A New Data Structure and Algorithm for Static Slicing Concurrent Programs

Huang Weiping and Xiao Jianyu
1Department of Computer, Shaoyang University, Shaoyang 710049, China
2Department of Computer, Wuhan University, Wuhan 430072, China

Abstract: Internal representation and static slicing algorithm of concurrent programs are studied. Based on the comparison among existed slicing algorithms and analysis of the fact that Krinke's algorithm produced imprecise program slice for the program structure which has loops nested with one or more threads, a conclusion is drawn that the reason for the impreciseness is that Krinke's data structure-threaded program dependence graph had very coarse definitions of data dependence relations between threads and the constraint put on the execution path in concurrent program is unduly loose. An improved threaded program dependence graph is proposed which adds a new dependence relation of loop-carried data dependence crossing thread boundaries. An improved slicing algorithm is also proposed which introduces a new concept of regioned execution witness to further constrain the execution path. The pseudo code of algorithm adding loop-carried data dependence relations crossing thread boundaries is given. The pseudo code of the new slicing algorithm is also given whose complexity has been analyzed. Examples shows that the improved slicing algorithm designed on the improved data structure can restrain the impreciseness of Krinke's.

Key words: Static slicing of programs, concurrent programs, program dependence graph, loop-carried data dependence, regioned execution witness

INTRODUCTION

Program slicing is based on the deletion of statements that preserve the original behavior of the program with respect to a slicing criterion which is a pair < p, x >, where p is a program point and x is a program variable. Program slicing is an important basis of static analysis of programs and is extensively used in program understanding and software maintenance (Binkley and Harman, 2004). Program slicing includes static slicing and dynamic slicing (Binkley and Harman, 2004). Static slicing technique for sequential program has been mature with over 20 years' development and the mainstream method is based on graphic reachability algorithm with Program Dependence Graph (PDG) as program's internal representation data structure. PDG is constructed basing on Control Flow Graph (CFG) with control flow edges deleted and data dependence and control dependence edges added. Data dependence includes flow dependence and def-order dependence and flow dependence includes loop independent dependence and loop-carried dependence. According to Horwitz et al. (1988), these dependence relations are adequate to express relations between statements of sequential program.

Concurrent program slicing now attracts more and more attention as concurrent systems were increasingly adopted. The execution order of statements in concurrent program is undetermined and dependence relations between statements include choice dependence, synchronization dependence, communication dependence and interference dependence etc. besides the conventional control and data dependence. So, static slicing concurrent program is very complex and the conventional slicing algorithm of sequential program cannot be adopted which is based on the assumption that the execution order of statements is determined. The first method for static slicing concurrent program was proposed by Cheng (1993) which was followed by Zhao (1999) and Chen et al. (2000). The principle of their method was to construct an extended PDG data structure as concurrent program's internal representation (such as Cheng's Process Dependence Net (Cheng, 1993), Zhao's Multi-threaded Dependence Graph (Zhao, 1999) and Chen's Concurrent Program Dependence Graph (Chen et al., 2000)) and use the graphic reachability algorithm. The problem of this method is that many dependence relations of concurrent program have no nature of transitivity which is the precondition of graphic reachability algorithm. The aftereffect of the problem is impreciseness of program slice. Krinke (2003, 1998) pointed out Cheng's problem and proposed a new static slicing algorithm to solve it which was based on a new data structure-threaded Program Dependence Graph (tPDG) and a new concept-threaded execution witness to

Corresponding Author: Xiao Jianyu, Department of Computer, Wuhan University, Wuhan 430072, China
constrain the execution path of program. We find that Krinko’s algorithm can not properly handle the data dependence in program structure with one or more threads nested in a loop and will produce imprecise program slice also. We concludes that the reason for the impreciseness is that tPDG has over coarse definitions of data dependence between statements and the slicing algorithm puts un Jordy loose constraint on the execution path in concurrent program.

This research is an extension of Krinko’s work which proposes a new strategy of static slicing concurrent program with improvement of program’s internal representation data structure and slicing algorithm. The improvement of data structure is to add a new dependence of loop-carried data dependence crossing thread’s boundary. The improvement of slicing algorithm is to introduce a new concept of region execution witness to further constrain the execution path in concurrent program. Examples shows that the improved slicing algorithm designed on the improved program dependence graph can restrain the impreciseness of Krinko’s.

