http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 7 (2): 253-260, 2008
ISSN 1812-5638

A New Data Structure and Algorithm for Static Slicing Concurrent Programs

[1,2]Huang Weiping and [2]Xiao Jianyu
[1]Department of Computer, Shaoyang University, Shaoyang 710049, China
[2]Department 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 over 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 dada 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 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 Krinke'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 unduly loose constraint on the execution path in concurrent program.

This research is an extension of Krinke'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 regioned 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 Krinke'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. $\Theta = \{\theta_0, \theta_1, ...\theta_n\}$ is assumed to be set of threads in program with θ_0 being the main thread. Function θ (p) is assumed to return identity of the innermost thread including statement p. Function $\| (\theta_i, \theta_j)$ is defined as {true, if θ_i and θ_j 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 ∈ N; control flow between two statements m, n ∈ N is expressed as an edge (n, m) (written n→m). 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 ref (n); The set of variables defined (assigned) at node n is noted as def (n).

Definition 2. Path and reachability on CFG: A path on CFG G is a sequence of nodes $p = < n_1,....n_k>$, where, exists $n_i \rightarrow n_{i+1}$ $(1 \le i < k)$. If there exists a path <p,...,p> in G, we say q to p is reachable (written q → p).

Definition 3. Execution witness: A node sequence $<n_1,...n_k>$ is said to be an execution witness on CFG G iff there exists $n_i \rightarrow^* n_{i+1}$ $(1 \le i < k)$.

Definition 4. Threaded Control Flow Graph (tCFG): tCFG < N, E, s, e, cobegin, coend > is an extension of CFG, where, cobegin, coend ∈ N represent cobegin and coend statements separately.

Definition 5. Threaded execution witness: A sequence of nodes $l = < n_1,...n_k >$ is a threaded execution witness on tCFG iff, $\forall_{t\in\Theta} : l|_t = < m_1,...,m_j > \Rightarrow \forall_{i=1}^{j-1} : m_i \xrightarrow{cf,pf}{}^* m_{i+1}$ where, $l|_t$ is a sub-sequence satisfying $\theta(m_i) = t (1 \le i \le j); \xrightarrow{cf}{}^*$ says reachable through sequential control flow edges and $\xrightarrow{pf}{}^*$ 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 $i \xrightarrow{dd} j$) iff: ① there is a path P from i to j (i.e., $i \rightarrow^* j$); ② there exists a variable v satisfying v ∈ def (i) and v ∈ ref (j); ③ for each node $k \ne i \Rightarrow v \notin def(k)$.

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 START to j must through *l*.

Definition 8. control dependence (cd): In CFG G, node j is said to be control dependent on node i (written $i \xrightarrow{cd} j$), if: ① there is a path P from i to j (i.e., $i \rightarrow^* j$); ② j is the post-dominant node of all nodes except i; ③ 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: ① there exists a path $P = < i = n_1,...,n_l = j >$, where, each node n_{k+1} is control dependent or data dependent on n_k; ② P is an execution path in the corresponding CFG.

In sequential program, control dependence, data dependence and their combination are transitive, i.e., $x \xrightarrow[cd]{dd} y \wedge y \xrightarrow[cd]{dd} z \Rightarrow x \xrightarrow[cd]{dd} z$.

Definition 11. Loop-carried dependence: In PDG G, node j is said to be loop-carried dependent on node i (written $i \xrightarrow{lc(L)} j$) if: ① i is data dependent on j, (i.e., $i \xrightarrow{dd} j$); 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 $i \xrightarrow{li} j$) if: ① i is data dependent on j, (i.e., $i \xrightarrow{dd} j$); ② in the corresponding CFG, there exists a 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 $i \xrightarrow{id} j$) if: ① $\theta(i) \neq \theta(j)$ and $\theta(i)$ may executes concurrently with $\theta(j)$; ② there exists a variable v satisfying $v \in def(i) \wedge v \in ref(j)$.

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, $S_\theta(p)$ is assumed to be program slice with respect to p, $S_\theta(p) = \{q \mid P = < n_1, \ldots, n_k >, q = n_1 \xrightarrow{d_1} \ldots \xrightarrow{d_{k-1}} n_k = p, d_{1 \le i < k} \in \{cd, dd, id\}\}$, 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_4, S_6, S_2> is a valid hreaded execution witness on tCFG as S_4 can reach S_2

