Precise Slicing of Concurrent Programs[*]

An Evaluation of static slicing algorithms for concurrent programs

Dennis Giffhorn · **Christian Hammer**

Abstract 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 (Krinke 2003; Nanda and Ramesh 2006). We present an empirical evaluation of both algorithms for Java. We demonstrate that both algorithms provide the same precision up to the model of concurrency in use and show that the concurrency model has strong impact on slice precision and computation costs. Furthermore, we extend both algorithms to support dynamic thread creation both in loops and recursion – a feature that the original algorithms could not fully handle. The worst case complexity of the algorithms being exponential, we developed several optimizations and compared these with each other and with algorithms that trade precision for speed. Finally, we show that one algorithm may produce incorrect slices and present a remedy.

Keywords slicing, program analysis, concurrency, threads

1 Introduction

Program slicing is a widely recognized technique for analyzing programs. Slices are often computed on a program representation called *system dependence graph* (SDG), where nodes represent statements or predicates and edges represent possible dependences (Horwitz et al. 1990). Slicing has many applications such as debugging

[*] This is an extended version of previous work (Giffhorn and Hammer 2007)

D. Giffhorn, C. Hammer
Universität Karlsruhe (TH), Karlsruhe, Germany
E-mail: giffhorn@ipd.info.uni-karlsruhe.de

C. Hammer
E-mail: hammer@ipd.info.uni-karlsruhe.de

(Kamkar et al. 1990; Sridharan et al. 2007), testing (Bates and Horwitz 1993), complexity measurement (Ottenstein and Ottenstein 1984), model-checking (Hatcliff et al. 1999), and information flow control (Hammer and Snelting 2008). Since contemporary languages, like Java or C♯, offer built-in support for concurrent execution and threads, slicing must cope with concurrent programs.

Unfortunately, the precise and efficient slicing algorithms known for sequential programs cannot be applied to concurrent programs directly. Concurrent programs contain new kinds of dependences, which need special treatment. Treating them like sequential dependences can result in incorrect slices (section 3 contains an example). 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 (Nanda and Ramesh 2006) and Krinke (Krinke 2003). It is the goal of our current work to thoroughly examine and compare both algorithms.

When we started this work, Krinke's algorithm had not been implemented and only one implementation of Nanda's algorithm had been reported (Nanda and Ramesh 2006), giving rise to several questions:

– Which algorithm is more precise?
– Which of them is more time efficient?
– Are the algorithms correct?
– Are the algorithms practical on practical programs?

As shown by Nanda, the algorithms have a worst case runtime complexity exponential in the number of the target program's threads. However, as this is only the worst case, the algorithms might behave well in most cases; this had not been investigated either.

Both algorithms cannot fully handle dynamic thread generation inside loops and recursion. Nanda's algorithm contains a technique to handle dynamic thread generation inside loops, but not inside recursive procedures, and Krinke's algorithm does not address dynamic thread generation at all.

In this article, we investigate the above questions and issues. The contributions of this work are manifold:

1. We provide the first implementation of Krinke's algorithm and the first combined implementation of both algorithms. We have implemented them for Java and use the same SDG generator framework (Hammer and Snelting 2004) for both algorithms, thus simplifying our comparison. We made the following observations: First of all, the model of concurrency used in the algorithms has a substantial impact on precision. Nanda and Krinke use different models, but if both algorithms are run with the same concurrency model, they produce the same results. Thus both algorithms offer the same precision.
2. Both algorithms have been shown to be expensive in terms of execution time, which inspired us to develop further optimizations and a new conservative algorithm that trades precision for speed. We indeed found one new optimization for Nanda's algorithm. Not implemented before, Krinke's algorithm offered more opportunities for optimization. To mention one, we developed a fast interproce-

dural and context-sensitive reachability analysis between two statements that can be computed intra-procedurally.