**TERMS RELATED TO STATIC SLICING CONCURRENT PROGRAMS**

Here, terms related to static slicing concurrent program will be formally defined for easy description of the proposed data structure and slicing algorithm. The assumed model of concurrent program includes a start point START and an end point EXIT and has many concurrently executable threads which are synchronized through statements cobegin/coend. \( \Theta = \{\theta_0, \theta_1, \ldots, \theta_n\} \) is assumed to be set of threads in program with \( \theta_i \) being the main thread. Function \( \Theta(p) \) is assumed to return identity of the innermost thread including statement p. Function \( \parallel \Theta_0 \Theta_1 \) is defined as \{true, if \( \Theta_0 \) and \( \Theta_1 \) may execute concurrently \}. false, else}.

**Threaded Control Flow Graph (tCFG)**

**Definition 1. Control Flow Graph (CFG):** CFG is a directed graph \( G = (N, E, s, e) \) where, \( N \) is the set of nodes and \( E \) is the set of edges. Statements and condition predicates are expressed as nodes \( n \in N \); control flow between two statements \( n_m, n_n \in N \) is expressed as an edge \( (n_m, n_n) \) (written \( n_m \rightarrow n_n \)). The two special nodes \( s \) and \( e \) represent program’s START and EXIT separately. The set of variables referenced at node \( n \) is noted as \( \text{ref}(n) \); The set of variables defined (assigned) at node \( n \) is noted as \( \text{def}(n) \).

**Definition 2. Path and reachability on CFG:** A path on CFG \( G \) is a sequence of nodes \( p = \langle n_0, n_1, \ldots, n_k \rangle \), where, \( n_i \rightarrow n_{i+1} \) (\( 1 \leq i < k \)). If there exists a path \( \langle p_0, p_1, \ldots, p_l \rangle \) in \( G \), we say \( q \) to \( p \) is reachable (written \( q \rightarrow p \)).

**Definition 3. Execution witness:** A node sequence \( \langle n_1, \ldots, n_k \rangle \) is said to be an execution witness on CFG \( G \) if there exists \( n_i \rightarrow \ast n_{i+1} \) (\( 1 \leq i < k \)).

**Definition 4. Threaded Control Flow Graph (tCFG):** tCFG \( \langle N, E, s, e, \text{cobegin}, \text{coend} \rangle \) is an extension of CFG, where, \( \text{cobegin} \in N \) represent cobegin and coend statements separately.

**Definition 5. Threaded execution witness:** A sequence of nodes \( I = \langle n_1, \ldots, n_k \rangle \) is a threaded execution witness on tCFG if \( \forall (i) \in I, i_{n_i} \rightarrow m_i \Rightarrow \forall (j) \in I: m_{n_j} \rightarrow \ast \Rightarrow m_{n_k} \) where, \( \delta_i \) is a sub-sequence satisfying \( \delta_{\sum_j t \leq 1} \), \( \ast \rightarrow \ast \) says reachable through sequential control flow edges and \( \ast \rightarrow \ast \) says reachable through concurrent control flow edges.

**Threaded Program Dependence Graph (tPDG)**

**Definition 6. data dependence (dd):** In CFG \( G \), node \( j \) is said to be data dependent on node \( i \) (written \( i \rightarrow_{dd} j \)) iff: \( \circ \) there is a path \( P \) from \( i \) to \( j \) (i.e., \( i \rightarrow j \)); \( \bullet \) there exists a variable \( v \) satisfying \( v \in \text{def}(i) \) and \( v \in \text{ref}(j) \); \( \oplus \) for each node \( k \neq i \rightarrow v \in \text{def}(k) \).

**Definition 7. Post-dominant node, Pre-dominant node:** In CFG \( G \), node \( j \) is said to be node \( i \)'s post-dominant node if all paths from \( i \) to EXIT must through \( j \); node \( i \) is said to be \( j \)'s Pre-dominant node if all paths from \( \text{START} \) to \( j \) must through \( i \).

**Definition 8. control dependence (cd):** In CFG \( G \), node \( j \) is said to be control dependent on node \( i \) (written \( i \rightarrow_{cd} j \)) if: \( \circ \) there is a path \( P \) from \( i \) to \( j \) (i.e., \( i \rightarrow j \)), \( \bullet \) \( j \) is the post-dominant node of all nodes except \( i \); \( \oplus \) \( j \) is not \( i \)'s post-dominant node.