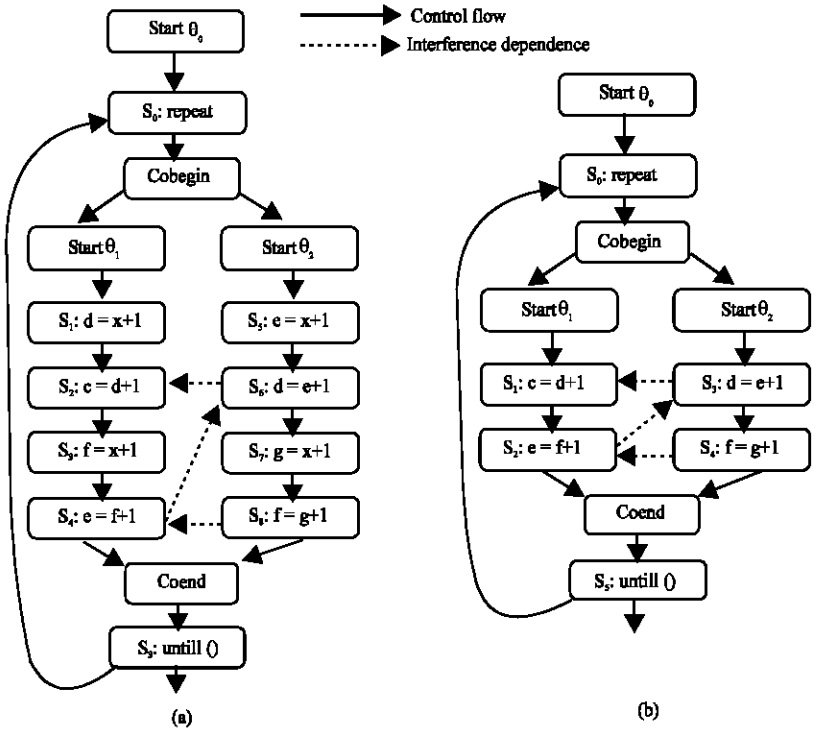

Fig. 1: The concurrent program structure which has loops nested with threads

through loop predicate S_0. As shown in the Fig. 1, there exists $S_4 \xrightarrow{id} S_6$ and $S_6 \xrightarrow{id} S_2$, so S_4 should be included in the program slice S_θ (S_2) according to Krinke's algorithm. But in fact, in program's behavior, if S_4 reach S_2 through S_0, then the definition to e by S_4 should have been redefined by S_5 and the definition to d by S_6 is redefined by S_1. This means that S_4's definition to e would not affect c or d in S_2 and S_4 should not be in S_θ (S_2). S_8 is just the same as S_4. 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 θ_i is said to be loop-carried (data) dependent between threads on a node S_j in θ_j if the corresponding tCFG satisfies: ① θ_i and θ_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 θ is said to be loop-carried (data) dependent on a node S_j in θ between instances of the same thread if the corresponding tCFG satisfies: ① θ is nested in a loop; ② S_i data dependent on S_j; ③ the path from S_j to S_i includes a loop back edge.

Definition 18. Loop-carried (data) dependence crossing thread's boundary (ddl): 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 $\xrightarrow{ddl}$).

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_s is assumed the In node of R and n_e is assumed the Out node of R. A regioned execution path in R is a path from n_s to n_e; ② In G, R is assumed to be nested with one or more SISO regions. R_i' 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_i' 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_i 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, $S'_\theta(p)$ is the program slice with respect to p, $S'_\theta(p) = \{q \mid P = < n_1,...,n_k >, q = n_1 \xrightarrow{d_1} ... \xrightarrow{d_{k-1}} n_k = p, d_{1\le i<k} \in \{cd,dd,id,ddl\}$, 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 $<n_1,n_2,...n_k>$ in region R is a regioned executing witness iff the sequence's two arbitrary nodes n_i and n_j (I < j) satisfy that $\| (\theta(n_i),\theta(n_j))$ returns true or path (n_i, n_j,R) returns true, where, Function path (n, m, R) is defined as {true, if there exists a regioned execution path in R from node n to node m | false, else}.

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

Examples

Example 1: In Fig. 1a, S_2 is assumed to be the slicing criterion. From the figure we can get that there exists $S_2 \xrightarrow{id} S_6$ and $S_6 \xrightarrow{id} S_4$. But there does not exist regioned execution witness from S_4 to S_2 because the loop back is not included in the coregion of S_4 and S_2 with In node cobegin and Out node coend. According to our slicing algorithm, S_4 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_1 is assumed to be the slicing criterion. From the Fig. 1b we can get that there exists $S_1 \xrightarrow{id} S_3$ and $S_3 \xrightarrow{id} S_2$. But there doesn't exist regioned execution witness from S_2 to S_1 in the coregion of S_3 and S_2 with In node cobegin and out node coend. According to our slicing algorithm, S_2 cannot be added to program slice in this scene. But there exists $S_2 \xrightarrow{ddl} S_3$ in the Fig. 1 and S_2 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 wiil 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.

