0

A variant of the automatic-partition function presented here selects the variables used for the initial partition randomly. Table 4.1 compares the algorithm previously described with the randomly initialized partition. This subset of the models sufficiently demonstrates

the behavior of the randomly selected initial partition. For both searches, the time required to complete the search, the final number of buckets, and the number of times a bucket was split are reported. The randomly initialized partition has both more buckets split and more total buckets. An excessive number of buckets slows the search down because overhead is incurred for each bucket. Also, splitting a bucket is expensive. The automatic-partition function outperforms the randomized initialization on all measured metrics.

Table 4.2

BFS used to select initial partition

	Sampling Initialization				BFS Initialization		
Model	Total Time *(hh: mm:ss)*	Init Time *(mm:ss)*	Number Final Buckets	Number Split Buckets	Total Time *(hh: mm:ss)*	Number Final Buckets	Number Split Buckets
sci1	41:27	53	19	3	44:28	33	6
ns	41:24	55	68	1	2:30:01	226	16
kerb	1:57:58	1:05	12	1	2:35:40	62	3
arbiter1	41:48	55	137	0	2:15:50	512	0

Table 4.2 compares the automatic-partitioning algorithm with a modification that uses BFS to sample for the initial partition. These models were selected to show the range of observed behaviors. Total search time required, final number of buckets, buckets split, and time required for the sampling search to create the initial partition are reported. The sampling search must take the extra step of sampling the state space, which requires more time compared to the BFS-initialized partition, which just uses the first portion of the search as a sample. However, the initializing time is roughly one minute long and does not

greatly alter the overall time. The BFS-initialized partition, which uses the same heuristics with a different sample set, produces more buckets overall. Unfortunately, the bucket sizes produced by the BFS-initialized partition are more unevenly distributed, so more splitting is required. The sampling initialization samples deeper states in the state space and produces a more useful partition. The combination of more buckets and more splitting make the BFS-initialized partition the slower search.

Table 4.3

Adding one more variable to initial partition

	Sampling Initialization			More Initial Variables		
Model	Total Time *(hh:mm:ss)*	Number Final Buckets	Number Split Buckets	Total Time *(hh:mm:ss)*	Number Final Buckets	Number Split Buckets
sci1	41:27	19	3	41:01	21	3
ns	41:24	68	1	43:07	68	1
kerb	1:57:58	12	1	1:57:51	12	1
arbiter1	41:48	137	0	43:07	144	0
newlist	5:13:32	384	0	6:06:02	444	0

The initial partition developed in this research stops after creating 1,000 potential buckets. In this section, we compare it to an altered algorithm, with one more variable added after the usual stopping place. Results appear in Table 4.3. These models were sufficient for proving the effect of additional variables on the initial partition. For both searches, time to complete the search, final number of buckets, and number of times a bucket was split were reported. In all but one case, there was not a significant time difference between the two searches. In all of those models, the number of observed buckets was nearly iden-

tical in both searches. But for the $newlist$ model, one more variable added 60 buckets. The overhead of those buckets made the modified search take longer. This result demonstrates the overhead of buckets and encourages use of the minimum number of buckets that satisfies the properties required by hash-based DDD.

4.4.2.2 Heuristics for Dynamic Repartitioning

This section examines several heuristics for repartitioning the buckets, establishing why the heuristic described in this chapter is effective. First a randomly selected split. Then a split heuristic that minimizes the number of buckets created.

Table 4.4

Hash function with randomly selected variables used for split

	Minimal Max Size Split			Random Split		
Model	Total Time *(hh:mm:ss)*	Number Final Buckets	Number Split Buckets	Total Time *(hh:mm:ss)*	Number Final Buckets	Number Split Buckets
sci1	41:27	19	3	N/A	>1,000	>1,000
ns	41:24	68	1	43:01	68	1
kerb	1:57:58	12	1	2:15:40	16	2

The repartitioning function was altered to select a random variable to split on, with results presented in Table 4.4. The models selected for this table all required repartitioning and demonstrated the range of behaviors observed. For both searches, time required to complete the search, final number of buckets, and number of times a bucket was split were reported. On the ns model, the two searches had nearly identical performance. On the

$kerb$ model, the first randomized split was not effective, requiring a second split. On the $sci1$ model, the randomized split reached a pathological case. Every time a bucket was split, all or most of the states went to a single bucket, requiring another split. The search was stopped when it reached 1,000 buckets. The newly developed heuristic does far better than randomized selection at splitting buckets.

Table 4.5

Hash function with heuristic that minimizes the number of generated buckets

	Minimal Max Size Split			Minimal Sub-Buckets		
Model	Total Time *(hh:mm:ss)*	Number Final Buckets	Number Split Buckets	Total Time *(hh:mm:ss)*	Number Final Buckets	Number Split Buckets
sci1	41:27	19	3	48:24	90	16
ns	41:24	68	1	1:14:39	563	90

The alternative heuristic described next attempts to minimize the number of buckets created. Each potential variable is used to split the sample states into new buckets, and then the number of new buckets containing one or more sample states is counted. This alternative selects the variable that produces the fewest observed buckets, but variables that have only one active child bucket are discarded ensuring that the new buckets are always smaller than the split bucket. Table 4.5 compares this heuristic with the original algorithm developed for this chapter. For both searches, time required to complete the search, the final number of buckets, and the number of times a bucket was split are reported. In both cases, the new heuristic resulted in more split buckets and more overall time. Although

minimizing the number of buckets is an important goal, as explained previously, ensuring that a large bucket is split into small new buckets is more important. For this section, only models that require repartitioning are used. Further models required even more run time and are not reported.

4.4.3 Comparison to Sorting-Based DDD

Table 4.6

Comparing time of hash-based to sorting-based DDD.

