On-the-fly Dynamic Dead Variable Analysis

Joel P. Self, Eric G. Mercer

Department of Computer Science
Brigham Young University
Provo, Utah, USA

Abstract. State explosion in model checking continues to be the primary obstacle to widespread use of software model checking. The large input ranges of variables used in software is the main cause of state explosion. As software grows in size and complexity, the problem only becomes worse. As such, model checking research into data abstraction as a way of mitigating state explosion has become more and more important. Data abstractions aim to reduce the effect of large input ranges. This work focuses on a static program analysis technique called dead variable analysis. The goal of dead variable analysis is to discover variable assignments that are not used. When applied to model checking, this allows us to ignore the entire input range of dead variables and thus reduce the size of the explored state space.
Prior research into dead variable analysis for model checking does not make full use of dynamic run-time information that is present during model checking. We present an algorithm for intraprocedural dead variable analysis that uses dynamic run-time information to find more dead variables on-the-fly and further reduce the size of the explored state space. We introduce a definition for the maximal state space reduction possible through an on-the-fly dead variable analysis and then show that our algorithm produces a maximal reduction in the absence of nondeterminism.

1 Introduction

Model checking is a way to automatically verify properties of a system [13, 4, 16, 11, 10]. The model of a system is a directed graph containing a set of vertices and a set of edges. In explicit state model checking, vertices represent states of the system and edges represent transitions between states. When used to verify software, model checking can discover subtle errors in deep execution traces that are easily passed over in traditional software testing techniques. Since model checking is a form of formal verification, the output of a model checker is a proof that the system does or does not satisfy the specified property.

When model checking software, a state is a snapshot of the program at a single program location. The state contains the value of the program counter and the values of all of the variables in the program. The program is used to generate successor states given a current state. Every state generated is stored in a *Visited* set, and every newly generated state is checked against the set to determine if the state is new. A breadth-first or depth-first search is used to explore the entire state space and ultimately verify or disprove the specified property. A single state may have multiple successors due to non-determinism in the program. Non-determinism represents input from an outside source such as user input from a keyboard or input from a sensor. The model checker must generate successor states that represent all possible input values in order to explore all possible scenarios when running the program. Since the size of the reachable state space is exponential in the branching factor of the model, the state space becomes large rather quickly, even for programs with relatively few variables. This rapid growth of the state space is called the state explosion problem.

An important technique for mitigating the state explosion problem in verification is data abstraction [5]. Data abstraction reduces the size of the generated state space by abstracting away data values; in other words, it removes variables from the state to make their value unconstrained. Variables that receive values from a non-deterministic input often have such large domains that removing even a single variable can greatly reduce the effect of state explosion.

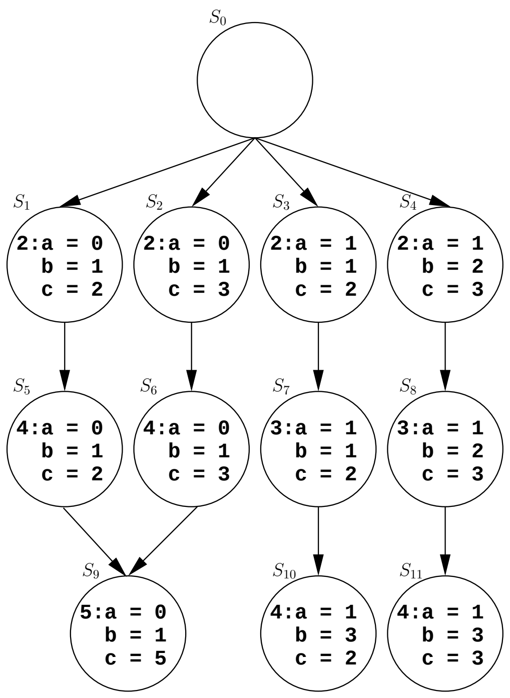
Dead variable analysis is a type of data abstraction that determines when the values of variables do not matter in order to simplify a program state. Variables can be either live or dead with respect to a program location. A variable is live at a location when its current value is used. A variable is dead at a location when it is redefined before it is used in some future location, or it is not used in any future location. When a variable is dead at a program location its value does not affect the behavior of the program since it is not used. *Static dead variable analysis* (SDVA) has been implemented in several model checkers including SPIN, XMC, Bandera, IF, and Bebop [12, 7, 6, 3, 2]. When SDVA discovers that a variable is dead at a location, it becomes unnecessary for the model checker to track values for that variable.

Figure 1(a) is a simple program with labeled locations that we use to illustrate how SDVA helps reduce the cost of state exploration. We must assume that any possible value may be passed into the function; however, for the sake of brevity, we only consider four input patterns. The reachable state space of the program from the four input patterns is shown in Figure 1(b). There are 11 states in the state space of this program when no dead variable analysis is used. SDVA marks c dead at locations **2**, **3**, and **4**, since c is reassigned at location **4**, and it marks b dead at location **3**, since b is reassigned at location **3**. We can coalesce multiple states into a single state by ignoring dead variables since the values of these variables do not matter. For example, the states s_1 and s_2 in Figure 1(b) become equivalent when the dead variable c is ignored. We combine these into one state in Figure **??**(a). Similar reductions to Figure 1(b) are applied to states (s_5, s_6), (s_7, s_8), and (s_{10}, s_{11}). The final reduced state space from SDVA is shown in Figure **??**(a).