The pseudo code of the algorithm follows:

INPUT: Krinke's threaded program dependence graph (tPDG)

OUTPUT: Improved threaded program dependence graph (tPDG') which has loop-carried data dependence edges crossing thread's boundary

SUB-PROGRAM DECLARATION:

sort_backedgmark (R)

input: The regioned R constructed by the current loop in tPDG

output: The sorted nodes of R in topological order computed by depth-first traversal algorithm where nodes in back edges are marked useless

$\| (\theta_i, \theta_j)$

input: Identities of threads θ_i and θ_j

output: If θ_i can execute concurrently with θ_j, then returns true; else false

VARIABLES DECLARATION:

R: The region constructed by current loop body

D: the set of variable-defining nodes in R

H: The loop head node of the current loop

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_backedgmark (R)
$p \xrightarrow{ddl'} n$: The loop-carried data dependence edge crossing thread's boundary which is found in the outer loop body
$p \xrightarrow{ddl} n$: The loop-carried data dependence edge crossing thread's boundary which is found in the current loop body

INITIALIZATION: D = D∩out (s)
BEGIN
FOREACH n ∈ sort _backedgemark (R) **DO**
IF n == H **THEN** in (n) = D
ELSE IF n.type = coend **THEN** $in(n) = \bigcap_{p \in pred(n)} out(p)$
ELSE $in(n) = \bigcup_{p \in pred(n)} out(p)$
FOREACH $p \in in(n) \wedge (def(p) \cap ref(n) \neq \varnothing)$ **DO**
IF ‖ (n, p) == true
THEN IF $p \xrightarrow{ddl'} n$ does not exist, **THEN** create dependence edge $p \xrightarrow{ddl} n$
ELSE
IF $p \xrightarrow{ddl'} n$ does not exist, **THEN** delete edge $p \xrightarrow{dd} n$ and create dependence edge $p \xrightarrow{ddl} n$
OD
out (n) = in (n)−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 exist 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:
‖ (θ_i, θ_j)
input: Identities of threads θ_i and θ_j

output: If θ_i can execute concurrently with θ_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

θ (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 $T=[t_0,t_1,...,t_n]$, $t_{0\le i\le n}$ corresponds to the thread θ (i), which records the algorithm's last reached node in θ (i) non-concurrently with θ (i). ⊥ 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: $c=(p,[t_0,...,t_n])$, WHERE $t_i=\{p,\text{if } \|(\theta(p),\theta_i=\text{false};\perp,\text{else}\}$;

w = {c}; S = {p};

BEGIN

REPEAT

Get a triple c from w and let w = w−{c}

FORALL $y\mid(y,x)\in dd\cup cd$ **DO**

FORALL $i\in(0..N)\wedge\|(\theta_i,\theta(y))=\text{false}$ $c'=(y,T)$ **DO** T [i] = y **OD**

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 $y\mid(y,x)\in id$ **DO**

$t=T[\theta(y)]$

$R=coregion(x,y)$

IF t == ⊥ || $path(y,t,R)==true$ **THEN**

FORALL $i\in(0..N-1)\wedge\|(\theta_i,\theta(y))==false$ **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)∈ddl **DO**

FORALL i∈ (0..N−1)∧i∈L **DO** T [i] = the source node of the backedge of L **OD**

FORALL i∈ (0.. N−1)∧|| (θ_i, θ (y)) == 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 $q\xrightarrow{id}x$, 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. ① If there is no nodes in s which executes sequentially with q, that is, the position of θ (q) in the trace record array associated

with x is $\perp$. 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. ② 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 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; ③ 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 $q \xrightarrow{ddl} x$, there is a valid regioned execution witness from q to x in the region of coregion (q, x) according to Definition16, 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 $(\frac{N}{t})$, that is, in the first iteration of the algorithm there are nearly $(\frac{N}{t})$ nodes to be evaluated; In the second iteration, there are nearly $(\frac{N}{t} \times \frac{N}{t})$ nodes to be evaluated; …; In the t-th iteration, there are nearly $(\frac{N}{t})^t$ nodes to be evaluated. The algorithm complexity is approximately 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

Binkley, D.W. and M. Harman, 2004. A survey of empirical results on program slicing. Adv. Comput., 62: 105-178.

Cheng, J., 1993. Slicing concurrent programs. A graph-theoretical approach. Proceedings of the First International Workshop on Automated and Algorithmic Debugging, pp: 223-240.

Chen, Z., B. Xu and H. Yang, 2000. An approach to analyzing dependency of concurrent programs[A]. Proceedings of the The First Asia-Pacific Conference on Quality Software, pp: 39-43.

Horwitz, S., J. Prins and T. Reps, 1988. On the adequacy of program dependence graphs for representing programs. Proceedings of Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, pp: 146-157.

Krinke, J., 1998. Static slicing of threaded programs. Proceeding ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE), pp: 35-42.

Krinke, J., 2003. Context-sensitive slicing of concurrent programs. Proceedings ESEC/FSE, pp: 178-187.

Muller-Olm, M. and H. Seidl, 2001. On optimal slicing of parallel programs. 33rd ACM Symposium on Theory of Computing, pp: 647-656.

Zhao, J., 1999. Slicing concurrent Java programs. Proceedings of the 7th IEEE International Workshop on Program Comprehension, pp: 126-133.