Locating Trace Change Points in Parallel Programs

Zulah K. F. Eckert and Gary J. Nutt
Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, CO - 80309

CU-CS-730-94 May 1994

Locating Trace Change Points in Parallel Programs

Zulah K. F. Eckert*and Gary J. Nutt
Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, CO - 80309

May 1994

Abstract

The problem of exactly locating the set of TCPs for a parallel program is NP-Hard. In this report, we demonstrate that the problem of locating TCPs in a parallel program is a global flow analysis problem. We introduce the shared variable dependence relation (SVD relation) and show that this relation can be used to approximate the set of TCPs for a parallel program. This relation can be computed in $O(N \times E \times V^2)$. Finally, we present a simplification of this algorithm, requiring only $O(N \times E)$ for SVD relation computation that trades run time for computational accuracy.

*This work was supported in part by Convex Computer Corporation.

Contents

1 Introduction

In [1] we have demonstrated that the problem of locating the exact set of TCPs for a program is NP-hard. In this report, we demonstrate that the problem of locating TCPs is a global flow analysis problem that can be solved in $O(N \times E \times V^2)$. The algorithm presented is similar to ones computing constant propagation.

In this report, we present two algorithms that approximate the set of TCPs for a parallel program. Both algorithms provide a conservative approximation to the actual set of TCPs in a parallel program. That is, it is assumed that any conditional or iterative statement that is dependent on the value of a shared variable, either directly or indirectly, is a TCP. This reduces the problem of locating TCPs to that of determining the set of variables that are dependent on the value of a shared variable either directly or indirectly. We call this set the *shared variable dependent variable* set (SVD variable set). Computing the set of SVD variables is similar to the problem of constant propagation, a global flow analysis problem [6]. As with constants, SVD variables *propagate* through a program. There are two fundamental differences between the problems. First, the goal of constant propagation is to determine the actual values of constants (whenever possible). In contrast, the SVD variable set problem requires no computation of program values. Instead, the knowledge of whether or not a variable is an SVD variable is collected. Second, a variable is a constant prior to the execution of a statement if it was a constant on all paths leading to the statement and the constant value was the same. A variable is an SVD variable prior to the execution of a statement if it was an SVD variable on some path leading to the statement.

Iterative statements complicate the process of locating dependencies in a program. For programs containing no iteration, each statement can only depend on preceding statements. That is, dependencies can be computed with a single top to bottom traversal of the program. For programs containing iterative statements, each statement can depend on succeeding statements due to a cycle in the control flow graph. This implies that collection of the set of SVD variables for an iterative statement requires iteration over the set of statements in the body of the iterative statement until the set of SVD variables converges. This method has the advantage of accurately locating the changing dependencies that may be present in an iterative statement. We present an algorithm using this method based on Wegman and Zadeck's solution to constant propagation, see *simple constant* [6].

The cost of accurate computations for iterative constructs is a worst case run time of $O(N \times E \times V^2)$. We present a simplification requiring only $O(N \times E)$ for SVD variable location that trades time for computational accuracy. Unlike constant propagation, the order in which statements occur is not as important for the SVD variable problem because actual program values are not needed. We introduce the *shared variable dependence graph* (SVD graph) of a loop and present a simplification for iterative statements using this representation. The SVD graph of a loop is a representation of the variable dependencies within the loop. Whenever a loop is encountered, the graph is sorted based upon the current SVD variables. This sorting results in a set of SVD variables for the loop. The loop can then be analyzed for TCP statements. This approximation can be less accurate than the previous algorithm but has a worst case runtime of $O(N \times V)$.

2 The Shared Variable Dependence Relation

Given two variables a and b, we say that a is *directly dependent* on b, denoted $a \rightarrow b$, if b appears on the right hand side of an assignment to a or if b is used to decide a conditional statement (or as an iterative statement index) in the body of which an assignment to a appears. If b is a shared variable, then a is *directly shared variable dependent* on b, denoted $a{\rightarrow_{sv}}b$. A variable a is *shared variable dependent* if it is either directly shared variable dependent on some shared variable or there is a chain of dependencies such that $a \rightarrow_{sv} v_1 \rightarrow_{sv} \dots \rightarrow_{sv} v_n$ where v_n is a shared variable. The *shared variable dependence relation* (SVD relation) is the set of all shared variable dependencies for a program.

2.1 SVD variables

Determining the set of TCP statements for a program can be accomplished by determining the set of program variables dependent on the value of a shared variable. A local variable's dependence on a shared variable can change over the execution of program statements. For example, a local variable may be assigned the value of a shared variable in one statement and assigned to a constant value in a subsequent statement. Immediately following the first statement, the local variable is SVD dependent, however immediately following the second, it is no longer SVD dependent. The SVD relation can be used to estimate the set of program variables dependent (either directly or indirectly) on a shared variable (the set of *SVD variables* for the program). The set of variables present in the SVD relation at any program statement is the set of SVD variables for that statement. For each variable, membership in the SVD variable set is either *preserved*, *killed*, or *generated* during execution of a statement. The SVD variable set of maintained after the execution of each statement by removing all killed variables from the set and by adding any generated variables to the set.

There are two ways in which a variable can become an SVD variable. First, a dependency on a shared variable may arise as a result of a chain of dependencies due to assignment statements, a *definition dependency*. Second, a dependency may result due to a chain of dependencies in which at least one dependency is due to a conditional or iterative statement, an *indirect dependency*.

