Dynamic Dead Variable Analysis

Micah S. Lewis
Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Computer Sciences Commons

BYU ScholarsArchive Citation
https://scholarsarchive.byu.edu/etd/639

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.
DYNAMIC DEAD VARIABLE ANALYSIS

by

Micah Lewis

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science
Brigham Young University
December 2005
BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date

Michael D. Jones, Chair

Date

Eric G. Mercer

Date

Sean Warnick
As chair of the candidate’s graduate committee, I have read the thesis of Micah Lewis in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date
Michael D. Jones
Chair, Graduate Committee

Accepted for the Department

Date
Bryan S. Morse
Assistant Department Chair

Accepted for the College

Date
G. Rex Bryce,
Associate Dean, College of Physical and Mathematical Sciences
ABSTRACT

DYNAMIC DEAD VARIABLE ANALYSIS

Micah Lewis
Department of Computer Science
Master of Science

Dynamic dead variable analysis (DDVA) extends traditional static dead variable analysis (SDVA) in the context of model checking through the use of run-time information. The analysis is run multiple times during the course of model checking to create a more precise set of dead variables. The DDVA is evaluated based on the amount of memory used to complete model checking relative to SDVA while considering the extra overhead required to implement DDVA. On several models with a complex control flow graph, DDVA reduces the amount of memory needed by 38-88MB compared to SDVA with a cost of 36 bytes of memory overhead. On several models with loops, DDVA achieved no additional reduction compared to SDVA while requiring no more memory than SDVA.
ACKNOWLEDGMENTS

I want to thank my wife, Julie, for being so supportive and encouraging while I spent so much time in that windowless lab. I want to also thank the Lewis and Olsen families, namely Max, Kary, David, Gloria, Patrick, Jenna, Whitney, Staci, Ethan, Natalie, Quinn and Jessica for always keeping me in their prayers. And of course, I couldn’t have done this without the help and guidance of Dr. Jones. Lastly, lest I forget, I want to acknowledge my fellow lab mates who incessantly distracted me from getting too much done during any single day. I believe it was their way of trying to keep me around longer.
# Contents

1 Introduction .................................................. 1
   1.1 Model Checking ............................................. 1
   1.2 Static Analysis ............................................ 3
   1.3 Data Abstraction .......................................... 5
   1.4 Dead Variable Analysis ................................... 7
   1.5 A Dynamic Dead Variable Analysis ...................... 9
   1.6 Tools ...................................................... 11
   1.7 Thesis Statement .......................................... 12

2 Related Work ................................................ 13
   2.1 Formal Verification ....................................... 13
   2.2 Model Checking .......................................... 14
   2.3 Explicit State Model Checking ......................... 15
   2.4 Predicate Abstraction .................................... 16
   2.5 Live Variable Analysis .................................. 17

3 Methods ....................................................... 19
   3.1 Dynamic Dead Variable Analysis ....................... 20
      3.1.1 Determining Use and Def ............................ 22
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.2 Handling Interrupts</td>
<td>22</td>
</tr>
<tr>
<td>3.1.3 When to run the analysis</td>
<td>25</td>
</tr>
<tr>
<td>3.1.4 Making Decision Paths</td>
<td>26</td>
</tr>
<tr>
<td>3.1.5 Using Decision Paths</td>
<td>29</td>
</tr>
<tr>
<td>3.1.6 Overlays</td>
<td>31</td>
</tr>
<tr>
<td>3.2 Implementation Details</td>
<td>33</td>
</tr>
<tr>
<td>3.2.1 Overlay Implementation</td>
<td>33</td>
</tr>
<tr>
<td>3.2.2 States in Estes</td>
<td>35</td>
</tr>
<tr>
<td>3.2.3 User Input</td>
<td>36</td>
</tr>
<tr>
<td>4 Results</td>
<td>37</td>
</tr>
<tr>
<td>4.1 Methods</td>
<td>37</td>
</tr>
<tr>
<td>4.2 Experimental Results</td>
<td>39</td>
</tr>
<tr>
<td>4.2.1 Dining Philosophers</td>
<td>40</td>
</tr>
<tr>
<td>4.2.2 Timed SSE</td>
<td>43</td>
</tr>
<tr>
<td>4.2.3 Multi-Branch</td>
<td>45</td>
</tr>
<tr>
<td>4.2.4 Multi-Branch II</td>
<td>48</td>
</tr>
<tr>
<td>5 Conclusions and Future Work</td>
<td>53</td>
</tr>
<tr>
<td>5.1 Future Work</td>
<td>54</td>
</tr>
<tr>
<td>A Use/Def Instructions</td>
<td>57</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Dining_Phil3 Results ........................................ 41
4.2 Timed SSE Results ............................................ 44
4.3 Multi-Branch Results ........................................... 48
4.4 Multi-Branch II Results ....................................... 50
A.1 The Use/Def interpretation for M68HC11 instructions. ....... 58
List of Figures

1.1 A simple control flow graph ................................. 5

3.1 Dynamic Dead Variable Analysis Algorithm ................ 23
3.2 Control flow graph without interrupt handlers .............. 25
3.3 Control flow graph with interrupt handlers .................. 25
3.4 A more accurate control flow graph ......................... 26
3.5 Using decision paths in DDVA ............................... 28
3.6 Dynamically pruning the CFG ................................. 30
3.7 Distributing an overlay across the CFG ....................... 34

4.1 Dining Phil3 DDVA results ................................. 42
4.2 Timed SSE DDVA results ..................................... 46
4.3 Multi-Branch CFG ........................................... 47
4.4 Multi-Branch DDVA results ................................... 49
4.5 Multi-Branch II DDVA results ............................... 51
Chapter 1

Introduction

The use of embedded devices in everyday technology affects significant portions of society. Such widespread use introduces chances for technical failure that could result in human and economic costs. One of the main culprits is the software used to control such technology. Such software is becoming increasingly more complex and difficult to verify. This chapter describes model checking, a technology used for verifying software. Static analysis is also introduced as a way to compute a data abstraction in model checking.

1.1 Model Checking

Model checking is a technique used to algorithmically prove that a transition graph is a model of a specification. The specification is a set of properties that define correct behavior. The transition graph is a representation of the behavior of a system. If the transition graph violates a given property in the specification, it is considered an error. In model checking, the transition graph is composed of the set of all possible states of a system and the relation between them. If the specification indicated that starting from a state $\alpha$, it is not possible to reach a state $\beta$, then any path in the transition graph starting at $\alpha$ and arriving at $\beta$ would be considered an error and the
path constitutes an error trace. Model checking, as opposed to testing and simulation, explores every state along every path. This makes it unique in its ability to discover certain kinds of errors. It is well-suited to find errors in which $\alpha$ and $\beta$ are distant from each other or in which $\beta$ is reachable from $\alpha$ only along a complex sequence of states.

Model checking has been shown to be useful in early stages of product development in order to prevent errors from occurring later where they become significantly more costly to fix. One of the best examples was in 1994 when Intel, after shipping their latest processor, discovered that it had an error in floating-point division. Intel spent nearly half a billion dollars to fix the error. Later, an internal study by Intel concluded that they could have found the error for a fraction of the cost using existing model checking techniques.

Formal verification of hardware components using model checking is done with symbolic techniques. Symbolic model checking uses Binary Decision Diagrams (BDDs) to represent the transition graph of the system and the properties to be checked. This allows the model checking algorithm to be defined in terms of BDD operations.

In model checking software and communication protocols, explicit state techniques are used in which all possible states of a program are generated and explicitly stored. Those states make up the nodes of the transition graph and edges between nodes indicate the possible transitions from one state to the next. For example, if the software in question was a threaded application with a critical section, a state of the system would include all the variables and values of each thread.

In the case of both explicit model checking and symbolic model checking, the set of all possible states makes up the state space. The size of the associated state space grows exponentially large in the size of the model description [1]. The model checking problem is known to be NP-complete and PSPACE-hard for both symbolic
1.2. STATIC ANALYSIS

and explicit techniques. Of the two types, this thesis deals with explicit state model checking.

The main challenge in model checking is dealing with the state space explosion. In explicit model checking, each state must be saved in a hash table after being generated in order to mark visited states and guarantee complete coverage of the state space. Due to the complexity of the model checking problem, generating every state and storing it consumes large amounts of resources in terms of time and space. Conversely, model checking is very precise, algorithmic, and completely covers the state space. It is also able to return an error trace indicating the sequence of events which must occur to reproduce the error.

1.2 Static Analysis

In this thesis, static analysis is used to mitigate the state space explosion problem. Static analysis is another methodology to verify properties of programs. The term “static analysis” encompasses a very broad collection of various techniques that verify properties of programs without executing the actual program. In the software development process, the term static analysis is used synonymously with static testing which covers the different forms of testing software before it has been compiled and linked. The term dynamic testing or runtime testing refers to the testing that is done on the object code and executables. The distinguishing difference between static analysis and dynamic analysis is when the analysis occurs, prior to or during program execution.

