An Evaluation of Slicing Algorithms for Concurrent Programs

Dennis Giffhorn
Universität Passau
Passau, Germany
giffhorn@uni-passau.de

Christian Hammer
Universität Passau
Passau, Germany
hammer@fmi.uni-passau.de

Abstract

Program slicing is a program-reduction technique for extracting statements that may influence other statements. While there exist efficient algorithms to slice sequential programs precisely, there are only two algorithms for precise slicing of concurrent interprocedural programs with recursive procedures. We implemented both algorithms for Java, applied several new optimizations and examined their precision and runtime behavior. We compared these results with two further algorithms which trade precision for speed. We show that one algorithm may produce incorrect slices and that precise slicing of concurrent programs is very expensive in terms of computation time.

1. Introduction

Program slicing is a widely recognized technique for analyzing programs. It has many applications such as debugging [1, 11], testing [2], complexity measurement [19], model-checking [9], and information flow control [8]. As most up-to date languages, like Java or C♯, have built-in support for concurrent execution and may even implicitly use threads (e.g. graphical user interfaces in Java programs), program slicing must be able to cope with concurrent programs.

Unfortunately, the precise and efficient slicing algorithms known for sequential programs cannot be applied to concurrent programs. Currently, there exist several algorithms for slicing concurrent programs, but only two of them can slice concurrent interprocedural programs with recursive procedures and yield precise slices. These two algorithms were developed by Nanda [16] and Krinke [14]. In this paper, we present an evaluation of these two algorithms and compare them with less precise algorithms in terms of precision and runtime behavior. We show that the precise algorithms may yield significantly smaller slices, but at the price of high execution times. We further show that Nanda's algorithm may compute incorrect slices and present a fixed version as well as an improved version of Krinke's algorithm with a set of new optimizations.

2. Slicing

A slice of a program consists of all statements and predicates that may influence a given program point of interest, the so-called *slicing criterion*. Slicing was first introduced by Weiser in 1979 for reducing programs during debugging [23]. His approach uses an iterative data flow analysis to compute slices. Today, most slicing techniques use a different approach: They compute slices using reachability analysis in *program dependence graphs* [19], where the nodes represent statements or predicates and the edges represent possible influences. Horwitz et al. [10] introduced the *system dependence graph* (SDG), an extension of the PDG for procedural programs and developed the *two-phase slicing algorithm*, which uses *summary edges* to compute context-sensitive slices in $\mathcal{O}(|graph|)$. An overview of fundamental slicing techniques can be found in Tip's survey [22].

A *statement-minimal* slice for a slicing criterion s is a slice that does only contain statements that are guaranteed to influence s. Weiser showed that computation of statement-minimal slices is undecidable, as the evaluation of conditional branches cannot be analyzed [23]. Therefore, conditional branching is handled as non-deterministic branching. We will refer to context-sensitive slicing algorithms of sequential programs that abstract conditional branching in this way as *precise slicing*.

2.1. Slicing of concurrent programs

SDGs can be extended to *concurrent system dependence graphs* (cSDG) to represent concurrent programs that communicate via shared variables. We will use the term 'thread' for concurrently executing tasks in the remaining of this paper. Concurrent programs exhibit a special kind of data dependence called *interference dependence*, represented in the cSDG by *interference edges*. A node n is interference dependent on node m if m defines a variable that n uses and

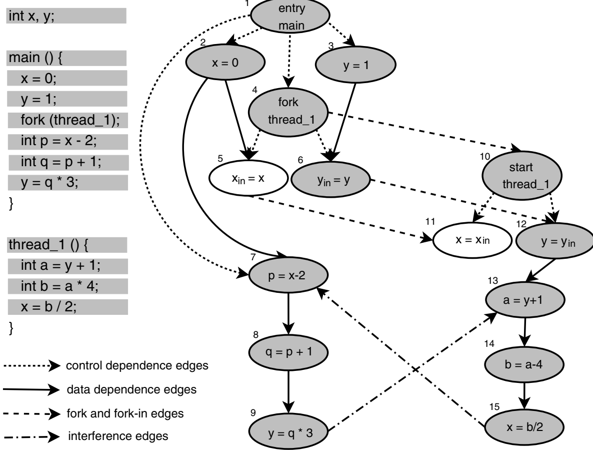

Figure 1. An example cSDG

Input: The cSDG G, a slicing criterion s.
Output: The slice S for s.

$W = \{s\}$, *a worklist*
$M = \{s \mapsto true\}$, *a map for marking the contents of W*
`repeat`
  $W = W \setminus \{n\},\ \ f = M(n)$ *remove next node n from W*
  `foreach` $m \rightarrow_e n$ *consider all incoming edges of n*
    `if` $m \notin \mathrm{dom}\, M \vee (f \wedge (m \mapsto false) \in M)$
      `if` $f \vee e \notin \{pi, c\}$
        $W = W \cup \{m\}$
        `if` $f \wedge e = po$
          $M = M \cup \{m \mapsto false\}$
        `elseif` $\neg f \wedge e = id$
          $M = M \cup \{m \mapsto true\}$
        `else`
          $M = M \cup \{m \mapsto f\}$
`until` $W = \emptyset$
`return` $\mathrm{dom}\, M$

Figure 2. Iterated two-phase slicer

m and n belong to different threads. Then the cSDG contains an interference edge $m \rightarrow_{id} n$. The invocation of a thread is modeled similar to procedure calls [10] using *fork sites*. Sharing of variables between different threads is simulated by passing them as parameters during a thread invocation. We define *fork edges* and *fork-in edges* in analogy to call and parameter-in edges [10]. There is no need for an equivalence to parameter-out edges, as changes in parameters (the shared variables) are propagated immediately via interference edges. We do not model join points of threads, as in many languages like Java or C♯ this would require must-aliasing between the target objects calling fork and join: We assume conservatively that all threads run until the last thread terminates. Figure 1 shows an example cSDG (for better readability, we will omit some control dependences in our figures that do not influence the result of our demonstrated slices). Several authors define further dependences in concurrent programs based on synchronization like *synchronization dependence* or *ready dependence* [3, 9, 25]. Both Nanda and Krinke suggest using synchronization-related constructs to prune interference dependences. As we do not consider data flow computation in this paper, these details are omitted.