2.2 A Motivating Example

```
                              A0:begin
                              shared int x = 0, y = 0;
                              int n = 2, z = 0;
                              A1:read(z);
                              A2:task_create(n)
task 0                                                  task 1
int i,j,k,l;                                            int p,q,r,s;
B0: read(j,k);                                          C0:read(p,q);
B1: for i = 1 to j do                                   C1:x = x + 1;
B2:     if k > j then                                   C2:for r = 1 to x do
B3:         l = y;                                      C3:     p = p + x;
B4:     k = i + x;                                      C4:     q = q + p;
B5: end;                                                C5:     s = 1;
                              C6:end;
                              A3:task_terminate(n);
                              A4:if z > n then
                              A5:     z = n;
                              A6:print(n);
                              A7:end;
```

Figure 1: A parallel program illustrating SVD set calculation

In this section, we use an example to motivate the calculation of the SVD variable set and location of possible TCP statements. Figure 1, is used to demonstrate the calculation of the set of SVD variables and the location of TCP statements in a parallel program. We use S to denote the set of SVD variables and N to denote the set of variables that are not SVD variables. The contents of these sets will vary at different points in the program. Before any calculation begins, statements can be classified depending on whether or not they have to possibility of being TCP statements. For example, an assignment statement containing no array references cannot be a TCP statement and **task_create** statements are all TCP statements. We call statements that have the possibility of being TCP statements *candidate TCP statements.* Figure 1 contains five *candidate TCP statements*: statements A2, B1, B2, C2, and A4. Three of these statements are are TCP statements: statements A2, B2, and C2. Statement A2 is a TCP statement because it is a **task_create** statement. The set of SVD variables is used only to determine whether or not a candidate statement is a TCP statement.

Initially $S = \{x, y\}$ and $N = \{z, n\}$. For task 0, the sets are initially $S = \{x, y\}$ and $N = \{i, j, k, l\}$ and for task 1 they are initially $S = \{x, y\}$ and $N = \{p, q, r, s\}$. The candidate TCP statements for task 0 are statements B1 and B2. Statement B0 generates no generated or killed definitions, resulting in no change to the S and N sets. Statement B1 is not a TCP statement because $j \in N$ immediately prior to this statement. Statement B2 is a TCP statement since on some iteration of the loop, it is possible that $k \in S$. For task 1 the candidate TCP statement is statement C2. Prior to statement C2, $S = \{x, y\}$. The iterative statement C2 depends on the value of the shared variable x and the sets have the following values at the end of the loop $S = \{x, y, p, q, r, s\}$ and $N = \{\}$. The dependencies that include variables p, q, and s in S are indirect (note that there are dependencies due to a definition in C3 and C4). Statement C2 is a TCP statement because $x \in S$. For the main thread, statements A2 and A4 are candidate TCP statements. At the termination of tasks 0 and 1, the SVD variable sets are merged for global variable (in this case there are none) and $S = \{x, y\}$ and $N = \{n, z\}$. Statement A3 has no effect on these sets. Statement A4 is not a TCP statement because neither $z \in S$ nor $n \in S$. Therefore, statements A2, B2, and C2 are the only TCP statements.

3 Computing the *SVD* Relation

Shared variables are by definition always SVD variables (i.e., members of the SVD variable set). Local variables become SVD dependent variables either as a result of a direct dependency, due to an assignment statement, or as a result of indirect dependencies that arise as a result of control flow[1].

3.1 Assignment Statements

Any assignment statement has the possibility of generating new definitions or killing old definitions. An SVD variable is *generated* by an assignment statement, if the variable of the left hand side of the statement is not already an SVD variable, and if some variable of the right hand side is an SVD variable. If the left hand side of an assignment statement is currently an SVD variable and is not a shared variable, and the right hand side of the assignment statement contains no SVD variables, then the variable is removed, *killed*, from the current set of SVD variables. Consider the example of Figure 2. Statement A1 generates a single SVD variable, i. If the set of variables dependent on shared variables is $S = \{x\}$ and $N = \{i, j, k, m\}$ before the execution of statement A1, then after its execution we have $S = \{x, i\}$ and $N = \{j, k, m\}$. Statement A2 does not generate or kill any SVD variables. Statement A3 generates SVD variable j yielding $S = \{x, i, j\}$ and $N = \{k, m\}$. Statement A4 kills the SVD variable i resulting in $S = \{x, j\}$ and $N = \{i, k, m\}$.

[1] We put off a discussion of the problem of determining SVD variables for procedure or function bodies until the end of this section.

```
int i,j,k,m;
shared int x;
:
A1: i = 2 * x + k;
A2: k = k + 1;
A3: j = k * i;
A4: i = k + 1;
A5: if j > 0 then
A6:       k = k + 2;
A7: else
A8:       m = m + k;
A9: ...
```

Figure 2: A program fragment containing assignment statements

The presence of array references complicates this simple calculation. Array elements are treated as individual variables except when an actual location (i.e., array element) cannot be determined. Because the actual element of the array that is SVD is unknown, the entire array must be considered to be SVD variables[2]. An array reference can become SVD dependent in two ways. First, if a reference appears on the left hand side of an assignment statement and an SVD variable appears on the right hand side of the same assignment statement then the reference is an SVD variable. Second, if the variable appears anywhere in an assignment statement and the index of the array reference is itself an SVD variable. In either case, the array element in question is added to the set of SVD variables. If an array reference appears on the left hand side of an assignment statement, the index for the reference is not an SVD variable, and there is no SVD variable appearing on the right hand side of the assignment statement, then if the array location was an SVD variable, the reference represents a kill. Consider the example of Figure 3. $S = \{x\}$ initially and $N = \{i, j, k, m, a[1], a[2], a[3], a[4], a[5]\}$ immediately preceding the execution of statement B1. Statement B1 is a TCP statement since the array reference $a[x]$ depends on the value of $x \in S$. This statement does not generate any shared variable dependencies. Statement B2 generates i as an SVD variable. Statement B3 is again a TCP statement since the reference $a[i]$ depends on the value of $i \in S$. This statement generates j as an SVD variable, $j \in S$. Statement B4 is not a TCP statement since the array reference $a[k]$ is not dependent on a shared variable. However, $a[k]$ is generated (because its value depends on SVD variables j and x) If we can determine the location of the array that