In the absence of concrete data values, static analysis must reason over all paths of a program which prevents it from pruning infeasible paths. For a given point of execution, it discovers a set of assertions about the state of the program that hold whenever execution passes through that point [2]. Static analysis is widely used in optimizing compilers [3] and code verifiers. As quality assurance and testing have
become larger factors in the software development life cycle, more attention has been
given to developing source code analysis tools. Some of the commercially available
tools include: Coverity, KlocWork, PREfix and PREfast. Some of the academic tools
available include: ESC, LC-Lint, Astree, and ckit.

Static analysis and model checking represent two extremes in software analysis.
Static analysis verifies properties of code by analyzing program flow in the absence of
data. Model checking verifies properties on all reachable concrete data values. Model
checking and static analysis become more similar as data and control abstractions are
applied in model checking and control flow graphs (CFG) are annotated with data
values in static analysis [4].

Figure 1.1 shows the CFG of a simple function. The nodes, called basic blocks,
contain a maximal sequence of instructions with a single point of entry
\(^1\). In many
analyses, the basic blocks and the graph itself are annotated during the computation
with additional information. This information is used to determine which paths of a
program are taken under various circumstances.

The CFG of a program is the simplest representation containing all possible trans-
sitions of the program; thus, making it the program’s abstract computation tree.
Static analysis uses the CFG as the main computational structure. By iteratively an-
notating the CFG with concrete data values, a static analysis generates a state space
that approaches the state space consisting of only reachable states that are generated
during model checking. Similarly, applying abstractions in model checking creates a
representation of the program that approaches the program’s abstract computation
tree or control flow graph. In static analysis, the property to verify is expressed as a
set of flow equations. In model checking, it is written in a modal temporal logic [4].

\(^1\)We ignore the effects of delayed branches as implemented in some pipelined processors; therefore,
this definition is sufficient. It would need to be modified slightly if this was not the case.
While static analysis is not as precise as model checking, it does not suffer from the state explosion problem; therefore, it is able to operate on significantly larger problem sizes and runs considerably faster. Additionally, it is better suited for finding errors that are more likely to occur in practice [5].

1.3 Data Abstraction

Abstraction is the most important technique for minimizing the effect of the state explosion problem in model checking [6]. In model checking, abstraction is the process of ignoring irrelevant or unimportant detail in an attempt to verify otherwise intractable systems. There are two kinds of abstraction: data abstraction and control abstraction. Data abstraction reduces possible data values and control abstraction reduces control structures. This work will focus on data abstraction. The most common kind of data abstraction is predicate abstraction. While predicate abstraction is not used in this thesis, a brief review of predicate abstraction will help set the context for data abstraction through dynamic dead variable analysis.

An abstraction, of either data or control, can be characterized by whether it is precise or imprecise. An imprecise abstraction is a conservative approximation
of the original system. In a conservative approximation, if a property is true of
the abstraction, it may be true in the original system. Stated more formally, a
conservative approximation forms a Galois connection 

\[ T(s) \in \gamma(T(\alpha(s))) \]

where \((\alpha, \gamma)\) is a Galois connection and \(T(s)\) is the transition function \(T\) for a given state \(s\). On the
other hand, a precise abstraction is one where 

\[ T(s) = \gamma(T(\alpha(s))) \]

The benefits of a precise abstraction are that every property we can verify for the original system
also holds in the abstraction and vice-versa. Thus we can preserve both liveness and
safety properties whereas other imprecise abstractions can not [7].

In predicate abstraction, the data values are abstracted using a set of predicates.
The concrete data values are then replaced by boolean variables that evaluate to a
predicate over the concrete data values. The same idea can be extended for multiple
variables and relationships between variables. Furthermore, predicates can be con-
structed to represent the effect of concrete transitions on the abstracted data values.
Most predicate abstraction tools are given the abstraction function and the verifi-
cation system automatically replaces concrete transitions with abstract transitions.
Using predicate abstraction, the abstracted state-space is an over-approximation of
the original; therefore, the abstracted state-space may contain extra behaviors and
errors that are not present in the original.

With extra behaviors, certain properties that are true in the original may be in-
validated. To overcome this it is common to go through a process of counter-example
guided abstraction refinement in which predicates are added to confine the behaviors
of the system to avoid infeasible property violations by using a more precise repre-
sentation of the data values. Unfortunately, adding the right predicates is difficult
because adding too many predicates results in a computationally intractable abstrac-
tion. Furthermore, predicate abstraction breaks down when confronted with data
operations that require reasoning in undecidable logics to determine the effect of a
concrete data operation. Perhaps the best example of such operations are non-linear statements that fall under the category of Peano arithmetic.

This thesis uses a precise data abstraction based on dead variable analysis. At certain states, the value of a variable may become irrelevant because that variable has no impact. When this is the case, the value could be anything in that state and yet, it wouldn’t change the behavior of the model. Such a variable is called “dead” at that program point.

A technique from static analysis, called dead variable analysis, determines at what points on a CFG a variable can safely be considered dead such that it has no impact on the behavior of the program; therefore, using a dead variable analysis allows model checking to abstract the data values of some variables within a state while preserving all behaviors of the model.

### 1.4 Dead Variable Analysis

Static analysis can be divided into two main categories. The first category is based on control-flow. The second is based on data-flow. Control-flow analysis examines the hierarchical flow of control within each procedure. Data-flow analysis attempts to identify how a segment of a program manipulates its data. Data-flow problems are further categorized based on the direction of information flow: in the direction of program execution (forward problems), opposite the direction of execution (backward problems), or in both directions (bidirectional problems) [3].

A dead variable analysis of a program fragment proceeds by first constructing the CFG. For each basic block $i$, the analysis identifies the set of variables, $def(i)$, that are defined or redefined before the next use. And it identifies $used(i)$ as those that are used before they are defined. Furthermore, it distinguishes between variables that
are dead on the entrance and exit to basic block $i$ as follows:

$$DeadIn(i) = (DeadOut(i) \cup def(i)) \cap \neg used(i)$$

$$DeadOut(i) = \bigcup_{j \in Succ(i)} DeadIn(j)$$

By representing a program’s CFG in terms of an equivalent Kripke structure, the meaning of dead variables in the context of model checking can be clarified by using the temporal logic CTL. A kripke structure is a non-deterministic finite state machine whose states are labeled with boolean variables, which are the evaluations of expressions in that state [8]. The equivalent Kripke structure is formed by defining a state for the entry and exit of each basic block in the CFG. The new states correspond to the $DeadIn$ and $DeadOut$ sets at each location $i$. The labels for each state in the Kripke structure $K$ are made from the set of $def()$ and $used()$ variables of the associated basic block. Using CTL, dead variables are expressed as follows:

$$x \text{ is dead in program state } s \text{ iff } K, s \models A[def(x) \land \neg used(x)]$$

in which a variable $x$ is dead in state $s$ if and only if along all paths from state $s$, $x$ is not used or it is (re)defined before it is used. The quantifier $A$ refers to all computation paths. The $R$ operator requires that the second property must remain true until a state where the first and second are both true. Intuitively, satisfying the first property releases an obligation to satisfy the second.

Dead variable analysis is orthogonal to traditional methods of model checking. It does not prevent the use of other capacity improvement methods such as partial order reductions, hash compaction, parallelism, etc. For this reason, it has been widely used in numerous model checkers and model checking tools such as Spin, XMC, Bandera, and IF [9, 10, 11, 12]. The analysis is separate from the model checking run and performed prior to model checking by examining the source code of a program or model. During a model checking run, when a variable becomes dead it is commonly
eliminated by resetting it to a null value. Setting the variable to null causes it to be ignored when generating successor states. Relative to the time to completely verify a system, the computational time to perform dead variable analysis is negligible.

Static analysis techniques, including dead variable elimination, do not have concrete values. This creates a challenge when dealing with pointers. In static analysis, the problem of determining whether two pointers may alias the same memory location is known to be undecidable while determining if they must alias is not recursively enumerable. Current static analysis techniques that allow pointers must, therefore, use simplifying assumptions. Performing a dead variable analysis at the level of assembly code, or C code, requires that simplifying assumptions be made such as assuming that dynamically allocated recursive data structures do not exist [13]. Without them, the problem of a dead variable analysis is at least as hard as the aliasing problem; thus, making it undecidable. In general, the more precise the analysis, the more computationally expensive it is in terms of time and space [14].

1.5 A Dynamic Dead Variable Analysis

Using information from the model checker, a DDVA determines which paths are feasible in the CFG for a given set of data values using an algorithm similar to [15]. Variables that are normally considered live along a small set of paths can be marked as dead if that set of paths is infeasible. By marking them as dead along the current path in the transition graph of the model, it can reduce the number of states that must be explored. The ideal scenario in which DDVA makes significant reductions in the state space is when the CFG of a program contains multiple paths of execution resulting from nested conditional statements.