Unfortunately, the two-phase slicing algorithm for sequential programs cannot be used to slice cSDGs, as summary edges do not capture interprocedural effects of interference dependences [16]. But a simple modification enables slicing of cSDGs: A two-phase slice is computed for the slicing criterion s and each time an interference dependence edge is traversed the reached node becomes a new slicing criterion. This *iterated two-phase slicer* was first described by Nanda [16] as a two-phase slicer nested in an outer while loop. It can be modified to yields correct slices in $\mathcal{O}(|graph|)$ (Figure 2).

The computed slices are, however, imprecise as interference dependence is not transitive. Consider the example in figure 1, where the slice for node 14 computed by the iterated two-phase slicer is shown in gray. The computation will leave `thread_1` at node 13 towards node 9 and later return to node 15 via the interference edge from node 7. Obviously, node 15 cannot influence the slicing criterion 14, as it cannot execute before node 14 (Krinke calls this effect *time travel* [12]). The intransitivity of interference dependence results from its weaker requirements: Unlike the dependences in sequential programs, interference dependence does not require an execution order between the interfering nodes. As the scheduling between concurrent statements is non-deterministic, it cannot be made a prerequisite for interference dependence. Therefore transitive traversal of interference edges can result in invalid execution orders similar to the example above. One approach to analyze which interference edges are valid to traverse is to trace the execution states of the threads. For this, each visited node is annotated with a *state tuple* Γ that contains for every thread the node last visited in that thread. If the slicing algorithm wants to traverse an interference edge $q \rightarrow_{id} m$ towards q, where t is the thread of q, and p is the state of t in m's state tuple, then a reachability analysis on the CFG of t checks whether q may reach p. If not, the traversal forms an invalid execution order and is rejected. This approach is still imprecise as the states of threads are only represented by a node. To gain more precision, one can additionally consider the calling contexts of the nodes. Both Nanda's and Krinke's algorithms use this approach. In the remainder, we will use the term *context* for a node and its calling context.

As Müller-Olm et al. have shown, precise slicing of concurrent interprocedural programs is undecidable [15]. Basically, if two nodes n and m are interference dependent $n \rightarrow_{id} m$ due to some variable v, then it is not decidable

whether another statement s that redefines v may execute between n and m (i.e. s is a *killing definition*). This follows from the conservative assumption that scheduling is non-deterministic to abstract from the scheduler. Therefore slicing of concurrent interprocedural programs may only be precise up to killing definitions for interference dependences. We will refer to it as *precise concurrent slicing*.

3. Nanda's and Krinke's algorithms

To gain a precise concurrent slice, one has to determine the calling contexts of the nodes at which the slice computation leaves and enters threads. For this purpose Nanda and Krinke use slicers that work with contexts instead of nodes. These slicers are called for a context c as slicing criterion and return its intra-thread slice $\bar{S}(c)$ and the contexts $I \in \bar{S}(c)$ where the thread can be left. Figure 3 shows the basic structure of both algorithms: First they compute all possible contexts C of slicing criterion s. These contexts are annotated with an initial state tuple Γ_0, where the execution state of every thread is at its end context, and inserted into a worklist W: Every interference edge traversal towards a thread with initial execution state is valid according to the reachability analysis. Now the algorithms iterate over every element (c, Γ) of W and compute an intra-thread slice $\bar{S}$ for c and the set of visited contexts I where the thread can be left. Then they compute the valid interference edges: For every context $i \in I$ they determine for every incoming interference edge $m \rightarrow_{id} n$, where n is the node of i, the set of valid contexts C' of m. A context c' of m is considered valid if c' may reach the context saved in Γ as the state for thread t', where t' is the thread of c'. The valid contexts $c' \in C'$ are then annotated with an updated state tuple $\Gamma' = [i/t]\Gamma$, where t is the thread of i, and inserted into worklist W. The slicing result ist the union of all slices $\bar{S}$.

Krinke's slicing algorithm is described in detail in [14]. Nanda describes two versions of her algorithm in [16]; a generic version with cobegin-coend parallelism and a special version with fork-join parallelism suitable for Java. As we implemented the algorithm for Java, we will only refer to the latter.

3.1. Differences between the algorithms

In Krinke's algorithm, a context of a node is represented by the node annotated with a call string [21]. A call string for a node n represents a sequence of procedure calls leading to the procedure that n belongs to. The contexts are computed dynamically during the slice. To compute them, the algorithm uses a slightly modified *explicitly context-sensitive slicer* (ECSS, [13]). This intra-thread slicing algorithm does not use summary edges but call strings to gain context-sensitive slices. It returns a precise intra-thread slice for the context that serves as slicing criterion, and the set of visited contexts where the thread may be left.