	Sorting-Based		Hash Based		
	Simple	Phased	Automatic		Saved
	Total Time	Total Time	Total Time	Num Bucket	Total Time
Model	*(d:hh:mm:ss)*	*(d:hh:mm:ss)*	*(d:hh:mm:ss)*	Split	*(d:hh:mm:ss)*
arbiter1	55:49	42:57	41:48	0	41:15
sci1	22:28	22:06	41:27	3	20:49
newlist	5:43:28	5:26:29	5:13:32	0	4:51:54
kerb	2:40:12	2:32:02	1:57:58	1	1:57:22
ns	35:31	34:59	41:24	1	32:58
arbiter2	12:19:01	11:41:29	10:48:46	76	8:33:02
eadash	1:16:56:53	1:03:05:55	1:03:01:37	27	1:02:16:59
directory	8:42:22	8:05:33	8:17:24	7	7:03:29
sci2	11:30:19	10:44:41	10:23:32	151	8:58:16

Automatic hash generation for hash-based DDD significantly reduces the time required to complete the search, mostly by reducing the number of state comparisons, but also by removing the sort operation. Table 4.6 reports the total time required for sorting-based and hash-based DDD, as well as the number of times a bucket is split during the search and

the total time required for a search using a saved partition. Searching with repartitioning is faster than searching with sorting-based DDD on the larger models, but searching with a saved partition is always the fastest. The difference in time between saved-partition and automatic-partition searching is almost entirely due to repartitioning time. In the larger models, searching time dominates over repartitioning time; therefore, a speed-up is seen even when repartitioning is included. The time to complete a phased DDD search is between the time for the hash-based and sorting-based approaches.

The time savings exhibited by hash-based DDD largely comes from reducing the number of state comparisons, as can be seen in Table 4.7. This table shows the total time spent, time required for DDD, and number of state comparisons completed during DDD for a single search with simple sorting-based DDD, sorting-based phased DDD, and one with hash-based DDD. The layer column in the table reports the static bound for partial DDD over the total number of layers in the state space. Since state vectors are large, state comparisons are expensive. The three largest models, as measured by GB for storing the state set, require much more time during DDD than the smaller models. When measured this way, the larger state set is partially due to the number of states and partially due to the size of an individual state. Both of those factors increase the time spent on the closed set during DDD. Remember each state in layer l is part of DDD $L - l$ times as a closed state, where L is the total layers in the graph. If layer l has n states that is $n * (L - l)$ states used as a closed state in DDD. The multiplicative nature of closed states during DDD account for larger models spending a larger portion on DDD.

Table 4.7

Comparing time of hash-based to sorting-based DDD; state comparisons

Model	Algorithm	Total Time *(d:hh:mm:ss)*	DDD Time *(d:hh:mm:ss)*	State Comparisons	Lay-ers
arbiter1	Sorting-based	55:49	20:32	2,812,276,020	15/31
	Phased DDD	42:57	10:55	1,013,046,619	
	Hashed-based	41:15	1:33	727,540,638	
newlist	Sorting-based	5:43:28	33:45	39,840,685,268	55/110
	Phased DDD	5:26:29	13:33	1,592,461,317	
	Hashed-based	4:51:54	2:41	755,429,862	
kerb	Sorting-based	2:40:12	22:52	539,591,146	2/28
	Phased DDD	2:32:02	21:57	415,673,648	
	Hashed-based	1:57:22	11:19	21,493,558	
mcslock	Sorting-based	1:12:01:30	1:07:01:17	675,461,989,138	86/154
	Phased DDD	1:09:25:53	1:03:02:57	55,803,846,707	
	Hashed-based	23:34:54	17:41:02	3,141,882,730	
eadash	Sorting-based	1:16:56:53	1:09:43:14	95,272,642,030	34/63
	Phased DDD	1:03:05:55	16:10:25	9,925,349,583	
	Hashed-based	1:02:16:59	12:37:05	961,697,774	
directory	Sorting-based	8:42:22	6:33:54	47,860,383,324	8/114
	Phased DDD	8:05:33	5:54:09	18,895,064,859	
	Hashed-based	7:03:29	4:08:36	3,500,445,738	
sci2	Sorting-based	11:30:19	4:22:26	8,483,872,618	2/7
	Phased DDD	10:44:41	3:35:09	8,434,772,822	
	Hashed-based	8:58:16	2:04:08	114,777,097	

Sorting-based DDD requires more state comparisons during DDD than hash-based DDD. The actual comparisons are done with memcmp. We will now describe what state comparisons are necessary for sorting-based DDD. As explained in Section 3.3.1, in our sorting-based DDD, states from several files are merged into a single sorted list using a priority queue implemented as a heap. Each update to the heap involves $log(10 * n)$ state comparisons, where n is the number of files merged and ten states from each file are included in the heap at all times. There are two heaps, one to merge the candidate files and one to merge the closed files. Each candidate state s dequeued from the heap will be compared to additional candidate states from the heap until a state is found that is not a duplicate of s. The candidate state s will then be compared to closed states dequeued from the other heap until a duplicate of s is found or a state past s in the sorted order is found. All of these state comparisons (heap reorder, candidate to candidate, and candidate to closed) are counted and put into the comparisons column in Table 4.7.