Model checking is well suited to deal with pointers and arrays. It generates and uses concrete data values for each variable. In a model checker, deciding whether two pointers reference the same memory location is as easy as computing equiva-
lence between the values generated for each variable. Viewing model checking as a data-flow analysis that uses concrete values [4], the Dynamic Dead Variable Analysis (DDVA) extends a dead variable analysis by performing multiple analyses during a model checking run and using the concrete values generated by the model checker to dynamically identify dead variables and remove them from the state vector at key points of execution. The advantage of this approach over regular dead variable analysis is threefold:

1. concrete values can be used to perform an accurate must alias analysis, and

2. concrete values can be used to prune conditional branches of the CFG, and

3. the analysis can be extended to the runtime stack and heap.

Additionally, a dynamic dead variable analysis, unlike predicate abstraction, is able to function whether or not a program uses non-linear statements.

Compared to performing a dead variable analysis at the level of high-level language code, working at the level of assembly code introduces two differences. First, the analysis must be aware of indirect addressing. Indirect addressing occurs when an address is loaded into register and then used as an offset to reference a nearby location. It is the equivalent of a pointer in C/C++. Secondly, compiler optimizations in some cases confound regular static analysis. This arises when the compiler uses indirect addressing in the assembly code because it detects that variables of the same type will be spatially near to one another. Even though the high-level code doesn’t specify an array, the compiler treats the declared variables as if it were an array in order to save time. This presents difficulty for static analysis because a static analysis lacks the runtime information which would indicate the memory addresses the program uses.
1.6 Tools

The DDVA is an extension of the Estes model checker. Estes is an ideal tool to extend because it provides a readily available framework for model checking assembly code.

Estes is a software model checking tool for embedded devices developed at Brigham Young University [16]. One of the main difficulties in model checking software is deriving a formal statement of what a program means. Estes gives a formal interpretation to a program by using the actual processor on which it will be executed. This interpretation is automatically generated using GDB to simulate the target processor. The current implementation works with the 68hc11 family of processors by Motorola and the Hitachi H8/300 processor.

The interpretation is both precise and accurate, but makes an already difficult problem even more computationally difficult because the formal interpretation given by GDB is the state of the entire system at any given point in time. It includes internal registers and the contents of memory. Therefore, the size of a single state is the sum of the registers and memory locations which increases memory and time resource demands. Regardless of what high level language someone uses in writing software, Estes performs model checking at the level of assembly code. In other words, it uses assembly code as the specification language. Working at that level reveals subtle concurrency errors that would otherwise go undetected. There are currently model checkers that operate on other commonly used languages in practice such as the Java PathFinder which does model checking on Java byte-code.

In Estes, the state space explosion problem is especially pronounced. Since a state includes the processor’s registers and the contents of memory, the space required for model checking in Estes is unusually large due to the size of the state representation. Moreover, to determine equivalence between states requires checking that every bit
in the processor state is the same. This increases computation time by a constant that, in practice, can be significant.

1.7 Thesis Statement

A dynamic dead variable analysis will reduce the size of the state space compared to static dead variable analysis at a negligible additional cost in space and time.
Chapter 2

Related Work

The work in this thesis focuses on a new form of data abstraction for explicit state model checking. This chapter describes where explicit state model checking fits under the umbrella of formal verification. It then briefly describes other forms of data abstraction, namely predicate abstraction and a live variable analysis, and how they relate to the thesis.

2.1 Formal Verification

Formal verification refers to the systematic act of proving or disproving that a system is correct with respect to a formal specification or property using formal methods. Formal methods is frequently defined as mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. Formal verification is an alternative to other forms of verification such as simulation and testing which only explore some of the possible behaviors of a system.

A system, in the context of formal verification, is often given in the form of finite state machines, labelled transition systems, Petri nets, timed automata, and hybrid automata. Systems described using one of the above methods are mathematically precise. With a mathematically precise definition of a system, there are two main
techniques to reason about its behavior: theorem proving and model checking. This work focuses on model checking.

In automated theorem proving (ATP), a formal proof about a system is derived from scratch by providing a description of the system, a set of axioms, and a set of inference rules. An ATP is often not completely automatic and requires user intervention to prove a desired conjecture \[17, 18\]. To be effective, the user needs to have expertise. Automated theorem proving has been found to be useful in software generation, software verification, security protocol verification, and hardware verification.

2.2 Model Checking

Model checking takes a different approach in verifying properties of a system by exhaustively enumerating and searching all the states of a system. Despite being largely automatic, model checking relies on the user to provide a model and a precise specification. The specification is normally expressed in a temporal logic such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL). The two main divisions in model checking are in the type of algorithm used: symbolic model checking, and explicit model checking. This work uses explicit model checking.

Symbolic model checking \[19\] is well-suited for hardware verification. Using this approach, a model is most frequently represented with BDDs \[20\] which encode sets of boolean formulas using the characteristic equation. The property to be checked is expressed in CTL and its negation is translated into an equivalent BDD. The algorithm then performs a fixed point computation to find the set of states that matches the property. A violation is found if intersecting the negated property with the resulting set of states returns a non-empty set.

Symbolic model checking has verified systems of several hundred million states in a matter of seconds, proving to be quite fast in practice. However, this is not always the
2.3. EXPLICIT STATE MODEL CHECKING

More recent research has studied the viability of replacing BDDs in symbolic model checking with efficient SAT solvers [22, 23, 24].

2.3 Explicit State Model Checking

Explicit state model checking, as the name implies, generates explicit representations of a system’s state. It does not use symbolic methods in state representation of computation. Each state is explicit in that it contains a value for each variable of the system. To avoid generating duplicate states, each newly created state must be checked for equality against the set of previous states. This is often done using a hashtable. Due to the state explosion problem, the number of states that must be generated consume large amounts of space and time.

Active research in the area of explicit model checking focuses on reducing the space or time complexity. The most relevant to this thesis involve methods of abstraction. Abstraction takes one of two forms: control abstraction or data abstraction.

Control abstraction is an abstraction of actions within the system. For example, writing a subroutine in a program is a form of control abstraction. From that point on, the programmer will use a single statement to refer to the sequence of instructions associated with the subroutine. In the context of software model checking, control abstraction is the process of removing portions of a program that do not alter the behavior of the system in such a way that the presence of the property being checked can still be decided. Control abstraction has been implemented in Bogor, Spin and the JavaPathfinder [25, 26, 27].

Program slicing is a prominent form of control abstraction. It is a static analysis technique in which portions of code are identified as relevant in preserving a specified behavior of a program. The reduced program, obtained by removing irrelevant sections of code, is called a slice and is guaranteed to preserve the specified behav-
The Bandera toolset performs model checking of concurrent Java programs. It creates a slice using the temporal logic specification of the property to verify. The resulting abstraction is then used to create a model and perform model checking. This approach has the advantage of removing unnecessary portions of code before the model checking which reduces the possible state space. The work in this thesis also uses static analysis techniques to perform abstraction, but it is best classified as a data abstraction.

The state space of many interesting programs is large or possibly infinite owing to the domain of state variables. Data abstraction techniques attempt to map possible data values of each variable into a smaller domain of abstract values thereby reducing the size of the state space and making verification easier. The function for mapping possible data values to abstract values is called the abstraction function. Data abstraction techniques are widely used in model checking [7, 29, 30]. The two most closely related to this work are predicate abstraction and live variable analysis.

### 2.4 Predicate Abstraction

Predicate abstraction is a well known technique for doing data abstraction that has been used in model checkers such as BLAST [31] and SLAM [32, 33]. In [31], Visser et al. were the first to use predicate abstraction [7] on a real software system at the level of C++ source code. In the process, they found that it required a high level of expertise with the code they were verifying. The difficulty was largely due to the fact that most predicate abstraction tools relate the predicates to static variables whereas they needed to retain dynamic information relating instantiated objects of C++ classes. The algorithm they propose for calculating the abstract transitions used normal techniques to generate an initial set of abstract transitions and then augment them dynamically with run-time information.

DDVA uses a similar concept as Visser et al. in that it uses normal static analysis
techniques for an initial analysis and augments the initial analysis with dynamic information available at run-time. The main difference between DDVA and predicate abstraction is that:

1. DDVA is a precise abstraction, and
2. uses a modified dead variable analysis algorithm as the abstraction function.

2.5 **Live Variable Analysis**

A live variable analysis originated with optimizing compilers to determine the points during program execution at which a variable is considered live. A live variable is one in which later computation may depend on the current value of the variable. The dual to a live variable is a dead variable. A dead variable is one in which later computation does not depend on the current value of the variable. Calculating live variables is a classical backward data-flow analysis problem. The algorithm for calculating live variables is a worklist algorithm widely used and published in compiler literature [35, 3].

Live variable analysis has since been employed in various model checkers such as Spin, XMC, Bandera, and IF [9, 10, 11, 12]. The use of a live variable analysis in each is basically the same. The sets of live variables are calculated prior to commencing the model checking run. During model checking a variable is “reset” to a null value when it can safely be considered dead. Setting the variable to an accepted null value causes it to be ignored during successor state generation which reduces the number of successors.