**Definition 9. Program Dependence Graph (PDG):** PDG is a variant of CFG with two kinds of edges-control dependence and data dependence added.

**Definition 10. Transitive dependence:** In PDG \( G \), node \( j \) is transitively dependent on node \( i \) if: \( \circ \) there exists a path \( P = \langle i = n_1, \ldots, n_i = j \rangle \), where, each node \( n_k \), is control dependent or data dependent on \( n_k \); \( \bullet \) \( P \) is an execution path in the corresponding CFG.
In sequential program, control dependence, data dependence and their combination are transitive, i.e., \( x \xrightarrow{\text{ct}} y \land y \xrightarrow{\text{ct}} z \Rightarrow x \xrightarrow{\text{ct}} z \).

**Definition 11. Loop-carried dependence**: In PDG \( G \), node \( j \) is said to be loop-carried dependent on node \( i \) (written \( i \rightarrow^{\text{lc}} j \)) if: 1) \( i \) is data dependent on \( j \) (i.e., \( i \xrightarrow{\text{dt}} j \)); 2) \( i, j \), are nested in loop \( L \); 3) in the corresponding CFG, there exists a execution path \( P \) from \( i \) to \( j \) which includes a back edge pointing to \( L \)'s condition predicate.

**Definition 12. Loop-independent dependence**: In PDG \( G \), node \( j \) is said to be loop-independent dependent on node \( i \) (written \( i \rightarrow^{\text{li}} j \)) if: 1) \( i \) is data dependent on \( j \) (i.e., \( i \xrightarrow{\text{dt}} j \)); 2) in the corresponding CFG, there exists an execution path from \( i \) to \( j \) which has no back edge pointing to \( L \)'s condition predicate.

**Definition 13. Interference dependence (id)**: In PDG \( G \), node \( j \) is said to be interference dependent on node \( i \) (written \( i \rightarrow^{\text{id}} j \)) if: 1) \( \theta (i) = \theta (j) \) and \( \theta (i) \) may executes concurrently with \( \theta (j) \); 2) there exists a variable \( v \) satisfying \( v \in \text{def}(i) \land v \in \text{ref}(j) \).

Interference dependence is a kind of data dependence relation between threads which is not transitive.

**Definition 14. Threaded Program Dependence Graph (tPDG) (Krinke, 2003)**: tPDG is an extension of tCFG with control dependence, data dependence and interference dependence edges added.

**Definition 15. Slicing criterion**: A program's slicing criterion \(< p, x >\) (where, \( p \) is a statement and \( x \) is a variable) is represented as a node in threaded program dependence graph.

**ANALYSIS OF KRINKE'S SLICING ALGORITHM**

**Principle of Krinke's algorithm**: Krinke’s method is based on the data structure tPDG. It’s principle is: In tPDG \( G \), node \( p \) is assumed to be the slicing criterion, \( S_p(p) \) is assumed to be program slice with respect to \( p \), \( S_p(p) = (q, \tau \leftarrow n_1, n_2, \ldots, n_k) \), where, \( p \) is a threaded execution witness.

**The fact of Krinke’s impreciseness**: We found that Krinke’s algorithm produce imprecise program slice for the program structure which has loops nested with one or more threads. The tCFG showed in Fig. 1a has a loop nested with two threads. \( S_2 \) is assumed to be slicing criterion. According to Definition 5, \(< S_2, S_2, S_2, S_2> \) is a valid threaded execution witness on tCFG as \( S_2 \) can reach \( S_2 \).

---

**Fig. 1**: The concurrent program structure which has loops nested with threads
through loop predicate \( S_0 \). As shown in the Fig. 1, there exists \( S_1 \rightarrow S_0 \) and \( S_1 \rightarrow S_0 \), so \( S_0 \) should be included in the program slice \( S_0 \) (S) according to Krinke’s algorithm. But in fact, in program’s behavior, if \( S_0 \) reach \( S_1 \) through \( S_0 \), then the definition to e by \( S_0 \) should have been redefined by \( S_1 \) and the definition to d by \( S_1 \) is redefined by \( S_1 \). This means that \( S_0 \)’s definition to e would not affect c or d in \( S_0 \) and \( S_0 \) should not be in \( S_0 \) (S). \( S_0 \) is just the same as \( S_0 \). That is, the program slice computed by Krinke’s algorithm includes unrelated statements.