[2]Methods for array sectioning exist but are out of the scope of this work. See [5] for more details

is shared variable dependent, then it is added to S. If we cannot determine the exact location, then in order to guarantee that all TCP statements are located, the entire array is placed in S. Statement B5 may or may not be a TCP statement depending on which of the above decisions we make at statement B4 (i.e., statement B5 depends upon the contents of S). In the algorithm that follows, we omit the details of determining dependencies for array references, see [5] for details of array analysis in data flow problems.

```
int i,j,k,m;
int array a[1..5];
shared int x;
⋮
B1: a[x] = i + j;
B2: i = i + x;
B3: j = a[i];
B4: a[k] = x + j;
B5: if a[m] then
⋮
```

Figure 3: A program fragment containing array reference assignment statements

3.2 Conditional Statements

A conditional statement can result in indirect dependencies. In the event of an indirect dependency, any definition that occurs within the body of the conditional statement, regardless of the outcome, is generated by the execution of the conditional statement. Consider statement A5 in Figure 2. Before execution of this statement $S = \{x, j\}$ and $N = \{i, k, m\}$ hence A5 is a TCP statement. After execution of this statement (immediately preceding the execution of statement A9), we have $S = \{x, j, k, m\}$ and $N = \{i\}$. The variables k and m are generated by the conditional statement.

If a conditional statement is not a TCP statement, then the SVD variable set is calculated along each branch of the conditional and TCP statements are located. Each branch is evaluated with the SVD variable set that is present on entry to the conditional statement. Because each branch may possibly execute and execution will occur independent of one another, the resulting SVD variable sets are merged at the end of the conditional statement. That is, the SVD set at the end of a conditional statement is the union of the SVD sets from each branch of the conditional statement. Consider the example of Figure 4. Initially, $S = \{x\}$ and $N = \{i, j, k, m\}$. Statement C1 is not a TCP because $j \in N$. The SVD variable set S after the execution of the conditional

```
int i,j,k,m;
shared int x;
⋮
C1: if j > 0 then
C2:       i = i + x;
C3:       j = j + 1;
C4: else
C5:       j = a[i] + 1;
C6:       k = x + 1;
C7:       m = i + k;
C8:
⋮
```

Figure 4: A program fragment containing a non TCP conditional statement

statement (i.e., at the beginning of C8) is $S = S_1 \cup S_2$ where S_1 is the SVD variable set generated by statements C2 and C3 and S_2 is the SVD variable sets generated by statements C5, C6, and C7. Given the initial set $S = \{x\}$, statements C2 and C3 generate $S_1 = \{x, i\}$ and given the initial set $S = \{x\}$, statements C5, C6, and C7 generate $S_2 = \{x, k, m\}$. The set $S = S_1 \cup S_2 = \{x, i, k, m\}$ and $N = \{j\}$.

3.3 Iterative Statements

An iterative statement can induce indirect dependencies. In this case, as with conditional statements, any definition occurring within the body of the iterative statement represents a generation by this statement (of SVD variables). Iterative statements that do not fall into this category require a more complicated calculation to determine dependencies. Consider the following example:

On entry to statement D1 we have $S = \{x, y\}$ and $N = \{q, r, s, t, n\}$. For the first iteration of the loop, this is true until statement D7 is executed and generates q. On the second iteration of the loop, after the execution of statement D4 (and immediately preceding the execution of statement D7) we have $S = \{x, y, q, s, t\}$ and $N = \{r, n\}$. At this point, we have identified statement D4 as a TCP statement but have failed to identify statement D2 because t was generated by statement D6 in the second iteration of the loop. Statement D4 will be identified as a TCP statement in the third iteration of the loop. In order to identify all TCP statements, the loop will have to be scanned until the set of SVD variables remains constant over successive iterations.

```
shared int x, y;
int q, r, s, t, n;
⋮
D1: for r = 1 to n do
D2:      if t > n then
D3:           t = t - n;
D4:      if q > s then
D5:           s = s + t;
D6:           t = t + r;
D7:      q = (x * y) + (t - r);
D8: end
```

Figure 5: A program fragment containing a TCP iterative statement

4 Preprocessing

We use a control flow graph [2] of the program decorated with a single pointer to the convergence point for a TCP (if there is one in a block), and a flag denoting whether or not a block contains a TCP. To ensure that each block contains only a single TCP candidate, we include in the set of block delimiters, for the control flow graph, assignment statements containing array references. This suffices to ensure a single TCP candidate per block since all TCP candidates are conditional statements or other parallel program structuring statements (e.g., task_create). We call this control flow graph the *TCP control flow graph* for a parallel program.

We use an array, one element for each program variable, to maintain the set of SVD variables.

In the following sections, we use N, E, and V to describe the size of an input problem. Let N be the number of assignment, conditional, and iterative statements in the program, E be the number of edges in the modified program flow graph, and V be the number of variables in the program.

The program flow graph and SVD variable array requires $O(N \times V)$ space, and $O(N)$ time to construct.

5 An Algorithm for locating TCPs

The problem of computing the set of SVD variables for a program is similar to that of constant propagation [6]. Both problems can be solved using data flow analysis techniques. Wegman and Zadeck's version of Kildall's simple constant algorithm [4], determines the lifetime and extent of constant definitions in a program when simple constant propagation is considered. The algorithm can be modified to determine the SVD variable sets for a program. Wegman and Zadeck's algorithm is a worklist algorithm [3]. We will use this algorithm to illustrate the worklist algorithm technique.