DDVA extends this idea by doing multiple analyses during a model checking run and using the additional information available at run-time to construct a more precise set of dead variables.
Chapter 3

Methods

DDVA is an extension of the static dead variable analysis. A static dead variable analysis involves building a control flow graph, determining use and def sets for each basic block, determining dead variables at each basic block, and appropriately annotating the control flow graph with sets of dead variables. A set of dead variables corresponds to a specific basic block and all the instructions within that block. Annotating the basic block with the associated dead variables provides a simple way to retrieve the appropriate set given an instruction’s address. After a state is generated, the PC value in the state is used as the index into the CFG to retrieve the associated dead variable set. Using the retrieved set, parts of the state vector are marked such that they are ignored when successor states are generated.

The main limitations with using a static dead variable analysis in model checking is that the analysis is, by definition, conservative. A dead variable analysis is conservative because it must assume that every block in the control flow graph is reachable. Furthermore, the analysis lacks run-time information and is unable to resolve pointers; therefore, it must assume a pointer can reference any memory location. If the analysis must assume that any memory location may be referenced, then the problem
of determining which variables are dead is, in general, undecidable.

The DDVA does not need to assume that every block in the CFG is reachable, nor that a pointer may reference any memory location. It gathers run-time information during model checking and utilizes the information to dynamically prune the CFG and resolve pointer references. The next section presents the analysis in more detail.

3.1 Dynamic Dead Variable Analysis

The pseudo-code for the core parts of the algorithm is given in Figure 3.1. The algorithm is called during model checking with three parameters: the control-flow graph, a maximum explore depth (exploreDepth), and a maximum number of reachable states to use in the analysis (numToSave). The exploreDepth indicates the maximum number of steps the analysis can take through the state space to collect reachable states for the analysis. Of those generated states, only some are necessary for the analysis. The numToSave indicates the maximum number of such states to use in each analysis. When that amount has been encountered, the analysis can stop generating reachable states before reaching the exploreDepth or a point of non-determinism.

The CFG serves as the basic computational structure on which information (e.g., dead variable sets) is stored. The DDVA in this thesis depends on the use of DFS as the search technique in model checking. The reason being that DDVA exploits the manner in which a DFS explores the state space in order to efficiently store and manage information that the analysis uses to create a data abstraction. DDVA can be used with any search technique, such as BFS; however, it becomes increasingly more difficult to manage the information used to create the abstraction.

The analysis collects run-time information by interrupting the normal exploration process to generate reachable states along the current path of execution. During that exploration process, select states are saved which can be used to prune parts of the
3.1. DYNAMIC DEAD VARIABLE ANALYSIS

CFG and determine memory locations referenced by pointers. A state is saved if the transition to the next state is a conditional branch instruction or the instruction uses indirect memory addressing. The run-time information available at these states is used to determine a pointer’s value and which CFG branches are to be followed as the program executes.

The current active state is temporarily saved (line 2) as a reference point of where to continue model checking later. This state is called the reference state. The algorithm drives the state generator along the current DFS path while saving a small subset of reachable states that are used only during the course of the analysis (lines 3-9). This subset of states is not saved in the hashtable, and consequently, must still be explored by the model checker. The process of collecting reachable states is stopped when one of three conditions occur:

1. The number of steps taken to collect reachable states equals the exploreDepth.

2. The number of reachable states temporarily stored by the analysis equals the numToSave.

3. The state generator reaches a point of non-determinism indicated by a state with multiple children.

The context of the previously saved currentState is finally restored before proceeding with the previously interrupted model checking (line 11).

The call to create_use_def_sets (line 12) takes the CFG and the set of reachable states to compute the use and def set for each basic block in the CFG. The reachable states where the instruction uses indirect memory addressing provide the concrete values of registers which can be substituted into the instruction to resolve pointers. In order to reason about dead variables in the presence of interrupts, the algorithm
computes the set of variables that are considered used in any interrupt handler (line 13).

The method `calc_dead_variables` shows how the purely static dead variable analysis (lines 18-26, 39-40) is augmented with reachable states and decision path information to prune the CFG on-the-fly (lines 27-38). This is done when deciding whether to include a basic block in the analysis through a process of voting. A unanimous decision of `TAKE` must occur for a block to be included (line 31-40). The successors of a given block (line 24) only include blocks that have previously been marked as visited. After dead variables are computed, the results, which are a superset of the original dead variables, are saved in an overlay structure (line 27). The overlay provides the context of dead variables in which successive states are generated.

The following sections explain in greater detail how the algorithm determines the set of use and def variables, deals with interrupts, creates and uses decision paths, decides when to run the analysis, and uses overlays.

### 3.1.1 Determining Use and Def

The sets of used and defined variables in a basic block is perhaps easier to compute at the level of assembly code than in high-level languages. Statements in high-level languages are often compiled into multiple assembly instructions. The ISA of the M68HC11 is defined such that memory can not be read and written within the same instruction. This simplifies the task of determining the set of addresses that are used versus defined. The computation only needs to consider the type of the instruction and the address it uses. Appendix [A] contains a list of the instructions for the M68HC11 processor and whether they use or define an address.

### 3.1.2 Handling Interrupts

Allowing interrupts in a program changes the CFG we construct by introducing execution paths. Figure [3.3] shows the CFG that would result if an interrupt handler
3.1. DYNAMIC DEAD VARIABLE ANALYSIS

```c
void dynamic_dead_variable_analysis (CFG, exploreDepth, numToSave)
    currentState = get_current_state()
    for ( i < exploreDepth && reachableStates.size() < numToSave ) {
        reachable_state = get_next_state()
        if (isNonDeterministic(reachable_state) break;
        if (is_Branch_or_Pointer(reachable_state)) {
            reachableStates.add(reachable_state)
            i++;
        }
    }
    set_current_state(currentState)
    create_use_def_sets(CFG, reachableStates)
    useIntrpt = determine_interrupt_use_set()
    calc_dead_variables(CFG, reachableStates)
}

void calc_dead_variables (CFG, reachableStates) {
    let allVars be the set of all variables in a program
    let partialPath be the path that will be followed given reachableStates
    add CFG.exit toVisit_Q
    while (toVisit_Q != empty) {
        remove i from toVisit_Q
        mark i as visited
        out[i] = Union( in[s] ) for all visited successors s of i
        oldIn = in[i]
        in[i] = allVars - [(use[i] ∪ useIntrpt) ∩ (out[i] ∪ def[i])]
        create_overlay(CFG)
        if (oldin != in[i]) {
            for all predecessors p of i {
                if p is not visited {
                    let vote be TAKE
                    for each decision d in i {
                        if d in partialPath
                            let ds = decision_from(partialPath, d)
                            if d and ds not irrelevant and d != ds
                                let vote be IGNORE
                    }
                        if vote == TAKE {
                            add p toVisit_Q
                            mark p as visited
                        }
                    }
                }
            }
        }
    }
}
```

Figure 3.1: Dynamic Dead Variable Analysis Algorithm
were added to the CFG in Figure 3.2. When an interrupt occurs, the handler is called immediately following the current instruction. We assume interrupts can occur at any time, the CFG must include an edge from every basic block to the handler and from the handler to every basic block. It also forces basic blocks to be exactly one instruction long.

The CFG as shown in Figure 3.3 is an over-approximation of the true control flow because it includes interrupts in its construction. In reality, the additional paths of execution do not exist. For example, if the interrupt occurred after executing the instruction in $B_1$, the interrupt handler at $B_i$ would execute and then return control to the next instruction at $B_2$. But from the CFG it appears that the program could return to any instruction.

A dead variable analysis on a CFG that over-approximates the actual control flow results in dead variable sets that under-approximate the actual dead variables in each basic block. A more accurate CFG is shown in Figure 3.4 in which the interrupt handler is duplicated to show when it might occur in reality and what the true flow of control is. The modified CFG more accurately depicts the flow of control and removes many extraneous paths.

The CFG in Figure 3.4 is not, however, ideal since the size of each basic block is no greater than a single instruction long. This unnecessarily increases the number of basic blocks that must be considered in the analysis. With more basic blocks, the amount of computation per analysis increases. At the same time, the influence of the interrupt handler causes any global variable shared by the interrupt handler to be considered live until the end of the program. DDVA avoids the extra computation while deriving the same information including interrupt handlers in the CFG by considering the set of all used variables from interrupt handlers as used variables in every basic block. By doing so, the dead variable calculation always makes the
3.1. DYNAMIC DEAD VARIABLE ANALYSIS

The goal of the analysis is to use run-time information to perform a more precise dead variable analysis. At one extreme, the analysis could be performed after every instruction execution. At the other extreme, the analysis could be performed only once at the start. Both extremes offer little benefit. Running the analysis after every instruction would be costly in terms of computational overhead while running it once is unlikely to provide useful information.

The analysis takes a more balanced approach by running when model checking hits a conditional branch instruction. The rationale for this design decision is that evaluating a conditional removes at least one branch from the control flow graph and enables the analysis to possibly prune the other branches in the CFG.