The reason for Krinke’s impreciseness: We think that the reason of Krinke’s impreciseness be that it improperly handle the loop-carried data dependence relation crossing thread’s boundary. Krinke didn’t realize that during the execution path constructed the loop-carried dependence, the included loop back edge’s execution meant the old s-instance of the thread has finished and the new created instance’s execution has overlaid some statements’ behavior of the old instance. So, the data dependence relation in the old instance of thread should be re-computed. But Krinke still used the invalid data dependence which produced impreciseness.

THE STRATEGY OF IMPROVED DATA STRUCTURE AND SLICING ALGORITHM

The improvement of data structure: Based on Krinke’s tPDG, we further refine the data dependence relation between threads. We introduce a new kind of relation of loop-carried data dependence crossing thread’s boundary. The improved concurrent program’s internal representation is named tPDG’.

Definition 16. Loop-carried (data) dependence between threads: In tPDG G, a node \( S_i \) in thread \( \theta_i \) is said to be loop-carried (data) dependent between threads on a node \( S_j \) in \( \theta_j \) if the corresponding tCFG satisfies: 1 \( \theta_i \) and \( \theta_j \) were nested in the same loop; 2 \( S_i \) interference dependent on \( S_j \); 3 the execution path constructing the interference dependence includes a loop back edge.

Definition 17. Loop-carried (data) dependence between instances of the same thread: In tPDG G, a node \( S_i \) in thread \( \theta \) is said to be loop-carried (data) dependent on a node \( S_j \) in \( \theta \) between instances of the same thread if the corresponding tCFG satisfies: 1 \( \theta \) is nested in a loop; 2 \( S_i \) data dependent on \( S_j \); 3 the path from \( S_i \) to \( S_j \) includes a loop back edge.

Definition 18. Loop-carried (data) dependence crossing thread’s boundary (dd): Loop-carried (data) dependence between threads and Loop-carried (data) dependence between instances of the same thread are called by a joint name — loop-carried (data) dependence crossing thread’s boundary (written \( \rightarrow \)).

Definition 19. Improved threaded program dependence graph tPDG’: tPDG’ is based on tPDG with loop-carried data dependence edges being changed to be ordinary data dependence edges and loop-carried (data) dependence crossing thread’s boundary being added.

The improvement of slicing algorithm: The root cause of the impreciseness of Cheng’s algorithm (Cheng, 1993) is that some sequences of nodes in paths which reach slicing criterion through all kinds of dependence edges do not obey the constraint upon concurrent program’s execution behavior. Krinke (2003) improved Cheng’s algorithm by introducing a concept of threaded execution witness to constrain the qualification of execution paths. We considered Krinke’s algorithm still put unduly loose constraint on execution paths. Our improved slicing algorithm built on the new data structure tPDG’ introduce another new concept of regioned execution witness to further constrain the valid execution paths in TCFG.

Definition 20. Region: Region \( R \) is a sub-graph of tCFG G. A node \( m \) in \( R \) is said to be In node if there exists an edge (v, m) in G where, v is not in \( R \); A node \( n \) in \( R \) is said to be Out node if there exists as edge (n, v) in G where, v is not in \( R \). \( R \) is said to be a Single-In-Single-Out (SISO) region if there is only a pair of In and Out nodes in \( R \).

Definition 21. Regioned execution path: Regioned execution path in tCFG G is a sequence of nodes \( <n_1, ..., n_k> \) satisfying one of the following: 1 In G, \( R \) is assumed to be a SISO region representing a basic thread which has no nodes of cobegin/coend. \( n_i \) is assumed the In node of \( R \) and \( n_k \) is assumed the Out node of \( R \). A regioned execution path in \( R \) is a path from \( n_i \) to \( n_k \); 2 In G, \( R \) is assumed to be nested with one or more SISO regions. \( R' \) is assumed to be a SISO region between cobegin and the corresponding coend. A regioned execution path in \( R \) is constructed as a regioned execution path according to 1 with \( R' \) being looked as a single node \( N_i' \) and then \( N_i' \) is replaced by a regioned execution path in \( R_i' \); 3 In G, \( R \) is assumed to be an SISO region and is nested with a set of SISO regions, each of whose element \( R \), represents a basic thread. A regioned execution path in \( R \) is an arbitrary interleaving of regioned execution paths in \( R_i \).
Definition 22. Regioned execution witness: In tCFG G, a regioned execution witness is a sub-sequence of a regioned execution path in G.