5.1 Simple Constant

Both simple constant and the SVD variable set computation for programs represents the computation of a property of variables that vary over the program (e.g., a variable is constant or not constant, a variable is an SVD variable or not an SVD variable, etc.). The desired property of the variable is either killed, preserved, or generated by a statement. For example, in Figure 6, if the variable y is constant and variable z is not constant prior to executing statement B, then after execution, the variable y is no longer constant (i.e., killed by the statement) and the variable z remains not constant (i.e., preserved by the statement). Clearly, the property of individual variables reaching a statement can effect the resulting property of variables at termination of a statement. In the previous example, if z had been constant, then y would be constant at the termination of statement B. Iteration and branching complicates the calculation of a property by introducing the potential for conflicts between multiple possible paths reaching a statement. Notice that in Figure 6, statement A has two predecessors due to the control flow of the program. If y is a constant and z is not a constant, then for the first iteration of the loop, x is a constant (using simple constant propagation) at the termination of statement A. However for subsequent iterations, y is not a constant (due to statement B), and x is not a constant at the termination of statement A.

In simple constant, and for the SVD variable computation, a lattice of possible states for a variable is used to resolve conflicts in property. Conflicts may arise when a statement has multiple predecessors for which the property of a variable (or state) is different on termination. Recall that a lattice is a partially ordered set containing a greatest element $\top$ and a least element $\bot$. The elements of a lattice correspond to states. For simple constant, the elements of the lattice correspond to the three possible states of interest for a variable, those of *no information* ($\top$), *constant*, and *not constant* ($\bot$). In reality, each possible constant value (of which there are many) must represent a different state, because if for a particular variable, two paths leading to a statement are constant but have different values, then the variable is not constant prior to the execution of the statement. Figure 7 has been simplified to include only the integer constants. The state lattice is equipped with an operation $\sqcup$ that serves to resolve conflicts between states in the lattice. The rules for $\sqcup$ are given in Figure 7. Rule 1 indicates that if on two paths leading to a statement, the state of a variable is unknown and *any* state, then the conflict is resolved by assigning the state of the variable on entry to the statement to *any* (e.g., any information about the state of a variable overrides no information). Returning to Figure 6, for the first iteration of the loop, y is an unknown state for the first path leading to statement A and is a constant for the second, therefore, it is a constant

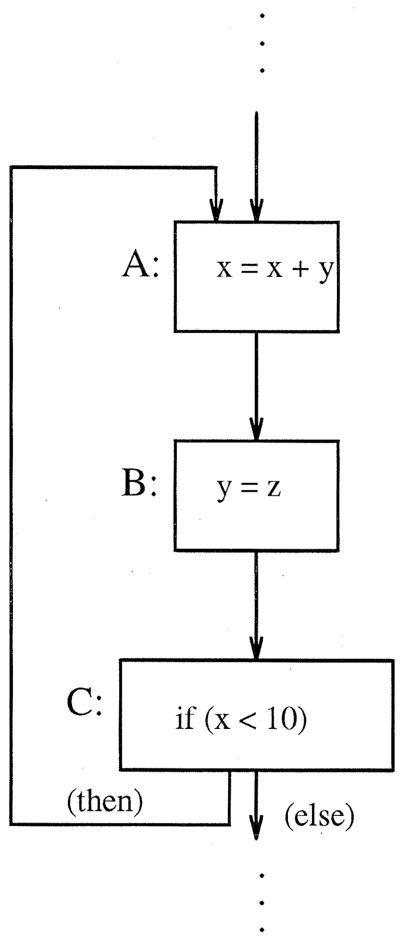
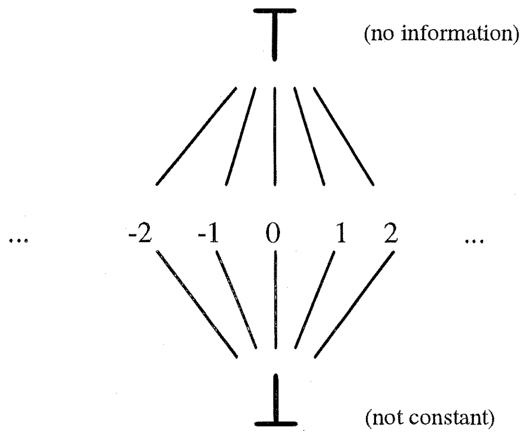


Figure 6: An example control flow graph



(1) $\text{any} \sqcup \top = \text{any}$

(2) $\text{Const} \sqcup \text{Const} = \text{Const}$

(3) $\text{any} \sqcup \bot = \bot$

Figure 7: The lattice and associated rules for simple constant

on entry to statement A. Rule 2 indicates that if two paths have the same constant state for a variable, then the variable remains a constant. Finally, rule 3 indicates that if on any path leading to a statement, a variable has been determined to be not constant ($\perp$), the the variable is not constant on entry to the statement.

A statement level control flow graph of the program suffices to identify statements and the possible paths leading to them. Each node in the control flow graph has two arrays representing the state of each variable in the program on entry to the node and on exit. A worklist algorithm iterates over the nodes of the control flow graph in order to determine the desired property for variables at each node in the graph. The algorithm begins by adding the root node for the control flow graph to the worklist and proceeds by removing nodes from the worklist and processing the nodes until the worklist is empty. Initially, the state of all variables at the entry and exit of each node are $\top$ (no information). When node is removed from the worklist the killed, generated, and preserved definitions are computed for the node based on the state of incoming variables to the node. If at any time, the state of a variable is changed by the killed or generated definitions of a node, all nodes that are immediate successors of the current node, in the control flow graph, are added to the worklist. These nodes need to (re)processed to reflect changing variable states.