Input: The cSDG G, a slicing criterion s.
Output: The slice S for s.

let $\bar{C}_t(n)$ return all possible contexts for node n
let $\theta(c)$ return the thread context c belongs to
let $SeqSlice(c)$ return the intra-thread slice $\bar{S}$ for context c
and the contexts I where the thread of c can be left

Initialize the worklist W with an initial state tuple:
$\Gamma = (\bot, ..., \bot)$, *every thread is at its end node*
$W = \{(c, \Gamma') \mid t = \theta(s) \wedge c \in \bar{C}_t(s) \wedge \Gamma' = [c/t]\Gamma\}$
$M = \{s\}$, *a list for marking the contents of W*
repeat
  remove next element $w = (c, \Gamma)$ from W
  Compute a sequential slice ($\bar{S}$) for c and the visited
  contexts with incoming interference edges I
  $(\bar{S}, I) = SeqSlice(c)$
  $S = S \cup \bar{S}$

  foreach $i \in I$, *compute valid interference edges*
    foreach $m \rightarrow_{id} n \mid n\ is\ node\ of\ i$
      $t = \theta(n)$, *current thread*
      $t' = \theta(m)$, *reached thread*
      $\Gamma' = [i/t]\Gamma$, *save where thread t is left*

      Compute the valid contexts of m
      $C' = \{c' | c' \in \bar{C}_{t'}(m) \wedge c'\ reaches\ context\ \Gamma[t']\}$

      foreach $w' \in \{(c', [c'/t']\Gamma')|c' \in C'\}$
        if $w' \notin M$
          $W = W \cup \{w\}$
          $M = M \cup \{w\}$
until $W = \emptyset$
return S

Figure 3. Slicing concurrent programs

The reachability analysis to determine if an interference dependences between two contexts is valid uses a similar approach. The call strings of the contexts are used to traverse the related control flow graph in a context-sensitive manner. To avoid infinite stacking of call strings, it folds cycles in control flow graphs and dependence graphs resulting from loops and recursion. For this, it applies a folding algorithm that preserves context-sensitivity [14].

Nanda uses a special folding method for cycles in control flow graphs with the resulting graph called *interprocedural strongly connected regions* (ISCR) graph. This method allows enumerating the contexts of a CFG topologically in reverse preorder, such that contexts are represented as a single integer. For that purpose instances of methods that are called (transitively) from within a recursive or loop-based cycle are included into the cycle (called *virtual inlining* by Nanda [16]). This results in a stronger folded graph than in Krinke's algorithm and thus in a smaller number of contexts. The reachability analysis is done by a traversal on

the ISCR graph, where the context enumeration is used to preserve context-sensitivity.

The intra-thread slicer used in Nanda's algorithm is a modified two-phase slicer that works on contexts instead of nodes. Unlike Krinke's algorithm, it does not compute contexts itself but queries them from the ISCR graph. This basically works as follows: After traversing a dependence edge $m \rightarrow n$ towards m where c is the current context of n, all contexts C' of m are queried from the ISCR graph of m's thread. Then a reachability analysis on the ISCR graph is used to determine every context $c' \in C'$ that reaches c.

The algorithm contains a conservative approximation to handle dynamic thread generation inside of loops: Let l be a loop that dynamically invokes instances of a thread t. During the ISCR Graph computation, all nodes of l and the nodes of t are folded into a single fold node f. As now every node of thread t has the same context, every interference edge traversal towards an instance of t during the slice computation will be considered to be valid by the reachability analysis. It does not handle dynamic thread generation inside of recursive methods.

Nanda identifies combinatorial explosion in the thread state tuples to be a major issue of the algorithm's performance. She defines *restrictive state tuples* as a way to reduce that combinatorial explosion. Let $[c_1, ..., c_n]$, $[c'_1, ..., c'_n]$ be two state tuples. If $\forall i \in 1, ..., n : c'_i \ reaches \ c_i$ holds, then $[c'_1, ..., c'_n]$ is a *restrictive state tuple* according to $[c_1, ..., c_n]$. If c is a context, t and t' are state tuples and t' is restrictive according to t, then a slice for the slicing criterion (c, t') is a subset of the slice for slicing criterion (c, t). This results from t' making higher restrictions to the set of valid interference edges than t. The algorithm uses this property to identify redundant pairs of contexts and state tuples: After a dependence edge e is traversed towards a context c, the associated state tuple t' is computed. Then t' is compared with all state tuples T of earlier visits of c. If t' is restrictive to a tuple $t \in T$, the traversal of e towards c is discarded. The algorithm uses this optimization after every edge traversal.

3.2. Model of concurrency

Krinke uses a conservative model of concurrency where all threads of a program execute entirely in parallel. He defines a *threaded interprocedural program dependence graph* (tIPDG) consisting of the IPDGs of each thread that are only connected with interference edges. The control flow graphs of the threads are entirely disjoint. Furthermore, he does not allow dynamic generation of threads.

Nanda's algorithm uses a more precise model of concurrency, a fork-join mechanism to model thread invocation and joining. As a result, the dependences are represented in a *threaded system dependence graph* (TSDG) that is similar to the cSDG introduced in section 2. The *threaded control flow graph* (TCFG) consists of the CFGs of each thread that are connected at fork nodes and join nodes, respectively. Using this concurrency model enables a more precise analysis of whether two nodes in different threads may execute in parallel. This is done by dividing the threads into *thread regions* according to fork and join points of threads. Parallel execution is then determined on the level of thread regions.

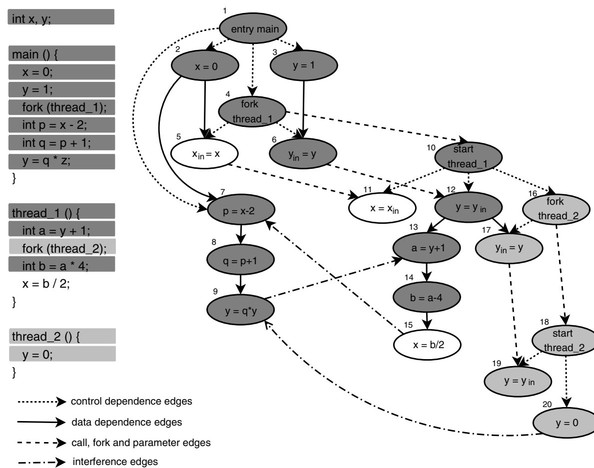

Figure 4. Thread regions allow higher precision

The state tuples for tracing the execution states of the threads also work on the level of thread regions and contain one element per region. If the entry of a thread region p is to be updated to a new value, context c, then all entries of thread regions that execute sequential to p are given the same value c (this may result in an entry of a thread region containing a context of a different thread). Using this, the algorithms are able to find more time travel situations: Consider figure 4 as an example. With Krinke's model of concurrency the algorithms compute the set of shaded nodes as the slice for node 14. With Nanda's model, however, they only compute the set of dark gray nodes, as they identify the interference edge traversal $20 \rightarrow_{id} 9$ towards 20 as invalid. To influence the slicing criterion node 14, node 20 must be executed before nodes 9 and 13. As thread 2 is started after the statement of node 13, this would require time travel. To keep track of the thread regions' states, the intra-thread slicer has to annotate every visited context with a state tuple and to update it after every edge traversal. By comparison, Krinke's model of concurrency allows updating state tuples only after interference edge traversals, his intra-thread slicer does not consider state tuples at all.