Hash-based DDD requires fewer state comparisons during DDD than sorting-based DDD. During hash-based DDD, the candidate states are read from the candidate file and placed into a chained hash table. When a collision occurs in a chained hash, the states are kept in a linked list. So when state s hashes to an entry in the table containing n states, state s is compared to all n states to see if s is a duplicate of any of them. If it is not, state s is added at the tail of the linked list. After all candidate states are in the linked list, the closed states are used to find duplicates in the hash table. Each closed state is hashed and compared to all elements in the linked list it collides with. All state comparisons looking for duplicates in the linked list are counted and put into the comparisons column in Ta-

ble 4.7. Hash-based DDD always reduces the number of state comparisons dramatically. This is because the number of comparisons for hash collisions is much less than comparisons to maintain a priority queue in a sorted order. As explained in Section 2.2.5, in model checking state vectors are large and state comparisons are costly. Large reductions in the number of state comparisons result in a corresponding reduction in total time required and time spent for DDD.

The phased DDD algorithm combines sorting-based DDD with some elements of hash-based DDD. Phased DDD also ends up between these two algorithms in performance and state comparisons. In phased DDD, the candidate states are merged into a single sorted order with a heap. The candidate merging process is the same as the simple sorting-based DDD algorithm, so it uses the same number of state comparisons. Then the candidate states are inserted into a hash table in a similar manner as hash-based DDD. Inserting states into a hash table involves state comparisons due to hash collisions. The closed states belonging to a single phase are read from disk and detected by a binary search, also requiring state comparisons, as explained in Section 4.2.4. Phased DDD has fewer state comparisons than the simple sorting-based search because it does not employ a heap to merge the closed states. Using a hash table combined with binary search requires fewer state comparisons than simple sorting-based DDD. The reduced number of state comparisons is why phased DDD is faster than simple sorting-based DDD.

Phased DDD is not as efficient as hash-based DDD. Phased DDD requires more state comparisons than hash-based DDD. The main reason phased DDD requires more state comparisons than hash-based DDD is the merging of the candidate states. Many candidate

files are merged into a single sorted order using a heap. The heap requires many state comparisons that the hash-base DDD does not require. Since hash-based DDD does not need the candidate states in any particular order, they can be put into the hash table in the same order they are in the files. Because hash-based DDD uses fewer state comparisons than phased DDD it is the faster search. Phased DDD does reduce state comparisons and time when compared to the simple sorting-based DDD, but not as much as hash-based DDD.

The models that use a higher layer bound show a larger reduction in the number of state comparisons. Higher bounds mean more layers are used during DDD. More layers in DDD equates to more closed states used in DDD. When more closed states are used, methods that reduce the required state comparisons with closed states, such as phased DDD, will have more of an effect. You can see that difference in Table 4.7. In the *kerb* and *directory* models, where the layer bound is very low relative to the total number of layers, there is a small difference in the number of state comparisons between simple sorting-based DDD and phased DDD. In the *newlist*, *mcslock*, and *eadash* models the layer bound is high relative to the total number of layers. For these high bound models, there is a comparatively large reduction in the number of state comparisons between the simple sorting-based DDD and phased DDD searches. The number of layers used during DDD effects the reduction in state comparisons observed in phased DDD.

State comparisons are always expensive, but the cost of a state comparison is proportional to the size of the state vector. The *eadash* model has much larger state vectors than the *mcslock* model. The size of the state vector helps explain why phased sort reduces the

number of state comparison for both *eadash* and *mcslock* by a factor of ten but *eadash* has a much bigger reduction in DDD time than is observed for *mcslock*. Models with larger state vectors demonstrate a larger speedup relative to the number of state comparisons reduced.

CHAPTER 5

CONCLUSION

"State explosion" is a central problem for graph search and model checking. State spaces increase exponentially with the number of variables in the state description, making the large models common in the real world difficult to verify. Use of external memory can circumvent the limits on RAM size imposed by cost and computer architecture, but external-memory algorithms come with their own set of challenges, including slow I/O time and seek time latency. This dissertation explores a variety of means to improve the scalability of model checking in external memory.

5.1 Contributions

A brief summary of some of the key contributions of this dissertation follows.

5.1.1 Improvements to Sorting-based DDD

Several improvements of sorting-based DDD have been developed that improve its efficiency. The two most effective are use of a more efficient internal sorting algorithm and use of phased delayed duplicate detection.

5.1.1.1 Internal Sort Algorithm

The sorting algorithm used can have a large influence on performance. It was found that sorting states internally before they are written to disk is most efficient. The states are sorted in the state buffer that is already used for buffered I/O, and an indexed sort is used to move small state pointers instead of large state descriptions. A three-way radix quicksort [5, 6] designed specifically for sorting large keys of fixed size since model checking states meet this criteria.

5.1.1.2 Phased Delayed Duplicate Detection

Phased DDD combines features of sorting-based and hash-based DDD. A hash table is used to eliminate duplicates, which is more efficient than comparing two sorted lists for duplicate elimination. No partitioning (i.e. hash) function is needed. Instead the state space is divided into phases in a way that never requires repartitioning. All candidate states that fit into the hash table belong to a single phase. Using the last candidate state to be entered into the hash table, it is possible to determine phase membership of all closed states. Duplicates are removed by only using closed states from the same phase, much like hash-based DDD uses only closed states in the same bucket. Phased DDD reduces the number of state comparisons relative to a simple sorting-based algorithm, which is how phased DDD saves time. When the time to generate the partition is not considered, phased DDD is not as efficient as hash-based DDD. However, phased DDD does not require a hash function, so it can be a good alternative when hash functions are difficult to obtain.

5.1.2 Hash-Based DDD with Automatic Partitioning

Use of hash-based DDD effectively reduces DDD to a linear-time operation [47, 48]. It also reduces the number of state comparisons necessary for DDD. However, hash-based DDD requires a specialized hash function that ensures buckets fit in RAM. Finding the best bucketing scheme is as difficult a problem as performing the verification itself [36]. To be domain independent, a hash function must be generated automatically. To automatically generate a hash function that satisfies the requirements of hash-based DDD, the state space must be dynamically repartitioned. An initial hash function is used until a bucket becomes too large, then a new hash function is generated. Both the initial hash function and the subsequent alterations are created by heuristics. Automatic partitioning allows the domain-independent use of hash-based DDD, which saves time by eliminating many of the state comparisons.