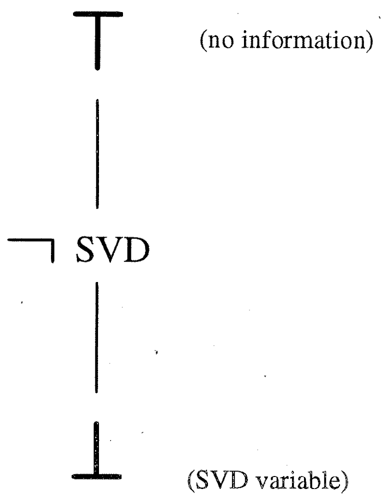
5.2 The SVD Variable Problem

Notice that simple constant requires an intersection over the predecessors of a statement[3]. This is illustrated in Figure 7 rules 2 and 3, where all of the predecessor states of a variable must be the same constant value in order for a variable to be a constant on entry to a statement and if in any predecessor state a variable is $\perp$ (i.e., not constant), then the variable state is $\perp$ (rule 3) on entry to a statement. In contrast, the SVD variable set computation requires the union over the predecessors of a statement[4]. Here, a variable is an SVD variable if it was an SVD variable for some predecessor. The algorithm described below is identical to Wegman and Zadeck's except that the lattice under consideration has been redefined for the SVD variable computation.

For SVD variable set computations, each statement can affect the set of SVD variables, however only those changes to the set of SVD variables immediately preceding and immediately following a TCP candidate are of interest. We use the modified program flow graph of the previous section to represent a program as opposed to a statement level control flow graph. Once a statement has been determined to be a TCP statement, it is added to the set of TCP statements for the program and can never be removed. Two arrays of size V are associated with each block – one for the block entry state of the SVD variable set and one for the block exit state.

[3]Simple constant is a top down intersection data flow problem.

[4]The SVD variable problem is a top down union data flow problem



$$(1) \quad \text{any} \sqcup \top = \text{any}$$

$$(2) \quad \neg\,\text{SVD} \sqcup \neg\,\text{SVD} = \neg\,\text{SVD}$$

$$(3) \quad \text{any} \sqcup \bot = \bot$$

Figure 8: The lattice for the SVD variable problem and associated rules

The algorithm uses a lattice containing three states and propagates values in a program flow graph. The lattice for SVD variables is shown in Figure 8. Notice that we are only interested in whether or not a variable is an SVD variable (i.e., the value of the variable is not of interest). This simplifies the lattice. Figure 8 includes the $\sqcup$ rules for the lattice. The highest element, $\top$ represents an undetermined state for a variable. That is, if a variable has a value of $\top$, then it is not known whether or not the variable is in the SVD variable set. The lowest element, $\bot$, represent the set of SVD variables. The middle element $\neg SVD$ represents variables that are currently known not to be in the SVD variable set. Our intuition about the state of variables is the following: If we have no information about the state of a variable on some path leading to a node, then information on other paths overrides this lack of information (Figure 8, rule 1). If a variable is not SVD on all paths leading to a node, then it is not SVD for the node (Figure 8, rule 2). If a variable is found to be SVD on some path leading to a node, then it is SVD for that node (Figure 8, rule 3).

The algorithm begins by adding the start node of the graph to the worklist of nodes to be processed. Initially, the value of every variable is $\top$ with the exception of the shared variables which have a value of $\bot$. Nodes are removed from the worklist and processed until the list is empty. The entry state values for each node are the $\sqcup$ of the exit state values from all predecessor nodes. The variable *flag* is used to denote when a TCP conditional (or iterative) statement has been encountered. The algorithm processes statements as outlined in Section 3. The assignment statements in the node are evaluated by taking the $\sqcup$ of the state values for all variables on the right hand side to be the state value for the variable on the left hand side. As each statement in a node is processed, candidate statements are checked against the current state to determine whether or not they are TCP statements. Once a statement has been determined to be a TCP statement, it cannot be removed from the set of TCP statements. When a TCP conditional or iterative statement is encountered, a flag is set signifying that each node encountered between the node and its continuation must be processed such that any variable appearing on the left hand side of an assignment statement is an SVD variable. If during processing of a node, the exit state does not match the entry state (for some variable(s)), then each successor to the node must be added to the worklist for processing. If the value of the entry state does not change for a node, then it need not be processed. Similarly, if the exit state for a node does not change, then no successor nodes are added to the worklist.

5.3 Complexity and Correctness

In simple constant, the state of any variable can only change twice (i.e., the lattice value for the variable can only be lowered twice per block). The same is true for an SVD calculation. Given that the number of predecessors for a node is I, each node can only appear on the work list a maximum of $2 \times V \times I$ which is $O(E \times V)$ for each node. At each node, $C \times V$ meet operations are performed, where C is a constant describing the number of statements in a block[5]. The worst case time for the algorithm is $O(N \times E \times V^2)$ and the space requirement is $O(N \times V)$. The lower bound for runtime for the algorithm is $O(N \times E \times V)$ in the case that each node appear on the work list a single time.

In [6], Wegman and Zadeck present formulations of the constant propagation problem with asymptotically better worst case run times. It is unknown if these methods can be applied to the SVD variable problem.

6 An Asymptotically Faster Approximation

The problem of constant propagation require more than a simple determination of whether or not a variable is a constant at a particular point in the program. In addition, the value of the constant is of interest. The SVD variable set calculation does not require detailed information about values being computed. Rather, the only information of interest is whether or not a variable is a member of the SVD variable set. This difference makes it possible to simplify the SVD calculation for a program by reordering program statements. This reordering can produce a program that computes incorrect values, with the same statements contained in each block. For example, it is possible to reorder statements within an iterative statement but it is not possible to move statements into or out of an iterative construct. Returning to the example of Figure 5, notice that if it were possible to reorder the statements of the iterative statement body, then a single pass would suffice to collect SVD variables. This implies that a variable has a single state within the entire loop – that is, it is

[5]The number of statements in a block must either be a constant when compared to N, or the program flow graph can consist of a single statement per block.

either an SVD variable or it is not. We know that this is not guaranteed to be the case and this assumption may result in a potentially larger set of TCP statements than the previous method. However, we are guaranteed that the set of TCP statements located contains the set of actual TCP statements. If a variable is generated by any statement within the body of the iterative statement, then it is generated by the iterative statement. If a variable is killed by a statement within the body of the iterative statement and is not generated by any other statement within the body of the iterative statement, then it is killed by the iterative statement.