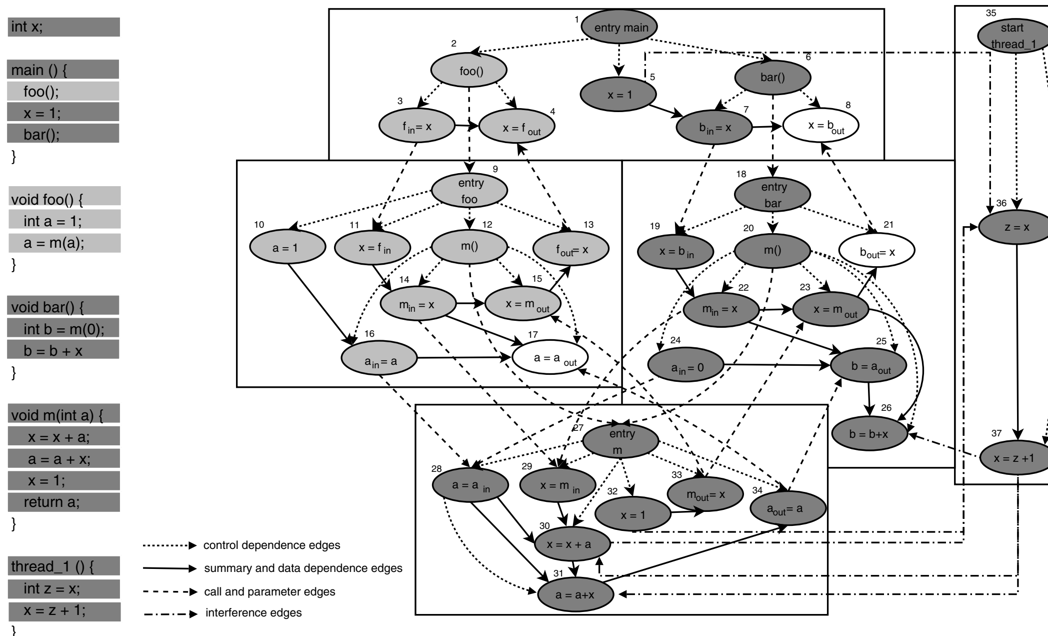

Figure 5. Incorrect Slice by Nanda's Algorithm

3.3. Further development

During our work, we developed and applied several further optimizations. We extended Krinke's algorithm with an approach to handle dynamic thread invocation inside of loops and recursion using conservative approximation. The initial execution state of every thread is set to the exit node of that thread, to allow initial interference edge traversal. With dynamic thread invocation inside of loops and recursion, we conservatively assume that these threads have an infinite number of instances, so every traversal of an interference edge towards such a thread is able to find an instance whose execution state is at its exit node. Thus the algorithm omits the reachability analysis when traversing towards a thread that is invoked dynamically in a loop or recursion (note that therefore the iterated two-phase slicer is able to handle dynamic thread invocation as well). We also adopted Nanda's more precise model of concurrency to Krinke's algorithm. For this the intra-thread slicer must be modified to update state tuples after each edge traversal to keep track of the thread region execution states. We applied further optimizations to the reachability analysis and to the intra-thread slicer which we will omit here due to space restrictions.

We applied an optimization to Nanda's algorithm that eliminates the reachability analysis after each traversal of an intraprocedural dependence edge. During the ISCR graph construction, we annotate every context with an ID of the method instance it belongs to. Then, after traversing an intraprocedural dependence edge, the algorithm can determine the context of the reached node n by retrieving that context of n that is annotated with the same method instance ID as the current context.

3.4. Imprecise Algorithms

We further use two imprecise algorithms in our evaluation. The first algorithm is the iterated two-phase slicer. The second algorithm trades precision for speed by using nodes instead of contexts to mark the thread execution states. It uses Krinke's model of concurrency and applies Nanda's optimization of restrictive state tuples after every interference edge traversal. We are not aware of any previous work describing the latter algorithm.

4. Implementation and evaluation

We implemented all algorithms in Java. All algorithms work on dependence graphs computed by Hammer's dataflow analysis for Java [7]. This analysis computes a variant of the standard SDG that represents nested parameter objects precisely. For the tests we used a uniprocessor 2.2Ghz AMD 3200+ workstation with 2GB of memory running Fedora 2.6.16 Linux. We expected that Nanda's and Krinke's algorithms do not scale well and that their gain of precision do not outweigh their costs.