For programs with few instructions between conditional branches, this strategy causes the analysis to run more often. This can happen, for example, in short loops. Running more often duplicates some of the same reachable states used in prior analyses. Using some of the same states as a prior analysis reduces the amount of additional
Figure 3.4: A more accurate CFG of a program with interrupts

information gained in the current analysis. For this reason, the analysis counts the number of reachable states it generates and uses. Another analysis is inhibited from occurring until after the model checker has explored the set of reachable states used in the analysis. This helps reduce the number of redundant analyses while still generating the same information.

3.1.4 Making Decision Paths

A subset of reachable states are generated for the analysis by simulating the program $n$ instructions forward and saving the state at each instruction. These reachable states lie on a partial path of the CFG that the program will follow for the next $n$ deterministic steps given the current state. Knowing which path of the CFG the program takes enables the dynamic dead variable analysis to reduce the set of paths that it must consider when computing dead variables. Removing paths reduces the set of live variables so that it includes only those that are live on the remaining, reachable paths. For example, in Figure 3.5 the analysis is called while exploring a state in basic block $B2$ and generates the next $n$ states to stop in basic block $B7$. This indicates that, given the current state, the evaluation of the conditional in basic
block $B_3$ leads to basic block $B_4$ followed by $B_7$. If some set of variables are only used within blocks $B_5$, $B_6$, and $B_8$, then the analysis should include them in the set of dead variables for successors of the current active state. The analysis is slightly more complicated when the number, $n$, of reachable states in the forward analysis does not form a complete path to the exit. For example, in Figure 3.5 the first $n$ reachable states only form a partial path to $B_7$. A simple forward analysis would reveal that there is only one path from $B_7$ to the exit.

The dead variable analysis, however, is a *backwards* data-flow analysis. It does a breadth-first search on the CFG starting at the *exit* and ending at the *entry*. Given the knowledge gained from the reachable states, it should not consider the basic blocks $B_5$, $B_6$, and $B_8$ when computing dead variables. Instead, it should follow the shaded path shown in the figure that starts at the exit and ends at the entry.

Essentially, the dead variable analysis needs the new CFG resulting from pruning the full CFG based on the reachable states. However, we can avoid the cost of creating duplicate partial CFGs by having the analysis dynamically “carve” off pieces of the CFG as it calculates the dead variables. This requires little time and space since the original CFG is re-used. To perform this dynamic carving of the CFG, the analysis needs to have a way of computing the set of all paths that lead back to the last instruction contained in the set of states reachable given the current state. The key to performing this pruning is the use of *decision paths*.

DDVA annotates the CFG with additional information prior to model checking. The additional information allows the analysis to infer which paths lead to the correct basic block. Section 3.1.5 explains how the information is used during dead variable analysis while this section addresses how the information is computed.

A *decision* refers to the outcome of a conditional statement, more specifically, a conditional branch instruction. Each basic block is annotated with a list of decisions,
Figure 3.5: Using a decision path in DDVA
called the decision path, that must be made to reach a given block during model checking. A decision path is composed of pairs \((PC, \phi)\) where \(PC\) is the program counter address of a branch instruction and \(\phi\) is one of three possible values \(T, F\) or \(?\), that indicate the necessary outcome of the conditional in order to reach the block. The question mark, 
\(\?\), indicates that the outcome is irrelevant.

The decision path is calculated by finding a fix-point. Initially, the decision path for each basic block is empty. For the child block along the false path of a branch instruction at \(PC = \beta\) the pair \((\beta, F)\) is added to its decision path. Similarly, the pair \((\beta, T)\) is added to the decision path of the block along the true path. The fix-point is then calculated by considering each block and making its decision path the join of all its parents’ decision paths with its own.

The join operator operates as follows: If two decision paths are such that one contains the pair \((\beta, F)\) and the other contains \((\beta, T)\), then the result will have the pair \((\beta, ?)\) instead. Similarly, a pair \((\beta, F)\) or \((\beta, T)\) joined with \((\beta, ?)\) returns \((\beta, ?)\). In other words, the outcome at \(\beta\) is irrelevant.

### 3.1.5 Using Decision Paths

When generating the reachable states to be used in the analysis, the number of states temporarily stored is bound by the parameters of exploreDepth and numToSave. It is likely that the depth of exploration will not reach the end of a large CFG; therefore, it is not certain which path of a subset of possible paths will be taken to the exit. The analysis overcomes this challenge by using the decision paths computed prior to model checking.

Figure 3.6 shows the relevant code in which the analysis is computing dead variables using a breadth-first search. The choice of adding or not adding a parent block to the toVisit\_Q is determined by using its decision path and the information from the subset of reachable states. The states indicate a partial path that will be taken
from the reference state. The decision path of the parent block is compared with the partial path from the reachable states. If there is a conflict in the decision path of the current block and the partial path, the block is not added to the \textit{toVisit}.Q.

\begin{verbatim}
29     for all predecessors p of i {
30         if p is not visited {
31             let vote be TAKE
32             for each decision d in i {
33                 if d in partialPath
34                     let ds = decision_from(partialPath, d)
35                     if d and ds not irrelevant and d != ds
36                         let vote be IGNORE
37                 }
38             if vote == TAKE {
39                 add p toVisit_Q
40                 mark p as visited
41             }
42         }
43     }
\end{verbatim}

Figure 3.6: Dynamically pruning the CFG

A conflict occurs when the partial path indicates a decision of \((\beta, \tau)\) where \(\tau\) is either \(T\) or \(F\) and the decision path of the block being considered contains a decision of \((\beta, \neg \tau)\). When this occurs, the block is guaranteed to not lie on any possible path of execution between the exit and the current active state examined by the model checker. Excluding that parent block “prunes” potential sub-paths in the CFG and reduces the set of blocks to be explored by the analysis. The removal of sub-paths in the CFG increases the precision of the dead variables analysis.

The set of dead variables computed by any given run of the DDVA during model checking is context sensitive. The variables can only be considered dead in a subset of the states in the state space. That subset is precisely the set of reachable states from the active state being explored in the model checker at the time the analysis was called; therefore, the results of an analysis must be stored and associated with the correct subset of reachable states. This is accomplished with the use of an \textit{overlay}. 
3.1. DYNAMIC DEAD VARIABLE ANALYSIS

3.1.6 Overlays

An overlay is a simple structure that contains a set of dead variables. Two or more overlays can be joined together in the form of a stack. When one overlay is stacked on top of another, the one on top only retains the set difference between its set of dead variables and the set contained in the one below it. If the top contains the same set as the bottom, the top is ignored and discarded. This structure is used for three reasons. The first is that it saves memory when dealing with subsets of sets. The second is that a prior set of dead variables can be reconstructed by removing overlays at the top of the stack. The third reason is more complex and requires an understanding of the relation between sets of dead variables produced by the DDVA.

Each call to DDVA produces a set of dead variables that is at least as large as that produced by an analysis prior to model checking. Each resulting dead variable set is applicable\(^1\) to the set of all reachable states from the reference state\(^2\). However, a dead variable analysis may not contain the dead variables from a prior analysis. This is true even if the reachable states used in the analysis are on the same path in the transition graph. Stated more formally, given two sets of dead variables derived from separate runs of DDVA, \(\Gamma\) and \(\Sigma\), where \(\gamma\) is the reference state for \(\Gamma\) and \(\sigma\) is the reference state for \(\Sigma\), if \(\gamma\) is reachable from \(\sigma\) then the following is true:

1. \(\Gamma \cap \Sigma \neq \emptyset\),

2. \(\Gamma\) may not be equivalent to \(\Sigma\),

\(^1\)This means that the set can be used to ignore some parts of a state vector when generating states

\(^2\)Recall that the reference state is the current active state of the model checker at the time that DDVA runs. The set of reachable states computed by the analysis is with reference to that current active state. Using a different reference state changes the set of dead variables that the analysis computes.
3. and at a minimum $\Gamma \cap \Sigma = \Omega$ where $\Omega$ is the set of dead variables calculated by SDVA.

Each analysis uses only a subset of reachable states from the reference state, and each call to DDVA uses a different reference state; therefore, we can expect each analysis to use a different set of reachable states which leads to a partially unique set of dead variables.

To compute the largest, and hence most accurate, total set of dead variables for a single reference state would require DDVA to use every state in the transition graph from which the reference state is reachable together with every state reachable from the reference state. Unsurprisingly, this would entail a considerable amount of computation. Instead, the analysis approximates the total set of dead variables by exploiting the manner in which the model checker searches the state space, DFS, together with overlays. Despite that fact that the resulting set of dead variables from each analysis is just a subset of the true set, the true set can be approximated using a dynamic programming approach. The subsets of dead variables are unioned such that the resulting set is an approximation of the true set.

A depth-first search guarantees that every state placed on the model checker’s search stack is reachable from every state before it in the stack; therefore, each state is related to the ones before it and can be considered the “history” of that state’s evolution. Each dynamic dead variable analysis that is done using a “historic” state as the reference state produces a subset of the true set of dead variables for every state that succeeds it; therefore, the union of the results for each analysis done using a prior state in the search queue as the reference state creates an approximation of the true set. This is easily accomplished using overlays.