6.1 The SVD Graph

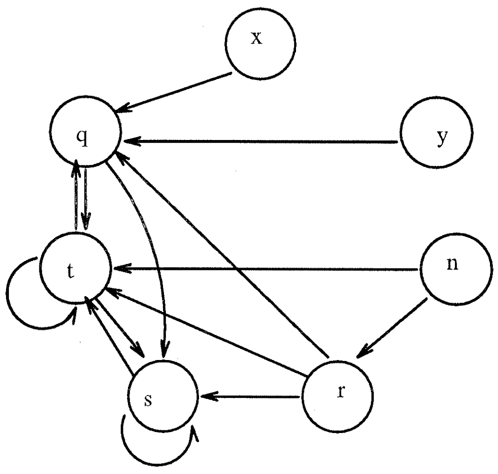


Figure 9: SVD graph of program fragment in Figure 8.5

Rather than reorder the statements in an iterative construct, the generated and killed dependencies for the body of the construct can be represented using a *shared variable dependency graph* (SVD graph). Figure 9 is the SVD graph for the iterative statement in Figure 5. Notice that q is dependent on x and y (statement D7), and that t and s are dependent on q (statements D5 and D6). In the SVD graph, a directed edge from node a to node b exists if variable b depends on variable a at some time in the body of the iterative statement. In Figure 9, variable q depends on variables x, y, t, and r. The SVD graph represents the dependencies for the entire loop as opposed to individual statements of the loop. Dependencies can be placed in the graph during a single traversal of a section of code. Once an SVD graph has been constructed for a section of code, the graph is topologically sorted from each node known to represent an SVD variable. The topological sorting produces the set of SVD variables generated by the code. Similarly, a topological sort can be used to determine the set of SVD variables killed by the section of code. For nested iterative statements, a single shared variable dependency graph is constructed[6].

given: A section of a parallel program p and a input dependence set V.
compute: The shared variable dependence graph for p.

algorithm:

```
1  Compute_svd_graph($p$, $V$):
2  graph $G$ = EMPTY;
3  for each statement $s$ of $p$ do
4        case $s$ of
5              assignment : given $l = lhs(s)$,
6                           add_node($l$, $G$);
7                           for each $v \in rhs(s)$ do
                                  add_directed_edge(($v$,$l$), $G$);
8                           for each $v \in V$ do
                                  add_directed_edge(($v$,$l$), $G$);
9              conditional : given $V_s \in condition(s)$, $p_{s_{TRUE}}$ and $p_{s_{FALSE}}$,
10                          $G$ = union_graphs($G$, Compute_svd_graph($p_{s\_TRUE}$, $V \cup V_s$));
11                          $G$ = union_graphs($G$, Compute_svd_graph($p_{s_{FALSE}}$, $V \cup V_s$));
12             iterative :  given $v = index(s)$ and $p_s$,
13                          $G$ = union_graphs($G$, Compute_svd_graph($p_s$, $V \cup V_s$));
14       end case
15 end for
16 return($G$);
17 end.
```

Figure 10: An algorithm for computing the SVD graph for a parallel program

Figure 10 computes the SVD graph for a section of code given a set of variables on which the entire section is dependent. This set includes the index variables from any enclosing iterative statements. Function *Compute_SVD_graph* is initially applied to the body of an iterative statement. Initially, V contains only the loop indexing variable. For each assignment statement, an edge is added to the graph from each variable appearing on the right hand side of the assignment to the variable appearing on the left hand side (line 7). In addition, the variable on the left hand side is dependent on each variable in V and an edge from each variable in V to this variable is added to the graph (line 8). For a conditional statement, *Compute_SVD_graph* is called recursively on the body of each branch of the conditional statement with the variables appearing in the condition added to the dependent variable set. The results of each of these are added to the existing graph (see lines 9-11). For an iterative statement, *Compute_SVD_graph* is called recursively on the body of

[6]For some programs, this method can approximate the set of program TCPs poorly. An example of the type of program for which this might happen is a program that consists almost entirely of a single iterative statement.

the iterative statement with the index variable added to the dependent variable set. The resulting graph is added to the existing graph (lines 12-13).

We use this representation to approximate the TCP statements within an iterative statement. Each iterative statement in a program is preprocessed to yield its associated shared variable dependence graph. During the process of approximating the TCP statements of a program, the shared variable dependence graph coupled with the current set of shared variable dependent variables is used to produce the set of shared variable dependent variables for the iterative statement. This is accomplished by computing a topological sort of the graph starting at each vertex that represents a known shared dependent variable. Consider again the shared variable dependence graph in Figure 9. If the set of SVD variables at the beginning of the loop (line D1) is $S = \{x, y\}$ then a topological sort of the graph starting at the variable x yields the tree highlighted in Figure 11. Each of the variables in this tree is added to the shared dependent variable set and deleted from the graph[7]. Figure 11 also shows the graph resulting from this deletion. A topological sort of the graph beginning with y, yields no new shared dependent variables. Finally, n and t are not determined to be shared dependent by any topological sort on known shared dependent variables. Once the set of shared dependent variables for a loop has been determined, the set can be used to approximate the set of TCP statements in the body of a loop.

6.2 The Algorithm

Notice that the SVD graph simplification alleviates the need for a per node copy of the entry and exit states for variables. This is because each node will now be traversed a single time allowing a single array to record the current SVD variable set.