4.1. Correctness

In this section we will show that Nanda's algorithm may compute incorrect slices. This incorrectness results from

Name (Nodes / Edges / Classes / Methods / Threads)	Instances	I2P	S	K	GK*	GK	N	ON
	1	.001	.001	.005	.003	.003	.001	.001
PrecisionTest	2	.001	.002	.212	.076	.048	.005	.005
(328 / 904 / 6 / 10 / 2)	3	.001	.004	6.184	.489	.312	.016	.016
	1	.001	.001	.001	.001	.001	.001	.001
TimeTravel	2	.001	.001	.010	.002	.002	.001	.001
(413 / 1136 / 7 / 14 / 2)	3	.001	.003	.149	.005	.007	.002	.002
	1	.001	.001	.007	.004	.002	.002	.001
ProducerConsumer	2	.001	.001	.239	.006	.009	.003	.002
(420 / 1159 / 6 / 10 / 2)	3	.001	.003	5.464	.022	.031	.004	.003
	1	.001	.004	.371	.023	.043	.091	.053
BoundedBuffer	2	.001	.037	67.325	.106	.233	.106	.071
(1324 / 3900 / 14 / 25 / 3)	3	.001	.127	–	.411	.879	.126	.088
Primes	1	.001	.023	3.883	.181	.258	.124	.050
(2906 / 9693 / 18 / 36 / 2)	2	.001	.178	–	5.411	7.916	.692	.416
AlarmClock	1	.003	.099	–	3.641	3.358	.832	.202
(4085 / 13842 / 17 / 74 / 2)	2	.003	.895	–	346.413	281.477	4.430	1.467
LaplaceGrid	1	.007	.323	111.900	1.945	.428	.476	.126
(10022 / 100730 / 22 / 95 / 1)	2	.007	1.868	–	27.836	4.775	2.141	.315
SharedQueue	1	.034	.385	–	49.198	5.052	.566	.309
(17998 / 139480 / 23 / 122 / 1)	2	.034	31.930	–	–	–	15.370	11.334

Table 1. Average execution times per slice for concurrent programs (in seconds)

the application of the restrictive state tuple optimization after every edge traversal. Consider Figure 5 as an example. The program consists of two threads - the main thread and thread_1. For simplicity we will apply Krinke's model of concurrency: Both threads execute entirely in parallel. Method `m` is called by both methods `foo` and `bar`, where `foo` can reach `bar` in the corresponding CFG. Thus each node of method `m` has two different contexts, where the context resulting from method `foo` can reach the context resulting from method `bar`. All other nodes have one context. We will denote every context of a node with the node itself, for nodes n of `m` we will append a suffix *foo* or *bar*, respectively (e.g. 30_{bar} denotes the context of node 30 in the calling context of method `bar`). The shaded nodes represent the precise concurrent slice for node 26, the darker gray shaded nodes represent the slice computed by Nanda's algorithm. It performs the following steps:

Initialization: Worklist W0 is initialized with element (26, [26, $\perp$]), where [26, $\perp$] is the state tuple, $\perp$ is the initial state of thread thread_1.

First intra-thread slice for (26, [26, $\perp$]): In phase 1, the algorithm visits the nodes {26, 25, 24, 23, 34, 33, 22, 20, 19, 18, 7, 6, 5, 1}, traverses the interference edge $37 \rightarrow_{id} 26$ towards node 37 and inserts element (37, [26, 37]) into worklist W0. The elements (33_{bar}, [33_{bar}, $\perp$]) and (34_{bar}, [34_{bar}, $\perp$]) are inserted in worklist W2. The elements (33_{foo}, [33_{foo}, $\perp$]) and (34_{foo}, [34_{foo}, $\perp$]) are also visited, but are discarded due to restrictive state tuples (`foo` can reach `bar`). In phase 2, the algorithm visits the nodes {34, 33, 32, 31, 30, 29, 28, 27}, where every node n is inserted as an element (n_{bar}, [n_{bar}, $\perp$]) in W2. Again, the also visited elements (n_{foo}, [n_{foo}, $\perp$]) are discarded as containing restrictive state tuples. Additionally, it traverses the interference edges $37 \rightarrow_{id} 31$ and $37 \rightarrow_{id} 30$ and thus inserts elements (37, [31_{bar}, 37]) and (37, [30_{bar}, 37]) into the outer worklist W0.

Second intra-thread slice for (37, [26, 37]): In phase 1, the nodes {37, 36, 35} are visited. At node 36, with state tuple [26, 36], the thread can be left via interference edge $30 \rightarrow_{id} 36$ towards node 30. Contexts 30_{foo} and 30_{bar} are valid according to the reachability analysis, as both can reach the saved context 26. But the state tuples of the resulting elements (30_{foo}, [30_{foo}, 36]) and (30_{bar}, [30_{bar}, 36]) are restrictive according to the state tuple of the earlier inserted element (30_{bar}, [30_{bar}, $\perp$]), as context 30_{foo} can reach 30_{bar} and context 36 can reach $\perp$. Thus the traversal towards node 30 is discarded. The same happens for interference edge $32 \rightarrow_{id} 36$ towards node 32 and later for the slices of (37, [31_{bar}, 37]) and (37, [30_{bar}, 37]): Method `m` cannot be entered again an thus the algorithm omits nodes that belong to the slice. This problem can be fixed by only applying the optimization when traversing interference edges.

4.2. Precision and runtime behavior

We use the following algorithms for our evaluation: Krinke's algorithm (K), a fixed version of Nanda's algorithm (N), another version of Nanda's algorithm using the optimization proposed in section 3.3, (ON), our modification of Krinke's algorithm 1. using Nanda's model of concurrency (GK), and 2. using Krinke's model of concurrency (GK*), the imprecise slicer described in section 3.4 (S), and the iterated two-phase slicer (I2P).