5.1.3 Leveraging Local Structure

This dissertation shows that model checking graphs exhibit two types of locality that can be exploited for a more efficient search: interlayer locality and intralayer locality. Both forms of locality are shown to exist in hash-based and sorting-based DDD.

5.1.3.1 Interlayer Locality

Interlayer locality is the principle that short interlayer edge distances are more common than long interlayer edge distances. Interlayer locality has been shown to exist for some time [64]. Delayed duplicate detection consumes a large portion of external-memory search time. By leveraging interlayer locality in external memory to perform partial de-

layed duplicate detection, which means only checking the most recent layers for duplicates, DDD time is reduced, while most delayed duplicates are still eliminated.

5.1.3.2 Intralayer Locality

Intralayer locality refers to duplicate states within a layer having small cache distances. Intralayer locality has not previously been explored in model checking. Structured duplicate detection (SDD) [81] is a complicated predictive cache algorithm that organizes a state cache to exploit intralayer locality in planning problems. However, simple methods, such as sorting and grouping by abstract states, can effectively create intralayer locality in model-checking graphs.

Intralayer locality can be created in problems other than traditional model checking, including the multiple sequence alignment problem in bioinformatics, in search problems common in artificial intelligence, and in learning the optimal structure of a Bayesian network [57]. Evidence supports the idea that intralayer locality can be created in many graphs.

In external-memory search, the option to use a state cache has often been ignored [17, 75] or implemented inefficiently [35, 66]. The state cache presented in this dissertation exploits both intralayer and interlayer locality [64]. By maintaining the most recent states in the cache, it can eliminate many duplicates in RAM. Time is saved because fewer duplicates needed to be considered by DDD and space is saved because fewer duplicates are stored in external memory.

5.2 Publication Summary

Portions of this dissertation have been published on two occasions. A first non-refereed four page paper detailed the layered state cache [53] that was the origin of the LRU state cache explained in Section 3.3.2.2. A second referred publication retained the same state cache and included the partial-DDD algorithm [54] described in Section 3.3.4.

5.3 Future Work

Two natural extensions to the algorithms in this dissertation are proposed and explained.

5.3.1 Heuristic for Partial Delayed Duplicate Detection

The partial delayed duplicate detection algorithm in this dissertation requires a manually set layer bound. This bound is best set only after extensive knowledge of the state space has been obtained. Ideally a heuristic would be used to create the bound. The algorithm would sample the state space to determine a bound. The bound would not have to be static, instead it could be adjusted as the search progressed. Such a heuristic would have to account for potential exponential increases in duplicates if the bound were too short. Not only would a heuristic make partial DDD domain independent, but it has the potential to increase the efficiency of the partial DDD.

5.3.2 Parallel Algorithm

One possible method of increasing the speed of external-memory search is to parallelize the approach. Model checking has been parallelized in multiple ways. Murϕ was

first parallelized by Stern and Dill [74]. Kumar and Mercer [50] added load balancing to the same design. Parallel and external-memory approaches were added to the SPIN model checker by Jabbar, Edelkamp, and Stefan Schrödl [17, 41]. Jabbar expanded on this implementation in his dissertation [40]. Both the sorting-based and hash-based DDD algorithms described in this dissertation could be parallelized. We will now explain a shared memory parallel algorithm and two parallel distributed memory algorithms. One distributed memory algorithm is based on sorting-based DDD and the other on hash-based DDD.

5.3.2.1 Parallel Shared Memory Algorithm

The algorithm could be parallelized with shared memory. In this approach all threads would share access to the same data structures. This could be based on the shared hash table described by Zhang and Hansen [80]. Each thread expands different states, but checks for duplicates using the same hash-table. The hash-table is a variant of a chained hash meant for threaded access. This approach is best suited to a single node multithreaded architecture.

5.3.2.2 Parallel Sorting-based Delayed Duplicate Detection

As demonstrated by the Jabbar algorithm [40], parallel external-memory search can be combined with sorting-based DDD. A parallel external memory search with sorting-based DDD still needs a hash function to divide the states into buckets. The buckets are used to divide the work among the threads. In the Jabbar algorithm a heuristic was used as the hash function. Unlike search with hash-based DDD, the hash function used by a parallel sorting-based approach does not have to be constructed to ensure all buckets are

less than the size of RAM. This is an advantage over hash-based DDD, which has stringent requirements on the hash-function employed. A suitable hash function for a hash-based search can be difficult to obtain, where almost any function will work for a sorting-based search. The primary goal for a parallel sorting-based DDD hash function is to divide the state space such that each bucket is approximately the same size.

The parallel implementation is based around a work queue, with access to the queue controlled by a mutex. Any thread that has finished its current work will take the next chunk of work from the queue. Work comes in two types: open sets and candidate sets. The work queue is sorted to ensure the files are processed in the correct order and is guided by two factors: the layer of the file (files of layer l will be processed before files of layer $l + 1$) and the type of state in the file (files containing candidate states will be processed before files containing open states of the same layer). Candidate files are refined first because duplicates must be eliminated before states can be expanded. Ordering the work queue ensures that all duplicate states are recognized and that the shortest error path is always discovered. The work queue allows threads to share the search tasks in a balanced way.

The threads must synchronize which layer they are processing because parent states from multiple open files can generate children states that belong to a single candidate file. To ensure that all candidate states belonging to bucket b and layer l are correctly merged into a single file, no work can begin on layer l until all open states of layer $l - 1$ have been expanded. Some idle time would results from each thread waiting for the last open bucket of a particular layer to be expanded.