The result of a DDVA for a given reference state is placed into an overlay. The set of dead variables in the overlay is applicable for all the states that are generated while
the reference state remains on the model checker’s search stack. Its removal from the stack indicates that every reachable state from it has been explored; therefore, the corresponding overlay can be safely discarded. Each successive analysis stores the results in an overlay and stacks it on top of the other overlays. The set of dead variables represented by the stack of overlays represents an approximation of the true set of dead variables generated by successive dead variable analyses.

3.2 Implementation Details

3.2.1 Overlay Implementation

Section 3.1.6 presented overlays as a data structure for storing sets of dead variables associated with appropriate subsets of the reachable state space. In reality, the implementation of the overlay stack is not monolithic, but distributed across each block of the CFG. Figure 3.7 shows a portion of a CFG. The stack beside each block in the CFG is a stack of overlay fragments. The term overlay is used to refer to the sets of dead variables for all basic blocks in the entire graph. A fragment refers to the set of dead variables for a single basic block and behaves in the same way as an overlay described in the previous section. The black square at the bottom of each is the fragment of the results from the initial analysis performed prior to model checking. Since a progressively higher level on a fragment stack only contains differences from the one below it, the set of dead variables at a given block in the CFG is found by computing the union of the overlay stack.

Each DDVA produces a potentially new set of dead variables for each block of the CFG. Figure 3.7 shows the state of the CFG after two DDVAs have occurred. The first DDVA uses a reference state with a PC value in B2, the second with B3. The large squares in the background represent the subset of reachable states used in each analysis. The lighter shades were used in the first analysis and the darker ones in the second. Each fragment in the overlay stack is colored to match the color of the basic
block at which DDVA was called.

With the “stack” distributed across each block of the CFG, some stacks grow/shrink at different rates due to similarities and differences in the fragments. In block $B_2$, the second analysis provided no new information and the stack remained unchanged. Since the model checker is using DFS, the reference state from the second analysis is removed from the search stack before the other. When the reference state is removed from the search stack, the top of the overlay “stack” should be discarded. But since the implementation is distributed across the CFG, it is not as simple as removing the top fragment from each block’s stack. Instead, each fragment in a stack is given a label that associates that fragment with the reference state from which the analysis was called. The label is made by combining the PC value of the reference state with a counter value. Multiple reference states with the same PC value may be generated due to loops. So the counter is used to create a unique label. The counter value is
3.2. IMPLEMENTATION DETAILS

stored in each basic block. It is incremented every time that a DDVA uses a reference state whose PC value is within the block. The counter is decremented every time an associated reference state is removed from the model checker’s search stack.

To save time in repeatedly computing the union of a stack, each block contains what is considered the “working set” of dead variables. Each new fragment that is added to the stack is also unioned with the working set. When the top-most fragment is removed, the set difference between the current working set and the set contained in the fragment is saved as the new working set. In this manner, the set represented by the stack of overlay fragments is changed incrementally as necessary.

3.2.2 States in Estes

When Estes saves the state of GDB in a state structure, it does so as a sequence of bytes contained within a state wrapper. The bytes represent values in different memory addresses. Instead of saving the entire 16-bit address space, Estes only saves address ranges specified by the user in the environment. In order to interpret the contents of a state, a map structure is used which associates a location in the byte sequence with a memory address. In order to mark variables as dead, the state wrapper also contains a bit vector that corresponds to the byte sequence called the ignorevec. If the ith bit is a 1, then the corresponding ith byte in the byte sequence is considered dead.

Before a state is actually saved in the hashtable, it is passed through a linearize() function. Linearizing the state returns a new sequence of bytes composed of the sequence of bytes from the state after removing all the bytes that are marked as dead in the ignorevec. The reduced byte sequence is then saved in the hashtable. For this reason, a state in the hashtable is only considered equivalent to the state being inserted when they both have the same set of dead variables and the live variables in each have the same values. The approach has a significant drawback.
Consider two states $\sigma$ and $\gamma$ where the following is true:

1. The set of dead variables in $\sigma$ is a subset of the dead variables in state $\gamma$.

2. If a variable is live in $\sigma$ and in $\gamma$ then it has the same value.

3. The state $\gamma$ is already in the hashtable.

Inserting $\sigma$ into the hashtable should fail. For all intents and purposes, the states are equivalent. The additional live variables in one state are considered dead in the other and should be ignored when checking equality. For Estes, however, this is not the case. Estes has strict equality and considers “equivalent” states as different and allows insertion into the hashtable. With DDVA, this can lead to exploring more states than necessary.

### 3.2.3 User Input

In order for DDVA to correctly work with a model, it requires some input from the user beyond creating the environment for Estes. When a model is given to Estes it is in the form of a binary executable. The analysis currently does not implement functionality to extract the assembly code that it needs to create the CFG. Instead, the user must use `m6811-elf-objdump` from the binary utilities package that is targeted for the M68HC11. The resulting object dump is then given to the analysis which parses it and uses it to construct the CFG. Additionally, the analysis requires the user to specify PC values for the start of the `main` function, the start of each interrupt handler, and the end of each interrupt handler.
Chapter 4

Results

The dynamic dead variable analysis has been implemented as an extension to the Estes model checker. This chapter presents the experimental methods and results. The results support the thesis that a dynamic dead variable analysis reduces the size of the state space with a negligible additional cost in space and time.

4.1 Methods

The dynamic dead variable analysis and the static dead variable analysis are evaluated in terms of the number of states generated, time used, and memory consumed.

The dynamic dead variable analysis and the static dead variable analysis are a form of abstraction used to reduce the size of the state space. The number of total states generated provides a simple measure of how well the abstraction works. A reduction in the number of generated states indicates a better abstraction because less memory is required to generate all the reachable states.

The measurements of CPU time include the abstraction time, state generation time, and total time. The abstraction time is the time required by the analysis to compute the set of dead variables for the CFG. The state generation time is the amount of time spent by the state generator to create states. The total time is the
total amount of time required by the model checker to explore the state space till an error is found or until states to explore are exhausted.

Both analyses use the same CFG in their computation; therefore, it is not considered to be extra overhead in the comparison. Beyond the CFG, the static dead variable analysis does not have any additional memory requirements. The dynamic dead variable analysis, however, requires extra memory to store overlays. The comparison between the two examines the difference in total memory savings given the extra memory required for overlays. The total memory savings refers to the reduction in the hash table size achieved by using an analysis versus no analysis. The approximate overlay size is an approximation of the amount of memory required to save the dead variables for DDVA during model checking. It is found by recording the maximum size of the overlay stack during the model checking run.

The analysis was run multiple times to test the performance of DDVA with different choices for the \textit{numToSave} and the \textit{exploreDepth}. In each run, the \textit{numToSave} was set to half of the \textit{exploreDepth} and the \textit{exploreDepth} was run for values of 5, 10, 15, 20, 25, 30, 50, 70, 100, 150, 200, 300, 400 and 500. In each test case presented below, the comparison of DDVA with the static dead variable analysis uses the run of DDVA that took the most time and the one that took the least time. A separate graph is also given which shows the performance of DDVA for all of the different \textit{exploreDepths}. All of the tests were run on a Linux machine with a Pentium 4, 3.2 GHz CPU with 2GB of RAM.

The models used for comparing the two analyses have some combination of the following characteristics because some factors favor DDVA and some favor SDVA.

1. Loops in the CFG,

2. Interrupt Handlers,
3. Indirect memory addressing instructions (Pointers),

4. Complex control-flow,

5. Linear statements in conditional statements, or

6. Non-linear statements in conditional statements.

The inclusion of loops causes some decision paths to be of the form \((\beta,?)\) such that DDVA is unable to rule out paths of the CFG. When fewer paths are excluded, the number of dead variables detected by the analysis decreases. Interrupt handlers force the analysis to consider a subset of variables as always *live*, degrading performance for DDVA. Instructions that use indirect memory addressing are the equivalent of pointers in assembly code and negatively affect performance by forcing SDVA to consider all addresses as live within the basic block in which it occurs. When DDVA does not have enough run-time information to resolve an indirect memory reference, it also must consider all addresses as live. A CFG with complex branching creates multiple paths with common sub-paths which reduces the utility of the static dead variable analysis. Linear arithmetic statements are decidable since they fall under the category of Presburger arithmetic. Their decidability enables abstraction techniques such as predicate abstraction to be used. Non-linear statements, however, include multiplication of two or more variables; therefore, they fall under Peano arithmetic. Predicate abstraction can not handle Peano arithmetic because it can not compute the outcome of a branch based on the truth values of a collection of predicates.