SDVA, being a static analysis technique, does not use any of the dynamic run-time information available during model checking. For this reason, SDVA is conservative and only considers a variable dead if its current value is not

```
1: f (int $a$, int $b$, int $c$);
2: if ($a > 0$) then
3:     $b = 3$;
4: $c = 5$;
5: print a, b, c;
```

(a)



(b)

Fig. 1. A simple program and its reachable state space. (a) A simple program with variables dead at several locations. (b) The reachable state space of the program in (a).

used on any future paths including infeasible paths that are unreachable in any program execution. Additionally, when there is a pending pointer dereference in the program, the variable referenced cannot be known until run-time. Variable aliasing in general cannot be computed statically; therefore, in order to be safe, SDVA must assume all variables could be used at the pointer dereference and declare all variables as live. These two issues cause the SDVA to not find the **true** dead variable set for a state; however, run-time information that is readily available during model checking resolves memory aliasing allowing variables to be positively marked as live and other variables to be marked as dead. Run-time information also reveals the exact path taken through the program. A dead variable analysis that uses run time information during model checking is able to discover a more true dead variable set for each state and possibly generate smaller state spaces. This is the idea behind *dynamic dead variable analysis* (DDVA).

An example of the effects of DDVA can be seen in Figure **??**(b). This state space is generated when variable valuations in addition to program location are used to refine dead variable analysis. When the variable a is greater than zero, it causes the program to go to location **3** which makes b dead at locations **2** and **3**. This allows s_3 and s_4 from Figure **??**(a) to be represented with just a single state, s_3, in Figure **??**(b).

Recent work in DDVA labels variables live or dead dependent on specific future execution paths and is tied directly to the reachable state space of the system [14]. During model checking, [14] simulates single procedure programs forward to discover a partial future path and the variables that are referenced at pointer dereferences. The paths in the program that are not taken in the future are removed from the program. A dead variable analysis on this new program marks more variables as dead because of the missing paths; however, the DDVA algorithm requires user input to determine how far forward to simulate the program in order to achieve the greatest reduction in the state space. Without the correct input value, the algorithm achieves little to no reduction with a substantial increase in verification time and memory used. It is not possible to know what the best explore depth is *a priori* without further analysis. Additionally, the algorithm does not handle programs with loops and non-determinism making DDVA as implemented in [14] impractical to use.

This paper presents a definition of the maximal state space reduction possible from a dead variable analysis and a new algorithm for intraprocedural dynamic dead variable analysis that yields a maximal reduction on single procedure programs with no non-determinism. By triggering analyses only after each trace has been fully determined and by updating states in the reachable state space with new dead variable information, our new algorithm discovers the true set of dead variables for any state. Without non-determinism, the future of an execution path is fixed; however, with non-determinism, the future path is uncertain. A single state can have a future that causes one of its variables be dead and another future where that same variable is live. Variables that become dead after a point of non-determinism cannot be reliably marked as dead before the point of non-determinism without first analyzing the entire reachable state space. In the presence of non-determinism, our algorithm yields the maximum state space reduction that is possible from an on-the-fly dead variable analysis.

2 Related Work

There are currently two relevant works on dead variable analysis in model checking known to us. The work in [3] focuses primarily on showing that live variable analysis defines an equivalence stronger than bisimulation. Static live variable analysis comes at virtually no cost compared to the cost of model checking and is completely orthogonal to other techniques used to attack the state explosion problem. Although SDVA is relatively quick, it only considers program locations in its analysis and can only discover unconditionally dead variables. An analysis that makes use of variable valuations available during model checking, in addition to program locations, can determine more precise paths through the program and find variables that are conditionally dead.

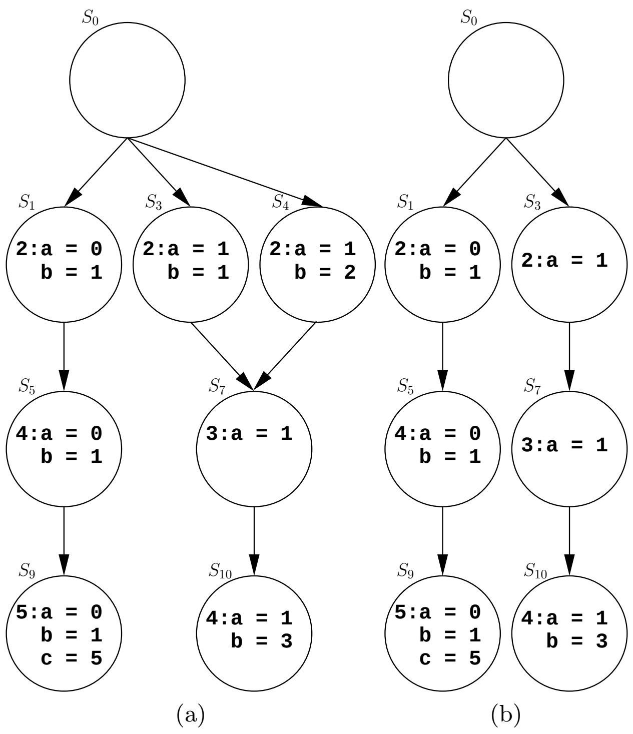


Fig. 2. Two state spaces showing the results of SDVA and DDVA. (a) A reduction of several states from using SDVA. The additional reduction of one state from using DDVA (b).