The principle of our new slicing algorithm designed on tPDG' is: In tPDG' G, node p is assumed to be the slicing criterion, S_k (p) is the program slice with respect to p, S_k (p) = \{ q | P \leftarrow q \leftarrow n_1 \leftarrow \ldots \leftarrow n_k \}, q = n_1 \longrightarrow \ldots \longrightarrow n_k = p, d \in \{d_{dd}, d_{dd}, d_{dd} \}, where, p is a regioned execution witness in the corresponding tCFG).

For describing and analyzing the new slicing algorithm, a theorem is introduced as follows:

Theorem 1: A sequence of nodes <n_i, n_i, \ldots, n_j> in region R is a regioned executing witness if and only if the sequence's two arbitrary nodes n_i and n_j (i < j) satisfy that ||θ(n_i), θ(n_j)|| returns true or path (n_i, n_j, R) returns true, where, Function path (n_i, m, R) is defined as {true, if there exists a regioned execution path in R from node n_i to node m | false, else}.

Proof: It can be directly drawn from Definition 22 and 21.

Examples

Example 1: In Fig. 1a, S_k is assumed to be the slicing criterion. From the figure we can get that there exists S_i \longrightarrow S_k and S_k \longrightarrow S_j. But there does not exist regioned execution witness from S_i to S_j because the loop back is not included in the coregion of S_i and S_j with In node cobegin and Out node coend. According to our slicing algorithm, S_k cannot be added to program slice, which conforms to the analysis in 2.2 and has conquered the impreciseness of Krinke's.

Example 2: In Fig. 1b, S_k is assumed to be the slicing criterion. From the Fig. 1b we can get that there exists S_i \longrightarrow S_k and S_k \longrightarrow S_j. But there doesn't exist regioned execution witness from S_i to S_j in the coregion of S_i and S_j with In node cobegin and out node coend. According to our slicing algorithm, S_k cannot be added to program slice in this scene. But there exists S_i \longrightarrow S_k in the Fig. 1 and S_k will be included in program slice, which conforms to intuitive analysis. This means that the improved algorithm would not lose any related statements in program slice.

Examples analysis shows that the improved slicing algorithm designed on the new data structure tPDG' has restrained the impreciseness of Krinke's for the program structure with loops nested with threads.

**ALGORITHM FOR CONSTRUCTING THE IMPROVED DATA STRUCTURE**

The difference between the improved program dependence graph (tPDG') and Krinke's tPDG is that tPDG' further refines data dependence between threads and introduces loop-carried data dependence crossing thread's boundary. Here, the algorithm for adding loop-carried data dependence edges on tPDG will be given in pseudo code. The principle of this algorithm is: given tPDG of a concurrent program structure with loops nested with threads, for each loop body from the outermost one to the innermost one, all variable-defining nodes which can reach loop head node are first computed and recorded; then all variable-referencing nodes in the current loop which can be reached from loop-head node are computed; lastly, the corresponding loop-carried data dependence edges crossing thread's boundary can be found.
s: The source node of loop back edge
n: A node in tPDG
in (n): Values of all variables in R when program executes at the point just before node n
out (n): Values of all variables in R when program executes at the point just after node n
kill (n): The set of variables which are reassigned in n
def (n): The set of variables which are defined in n
pred (n): The immediate predecessor of node n which is not marked as useless in the output of sort_backedge_mark (R)
p \rightarrow n: The loop-carried data dependence edge crossing thread's boundary which is found in the outer loop body
p \rightarrow a: The loop-carried data dependence edge crossing thread's boundary which is found in the current loop body

INITIALIZATION: D = D \cap \text{out} (s)

BEGIN
FOREACH n ∈ sort_backedge_mark (R) DO
    IF n = P THEN in (n) = D
    ELSE IF n.type = coend THEN in(n) = \bigcap_{p \rightarrow n} \text{out} (p)
    ELSE in(n) = \bigcup_{p \rightarrow n} \text{out} (p)
    FOREACH p ∈ in(n) \cap (\text{def}(p) \cap \text{ref}(n) \neq \emptyset) DO
        IF \parallel (n, p) \parallel = \text{true}
            THEN IF p \rightarrow n does not exist, THEN create dependence edge p \rightarrow n
        ELSE
            IF p \rightarrow n does not exist, THEN delete edge p \rightarrow n and create dependence edge p \rightarrow n
    OD
    out (n) = in (n) \setminus \text{kill} (n)
    OD

Clear useless marks of all nodes.