4.2 Experimental Results

In this section we present the results for each example in the test suite using a table and three graphs. The table contains the results of model checking with no analysis, with SDVA, and with DDVA. In order to show the variation in DDVA performance
given different parameters, the fastest and slowest runs are shown in each table. The total time to complete is closely correlated with the memory savings. In each of the graphs, the $x$-axis is the $\text{exploreDepth}$ used. The top graph shows the Total time to finish the model checking, the Abstraction time, and the Exploration time. The Abstraction time is the time spent in the DDVA which includes the Exploration time. The Exploration time is the time spent inside the analysis generating the subset of reachable states with the $\text{exploreDepth}$ as an upper-bound. The middle graph shows the size of the hash table after exhausting the state space or after identifying an error in the model. The bottom graph displays the cumulative total of bytes marked dead in states that are placed in the hash table.

4.2.1 Dining Philosophers

The DiningPhil3 is an implementation of the classic Dining Philosophers problem for the M68HC11 with 3 philosophers. Instead of threads, each philosopher is an interrupt handler which fires at a pre-specified rate. The main function is a while loop that executes until deadlock. The example does not contain any indirect memory addressing and has simple control-flow with only linear statements in its conditionals. In this example, deadlock does occur.

With no indirect memory addressing, SDVA is able to compute a set of dead variables that can be used during model checking. Similarly, DDVA begins by performing the same analysis and later refining sets of dead variables. The DiningPhil3 example represents the normal situation in which SDVA is most applicable.

Table 4.1 shows the performance of model checking with no abstraction, with a static dead variable analysis, and with a dynamic dead variable analysis. Every run terminates after finding the error in the model. The additional time that SDVA uses to perform the analysis is negligible. Although it does not provide a reduction in the total number of states that must be explored, it did reduce the hash table by slightly
4.2. EXPERIMENTAL RESULTS

Table 4.1: Dining_Phil3 Results

<table>
<thead>
<tr>
<th></th>
<th>Total Time (seconds)</th>
<th>States Explored</th>
<th>Hash Size (KB)</th>
<th>Approx. Overlay Stack Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>1.05965</td>
<td>7722</td>
<td>1620.14</td>
<td></td>
</tr>
<tr>
<td>SDVA</td>
<td>1.21445</td>
<td>7722</td>
<td>1412.87</td>
<td></td>
</tr>
<tr>
<td>Fastest DDVA</td>
<td>1.92783</td>
<td>7722</td>
<td>1412.87</td>
<td>0.000</td>
</tr>
<tr>
<td>Slowest DDVA</td>
<td>2.15872</td>
<td>7722</td>
<td>1412.87</td>
<td>0.000</td>
</tr>
</tbody>
</table>

more than 0.2 MB or almost 12%.

The fastest model checking run that used DDVA is only marginally faster than the slowest. Similar to SDVA, it did not reduce the number of explored states and only reduced the hash table by the same amount as SDVA. In this scenario, DDVA does not compute any additional information beyond that done by SDVA and, therefore, does not require any extra space for saving overlays. For this example, the cost of DDVA is just the extra time it spends in its analysis.

Figure 4.1 displays the results gathered from each DDVA analysis. It is interesting to note that as more states are included in the reachable set, the majority of the time in the analysis is due to time spent generating the subset of reachable states for the analysis versus actually computing the dead variables. The current algorithm generates a subset of reachable states and then discards them after the analysis is finished. If instead of discarding them the analysis appropriately marked dead variables, checked each state for errors, and inserted them into the hash table, then the model checker would not have to re-explore the same states used in the analysis. This would effectively negate the cost of gathering reachable states for the analysis.
Figure 4.1: Dining Phil3 DDVA results.
4.2. EXPERIMENTAL RESULTS

4.2.2 Timed SSE

The Timed SSE is a program that checks the readings of two sensors in a loop. If the readings are not equivalent, then an alarm is raised. The sensor readings are updated by an interrupt that occurs at a pre-specified rate. The interrupt handler sets the sensor readings by copying the values read from two input ports and saving them into the appropriate memory location of the sensor variables. In assembly, when the readings are checked for equality, one value is loaded into a register and the other is taken from memory. It is possible for the alarm to be raised even if the interrupt handler always sets the sensor readings to equivalent values. This occurs because an interrupt can occur between the time that one value is loaded into a register in preparation for comparison and the time that the next value is read from memory. This error is found during model checking.

Similar to Dining_Phil3, this model has a main loop and interrupts. They differ in that Timed SSE makes extensive use of indirect memory addressing whereas Dining_Phil3 does not use any. The presence of indirect memory addressing can reduce the precision of a static dead variable analysis.

In Table 4.2, the SDVA does not provide any reduction in states explored, but does reduce the hash table by approximately 1.8% (15.38KB). The presence of pointers in the test model forces SDVA to consider most variables as live during the course of model checking which degrades its performance.

As in the Dining_Phil3 example, every run of DDVA explores the same number of states as SDVA. But for the hash table, DDVA achieved a higher reduction in every run. In the slowest run the reduction was 18.27KB more than SDVA and 16.85KB in the fastest run. Although relatively small, the slowest run achieves a reduction of 3.98% (33.65KB), more than twice as much as SDVA. The fastest run is close behind at a reduction of 3.81% (32.23KB). In every run of DDVA, the overlays do not use
Table 4.2: Timed SSE Results

<table>
<thead>
<tr>
<th></th>
<th>Total Time (seconds)</th>
<th>States Explored</th>
<th>Hash Size (KB)</th>
<th>Approx. Overlay Stack Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>2.17019</td>
<td>6514</td>
<td>844.86</td>
<td></td>
</tr>
<tr>
<td>SDVA</td>
<td>2.22276</td>
<td>6514</td>
<td>829.48</td>
<td></td>
</tr>
<tr>
<td>Fastest DDVA</td>
<td>2.78915</td>
<td>6514</td>
<td>812.63</td>
<td>0.016</td>
</tr>
<tr>
<td>Slowest DDVA</td>
<td>3.70257</td>
<td>6514</td>
<td>811.21</td>
<td>0.016</td>
</tr>
</tbody>
</table>

more than 16 bytes of space.

Figure 4.2 shows the results for each run of DDVA. With a greater value for the `exploreDepth` fewer analyses are called reducing the total time. Similar to Dining_Phil3 the time spent generating the subset of reachable states becomes the dominant factor for the amount of time spent in DDVA. The Timed SSE example exhibits an unexpected behavior. As the `exploreDepth` increases and more reachable states are used in the analysis, fewer variables are marked as dead and the size of the hash table increases. Although not shown, a sufficiently large `exploreDepth` results in a hash size equivalent to that using SDVA.

The exhibited behavior where fewer variables per state are marked dead even though more states are stored in the hash table is due to implementation details. It occurs as a result of how the current implementation functions when faced with loops in the CFG. The analysis creates an index into the set of reachable states based on the PC value for each state. When the reachable states correspond to a loop, multiple states contain the same PC value which means that the index refers to a subset of the reachable states generated by the analysis. By itself, this would not cause the reduction in variables marked dead; however, the analysis is not allowed to run again
4.2. EXPERIMENTAL RESULTS

until the model checker has explored all the states in the set of reachable states used by the analysis. The result is that as the exploreDepth increases, the analysis occurs less frequently and the analysis uses an increasingly smaller subset of the reachable states generated and this reduces its precision.

4.2.3 Multi-Branch

The Multi-Branch example is a single function with no loops or interrupt handlers. It reads the values from 8 different sensors, stores them in an array and then tests for relationships between different pairs of values read (e.g., Is sensor a multiplied by sensor b greater than 0x80?). The sequence of conditional statements that test the relationships creates a complex CFG shown in Figure 4.3. Although no pointers were specifically used in the original code, the optimizing compiler generated assembly code and makes extensive use of indirect memory addressing to access the values in the array.

This example is an ideal case for the DDVA analysis since it exhibits a complex CFG where some variables appear only on certain paths of the CFG. It is not possible to predict which path is taken on the CFG until run-time information is available; therefore, the static dead variable analysis is unable to mark variables as dead except in the blocks found in the bottom two layers of the CFG when no more instructions use indirect memory addressing. In this example, the conditional statements are linear. This implies that if predicate abstraction were used as the abstraction technique, then a first-order logic decision procedure could decide the outcome of branch instructions based on a suitable set of predicates.

Table 4.3 highlights the accomplishments of DDVA in the ideal scenario. In the fastest run, model checking with DDVA took only one third of the time as when no abstraction is used and less than half the time as when SDVA is used. Moreover, the state space was reduced by more than 38MB (approx. 75%) at the cost of 36
Figure 4.2: Timed SSE DDVA results
4.2. EXPERIMENTAL RESULTS

bytes for overlays. In the slowest run, DDVA still completed faster than with no abstraction and achieved a reduction in the hash table slightly better than that of SDVA. Although the reduction in state space was not as much as in other cases, it was still significant at a little more than 12MB (24%). The performance of the slowest run is directly tied to the \texttt{exploreDepth}. The slowest run did not gather enough information from the reachable states to be able to resolve pointers and, therefore, in many cases it was forced to consider all the variables in a basic block as live.