The dynamic dead variable analysis in [14] uses run time information to resolve conditional branches and pointer dereferences. In order to do this, the DDVA in [14] stops the model checker just before conditional branch points and pointer dereferences are processed and runs a forward analysis. This forward analysis determines a partial path that the program takes in the future and resolves memory aliasing. This forward analysis is terminated either at a user-specified explore depth or at a state with a non-deterministic assignment to a variable. Having a partial path through the CFG allows the analysis to use program locations and variable valuations to more precisely determine dead variable sets. The algorithm prunes off portions of the program that are now known to be unreachable given the observed program locations and variable valuations. The normal SDVA is then run on this reduced program to find more precise sets of dead variables.

Although the DDVA in [14] may find more precise sets of dead variables than SDVA, it presents two issues. The first issue is that there is no correlation between explore depths and state space reductions. This is a consequence of the starting point for each forward analysis and the fact that states cannot have their

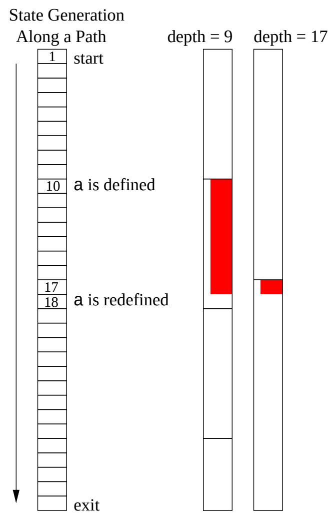


Fig. 3. On the left is the search stack with the variable a defined at state 10 and then redefined at state 18. On the right are two patterns of forward analyses with different explore depths. Highlighted regions show where each analysis marks a as dead.

dead variable sets updated once they have been stored in the $Visited$ set. The algorithm does not run a new forward analysis until the model checker runs past the end of the last forward analysis. Consequently, smaller explore depths have shorter analyses but the analyses happen more often. Whereas bigger explore depths have longer analyses, but the analyses are less frequent. An example of such a situation is illustrated in Figure 3.

Figure 3 demonstrates how longer explore depths do not always translate to greater state space reductions. In the figure, each box on the left represents a state in the search stack in the model checker. Of particular note are states 10 and 18, where a is defined and then redefined such that a is dead from state 10 to state 17. In the DDVA of [14], the forward analysis needs to reach state 18 to discover that a is dead. The way the algorithm is designed, it can only declare a dead in the window of states generated after the start of the forward analysis and before the next non-deterministic assignment. In the forward analysis patterns on the right, each empty rectangle represents the window of states explored by a single forward analysis. The analysis pattern with the smaller explore depth finds that a is dead on the second analysis, and since the analysis starts at state 10, can declare a dead in states 10 through 17. The pattern with the bigger explore depth also finds that a is dead on its second analysis, but since the analysis starts at state 17, it can only declare a dead at state 17; thus, it is impossible to know *a priori* the explore depth to produce the best state space reduction without further analysis of the program structure.

The second issue with the DDVA in [14] is that the true dead variable set for a state is not discovered no matter what explore depth is used. Once states are generated and stored in the $Visited$ set they cannot have their dead variable sets updated even if more dead variables are discovered. Additionally, [14] uses the CFG for DVA. In the presence of loops, the CFG conservatively captures all paths. It has no way to unroll loops and find the exact path taken. These limitations prevent the algorithm from achieving the maximal state space reduction. The goal of this work is to formally define the maximal reduction from DDVA and present an on-the-fly algorithm for computing it.

3 DVA Maximal Reduction

The dead variable abstraction in this work relies on the states and execution paths in the reachable state space and the *control flow graph* (CFG) of the system being verified. A state s is a mapping of variables to a finite domain or $\top$, $s : V \longrightarrow D \cup \{\top\}$, where V is the set of all variables in the system, D is a finite domain, and $\top$ represents an unconstrained or abstracted variable. We use the symbol S to represent the set of all possible mappings of variables to the domain or $\top$. For simplicity, we assume a single initial state, denoted by s_0, that contains the initial mapping; although, the results readily extend to systems with multiple initial states.

A control flow graph is a tuple, (N, E), where N is a set of nodes and $E \subseteq N \times N$ is a set of edges connecting nodes. Each node α in the CFG represents a transition that executes atomically in the system. A transition $\alpha \subseteq S \times S$ relates a state with its next state. A transition is enabled in s if and only if there exists an s' such that $\alpha(s, s')$ holds. A transition is deterministic if and only if for every state s there is at most one s' such that $\alpha(s, s')$ holds. The CFG is used in an iterative dataflow analysis to find dead variables [1]. SDVA and the DDVA in [14] use a CFG to find dead variables in the program. This work uses *execution paths* for the analysis.