A thread that takes an open set from the queue will expand states, which is the same as in the serial search. States in the open set are read from disk and designated as parent states, and children states are generated. A hash table is used to perform IDD, which reduces the number of candidate states written to disk and refined during DDD. The hash function described in the first paragraph of this section is used to distribute the states into candidate buckets. Expanded states that are not recognized as duplicates are sorted and written to disk with buffered I/O. When all states in the open set have been expanded, any remaining states in the buffers are sorted and flushed to disk. All candidate files are enqueued into the work queue, and the thread then dequeues the next chunk of work.

Some of the work items in the work queue are refinement tasks. These are candidate buckets that the algorithm refines through DDD to create open states. The sorting-based approach compares the sorted candidate states against sorted closed lists. Phased DDD, described in Section 4.2.4, should be used since it is more efficient than the simple sorting-based DDD. One change to sorting-based DDD is that delayed duplicate detection would only be applied to one bucket at a time, rather than the whole candidate layer. This would involve further subdividing buckets into phases.

The search continues until one of two possible conditions are met. First, an error state could be found by any thread expanding an open state. Since open states are expanded in layer order and threads are synchronized by layer, the first error state detected is guaranteed to have the shortest error path. The second termination condition is reached if all threads are idle and the work queue is empty. In such a case, the model is verified because no error

state is reachable. The parallel search will always terminate with the same conditions as a serial search.

One potential challenge with parallelizing our serial algorithm is that both the number of cores available and RAM space are limited resources. This algorithm has large hash tables for both IDD and phased DDD. It is possible to use a single hash table for both usages, which reduces RAM requirements. But using a single table would also reduce the effectiveness of the state cache, because it would no longer have a completely LRU replacement policy. Instead sometimes the state cache could be emptied and hash table repurposed for phased DDD. Korf and Schultze [48] found that the number of threads used should exceed the number of cores available, but when taken to the extreme, it would also be possible to use more threads than can be well supported by the cores. If ample RAM and processing cores are available, these challenges would not have to be addressed directly.

5.3.2.3 Parallel Hash-based Delayed Duplicate Detection

The hash-based delayed duplicate detection algorithm can also be parallelized. The same work queue structure could be used, with threads switching between expansion and refinement tasks. One difference is that the hash-function described in Section 4.2.3 is used to distribute the states into candidate buckets. This hash-function has to guarantee each bucket never exceeds the RAM size available.

The expansion portion of the parallel algorithm can be nearly identical for a sorting-based and hash-based approach. States are dequeued from open files and expanded to generate children states. Immediate duplicate detection can be applied to reduce the number

of candidate states. The expansion order of the states will be different between hash-based and sorting-based approaches, which effects intralayer locality and the performance of the state cache, as explained in Section 4.4.3. The remaining candidate states are written to candidate files. For the state expansion portion of a parallel search, hash-based and sorting-based approaches differ in only two respects: sorting candidate states before writing them to disk and the effectiveness of the state cache.

Threads may also take candidate sets from the work queue. Candidate sets are refined to remove duplicate states, which can be hash-based DDD. The thread goes through each state in a candidate file, adding unique states to a hash table. Duplicate states are detected when a copy of the state already exists in the hash table, with the second copy being discarded. Since adding a new state to the hash table and detecting a duplicate are the same action, no time is wasted. When all candidate states in the bucket have been processed, the thread then reads through all closed files belonging to the same bucket in previous layers. If any of the closed states are duplicated in the hash table, they are removed. When all unique candidates have been added to the hash table and all duplicates of closed states removed, the remaining states are written to disk. The open file is enqueued into the work queue, and the thread then dequeues the next chunk of work.

The same exit conditions as the parallel sorting-based algorithm apply. The search would terminate if an error state is found or the open state becomes empty. In all cases the algorithm would produce the same result as the serial algorithm.

The same resource limitations described for parallel sorting-based DDD apply to parallel hash-based DDD. Parallel hash-based DDD also has large hash tables for DDD and

IDD, which may be shared with some penalties. The number of threads used would have to be balanced against the available resources.

5.4 Significance

Developing techniques for improving the efficiency and scalability of a graph-search algorithm for an external-memory model checker will allow searches of larger spaces and make such searches faster. Model checking reduces the presence of critical errors in expensive and safety-critical systems, saving money and lives. The ability to verify larger models faster increases our ability to verify systems completely, including more complex, real-world systems. Verification of large problems is especially important for safety-critical systems like avionics controls. The algorithm developed in the research for this dissertation was tested on graphs that require hundreds of gigabytes of disk storage and days to search, pushing the boundaries of model-checking size.

Finally, since the techniques applied herein are domain independent, they apply to any large graph-search problem, including such search problems as those found in artificial intelligence, planning, high-performance computing, and other areas.

REFERENCES

[1] S. E. Adewmi, "Index Sort Algorithm for Positive Integers," *Science World Journal*, vol. 3, no. 3, 2008, pp. 23–25.

[2] D. Ajwani and U. Meyer, "Design and Engineering of External Memory Traversal Algorithms for General Graphs," *Algorithmics of Large and Complex Networks*, J. Lerner, D. Wagner, and K. A. Zweig, eds. 2009, vol. 5515 of *Lecture Notes in Computer Science*, pp. 1–33, Springer.

[3] T. Bao and M. Jones, "Time-Efficient Model Checking with Magnetic Disk.," *Proceedings of 11th International Conference Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005*, N. Halbwachs and L. D. Zuck, eds. April 2005, vol. 3440 of *Lecture Notes in Computer Science*, pp. 526–540, Springer.