END

The complexity of the algorithm is obviously O (n), where, n is the number of program's statements.

IMPROVED STATIC SLICING ALGORITHM

Pseudo code of the slicing algorithm: The improved slicing algorithm is an iterative one based on a work-list, which starts from the sling criterion and adds slice nodes successively. The algorithm takes an existed slice node as work node and traverses backwards along the data, loop-carried data, control and interference dependence edges. All nodes reached via only data or control dependence edges can be directly added to the slice nodes set as regioned execution witnesses for these nodes according to definitions of data and control dependence and their transitivity. When a node is reached via an interference edge, it is possible that a valid path on tPDG could have no valid regioned execution witness on the corresponding TCFG. The detail strategy to handle interference edge will be clear to you after you read the following pseudo code. Nodes reached via loop-carried data dependence edges crossing thread's boundary should be handled specially, which will be explained after pseudo code.

The pseudo code of the slicing algorithm follows.

INPUT: Slicing criterion p
OUTPUT: Program slice with respect to p which is a set of nodes in tPDG'

SUB-PROGRAMS DECLARATION:
\parallel (\theta_i, \theta_j) \parallel
input: Identities of threads \theta_i and \theta_j

output: If \theta_i can execute concurrently with \theta_j, then returns true, else false

coregion (i, j)
input: Statement node i, j
output: The smallest SISO region including i, j
path (i, j, R)
input: Statement node i, j
output: If there exists a regioned execution path in R from i to j returns true; else returns false
\( \theta (i) \)
input: Statement node i
output: Identity of the thread which i belongs to

VARIABLES DECLARATION:
N: Total number of threads in program
T[N]: Array for recording trace of nodes of threads which the algorithm accesses. In \( T = [t_1, t_2, \ldots, t_n] \), \( t_{seq} \) corresponds to the thread \( \theta (i) \), which records the algorithm’s last reached node in \( \theta (i) \) non-concurrently with \( \theta (i) \). \( \perp \) means the position has not been defined. This information is used to decide if there exists regioned execution witness.

c, c': Node triple \((x, T[N])\), where, x is a statement node and T [N] is the associated trace record array.
W: Work-list which is a set of node triples.
S: Program slice (a set of statement nodes named slice node.)
L: Loop body

INITIALIZATION: \( c = (p, \{t_1, \ldots, t_n\}) \), WHERE \( t_i = \{p, if||!(\theta (p), \theta = false, i, else)\} \);
\( w = \{c\} \); \( S = \{p\} \);

BEGIN
REPEAT
Get a triple c from w and let \( w = w - \{c\} \)
FORALL \( y \) \((y, x) \in dd \) DO
  FORALL \( i \in (0, N-1) \) \( (\theta (i), \theta (y)) \neq false \) DO \( T[i] = y \) OD
  \( c' = (y, T) \)
  IF \( c' \) has not been handled THEN mark \( c' \) as already handled and set \( w \) and \( S \) as \( w = w \cup \{c'\}, S = S \cup \{y\} \) OD
  FORALL \( y \) \((y, x) \in dd \) DO
    \( t = T[y, y] \)
    IF \( t = \perp \) path \((y, t, R)\) == true THEN
      FORALL \( i \in (0, N-1) \) \( (\theta (i), \theta (y)) \neq false \) DO \( T[i] = y \) OD
      \( c' = (y, T) \)
      IF \( c' \) has not been handled THEN mark \( c' \) as already handled and set \( w \) and \( S \) as \( w = w \cup \{c'\}, S = S \cup \{y\} \) OD
  OD
FORALL \( y \) \((y, x) \in dd \) DO
  FORALL \( i \in (0, N-1) \) \( (\theta i) \neq \perp \) DO \( T[i] \) - the source node of the backedge of L OD
  FORALL \( i \in (0, N-1) \) \( (\theta (i), \theta (y)) \neq false \) DO \( T[i] = y \) OD
  \( c' = (y, T) \)
  IF \( c' \) has not been handled THEN mark \( c' \) as already handled and set \( w \) and \( S \) as \( w = w \cup \{c'\}, S = S \cup \{y\} \) OD
UNTIL \( w \) is empty
END

This algorithm is different from Krinke’s in handling interference dependence edges and loop-carried data dependence edges crossing thread’s boundary.