An execution path, $\pi = s_0 \overset{\alpha_0}{\rightarrow} s_1 \overset{\alpha_1}{\rightarrow} \cdots$, is a finite or infinite sequence of states and transitions such that s_0 is the initial state and for every i, $\alpha_i(s_i, s_{i+1})$ is a valid transition and $(\alpha_i, \alpha_{i+1}) \in E$ is a valid edge in the CFG. A path suffix π^i is the suffix of the execution trace π starting at s_i. The set of all states that are in traces that begin with s_0 and contain only the transitions in E constitute the reachable state space of the system S_R.

The formal definition we use to mark live and dead variables in a trace makes use of some basic predicates. The predicate $def(v, \alpha)$ is true when the variable v is *defined* by the transition α. Similarly, $used(v, \alpha)$ is true when v is *used* by α. We now give the definition of a variable being live in a state of an execution path:

Definition 1 *A variable v is live in a transition α_i of an execution path $\pi^i = s_i \overset{\alpha_i}{\rightarrow} s_{i+1} \overset{\alpha_{i+1}}{\rightarrow} \cdots$ if and only if:*

- *there exists a $j \geq i$ such that $used(v, \alpha_j)$ and*

- $\neg def(v, \alpha_k)$ *for all* $i < k < j$

We use this definition of live variables in the function $live(\pi^i, v)$, which takes π^i, the suffix of the execution trace π starting at s_i, and returns whether the variable v is live in the first state on the trace. If a variable is not live in a state then it is dead. Intuitively, a dead variable is a variable whose current valuation is not used on any future path.

Variables mapped to $\top$ are abstracted and unconstrained. In this way, a state that has abstracted variables can represent many different states. The set of all abstracted variables in a state s is $abstract(s) = \{v \mid s(v) = \top\}$ and the set of concrete variables is $concrete(s) = \{v \mid s(v) \in D\}$.

In order to compare and match states that have differing sets of abstracted variables we define a relation between two states called *contains* denoted $\preceq_c$.

Definition 2 *A state s' is contained in s, denoted $s' \preceq_c s$ if:*

- $abstract(s') \subseteq abstract(s)$ *and*
- *For all variables v in $concrete(s')$, $s'(v) = s(v)$*

A state is contained in another state if the set of dead variables of the first state are a subset or equal to the set of dead variables of the second state and variables that are live in both states are equal.

SDVA only uses the information available in the CFG of the program to do the analysis which admits infeasible paths and produces an imprecise set of dead variables. When a precise execution path through the CFG is used to find dead variables, the true dead variable sets for every state on the trace can be calculated. Finding the true dead variable set for each state in the reachable state space produces an abstract state space that is a *DVA maximal reduction* of the concrete state space.

Definition 3 *An abstract state space S'_R is a* ***DVA maximal reduction*** *of the concrete state space S_R if and only if:*

- *For every reachable execution trace starting at the initial state $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \cdots$ in the concrete state space, there exists an abstract execution trace $\pi' = s'_0 \xrightarrow{\alpha_0} s'_1 \xrightarrow{\alpha_1} \cdots$ such that for all i, $s_i \preceq_c s'_i$ and $s'_i \in S'_R$*
- *For all states s' in S'_R, and for all variables v in V, if the value of v in s' is not $\top$, then there exists a reachable concrete trace $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \cdots$ and an $i \geq 0$ such that $s_i \preceq_c s'$ and $live(\pi^i, v)$*

The original DDVA in [14] uses some runtime information to refine SDVA and find more dead variables; however, it is not able to construct a DVA maximal reduction of a concrete state space and occasionally creates an abstract state space that is no smaller than the state space produced using SDVA. The dynamic dead variable analysis in this work implements Definition 4 on-the-fly to produce a DVA maximally reduced state space in the absence of non-determinism. In the presence of non-determinism, our dynamic dead variable analysis produces the closest approximation to a DVA maximally reduced state space that is possible to produce on-the-fly.

4 Maximal Dynamic Dead Variable Analysis

Our DDVA algorithm achieves a DVA maximal reduction by analyzing fully determined execution paths through the program instead of partial future paths generated from a forward analysis. A fully determined execution path is a single execution path that has been fully explored; it generates no more unique states. An execution path that has reached the exit of the program or a path that has reached an already visited state (representing a path that has entered an infinite loop or merged into an already explored path) are the two kinds of fully determined execution paths. Whenever the search generates a fully determined path, a dead variable analysis is performed. The equation in Definition 2 is used to calculate the new sets of dead variables for each state in the path starting with the last state in the path. The exception to this is when a prefix for a trace is unique but all states in its suffix are already in the $Visited$ set. In this case, we can use the dead variable information we already calculated for the suffix to start calculating the dead variables at the last state of the prefix.

When the model checker fully resolves an execution path through the program, the dead variable analysis may find more dead variables for states that have already been explored. A full execution path reveals dynamic run time information of all of the states in the path, allowing the analysis to positively declare variables live or dead. Updating the dead variable sets of visited states requires that they be re-stored in the $Visited$ set. In order to avoid storing states that are later found to be duplicates when their dead variable sets are updated, we use the *contains* relation to ascertain whether a state is unique even before its final dead variable set is generated. In our algorithm, if $s' \preceq_c s$, s' is a newly generated state, and $s \in$ *Visited*, then s' is **not** inserted into *Visited*, because it is contained in s. This pre-emptive duplicate detection saves us from having to generate and store states that are later found to be duplicates.