[4] J. Barnat, L. Brim, and P. Rockai, "On-the-fly parallel model checking algorithm that is optimal for verification of weak LTL properties," *Sci. Comput. Program.*, vol. 77, no. 12, 2012, pp. 1272–1288.

[5] J. Bentley and R. Sedgewick, "Fast Algorithms for Sorting and Searching Strings," *Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, New Orleans, 1997.

[6] J. Bentley and R. Segdewick, "Sorting Strings with Three-way Radix Quicksort," *Dr. Dobb's Journal*, 1998.

[7] R. Bhattacharya, S. M. German, and G. Gopalakrishnan, "Exploiting Symmetry and Transactions for Partial Order Reduction of Rule Based Specifications," *SPIN*, A. Valmari, ed. 2006, vol. 3925 of *Lecture Notes in Computer Science*, pp. 252–270, Springer.

[8] D. Bosnacki and S. Leue, eds., *Model Checking of Software, 9th International SPIN Workshop, Grenoble, France, April 11-13, 2002, Proceedings*, vol. 2318 of *Lecture Notes in Computer Science*. Springer, 2002.

[9] T. Bultan and P.-A. Hsiung, eds., *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*, vol. 6996 of *Lecture Notes in Computer Science*. Springer, 2011.

[10] E. Burns and R. Zhou, "Parallel Model Checking Using Abstraction," *SPIN*, A. F. Donaldson and D. Parker, eds. 2012, vol. 7385 of *Lecture Notes in Computer Science*, pp. 172–190, Springer.

[11] X. Chen and G. Gopalakrishnan, *Predicate Abstraction for Murphi*, Tech. Rep. UUCP-06-002, School of Computing Univeristy of Utah, Salt Lake City, UT 84112 USA, 2006.

[12] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 2000.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, vol. 3, chapter 12, Massachusetts Institute of Technology Press, 2009.

[14] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis, "Memory-Efficient Algorithms for the Verification of Temporal Properties," *Formal Methods in System Design*, vol. 1, no. 2/3, 1992, pp. 275–288.

[15] D. L. Dill, "The Murϕ Verification System," *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, R. Alur and T. Henzinger, eds., New Brunswick, NJ, / 1996, vol. 1102 of *Lecture Notes in Computer Science*, pp. 390–393, Springer Verlag.

[16] D. L. Dill, "A Retrospective on Murϕ," *25 Years of Model Checking*, O. Grumberg and H. Veith, eds. 2008, vol. 5000 of *Lecture Notes in Computer Science*, pp. 77–88, Springer.

[17] S. Edelkamp, S. Jabbar, and S. Schrödl, "External A*," *KI*, S. Biundo, T. W. Frühwirth, and G. Palm, eds. 2004, vol. 3238 of *Lecture Notes in Computer Science*, pp. 226–240, Springer.

[18] S. Edelkamp, S. Jabbar, and D. Sulewski, "Distributed Verification of Multi-threaded C++ Programs," *Electr. Notes Theor. Comput. Sci.*, vol. 198, no. 1, 2008, pp. 33–46.

[19] S. Edelkamp, M. Kellershoff, and D. Sulewski, "Program Model Checking via Action Planning," In van der Meyden and Smaus [77], pp. 32–51.

[20] S. Edelkamp, P. Sanders, and P. Simecek, "Semi-external LTL Model Checking," *CAV*, A. Gupta and S. Malik, eds. 2008, vol. 5123 of *Lecture Notes in Computer Science*, pp. 530–542, Springer.

[21] S. Edelkamp, V. Schuppan, D. Bosnacki, A. Wijs, A. Fehnker, and H. Aljazzar., "Survey on Directed Model Checking.," *MoChArt*, D. Peled and M. Wooldridge, eds., Berlin, Heidelberg, 2008, vol. 5348 of *Lecture Notes in Computer Science*, pp. 65–89, Springer-Verlag.

[22] S. Edelkamp and D. Sulewski, "Efficient Explicit-State Model Checking on General Purpose Graphics Processors," *SPIN*, J. van de Pol and M. Weber, eds. 2010, vol. 6349 of *Lecture Notes in Computer Science*, pp. 106–123, Springer.

[23] S. Edelkamp and D. Sulewski, "External Memory Breadth-First Search with Delayed Duplicate Detection on the GPU," In van der Meyden and Smaus [77], pp. 12–31.

[24] A. E. Emerson, S. German, J. Havlicek, and A. Venkataramani, "Model Checking A Parameterized Directory-based Cache Protocol,", 2002.

[25] S. Evangelista, "Dynamic Delayed Duplicate Detection for External Memory Model Checking," *SPIN*, K. Havelund, R. Majumdar, and J. Palsberg, eds. 2008, vol. 5156 of *Lecture Notes in Computer Science*, pp. 77–94, Springer.

[26] S. Evangelista and L. M. Kristensen, "Dynamic State Space Partitioning for External Memory Model Checking," *FMICS*, M. Alpuente, B. Cook, and C. Joubert, eds. 2009, vol. 5825 of *Lecture Notes in Computer Science*, pp. 70–85, Springer.

[27] S. Evangelista and L. M. Kristensen, *Search-Order Independent State Caching*, Tech. Rep., DAIMI - Aarhus University, Denmark, 2009.

[28] S. Evangelista and L. M. Kristensen, "Search-Order Independent State Caching," *T. Petri Nets and Other Models of Concurrency*, vol. 4, 2010, pp. 21–41.

[29] S. Evangelista and L. M. Kristensen, "Dynamic state space partitioning for external memory state space exploration," *Sci. Comput. Program.*, vol. 78, no. 7, 2013, pp. 778–795.