3. We compared the optimized algorithms with the original versions and several conservative algorithms in terms of runtime behavior and precision. The best algorithms are 30% more precise than the most conservative algorithm. Our optimized variant of Nanda's algorithm was 1.7 times faster than the original algorithm, our optimizations applied to Krinke's algorithm provided a speed-up of 142.2 times. The most performant precise algorithm was our optimized version of Nanda's algorithm.
4. Even the optimized algorithms do not scale well due to their exponential runtime complexity. Therefore, we investigated whether *hot spots* exist in the target programs, i.e. if certain slicing criteria take the lion's share of the average computation time. Unfortunately, our investigation strongly suggests that computation costs are not dominated by hot spots.
5. As mentioned before, Nanda's algorithm cannot handle dynamic thread generation inside recursive procedures, and Krinke's algorithm cannot handle dynamic thread generation at all. We present a technique that can handle both cases and which was integrated into both algorithms.
6. Nanda's original algorithm contains an optimization which significantly relieves the algorithms' exponential runtime complexity in practice. Unfortunately, its application at certain points in the original algorithm might prune valid program dependences, resulting in incorrect slices. We explain this problem and describe a remedy.

This paper is structured as follows: The next section introduces models of concurrency and describes Nanda's as well as Krinke's model. Section 3 introduces slicing basics, modeling of concurrent programs via SDGs and the basic idea of precise slicing of concurrent programs. It then proceeds to describe Nanda's and Krinke's algorithms. Section 4 describes our new optimizations and our method for handling dynamic thread generation. Section 5 contains our evaluation, section 6 describes future work, section 7 discusses related work, and section 8 concludes.

2 Models of Concurrency

To analyze concurrent programs one needs to model which parts of a program are actually executing in parallel. Nanda and Krinke use different models of concurrency, which influences precision and speed of their algorithms. We start with a discussion of these models, because they represent an independent dimension in the slicing algorithms (up to minor technical details) and thus are interchangeable. Since we have implemented these algorithms for Java, we focus on concurrency through threads invoked and joined by a fork-join mechanism and monitor-style synchronization. A more elaborate comparison of concurrency models including cobegin-coend parallelism as in Ada and rendezvous-style synchronization has been done by Chen et al. (Chen et al. 2000)[1].

[1] Note that their comparison includes Nanda's and Krinke's models of concurrency, but only of their earlier, intra-procedural slicing algorithms (Krinke 1998; Nanda and Ramesh 2000).

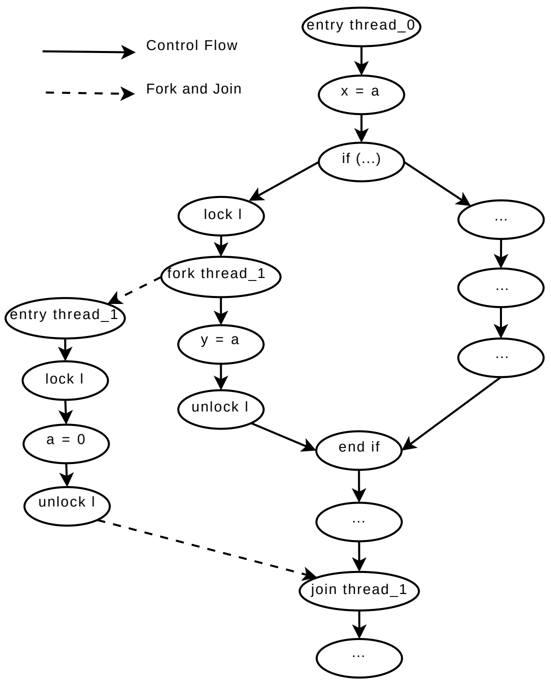


Fig. 1 A threaded program

The degree of parallelism in a program is influenced by the fork and join points of threads, as well as by synchronization. Figure 1 shows the control flow graph of a program consisting of two threads, where thread 0 forks and joins thread 1. The statements `lock l` and `unlock l` symbolize monitor-style synchronization, where `lock l` assigns a lock to monitor `l` and `unlock l` releases it. The three assignments `x = a`, `y = a` and `a = 0` will be discussed in the next section and can be ignored for now.