The new algorithm to dynamically find dead variables, shown in Figure 4, is remarkably simpler than the work in [14]. The function **dfs** performs a standard depth-first search to enumerate the entire state space of the model. *Stack* is the depth-first search stack. An entry in *Stack* consists of (s, A), with s being a state that includes the location and A being a set of transitions that can be applied to the state to get a next state and location. For our $Visited$ set, we use a hash table that implements the contains relation to compare states. The function $a(s)$ takes a state s and returns the set A, $a : s \longrightarrow A$. A transition $\alpha \in A$ maps a current state onto a next state as defined previously. When a duplicate state is generated (line **11**) or the exit is reached (line **16**), model checking is suspended and a dead variable analysis is run. In the case that the exit of the program is reached, **updateDeadVars** is called with **null** because the entire trace is contained in *Stack*. When a duplicate state is reached, **updateDeadVars** is called with the state in the hash table that matched the newly generated state.

The equation used in **updateDeadVars** to calculate dead variables sets in a state requires as input the previous state's set of dead variables. The variable used for this, $PreviousDeadVars$, is initialized to all variables at line **22** when the exit is reached due to the fact that all variables are dead at the exit.

When a partial path is in *Stack* and a path suffix is in $Visited$, we initialize $PreviousDeadVars$ to be the dead variables in the state we matched on, line **24**. Finally, in cases where the exit is not reached because the modeled program enters an infinite loop, the analysis is started with $PreviousDeadVars$ being empty, line **26**, as we cannot determine what the previous state's dead variable set is without entering into an infinite loop ourselves. When a state maps to a non-deterministic assignment in the program, as indicated by the return value of *nonDeterminism(*s_{trace}*)*, $PreviousDeadVars$ is set to the empty set, because dead variables discovered after a point of non-determinism cannot be used to calculate the set of dead variables for states before the non-determinism. This point is explained in greater depth at the end of this section.

The equation for the definition of a dead variable is applied at line **31** of **updateDeadVars** to find the set of dead variables for each state in the trace. The function $def(A) = \{v \mid \forall \alpha \in A, def(\alpha, v)\}$ returns the set of variables that are defined in a set of transitions and the function $used(A) = \{v \mid \forall \alpha \in A, used(\alpha, v)\}$ returns the set of variables that are used by a set of transitions. If the analysis finds more dead variables than are currently in the state, the states in *Stack* are updated with their new dead variable sets. Variables that are always live, such as the location, are never abstracted. The updated states are re-stored in the hash table using the function ***replace***$\boldsymbol{(}Visited, s_{trace}, s'\boldsymbol{)}$ (line **34**).

The following is an example run of the algorithm in Figure 4 that produces the state space shown in Figure **??**(b). Since some states shown in Figure **??**(a) are produced and then later have their dead variable sets updated to become the states in Figure **??**(b), we add a superscript, a or b, to states that differ between the two figures. Our model checking run starts with s_0 as our start state. The state s_0 is pushed onto *Stack* at line **2** and then the depth-first search is called at line **3**. In the main loop of **dfs**, s_0 is retrieved from *Stack*. Line **8** chooses a transition α from s_0's transition set, if there is more than one transition, and removes the transition from the set. Then line **9** uses the transition to produce s_1 from program location **1** of Figure 1. We check for uniqueness of the newly generated state in lines **10** and **11**. If the state is not contained in any other state in $Visited$, then it is a unique state. The new state in this example is unique so we add it to $Vistited$ and then push it onto *Stack* at lines **14** and **15**. We need to perform a dead variable analysis on each trace after it has been generated, so we check if this trace has finished at line **16** by checking to see if the current state's location is the program exit.

The current trace has not reached the exit so we return to the top of the loop and take s_1 off of the top of *Stack*. The state s_1 has a single action in its action set. This action is used to produce s_5 which is added to $Visited$ and *Stack*. The third time through the main loop of **dfs**, s_5 is retrieved from *Stack* at line **6**. The state s_8, the successor of s_5, is generated and pushed onto *Stack*. Since s_8 is generated at the exit location, we call **updateDeadVars** at line **17**. All variables are dead at the end of the program so the set $PreviousDeadVars$ is set to contain all the variables in the program at line **22**. We iterate backwards through the trace calculating the dead variables for each state starting at the last state. The dead variables of the current state are calculated using the formula on line **31** and then the appropriate variables are marked as dead at line **32**. In this example, no new dead variables are found, so we return from **updateDeadVars**. The third time through the main **dfs** loop, s_8 is at the top of *Stack*. It does not require a dead variable analysis, and it has an empty action set, so it is left off of *Stack*, and we look at s_5. The state s_5 also has no more children, so it is also popped off *Stack* and then the same process occurs for s_1.

The next action in s_0's action set produces s_3^a. The state s_3^a does not trigger a dead variable analysis and the successor of s_3^a, s_6^a, also does not trigger an analysis. The next state, s_9, is at the exit, so another dead variable analysis is run. This time the variable b is found to be dead at program locations **2** and **3**. Marking b as dead in s_3^a and s_6^a produces the states s_3^b and s_6^b which replace the previous states at lines **33** and **34**.