[30] S. Evangelista, L. Petrucci, and S. Youcef, "Parallel Nested Depth-First Searches for LTL Model Checking," In Bultan and Hsiung [9], pp. 381–396.

[31] S. Evangelista, M. Westergaard, and L. M. Kristensen, "The ComBack Method Revisited: Caching Strategies and Extension with Delayed Duplicate Detection," *CPN*, 2008.

[32] J. Geldenhuys, "State Caching Reconsidered.," In Graf and Mounier [34], pp. 23–38.

[33] P. Godefroid, G. J. Holzmann, and D. Pirottin, "State-Space Caching Revisited.," *CAV*, G. von Bochmann and D. K. Probst, eds. 1992, vol. 663 of *Lecture Notes in Computer Science*, pp. 178–191, Springer.

[34] S. Graf and L. Mounier, eds., *Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings*, vol. 2989 of *Lecture Notes in Computer Science*. Springer, 2004.

[35] M. Hammer and M. Weber, ""To Store or not to Store" Reloaded: Reclaiming Memory on Demand," *11th International Workshop on Formal Methods for Industrial Critical Systems*. 2006, Springer-Verlag.

[36] V. Holub and P. Tuma, "Streaming State Space: A Method of Distributed Model Verification," *TASE*. 2007, pp. 356–368, IEEE Computer Society.

[37] G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*, Addison-Wesley, 2003.

[38] IEEE Computer Society, *IEEE Standard for Scalable Coherent Interface (SCI)*, IEEE Computer Society, 1992, IEEE Standard 1596-1992.

[39] R. Iosif, "Symmetry reduction criteria for software model checking," *In Proceedings of Ninth International SPIN Workshop*. April 2002, vol. 2318 of *LNCS*, pp. 22–41, Springer.

[40] S. Jabbar, *External Memory Algorithms for State Space Exploration in Model Checking and Action Planning*, doctoral dissertation, Technischen Universität Dortmund and der Fakultät für Informatik, Dortmund, Germany, 2008.

[41] S. Jabbar and S. Edelkamp, "Parallel External Directed Model Checking with Linear I/O.," *VMCAI*, E. A. Emerson and K. S. Namjoshi, eds. 2006, vol. 3855 of *Lecture Notes in Computer Science*, pp. 237–251, Springer.

[42] M. Jones, E. Mercer, T. Bao, R. Kumar, and P. Lamborn, "Benchmarking Explicit State Parallel Model Checkers," *2nd International Workshop on Parallel and Distributed Methods in Verification*, July 2003.

[43] D. Knuth, *The Art of Computer Programming*, 2nd ed. edition, Addison-Wesley, 1998.

[44] J. T. Kohl, B. C. Neuman, and T. Y. Ts'o, "The Evolution of the Kerberos Authentication Service," *Proceedings of the Spring 1991 EurOpen Conference*. 1991, pp. 78–94, IEEE Computer Society Press.

[45] R. E. Korf, "Delayed Duplicate Detection: Extended Abstract," *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03), Acapulco, Mexico*, G. Gottlob and T. Walsh, eds. 2003, Morgan Kaufmann.

[46] R. E. Korf, "Best-First Frontier Search with Delayed Duplicate Detection," *AAAI*, D. L. McGuinness and G. Ferguson, eds. 2004, pp. 650–657, AAAI Press / The MIT Press.

[47] R. E. Korf, "Linear-time disk-based implicit graph search," *J. ACM*, vol. 55, no. 6, 2008.

[48] R. E. Korf and P. Schultze, "Large-scale parallel breadth-first search," *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05)*, July 2005, pp. 1380–1385.

[49] L. M. Kristensen and T. Mailund, "A Generalised Sweep-Line Method for Safety Properties.," *Proceedings of the International Symposium of Formal Methods Europe*, L.-H. Eriksson and P. A. Lindsay, eds. July 2002, vol. 2391 of *Lecture Notes in Computer Science*, pp. 549–567, Springer.

[50] R. Kumar and E. G. Mercer, "Load Balancing Parallel Explicit State Model Checking," *Electr. Notes Theor. Comput. Sci.*, vol. 128, no. 3, 2005, pp. 19–34.

[51] A. Laarman, R. Langerak, J. van de Pol, M. Weber, and A. Wijs, "Multi-core Nested Depth-First Search," In Bultan and Hsiung [9], pp. 321–335.

[52] A. Laarman, J. van de Pol, and M. Weber, "Boosting Multi-Core Reachability Performance with Shared Hash Tables," *CoRR*, vol. abs/1004.2772, 2010.

[53] P. Lamborn and E. A. Hansen, "Memory-efficient graph search in planning and model checking," *Doctoral Consortium of The International Conference on Automated Planning and Scheduling (ICAPS)*. June 2006, The International Conference on Automated Planning and Scheduling.

[54] P. Lamborn and E. A. Hansen, "Layered Duplicate Detection in External-Memory Model Checking," *In 15th International SPIN Workshop on Model Checking of Software*, 2008, pp. 160–175.

[55] L. Lamport, "Proving the Correctness of Multiprocess Programs.," *IEEE Transactions on Software Engineering*, vol. 3, no. 2, 1977, pp. 125–143.

[56] D. Lenoski, *DASH Prototype System*, doctoral dissertation, Stanford University, 1992.

[57] B. Malone, C. Yuan, E. A. Hansen, and S. Bridges, "Improving the scalability of optimal Bayesian network learning with external-memory frontier breadth first branch and bound search," *27th Conference on Uncertainty in Artificial Intelligence (UAI-11)*, 2011, pp. 479–488.