Our case study consists of eight programs. PrecisionTest and TimeTravel are small programs that model

Name	Instances	I2P	S	K	GK*	GK	N	ON
PrecisionTest	1	2.4	1.0	25.9	14.6	9.1	4.0	4.0
	2	2.4	1.1	615.8	76.1	39.7	9.3	9.3
	3	2.4	1.3	11000.2	219.6	118.3	16.5	16.5
TimeTravel	1	1.9	1.0	5.9	5.7	5.4	4.9	4.9
	2	1.9	1.0	41.0	15.1	12.1	6.9	6.9
	3	1.9	1.0	526.3	31.8	22.4	8.8	8.8
ProducerConsumer	1	3.2	1.1	24.7	11.1	11.1	10.0	10.0
	2	3.2	1.3	358.6	34.1	28.5	11.6	11.6
	3	3.2	1.6	6854.8	173.6	57.0	13.2	13.2
BoundedBuffer	1	15.3	1.0	360.5	96.7	96.7	90.2	90.2
	2	15.3	1.1	43112.0	316.8	232.1	99.9	99.9
	3	15.3	1.1	–	685.8	426.1	105.8	105.8
Primes	1	17.5	1.6	1189.1	195.5	142.6	108.6	108.6
	2	17.5	3.0	–	1458.5	788.2	338.9	338.9
AlarmClock	1	66.9	1.6	–	875.1	313.0	127.2	127.2
	2	66.9	3.5	–	2380.1	1607.7	397.2	397.2
LaplaceGrid	1	38.2	1.6	6376.7	167.6	59.4	27.4	27.4
	2	38.2	2.6	–	2409.3	441.9	100.5	100.5
SharedQueue	1	87.9	2.1	–	1599.7	145.1	111.7	111.7
	2	87.9	5.3	–	–	–	636.3	636.3

Table 2. Average number of elements inserted into the outer worklists per slice

nested thread invocation and potential time travel situations. ProducerConsumer implements a producer-consumer relation, BoundedBuffer is a bounded buffer example, Primes is a concurrent implementation of Eratosthenes' primes sieve, AlarmClock simulates an alarm clock, LaplaceGrid solves Laplace's equation over a rectangular grid and SharedQueue starts a set of threads that communicate via a shared queue. AlarmClock, BoundedBuffer, LaplaceGrid and SharedQueue are taken from the test suite of the Bandera project from the SAnToS Laboratory at the Kansas State University[1]. In our graph representation, threads are annotated with the number of their instances that exist at runtime. To observe how the algorithms cope with combinatorial explosion of thread states, we artificially raised the number of thread instances. This way we created several versions of the eight programs, resulting in a total of 20 programs. For each program, we computed 100 slices.

Table 1 shows the the average computation times per slice for our concurrent test programs in seconds. Omitted entries mean that the corresponding test suite run was not finished after 24 hours. The values for 'nodes' and 'edges' show the number of nodes and edges, respectively, of the dependence graphs, the value for 'thread' shows the number of different thread types in the programs, and the values for 'classes' and 'methods' show the number of classes and methods that are used in the programs. Table 2 shows the average number of elements inserted into the outer worklists due to interference edge traversals. Table 3 shows the average size of the computed slices in number of nodes. Column 'instances' in tables 1, 2 and 3 shows the number of instances of every thread that exist at runtime. For example: BoundedBuffer contains 3 threads. The column with 'instances = 2' means that the running program contains each two instances of every thread, resulting in 6 threads at runtime.

As table 3 shows, all algorithms are able to compute smaller slices than the iterated two-phase slicer I2P. The gain of precision ranges between 0%, for ProducerConsumer, and 30%, for LaplaceGrid and Shared Queue. The algorithms using Nanda's model of concurrency, N, ON and GK, are the most precise. The algorithms using Krinke's model of concurrency, K, OK and GK*, gain less precision. The imprecise algorithm S is more precise than the I2P slicer, but its gain of precision ranges only between 0% and 5%. It is further remarkable that increasing the number of thread instances decreases the benefit of the precise algorithms, whereas the needed computation times rise significantly: The more thread instances exist, the more interference edge traversals find a thread instance providing a fitting execution state.

We identified two major issues that influence the performance of the tested precise algorithms: combinatorial explosion of state tuples and the context computation and representation in the intra-thread slicers. The combinatorial explosion of state tuples directly influences the number of elements inserted into the outer worklist (table 2). Krinke's algorithm K suffers from both issues and could only slice our smaller test programs in reasonable time. Table 2 shows that the size of its outer worklists grows very fast when the number of thread instances is raised. Nanda's algorithm uses the restrictive state tuple optimization to ease this combinatorial explosion, which is very effective (table 2). Additionally, the ISCR graph construction creates fewer contexts as in Krinke's algorithm (see section 3.1), further reducing the possible combinations. Another advantage of Nanda's algorithm is, that it represents contexts as single integers instead of call strings like Krinke's algorithm. The call string rep-

[1] http://www.cis.ksu.edu/santos/

Name	Instances	I2P	S	K	GK*	GK	N	ON
PrecisionTest	1	31.2	27.0	26.6	26.6	24.7	24.7	24.7
	2	31.2	31.1	31.1	31.1	28.8	28.8	28.8
	3	31.2	31.1	31.1	31.1	28.9	28.9	28.9
TimeTravel	1	24.1	23.5	23.5	23.5	23.5	23.5	23.5
	2	24.1	24.1	24.1	24.1	24.1	24.1	24.1
ProducerConsumer	1	38.8	38.8	38.8	38.8	38.8	38.8	38.8
BoundedBuffer	1	211.9	211.1	211.1	211.1	211.1	211.1	211.1
	2	211.9	211.9	211.9	211.9	211.9	211.9	211.9
Primes	1	353.4	342.3	335.7	335.7	335.7	335.7	335.7
	2	353.4	353.4	–	353.4	353.4	353.4	353.4
AlarmClock	1	918.5	910.6	–	831.8	683.4	683.4	683.4
	2	918.5	910.6	–	909.8	762.3	762.3	762.3
LaplaceGrid	1	1534.6	1498.4	1179.5	1179.5	1019.3	1019.3	1019.3
	2	1534.6	1534.6	–	1301.1	1055.8	1055.8	1055.8
SharedQueue	1	2174.2	2082.3	–	2019.9	1479.9	1479.9	1479.9
	2	2174.2	2169.9	–	–	–	1890.1	1890.1

Table 3. Average size per slice (number of nodes)

resentation is likely to decline performance in bigger programs, as its size can grow arbitrarily. The performance of our improved versions of Krinke's algorithm, GK and GK*, is similar to the performance of Nanda's algorithm for the smaller programs. For the bigger programs, however, their performance declines, as they use the call site representation for contexts. Nanda's model of concurrency used in GK can gain a speed-up compared to Krinke's model of concurrency used in GK* (e.g. AlarmClock, LaplaceGrid, SharedQueue). Our optimized version of Nanda's algorithm, ON, has noticeably lower execution times, e.g. for AlarmClock, LaplaceGrid and SharedQueue. Of all precise algorithms it performed best. The S algorithm is less affected by the combinatorial explosion of thread state tuples, as it does not use contexts as thread states. The I2P algorithm is not affected at all as it does not consider thread states nor contexts.