After returning from **updateDeadVars**, s_9, s_6, and s_3 are popped off of *Stack*. The next successor of s_0 is s_4 which is contained in s_3^b, so it is not added to the *Stack* or *Visited*. Only s_0 is in *Stack* when **updateDeadVars** is called so no new dead variables are found. Now that s_0's action set is empty, it is popped from *Stack* and state generation has completed.

Our DDVA algorithm is designed on the definitions in the previous section. As such, we claim that using our algorithm to model check single procedure programs without non-determinism produces DVA maximally reduced state spaces on-the-fly by implementing Definitions 2 - 4; however, the presence of non-deterministic assignments to variables can affect the future path from a state so that a state with a non-deterministic assignment can have more than one possible future. These multiple futures of a single state may cause the state to have different sets of dead variables. It is possible that the non-determinism does not actually affect the state's dead variable set, but the only way to know for sure is to examine the entire reachable state space; however, once the entire reachable state space is produced, model checking has finished and there is no longer a need to find more dead variables.

An example of how an execution path can affect states produced before the point of non-determinism is presented in Figure 5. The function **get_input** represents non-deterministic input from an outside source that ranges over a large finite domain. The variable a is dead at location **2** if c is greater than 2 and the path goes through location **4**. A state generated at location **2** could not have a marked as dead because c might be assigned a value less than or equal to 2, making a live. It is possible that every single value returned by **get_input** at location **2** during model checking is greater than 2, which means we can mark a dead at location **2**; however, the only way to check if **get_input** always returns a value greater than 2 is to finish generating the entire reachable state space.

In order to not incorrectly mark variables as dead in the presence of non-determinism, dead variable knowledge gained after a non-deterministic assignment cannot be used on states generated before the assignment unless we first generate every possible assignment and future path for the analysis. It is possible

```
 1: verify $((l_0, s_0))$
 2: push($Stack$,  $s_0, a(s_0 red)$))
 3: dfs()

 4: dfs ()
 5: while $Stack \neq \oslash$ do
 6:    $(s, A)$ := peek($Stack$)
 7:    if $A \neq \oslash$ then
 8:       choose and remove transition $\alpha$ from $A$
 9:       $s' := \alpha(s)$
10:       for all $s_i \in Visited$ do
11:          if $s' \preceq_c s_i$ then
12:             updateDeadVars($s_i$)
13:             goto: line 5
14:       $Visited := Visited \cup \{s'\}$
15:       push($Stack, (s', a(s'))$)
16:       if $s$ is at $ExitLocation$ then
17:          updateDeadVars(null)
18:    else
19:       pop($Stack$)

20: updateDeadVars ($s_i$)
21: if $Stack.LastState$ is at $ExitLocation$ then
22:    $PreviousDeadVars := V$
23: else if $s_i \notin Stack$ then
24:    $PreviousDeadVars := abstract(s_i)$
25: else
26:    $PreviousDeadVars := \oslash$
27: for $s_{trace} := Stack.LastState$ to $Stack.FirstState$ do
28:    if nonDeterminism($s_{trace}$) then
29:       $PreviousDeadVars = \oslash$
30:    $A := a(s_{trace})$
31:    $DeadVars := (PreviousDeadVars \cup def(A)) \cap \neg used(A)$
32:    $s'$ = setAbstract($s_{trace}, DeadVars$)
33:    if $s' \neq s_{trace}$ then
34:       replace($Visited, s_{trace}, s'$)
35:    $PreviousDeadVars := DeadVars$
```

Fig. 4. Pseudocode of the maximal DDVA algorithm.

```
1: $a$ = **get_input**();
2: $c$ = **get_input**();
3: **if** $c > 2$ **then**
4:     $a = 5$;
5: **print** $a$, $b$, $c$;
```

Fig. 5. A program fragment that has a point of non-determinism that affects what can be declared dead above it.

that on some models this strategy does find the DVA maximal reduction as it may be the case that the non-determinism in a particular model does not affect dead variable sets in preceding states. We cannot determine on-the-fly whether this is the case, so our algorithm produces state spaces that are not technically DVA maximally reduced when non-determinism is present.

5 Results

We implemented our DDVA algorithm in the Estes model checker developed at the BYU Software Model Checking Lab [15]. Estes uses the GNU debugger as a state generator in order to verify software at the object code level. Since a single line of code from a high level language can easily translate into 2 or more object code instructions, ways to reduce the size of the explored state space are invaluable. The specific simulator we use as our state generator is based on the Motorola 68hc11 processor and can be found in the Gnu Debugger (GDB) [8]. We use the tools found in the GEL collection of libraries [9] to compile C source code into the binary files that run in the simulator.

In order to implement the contains relation, we need to be able to compare new states with existing states to see if the new state is contained in another state; however, comparing each new state with all the existing states in the $Visited$ set is too unwieldy as the set becomes larger. In order to mitigate this problem, we use a chained hash table, where each chain has a subset of variables that are all equal and that can never be dead. In all of our examples, we mark the registers and location as the set of variables that are never dead and hash on this set to find the correct chain. Once the correct chain is found, the state is compared to each of the states in the chain until an exact match or containing state is found, or the end of the chain is reached. If a match or containing state is found, then the new state is discarded. If the new state is unique, it is simply appended to the end of the chain.