[58] R. Mateescu and A. Wijs, "Hierarchical Adaptive State Space Caching Based on Level Sampling," *TACAS*, S. Kowalewski and A. Philippou, eds. 2009, vol. 5505 of *Lecture Notes in Computer Science*, pp. 215–229, Springer.

[59] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, 1991, pp. 21–65.

[60] MUG, "Explicit State Model Checking Benchmarks,", Available at http://vv.cs.byu.edu/emc_benchmarks/whole_query.html, 2004.

[61] D. R. Musser, "Introspective Sorting and Selection Algorithms," *Softw., Pract. Exper.*, vol. 27, no. 8, 1997, pp. 983–993.

[62] R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers.," *Communincations of the ACM*, vol. 21, no. 12, 1978, pp. 993–999.

[63] R. A. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory," *SC*. 2010, pp. 1–11, IEEE.

[64] R. Pelánek, "Typical Structural Properties of State Spaces," In Graf and Mounier [34], pp. 5–22.

[65] R. Pelánek, "Properties of state spaces and their applications," *STTT*, vol. 10, no. 5, 2008, pp. 443–454.

[66] G. D. Penna, B. Intrigila, I. Melatti, E. Tronci, and M. V. Zilli, "Integrating RAM and Disk based Verification within the Murphi Verifier," *12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2003)*, L'Aquila, Italy, October 2003, vol. 2860 of *LNCS*, pp. 277–282, Springer-Verlag.

[67] G. D. Penna, B. Intrigila, I. Melatti, E. Tronci, and M. V. Zilli, "Exploiting transition locality in automatic verification of finite-state concurrent systems.," *International Journal on Software Tools for Technology Transfer*, vol. 6, no. 4, 2004, pp. 320–341.

[68] G. D. Penna, B. Intrigila, E. Tronci, and M. V. Zilli, "Exploiting Transition Locality in the Disk Based Murϕ Verifier," *Formal Methods in Computer Aided Design*, M. Aagaard and J. O'Leary, eds., Portland, OR, USA, Nov. 2002, vol. 2517 of *LNCS*, pp. 202–219, Springer-Verlag.

[69] A. W. Roscoe, "Model-checking CSP," *A Classical Mind: Essays in Honor of C. A. R. Hoare*, Prentic-Hall, 1994, pp. 353 – 378.

[70] V. Schuppan and A. Biere, "Efficient reduction of finite state model checking to reachability analysis.," *Software Tools for Technology Transfer (STTT)*, vol. 5, no. 2-3, 2004, pp. 185–204.

[71] V. Schuppan and A. Biere, "Liveness Checking as Safety Checking for Infinite State Spaces.," *Electronic Notes in Theoretical Computer Science*, vol. 149, no. 1, 2006, pp. 79–96.

[72] K. Seppi, M. Jones, and P. Lamborn, "Guided Model Checking with a Bayesian Meta-heuristic.," *Fundamate Informatica*, vol. 70, no. 1-2, 2006, pp. 111–126.

[73] U. Stern, *Algorithmic Techniques in Verification by Explicit State Enumeration*, doctoral dissertation, Stanford University, 1997.

[74] U. Stern and D. L. Dill, "Parallelizing the Murϕ verifier," *Computer-Aided Verification, CAV '97*, O. Grumburg, ed., Haifa, Israel, June 1997, vol. 1254 of *Lecture Notes in Computer Science*, pp. 256–267, Springer-Verlag.

[75] U. Stern and D. L. Dill, "Using Magnetic Disk instead of Main Memory in the Murϕ Verifier," *Computer-Aided Verification, CAV '98*, A. J. Hu and M. Y. Vardi, eds., Vancouver, BC, Canada, June/July 1998, vol. 1427 of *Lecture Notes in Computer Science*, pp. 172–183, Springer-Verlag.

[76] E. Tronci, G. D. Penna, B. Intrigila, and M. V. Zilli, "Exploiting Transition Locality in Automatic Verification," *11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME) LNCS*, vol. 2144, September 2001, pp. 259–274.

[77] R. van der Meyden and J.-G. Smaus, eds., *Model Checking and Artificial Intelligence - 6th International Workshop, MoChArt 2010, Atlanta, GA, USA, July 11, 2010, Revised Selected and Invited Papers*, vol. 6572 of *Lecture Notes in Computer Science*. Springer, 2011.

[78] J. Wiese, "Most Recently Used (MRU) hashtable (LRU expiration),", Code Project, June 2004.

[79] A. Wijs and M. T. Dashti, "Extended beam search for non-exhaustive state space analysis," *J. Log. Algebr. Program.*, vol. 81, no. 1, 2012, pp. 46–69.

[80] Y. Zhang and E. A. Hansen, "Parallel Breadth-First Heuristic Search on a Shared-Memory Architecture," *AAAI-06, Workship on Heuristic Search, Memory-Based Heuristics and Their Applications*, Boston, MA, July 2006.

[81] R. Zhou and E. Hansen, "Structured duplicate detection in external-memory graph search," *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04)*, D. L. McGuinness and G. Ferguson, eds., San Jose, CA, July 2004, pp. 683–688, AAAI Press / MIT Press.

[82] R. Zhou and E. Hansen, "Breadth-first heuristic search," *Artificial Intelligence*, vol. 170, 2006, pp. 385–408.

[83] R. Zhou and E. A. Hansen, "Breadth-First Heuristic Search," *ICAPS*, S. Zilberstein, J. Koehler, and S. Koenig, eds. 2004, pp. 92–100, AAAI.

[84] R. Zhou and E. A. Hansen, "Dynamic State-Space Partitioning in External-Memory Graph Search," *ICAPS*, F. Bacchus, C. Domshlak, S. Edelkamp, and M. Helmert, eds. 2011, AAAI.