We first investigate the handling of interference dependence edge. x is assumed to be the current work node which is already a slice node. As for a node q where, exists \( q \leftarrow x \), there is a valid path in tPDG’ from q to slicing criterion through x. But there may be no corresponding regioned execution witness in the TCFG and we may draw different conclusions for different scenes. x being a slice node and according to Theorem 1, there is a regioned execution witness from x to slicing criterion which is a sequence nodes named s whose direction is assumed to be from left to right. \( \Theta \) If there is no nodes in s which executes sequentially with q, that is, the position of \( \theta (q) \) in the trace record array associated
with x is ⊥. Then in the new sequence s' which is formed by adding q to the tail of s, q and any other node v satisfy that \( (\Theta(q), \Theta(v)) \) returns true. According to Theorem 1, s' is still a valid regioned execution witness and q should be considered as slice node. \( \Theta \) If there are nodes in s which execute sequentially with q, the tail node of the sub-sequence s" formed by the nodes which execute sequentially with q is recorded on the position (assumed to be t) of \( \Theta(q) \) in the trace record array associated with x. If there is a path from q to t, (i.e., coregion \( (x, q) \) returns true), then in the new sequence s" formed by adding q to the tail of s, q and any other node v in s" satisfy path \( (q, v, \text{coregion}(q, v)) \) being true and q and any node w not in s" satisfy \( (\Theta(q), \Theta(w)) \) being true. According to Theorem 1, s" is still a valid regioned execution witness and q should be considered as slice node; \( \Theta \) Under any other conditions, q is not slice node.

Let's investigate the handling of loop-carried data dependence edge crossing thread's boundary. x is assumed to be the current work node. As for a node q where, exists \( q \rightarrow x \), there is a valid regioned execution witness from q to x in the region of coregion \( (q, x) \) according to Definition 6, 17, 18. According to Theorem 1, q is a slice node. The execution path s from q to x which builds the loop-carried data dependence edge crossing thread's boundary includes a loop back edge, meaning that s first traverses the source of the loop back edge and then successively reaches the loop head and the thread-creating statement nodes. As the source of the loop back edge executes sequentially with threads in the loop body, the position of the loop's nested thread in the trace record array associated with x should be set as the source of the loop back edge.

Algorithm complexity: The main complexity of the slicing algorithm comes from interference dependence. A program is assumed to have totally N statements nodes and t threads. In the worst condition, these threads execute concurrently and every node in a thread is interference dependence on a node in another thread. According to the algorithm, the number of nodes directly adjoining slicing criterion through interference dependence edge is approximately \( \binom{N}{t} \), that is, in the first iteration of the algorithm there are nearly \( \binom{N}{t} \) nodes to be evaluated; in the second iteration, there are nearly \( \binom{N}{t} \times \binom{N}{t} \) nodes to be evaluated; ...; in the t-th iteration, there are nearly \( \binom{N}{t^t} \) nodes to be evaluated. The algorithm complexity is approximately \( \mathcal{O}(N^t) \), which is exponential to the number of threads.

CONCLUSION

Internal representation and static slicing algorithm of concurrent program are studied. According to (Muller-Olm and Seidl, 2001), slicing algorithm of program written in a concurrent language which has procedure and synchronization primitives is always un-optimal, because the reachability of statements in this kind of concurrent program is undecidable. So, there is always room for optimization for the existed slicing algorithm of concurrent program. In this study, based on the analysis of the impreciseness of Krinke's slicing algorithm, an improved threaded program dependence graph is proposed and the algorithm constructing the new data structure is given in pseudo code. Upon the new data structure, the improved slicing algorithm is also given in pseudo code whose complexity is analyzed. Examples shows that the improved slicing algorithm designed on the improved data structure can restrain the impreciseness of Krinke's. Due to the fact that the worst complexity of the slicing algorithm is exponential to the number of program, which is the same for Krinke's, the future work is to optimize the slicing algorithm.

REFERENCES