We compare the implementation of our DDVA algorithm against normal model checking, model checking with SDVA, and the best and worst runs of the DDVA in [14]. We compare the different techniques running on 6 different models in the following areas:

– *States generated*: Size of the $Visited$ set at the end of model checking.
– *Wall clock time*: Total time taken to finish model checking.

– *Total memory used*: The total amount of memory used by the model checker to complete a model checking run.
– *Abstraction time*: Total amount of time taken in the dead variable analyses.

We test the algorithms on a number of artificial and real world tests including the main test used to benchmark the DDVA in [14]. The first three models are artificial with no real world objective other than to showcase the kind of state space reductions that are possible with a dynamic dead variable analysis. The last three models are mock-ups of real world functions or programs than can be found in embedded platforms or general purpose computers. The results are shown in Figure 6 and Figure 7.

The data in Figure 6 and Figure 7 show how the DDVA in [14] either results in no better reduction than SDVA or has widely varying results depending on the explore depth. Our DDVA on the other hand always has a smaller state space than SDVA, and thus, always has lower memory usage than all of the other methods. For simplicity, the DDVA algorithm in [14] is referred to as *original* in the tables, while our algorithm is referred to as *maximal*.

The `easy3` model is a program with several global integer variables that non-deterministically receive a value at the beginning of the program. The rest of the program contains conditional branches and, depending on values of the variables, all but one variable becomes dead in each branch. The results are shown in the top table of Figure 6. This is an example that benefits greatly from dead variable analysis. The original DDVA discovers dead variables at the exact same point that SDVA finds dead variables in this example and incurs the time penalty of extra analyses for no state space reduction. Our DDVA reduces the state space and is only slightly slower than SDVA. The original DDVA performs more analyses and thus takes almost twice as long as our DDVA to do its abstraction and yet gains nothing over the static analysis. Our DDVA produces a 35% smaller state space and correspondingly has a lower peak memory usage.

The `littleBranch` model is similar to `easy3`; although, it contains nested conditionals which the original DDVA can take advantage of with the right explore depth. The results are shown in the middle table of Figure 6. This model, however small, illustrates the difficulty in achieving a good result with the original DDVA. Our DDVA, on the other hand, gives the largest state space reduction, takes the least time to complete, and is able to do this every time without a user specified depth bound.

The `multiBranch` model shown in Figure 6 is a much larger version of the `littleBranch` model that is used to test the original DDVA. In addition to having deeper nesting than `littleBranch`, `multiBranch` makes use of local variables that are referenced as an offset from the frame pointer. Whenever there is an upcoming pointer dereference, SDVA is forced to declare all variables live. The results from this model are shown in the lower table of Figure 6. This is a good example of a situation where DDVA is engineered to surpass the performance of SDVA; however, again the performance of the original DDVA is unpredictable, and at its worst, generates more states than the static analysis due to the strict state comparison in the hash table. Please note that although our DDVA gener-

Model Name: easy3, **Lines of Code:** 38

Analysis	Explore Depth	States Generated	Total Time	Memory Used (MB)	Abstraction Time
None	N/A	34640	0m12.764s	34.5	0.0s
Static	N/A	15814	0m6.605s	33.80	0.001s
Original best	2	15814	0m10.765s	34.46	3.792s
Original worst	2	15814	0m10.765s	34.46	3.792s
Maximal	N/A	10330	0m8.105s	25.5312	2.017s

Model Name: littleBranch, **Lines of Code:** 57

Analysis	Explore Depth	States Generated	Total Time	Memory Used (MB)	Abstraction Time
None	N/A	864	0m0.442s	30.9	0.0s
Static	N/A	721	0m0.405s	31.4	0.001s
Original best	6	658	0m0.344s	31.43	0.074s
Original worst	2	721	0m0.34s	31.43	0.0492s
Maximal	N/A	530	0m0.223s	23.79	0.0138s

Model Name: multiBranch, **Lines of Code:** 140

Analysis	Explore Depth	States Generated	Total Time	Memory Used (MB)	Abstraction Time
None	N/A	294515	1m49.170s	87.1	N/A
Static	N/A	217454	1m21.780s	74.87	0.002s
Original best	16	176651	1m41.458s	75.79	42.67s
Original worst	5	217478	2m10.965s	83.46	46.35s
Maximal	N/A	145440	2m36.640s	57.99	7.513s

Fig. 6. Results for 3 artificial models. All 3 models are designed to showcase the benefits of using DDVA.

ates the smallest state space in this example, it incurs a higher run time due to the long chains in the chained table.