Figure 4.3 shows the results from all DDVA runs. It is interesting to note that while more variables are marked dead, the hash table actually becomes larger because the hash table uses a strict notion of equality as explained in Section 3.2.2. Two states that are identical for all live variables but have different dead variables are always considered different states. However, when a state has more dead variables than a
Table 4.3: Multi-Branch Results

<table>
<thead>
<tr>
<th></th>
<th>Total Time (seconds)</th>
<th>States Explored</th>
<th>Hash Size (MB)</th>
<th>Approx. Overlay Stack Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>99.3279</td>
<td>294515</td>
<td>51.6803</td>
<td></td>
</tr>
<tr>
<td>SDVA</td>
<td>73.631</td>
<td>216485</td>
<td>37.9660</td>
<td></td>
</tr>
<tr>
<td>Fastest DDVA</td>
<td>33.6372</td>
<td>73006</td>
<td>12.9419</td>
<td>0.036</td>
</tr>
<tr>
<td>Slowest DDVA</td>
<td>90.777</td>
<td>216462</td>
<td>37.9622</td>
<td>0.000</td>
</tr>
</tbody>
</table>

state already in the hash table and both states share the same values for the live variables, the new state does not need to be stored in the hash table.

If the hash table employed this weaker notion of equality, then fewer states would be inserted into the hash table and, therefore, require less space. Implementing such a hash table would require a way to quickly find both a state or states that might subsume another when dead variables are considered in the equality check. It is encouraging that marking 0.15MB of dead variables reduces the hash table by almost 39MB.

4.2.4 Multi-Branch II

The Multi-Branch II example is a modification of Multi-Branch. The CFG is almost identical to the CFG for Multi-Branch; therefore, this example is still considered an ideal case for DDVA analysis. This example differs from the Multi-Branch example because the conditional statements have been changed to be non-linear. As described earlier, predicate abstraction techniques do not work with non-linear statements.

For the Multi-Branch II example, the SDVA provides no benefit and costs very little in terms of time. In Table 4.4, the DDVA analysis which took the most time and consumed the most memory was still better than a normal model checking run.
4.2. EXPERIMENTAL RESULTS

Figure 4.4: Multi-Branch DDVA results
Table 4.4: Multi-Branch II Results

<table>
<thead>
<tr>
<th></th>
<th>Total Time (seconds)</th>
<th>States Explored</th>
<th>Hash Size (MB)</th>
<th>Approx. Overlay Stack Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>204.268</td>
<td>577478</td>
<td>118.957</td>
<td></td>
</tr>
<tr>
<td>SDVA</td>
<td>105.761</td>
<td>293585</td>
<td>60.4393</td>
<td></td>
</tr>
<tr>
<td>Fastest DDVA</td>
<td>67.588</td>
<td>150304</td>
<td>30.9283</td>
<td>0.026</td>
</tr>
<tr>
<td>Slowest DDVA</td>
<td>118.474</td>
<td>293469</td>
<td>60.4160</td>
<td>0.000</td>
</tr>
</tbody>
</table>

At a cost of a few bytes, the size of the hash table was reduced by 88MB (74%).

The graph of results for each DDVA run in Figure 4.5 show a similar behavior as in Multi-Branch. An increase in precision marks more variables dead, but degrades the overall savings in the hash table. The sharp dip indicates a point at which the precision of the analysis provides the greatest benefit in terms of time and memory and how the hash table checks equality.
4.2. EXPERIMENTAL RESULTS

Figure 4.5: Multi-Branch II DDVA results
Chapter 5

Conclusions and Future Work

The dynamic dead variable analysis achieves a 0-75% reduction in the size of the hash table compared to SDVA with a maximum memory overhead of 38 bytes for saving results in an overlay structure in our test benches. The dynamic dead variable analysis performs best on a complex control flow graph, such as Multi-Branch or Multi-Branch II where the reduction in the hash table size was almost 75% in our experiments. The analysis achieves little or no reduction on loops.

When the analysis does not produce any reduction in the hash table, it does not generate any overlays that must be saved requiring no additional space in memory.

The time spent in the analysis during the course of a model checking run is insignificant for large models. The reason is that the size of a model may increase by allowing a larger range of possible data values for variables, but the size of the control flow graph remains the same; therefore, the number of states explored by the model checker increases exponentially while the number of nodes in the CFG explored by the analysis remains constant. The complexity of the analysis is independent of the size of the state space and only dependent on the size of the CFG.

An important factor in the amount of reduction possible is the form of equality
used in the hash table. The hash table in Estes limits the amount of reduction when more variables are marked dead because it employs a strict notion of equality. A reimplementaion of the hash table with a weaker notion of equality should result in smaller hash tables when using the DDVA.

5.1 Future Work

The most immediate future work is to change the hash table implementation to allow a form of weak equality checking among states. Estes currently uses an open addressing hash table implementation. By using a variation of a chained hash table, states that share a similar property, such as the PC value, could all hash to the same bucket. This would simplify the problem of determining if a state in the hash table satisfies the weak equality constraint previously discussed. Such an implementation would assuage the effects of marking increasingly larger portions of the state vector as dead. It may also lead to a greater insight as to why the Timed SSE example behaves as it does when the \textit{exploreDepth} is increased.

Second, the largest overhead is time spent generating a subset of reachable states to be used in the analysis. By marking those states, checking them for errors, and inserting them into the hash table, the associated cost of state generation in DDVA effectively dissappears such that the only time used is the actual analysis itself. In addition to such a change, the analysis should be modified to collect reachable states beyond points of non-determinism instead of stopping. The additional reachable states could potentially increase the amount of dead variables found.

Third, the DDVA analysis could be modified to run with other forms of search techniques such as BFS or a guided search. This would require saving information in a structure other than the overlay and most likely increase the cost associated with saving information computed during each analysis.

Fourth, the greatest benefit of DDVA comes when used on sections of code with
5.1. FUTURE WORK

complex control-flow. An interesting avenue of future research is to look into ways

to automatically identify regions of a program similar to the ideal scenario in which

a DDVA could be run for parts of the CFG. Along the same lines, some future work

would include using heuristics to dynamically set the explore depth during model

checking. The goal is to reduce the scope of the CFG in which DDVA occurs while

still providing the major benefit of using DDVA to create a data abstraction.

Fifth, the analysis is generating a subset of reachable states, it stops if it encounters

a state with multiple successors indicating a point of non-determinism. Another

avenue of research is to modify the analysis such that it is not limited to just states

with a single successor.
Appendix A

Use/Def Instructions
### APPENDIX A. USE/DEF INSTRUCTIONS

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Use</th>
<th>Def</th>
<th>Instr.</th>
<th>Use</th>
<th>Def</th>
<th>Instr.</th>
<th>Use</th>
<th>Def</th>
<th>Instr.</th>
<th>Use</th>
<th>Def</th>
</tr>
</thead>
<tbody>
<tr>
<td>adca</td>
<td>X</td>
<td>X</td>
<td>bset</td>
<td>X</td>
<td>X</td>
<td>ldab</td>
<td>X</td>
<td></td>
<td>sbcb</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>adcb</td>
<td>X</td>
<td></td>
<td>cba</td>
<td>X</td>
<td></td>
<td>ldd</td>
<td>X</td>
<td></td>
<td>staa</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>adda</td>
<td>X</td>
<td></td>
<td>clr</td>
<td>X</td>
<td></td>
<td>lds</td>
<td>X</td>
<td></td>
<td>stab</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>addb</td>
<td>X</td>
<td></td>
<td>cmpa</td>
<td>X</td>
<td></td>
<td>ldx</td>
<td>X</td>
<td></td>
<td>std</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>addd</td>
<td>X</td>
<td></td>
<td>cmpb</td>
<td>X</td>
<td></td>
<td>ldy</td>
<td>X</td>
<td></td>
<td>sts</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>anda</td>
<td>X</td>
<td></td>
<td>com</td>
<td>X</td>
<td></td>
<td>lsl</td>
<td>X</td>
<td></td>
<td>stx</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>andb</td>
<td>X</td>
<td></td>
<td>cpd</td>
<td>X</td>
<td></td>
<td>lsr</td>
<td>X</td>
<td></td>
<td>sty</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>asl</td>
<td>X</td>
<td></td>
<td>cpx</td>
<td>X</td>
<td></td>
<td>neg</td>
<td>X</td>
<td></td>
<td>suba</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>asr</td>
<td>X</td>
<td></td>
<td>cpv</td>
<td>X</td>
<td></td>
<td>oraa</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bclr</td>
<td>X</td>
<td></td>
<td>dec</td>
<td>X</td>
<td></td>
<td>orab</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bita</td>
<td>X</td>
<td></td>
<td>eora</td>
<td>X</td>
<td></td>
<td>rol</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bitb</td>
<td>X</td>
<td></td>
<td>eorb</td>
<td>X</td>
<td></td>
<td>ror</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>brclr</td>
<td>X</td>
<td></td>
<td>inc</td>
<td>X</td>
<td></td>
<td>sba</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>brset</td>
<td>X</td>
<td></td>
<td>ldaa</td>
<td>X</td>
<td></td>
<td>sbca</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A.1: The Use/Def interpretation for M68HC11 instructions.
Bibliography