Nanda's algorithm might perform poorly for programs with deep call-chains or high usage of libraries. Both factors affect the cost of the reachability analysis which is computed after every edge traversal. We made a small case study with sequential programs to observe these factors; it is shown in table 4. We used the two-phase slicer (2P), the iterated two-phase slicer (I2P) and the intra-thread slicing algorithms of K, N and ON, abbreviated with (K'), (N') and (ON'). JavaCard Wallet is a program with deep call-chains and high usage of libraries. Here Nanda's algorithm performs worst. On the other hand, the algorithm might perform well for big programs that are highly recursive. As described in section 3.1, recursive cycles and all procedure calls within a cycle are collapsed into one single node. In highly recursive programs, this can reduce the number of contexts significantly.

Nanda provides an evaluation for her algorithm [16]; however, it is difficult to compare its results with ours, as her original algorithm may compute incorrect slices by pruning valid interference edges. We fixed the algorithm by only applying the restrictive state tuple optimization when traversing interference edges, which avoids such pruning but raises execution times. For example, our implementation of the original algorithm that computes incorrect slices needs 1.6 seconds on average to slice the SharedQueue program with 2 instances for each thread (not shown in our tables), the fixed algorithms N and ON need 15.4 seconds and 11.4 seconds respectively (table 1).

Krinke did not implement his algorithm. To the best of our knowledge, our implementation is the first, so we could not compare it with another evaluation.

4.3. Study Summary

The algorithms for precise slicing of concurrent programs are able to decrease the size of the slices significantly – up to 30% in our tests – but they pay a high price: The execution times rise dramatically and are dependent on the numbers of threads in the analyzed program. A vital requirement for an application of one of these algorithms is, in our opinion, to use Nanda's restrictive state tuple optimization. Nanda's more precise model of concurrency is not bound to increase the execution times – it can even decrease it – so we also recommend to use it. Two of the examined algorithms, Nanda's algorithm and our improved version of Krinke's algorithm, use both recommendations. Nanda's algorithm has an advantage as it represents contexts by single integers. The application area of these algorithms is, in our opinion, bound to concurrent programs with a low number of threads, as increasing numbers of threads decrease the precision benefits and at the same time raise execution times. The iterated two-phase slicer is by far the most efficient algorithm. Additionally, it is easy to implement, so we recommend its use for slicing bigger concurrent programs, for programs with high numbers of threads and in application areas where its imprecision is negligible.

4.4. Threats to validity

As evaluations depend on the quality of the used benchmark, we want to discuss possible flaws of our selected programs.

Our case study lacks bigger programs. As the size of a program does not necessarily influence the number of thread-shared data, the algorithms might work well for bigger programs with sparse interference dependences. In our case study we examined programs with a small number of threads and artificially raised their number of instances. As a result the performance and the precision benefit declined. However, that needs not be the case for programs with many threads but only few thread instances. Future work could therefore investigate how the algorithms cope with bigger programs with few threads or few instances of threads, e.g. graphical user interfaces written in Java. Further threats to validity are possible bugs in the implementations, as these algorithms are not easy to implement.

5. Related work

We only gave a summary of Krinke's and Nanda's algorithms. Both have described their algorithms in detail in several publications [14, 16]. Also, there exist earlier, intraprocedural variants of both algorithms [12, 17].

Chen [3] developed a different approach to handle the intransitivity of interference dependence. He uses execution orders, MHP analysis and synchronization information to detect time-travel situations during slicing. As his approach needs to inline synchronized methods, it cannot completely handle recursion.

Probably the first who addressed slicing of concurrent programs was Cheng [4, 5]. He developed a *Program Dependence Net* (PDN) for representing dependences in parallel or distributed programs without procedures, where the concurrent tasks communicate via channels. He defines *selection dependence* which is a kind of control dependence for nondeterministic selection of communication partners, *synchronization dependence*, a special control dependence based on synchronization, and *communication dependence*, a dependence between statements that are transitively dependent via a path that contains both data dependence and communication dependence. Slicing on PDNs is then performed using simple graph reachability.

The PDN is extended to the *System Dependence Net* SDN for representing concurrent object-oriented programs by Zhao et al. [24]. To slice on this SDN, they extend the two-phase-slicer to traverse the additional kinds of dependences in both phases. A similar algorithm is used by Zhao to slice concurrent Java programs [25]. His *Multithreaded Dependence Graph* (MDG) is similar to the cSDG and additionally contains *synchronization dependences* arising

	Dijkstra	MatrixMult	JavaCard Wallet
nodes	2927	3346	23340
edges	8837	20706	109360
K'	.128	4.826	636.137
N'	.006	.047	10130.578
ON'	.004	.037	4806.853
2P	.001	.007	.032
I2P	.001	.007	.031

Table 4. Average execution time per slice for sequential programs (in seconds)

from Java's operations for synchronization. To slice on MDGs he uses a modified two-phase slicer that additionally traverses interference and synchronization dependences in both phases. Nanda shows that such a simple inclusion of interference dependence into both phases of the two-phase slicing algorithm results in incorrect slices [16].