Figure 7 gives the results from the `lexer`, `robot` and `bintree` models. The `lexer` model is patterned after a function in a simple lexical analyzer. The model simulates input as a string of characters which the function reads and then returns a token based on what is in the first one or two characters. The `robot` model simulates a line following robot with three sensors. The robot changes the speed of its left and right motors based on input from the three sensors. In both of these models, our DDVA has the smallest state space and lowest memory usage while taking equal or less time to complete. The `bintree` model is the only model with a loop in it. This model searches a binary tree for a specific node. Due to algorithmic limitations, the original DDVA typically does not perform well on models with loops because its analysis is tied to the CFG. Our DDVA does much better because it analyzes entire traces through the program which is equivalent to unrolling the loop as many times as needed

Model Name: lexer, **Lines of Code:** 92

Analysis	Explore Depth	States Generated	Total Time	Memory Used (MB)	Abstraction Time
None	N/A	262843	1m28.391s	66.9	0.0s
Static	N/A	226169	1m17.633s	66.32	0.002s
Original best	2	225370	1m51.479s	71.30	31.66s
Original worst	3	226172	1m53.866s	71.13	33.46s
Maximal	N/A	74024	1m45.56s	37.69	4.898s

Model Name: robot, **Lines of Code:** 55

Analysis	Explore Depth	States Generated	Total Time	Memory Used (MB)	Abstraction Time
None	N/A	35865	0m12.838s	35.3	0.0s
Static	N/A	27940	0m10.377s	35.6	0.002s
Original best	2	27940	0m18.675s	36.21	7.947s
Original worst	2	27940	0m18.675s	36.21	7.947s
Maximal	N/A	27784	0m11.494s	29.21	0.552s

Model Name: bintree, **Lines of Code:** 31

Analysis	Explore Depth	States Generated	Total Time	Memory Used (MB)	Abstraction Time
None	N/A	157828	1m0.608s	66.5	0.0s
Static	N/A	154084	1m1.061s	68.4	0.005s
Original best	6	150964	2m14.807s	73.74	72.09s
Original worst	2	154084	2m7.356s	71.47	64.87s
Maximal	N/A	103839	1m7.530s	52.62	16.34s

Fig. 7. Results for 3 real-world models. The `lexer` model is a simple lexical analyzer. The `robot` model simulates a line following robot. The `bintree` model searches a binary tree for a specific node.

and then performing dead variable analysis on the unrolled loop as shown in the bottom table of Figure 7.

6 Conclusions and Future Work

Dead variable analysis is an effective means of reducing the size of the explored state space in model checking while retaining all relevant behaviors of the system. Dynamic dead variable analysis provides a way of finding a larger set of dead variables for each state resulting in even smaller state spaces than those generated using SDVA. Our DDVA greatly improves upon the ideas set forth in the original DDVA of [14] by eliminating the dependence on a user specified explore depth and by producing a DVA maximally reduced state space for models with no non-determinism and the closest possible approximation to a DVA maximally reduced state space in models that contain non-determinism. Our algorithm also correctly addresses looping structures in the analysis.

Our maximal DDVA algorithm is currently limited to single procedure programs. Future work focuses on modifying our DDVA algorithm to work on multi-procedural programs. The easiest way to do this is to declare all global variables as live, and treat every procedure and its local variables as a separate program. As the program returns from a procedure, a dynamic dead variable analysis is run on the trace of states generated through the procedure and dead variables sets for states generated in the procedure are updated.

Other areas of future work include finding ways to speed up run time, adapting the algorithm to different searches, and using a more efficient way of representing dead variables. The current implementation of the algorithm suffers from an increase in run time on large models that can make state space exploration infeasible. This increase in run time comes from the use of a chained hash table and the contains relation. An avenue for future work would be to look into ways to mitigate this problem. Another direction for future work adapts DDVA to work with other search algorithms such as breadth-first search. The benefit of breadth-first search is that paths that reach an error state are guaranteed to be the shortest path to the error. Lastly, the current data structure used to mark dead variables is highly inefficient. Some future work could be dedicated to creating data structures that take less memory to store dead variable information.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
2. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *7th International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer, August 2000.
3. M. Bozga, J. Fernandez, and L. Ghirvu. State space reduction based on live variables analysis. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis, 6th International Symposium, SAS '99, Venice, Italy, September 22-24, 1999, Proceedings*, volume 1694 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 1999.
4. A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model verifier. In *Computer Aided Verification*, pages 495–499, 1999.
5. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
6. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, R. Zheng, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.
7. Y. Dong and C. R. Ramakrishnan. An optimizing compiler for efficient model checking. In *FORTE XII / PSTV XIX '99: Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*, pages 241–256. Kluwer, B.V., 1999.

8. The gnu project debugger. Available at http://sources.redhat.com/gdb/, 2006.
9. Gnu embedded libraries for 68hc11 and 68hc12. Available at http://gel.sourceforge.net/, 2005.
10. K. Havelund and T. Pressburger. Model checking java programs using java pathfinder, 1998.
11. T. A. Henzinger, R. Jhala, R. Majumdar, , and G. Sutre. Software verification with Blast. In T. Ball and S.K. Rajamani, editors, *Proceedings of the 10th International Workshop on Model Checking of Software (SPIN)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239, Portland, OR, May 2003.
12. G. J. Holzmann. The engineering of a model checker: the gnu i-protocol case study revisited. In *Proc. of the 6th Spin Workshop*, volume 1680 of *Lecture Notes in Computer Science*, Toulouse, France, 1999. Springer Verlag.
13. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003.
14. M. S. Lewis and M. D. Jones. A dead variable analysis for explicit model checking. In *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program*, 2006.
15. E. G. Mercer and M. Jones. Model checking machine code with the GNU debugger. In *12th International SPIN Workshop*, volume 3639 of *Lecture Notes in Computer Science*, pages 251–265, San Francisco, USA, August 2005. Springer.
16. M. Robby and J. Dwyer. Bogor: an extensible and highly-modular software model checking framework, 2003.