Figure 12 approximates the set of TCP statements of a program. Starting at the top of the program flow graph p (the program begin statement) function *Approximate_TCPs* traverses the entire graph calculating the current state of the program variable (i.e., whether or not they are shared variable dependent). Each statement in the current block (line 4) is used to update the *state* of the program variables. If the statement is an assignment statement (line 6) then if *flag* is set (signifying that this statement is enclosed by a conditional or iterative TCP statement) the left hand side of the assignment is automatically added to the current state as a shared variable dependent variable. Otherwise function *Do_Definition* determines whether or not the left hand side of the assignment represents a generated shared variable dependency (line 24) or a kill of one (line 25). If the statement is an array assignment statement then it may or may not be a TCP statement. Hence the statement needs to be checked as to whether or not it is a TCP statement (line 7) as well as determining whether or not the left hand side of the assignment is a shared

[7]Each variable need only be added to the shared dependent variable set a single time.

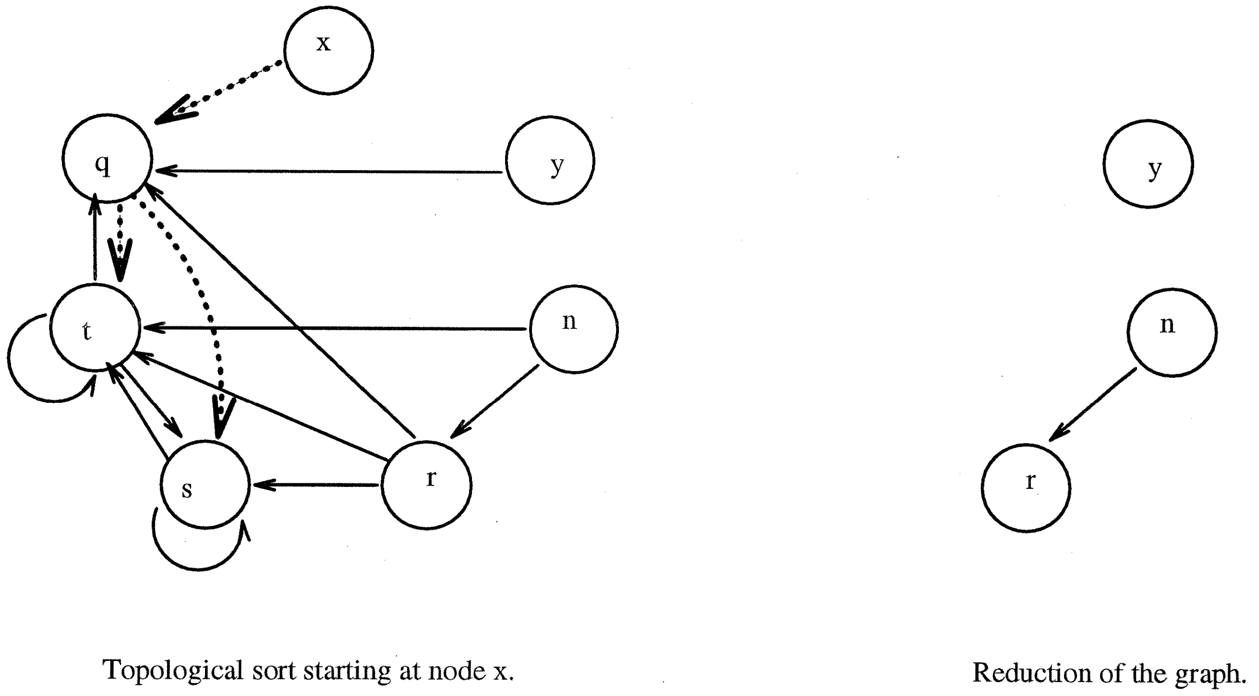


Figure 11: Reduction of the graph in Figure 8.9

variable dependent variable (line 8). If the statement is a conditional statement, it is first checked to determine whether or not it is a TCP statement (line 9). If this is the case, then *flag* is set signifying that all variables appearing on the left hand side of assignments within the body of the conditional statement are automatically placed in the shared variable dependent set. Whether or not the statement is a TCP statement, *Approximate_TCPs* is called recursively on each branch of the conditional (lines 10-11), given the state before the conditional, and the two resulting states are merged (line 12) a TCP statement. If the statement is an iterative statement, whether or not the statement is a TCP statement must be checked (line 13) and if it is, then *flag* is set. Function *do_SVD_graph*, see Figure 13, is used to determine the change in state due to the loop being executed and to determine the TCP statements present in the loop given the input state *state* (line 14). The current state and the state calculated during the loop are merged at the end of the loop (line 15). Finally, if the statement is a *task_create* statement then each thread must be evaluated using *Approximate_TCPs*.

The algorithm of Figure 13 uses the shared variable dependence graph for an iterative statement to determine the correct set of shared variable dependent variables for the loop. This set is subsequently used to approximate the TCP statements within the loop body. At line 3, if *flag* has been set, then a simple topological sort (line 4) of the graph beginning at the index variable for the loop, then all variables that appear on the left hand side of an assignment in the body of the

given: a parallel program flow graph p, a state *state*, and a conditional flag *flag*.
compute: approximate the set TCP of TCPs of p.
TCP is a set containing known TCP statements, is global and initially empty. *state* is an array of flags, one element for each program variable.
algorithm:

```
1  Approximate_TCPs(p, state, flag): returning state;
2  current_block = p;
3  while current_block != NULL do
4      for each s in current_block do
5          case s of
6              assignment :   if flag then state = add_to_state(lhs(s))
                              else state = do_definition(s,state,flag);
7              array_assn :   if isTCP(s,state) then $TCP = TCP \cup s$;
8                             if flag then state = add_to_state(lhs(s))
                              else state = do_definition(s,state,flag);
9              conditional :  if isTCP(s,state) then
                                  $TCP = TCP \cup s$;
                                  flag = TRUE;
10                            lstate = Approximate_TCPs(copy(left_branch), state, flag);
11                            rstate = Approximate_TCPs(copy(right_branch), state, flag);
12                            $state = state \cup rstate \cup lstate$;
13             iterative :    if isTCP(s,state) then
                                  $TCP = TCP \cup s$;
                                  flag = TRUE;
14                            loop_state = do_svd_graph(copy(loop_body), state, flag);
15                            $state = state \cup loop\_state$;
16             task_create :  for each thread in s do
17                                Approximate_TCPs(copy(thread), shared(state)), flag);
18             otherwise :    do_nothing;
           end
       end
19     current_block = current_block $\rightarrow$ continuation;
   end
20 return state;
   end.

21 do_definition(s,state, flag): returning state;
22 if flag then return(add_to_state(lhs(s)));
23 for each $v \in rhs(s)$;
24     if $v \in state$ then return(add_to_state(lhs(s));
25 return(remove_from_state(lhs(s)));
26 end.
```