Hatcliff et al. [9] use slicing in their Bandera project, a tool set for compiling Java programs into inputs of several existing model-checkers, to analyze and omit program parts that are unrelated to a given specification. They use similar dependences as in the cSDG and define further dependences to represent synchronization and indefinitely delays of execution. Their *synchronization dependence* captures dependences between a statement and its innermost-enclosing acquisition and release of a monitor. The *divergence dependence* represents the situation where an infinite loop may infinitely delay the further execution, *ready dependence* similarly represents the situation where a statement may block the further execution of a thread. They trade precision for efficiency by treating interference dependence as transitive.

Ramalingam shows that synchronization-sensitive context-sensitive slicing slicing of concurrent programs is undecidable [20]. This is achieved by reducing Post's Correspondence Problem to the synchronization-sensitive context-sensitive reaching problem.

6. Conclusion and future work

We evaluated Nanda's and Krinke's algorithms for precise slicing of concurrent programs. Our results can be summarized as follows: Both algorithms are able to raise precision significantly, but do not scale well for larger programs or higher numbers of threads. We showed that Nanda's original algorithm can compute incorrect slices, and a way to fix it. We further applied several optimizations to both algorithms. For most programs in our test suite, our optimized version of Nanda's algorithm performed best.

A major handicap of the precise concurrent slicing algorithms is, that in programs with many thread instances the computational costs may rise extremely, while the precision benefit decreases. This seems to restrict their usability to

programs with a small number of thread instances. One way to cope with this problem could be to conservatively approximate a thread, when its instances exceed a certain number, the way we treat threads generated in loops: Every traversal of an interference edge towards such a thread is considered to be valid. Apart from that, incremental precision can be employed in applications like e.g. information flow control [8]. If an illegal influence is detected with the I2P slicer, a more precise slicer may prune that dependence (but at a higher cost). Since this analysis is only done at compile time, a larger overhead for parts of the problem is probably acceptable.

Another problem we encountered is the usage of the program's CFG to determine valid execution orders. For that purpose, it it necessary that the generated CFG does in fact represent the execution order in the executed program. A Java compiler, for example, is permitted to reorder the instructions in a thread, as long as the reorderings do not affect the semantics of that thread in isolation [6]. This may result in spuriously rejected interference edge traversals and thus in incorrect slices.

A finer grained concurrency model – e.g. modeling join points of threads – based on the happens-before relation defined in the Java Memory Model (JMM)[6] or based on MHP (may-happen-in-parallel) analysis [18] would allow pruning of redundant interference dependence edges and detecting more time travels, resulting in fewer reachability checks and higher precision. Apart from that, reachability itself could become more precise, such that a smaller number of contexts is encountered

References

[1] H. Agrawal, R. Demillo, and E. Spafford. Debugging with dynamic slicing and backtracking. *SoftwarePractice and Experience*, 23(6):589–616, 1993.

[2] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. *Proc POPL '93*, pp. 384–396, ACM Press, 1993.

[3] Z. Chen and B. Xu. Slicing concurrent java programs. *ACM SIGPLAN Notices*, 36(4):41–47, 2001.

[4] J. Cheng. Slicing concurrent programs. *Automated and Algorithmic Debugging*, 1993.

[5] J. Cheng. Dependence analysis of parallel and distributed programs and its applications. *International Conference on Advances in Parallel and Distributed Computing*, 1997.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley Prof., 3rd edition, 2005. `http://java.sun.com/docs/books/jls/`.

[7] C. Hammer and G. Snelting. An improved slicer for Java. *PASTE'04*, 2004.

[8] C. Hammer, J. Krinke, and G. Snelting: Information flow control for java based on path conditions in dependence graphs. *Proc. IEEE International Symposium on Secure Software Engineering*, 2006

[9] J. Hatcliff, J.C. Corbett, M.B. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with jvm primitives. *Static Analysis Symposium*, pages 1–18, 1999.

[10] S.B. Horwitz, T.W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst.*, 12(1):26–60, 1990.

[11] M. Kamkar, N. Shahmehri, and P. Fritzson. Bug localization by algorithmic debugging and program slicing. *Proceedings of International Workshop on Programming Language Implementation and Logic Programming, LNCS*, 456pp.60–74, Springer, 1990.

[12] J. Krinke. Static slicing of threaded programs. *PASTE '98*, pp. 35–42, 1998.

[13] J. Krinke. Evaluating context-sensitive slicing and chopping. *International Conference on Software Maintenance*, pages 22–31, 2002.

[14] J. Krinke. Context-sensitive slicing of concurrent programs. *Proc. ESEC/FSE'03*, pages 178–187, 2003.

[15] M. Müller-Olm and H. Seidl. On optimal slicing of parallel programs. *STOC 2001 (33th ACM Symposium on Theory of Computing)*, pages 647–656, 2001.

[16] M.G. Nanda and S. Ramesh *Interprocedural slicing of multithreaded programs with applications to Java*. *ACM TOPLAS.*, 28(6):1088–1144, 2006.

[17] M.G. Nanda and S. Ramesh. Slicing concurrent programs. *ISSTA 2000*, pages 180–190, 2000.

[18] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing MHP information for concurrent java programs. In *Proc. ESEC/FSE '99*, pages 338–354. Springer, 1999.

[19] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development envi- ronment. *ACM Softw. Eng. Notes*, 9(3):177–184, 1984.

[20] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Prog. Lang. Syst.*, 22 (2):416–430, 2000.

[21] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, 1981.

[22] F. Tip. A survey of program slicing techniques. *Journal of Prog. Lang.*, 3(3):121–189, Sept. 1995.

[23] M. Weiser. Program slicing. *IEEE TSE*, pages 352–357 1984.

[24] J. Zhao, J. Cheng, and K. Ushijima. Static slicing of concurrent object-oriented programs. *Proceedings of the 20th IEEE Annual Int. Computer Software and Applications Conference*, pages 312–320, 1996.

[25] J. Zhao. Slicing concurrent java programs. *Proceedings of the 7th IEEE International Workshop on Program Comprehension*, pages 126–133, 1999.