Figure 12: An algorithm for approximating the set of TCP statements for a program

given: a parallel program flow graph p, a state *state*, and a flag *flag*.
compute: approximate the set TCP of TCPs of p.
TCP is a set containing known TCP statements and is global. *state* is an array of flags, one element for each program variable
algorithm:

```
1  do_svdgraph($p$, $state$, $flag$): returning ($state$);
2  $new\_state$ = create_state();
3  if $flag$ then
4       $ttree$ = topological_sort_with_delete($G$, index_variable($p$);
5       $new\_state$ = add_to_state(nodes($ttree$));
   else
6       for each dependent variable $v$ in $state \cup new\_state$ do
7            if $v \in G$ then $ttree$ = topological_sort_with_delete($G$,$v$);
8            $new\_state$ = add_to_state(nodes($ttree$));
9       for each non-dependent variable $v$ in $state \cup new\_state$ do
10           if $v \in G$ then $ttree$ = topological_sort_with_delete($G$,$v$);
11           $state$ = remove_from_state(nodes($ttree$));
12 $TCP$ = do_code($p$, $state \cup new\_state$);
13 return $state \cup new\_state$;
   end.

14 do_code($p$, $state$): returning $TCP$;
   $TCP$ = EMPTY;
15 $current\_block = p$;
16 while $current\_block$ != NULL do
17      for each $s$ in $current\_block$ do
18           case $s$ of
19                array_assn :  if isTCP($s$,$state$) then $TCP = TCP \cup s$;
20                conditional : if isTCP($s$,$state$) then $TCP = TCP \cup s$;
21                iterative :   if isTCP($s$,$state$) then $TCP = TCP \cup s$;
22                otherwise :   do_nothing;
           end
      end
23 $current\_block = current\_block \rightarrow continuation$;
   end
24 return $TCP$;
   end.
```

Figure 13: An algorithm for approximating the set of TCP statements given an SVD graph

loop will add to the current set of SVD variables (line 5). If *flag* is true, then this loop is either enclosed in an outer conditional statement which is a TCP statement or is itself a TCP statement. If this is not the case then *flag* is false and the set of SVD variables generated by the loop body is calculated in lines 6-8. For each variable in the current SVD variable set, the SVD graph is topologically sorted and the resulting tree of nodes are removed from the SVD graph (lines 6-7). This continues until no nodes in the SVD variable set remain in the SVD graph. The set of SVD variables killed by the loop body is determined in lines 9-11. For each variable that is not an SVD variable the remaining graph is topologically sorted and the resulting tree of nodes removed from the graph. These nodes are killed by the loop body and are removed from the SVD variable set at line 11. Line 12 uses function *do_code* to locate the TCP statements with the loop body given the set of SVD variables that has just been calculated.

6.3 Complexity and Correctness

The algorithm of Figure 10 requires a single pass over each program loop with the possibility of V operations at each statement. The worst case runtime for the algorithm is $O(N \times V)$ and the space required is $O(N + V)$. The algorithm of Figure 13 requires a topological sort of the SVD graph costing $O(V)$ since a variable is only located a single time before it is removed from the SVD graph. The total time for the algorithm of Figure 13 is $(V + n)$, where n is the size of a loop body. The algorithm of Figure 12 traverses each node in the flow graph a single time with the possibility of V operations at each node. The worst case runtime for this algorithm is $O(N \times V)$ since $N \times V > N + V$, the runtime for the algorithm of Figure 12 . The algorithm requires $O(N + V)$ space.

7 Procedures and Functions

Procedures and functions complicate the TCP location problem. If a procedure or function can side effect the state of a program, then it has the possibility of affecting the SVD variable set immediately after execution of the procedure or function. For example, a function invocation on the right hand side of an assignment can change the SVD state of variables creating a situation where the left hand side of the assignment is either generated or killed by the assignment based on side effects to the SVD variable state as a result of executing the function. If procedures and functions are not side effect free then they must be treated as inline sections of code at each potential invocation. For the worklist version of SVD variable approximation, the increase in complexity due to inline analysis of procedures and functions is $O(N \times E \times V \times P)$ where P is the total number of procedure and function invocations in the source program.

If procedures and functions are side effect free, the SVD graph representation of a function of procedure can be used to approximate the set of SVD variables introduced by the function or procedure body. This approximation requires a single pass over the body of each function and procedure. During the collection of SVD variables for the main program body, the set of SVD variables at *each* possible invocation of a procedure or function can be collected. That is, the set of SVD variables at the entry to a procedure or function is the union of the sets of SVD variables at each possible invocation of the function or procedure. A single application of the topological sorting algorithm at the end processing the main program, will suffice to approximate the set of TCPs in a procedure or function body. This simplification does not increase the complexity of either SVD variable approximation algorithm.

References

[1] Zulah K. F. Eckert. Trace extrapolation for parallel programs on shared memory multiprocessors. In *Doctoral Thesis*. University of Colorado, 1995.

[2] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[3] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, Inc., New York, NY, 1977.

[4] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the First ACM Symposium on Principles of Programming Languages*, pages 194–206, October 1993.

[5] D. A. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications ACM*, 29(12):1184–1201, December 1986.

[6] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(3):319–349, July 1991.