For many analyses, including slicing, it is sound to model more parallelism than possible in any actual execution but unsound to omit some. Thus a simple and sufficient concurrency model is to assume that all threads run entirely in parallel. In Figure 1, this model would ignore fork and join as well as the synchronization and pretend that thread 0 and thread 1 execute completely in parallel. This model was described by Krinke.

Nanda leveraged a more precise model that considers fork and join points but ignores synchronization. It enables a more precise analysis of whether two nodes in different threads may execute in parallel. To this end, threads are split into *thread regions* at fork and join points and parallel execution is determined on the level of thread regions. In summary, a thread region starts after a fork, at a join or at a point where two distinct thread regions meet. A thread region ends where another begins. The left hand graph in Figure 2 shows the thread regions for our example. It can now be determined that only regions 2 and 3 as well as 2 and 4 might execute in parallel.

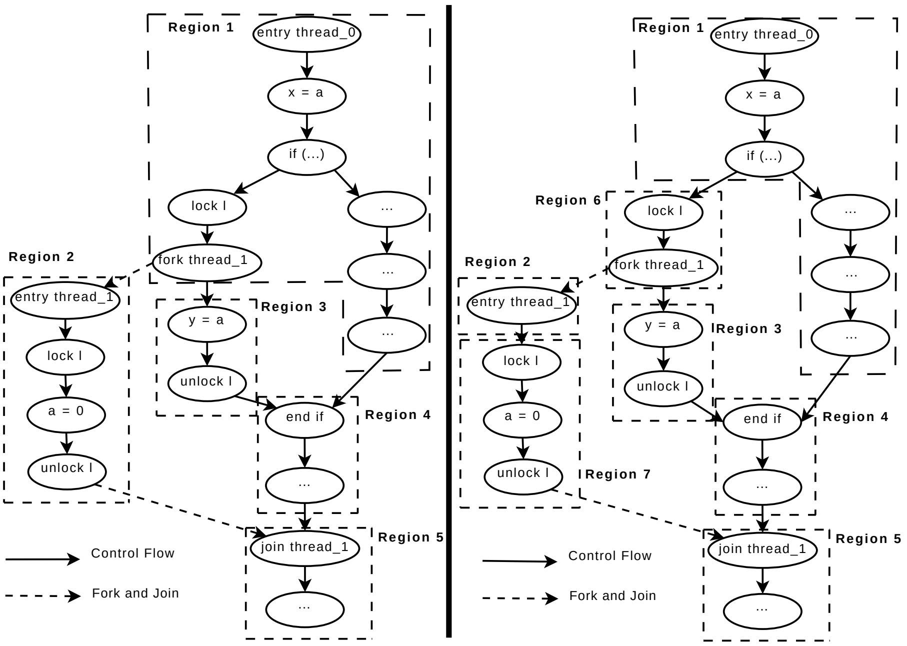


Fig. 2 Thread regions – without (left side) and with synchronization (right side)

The thread regions model might further be improved by considering synchronization. The acquisition of monitor l before the spawning of thread 1 assures that regions 2 and 3 are not executed in parallel. The computation of thread regions would have to consider `lock` and successors of `unlock` as additional starting points of thread regions. The result is shown in the right hand graph in Figure 2. Here regions 2 and 3, 2 and 4 as well as 7 and 4 might execute in parallel. This approach could be extended to full-fledged may-happen-in-parallel (MHP) analysis (Naumovich et al. 1999). However, to date no scalable implementation for full Java has been reported.

3 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*. Today, many slicing techniques are based on reachability analysis in *program dependence graphs* (Ottenstein and Ottenstein 1984). Horwitz et al. introduced the *system dependence graph* (SDG), a dependence graph for procedural programs which enables to compute context-sensitive slices in $\mathcal{O}(|edges|)$ via the *two-phase slicing algorithm* (Horwitz et al. 1990). Hammer and Snelting developed a dataflow analysis for object-oriented programs which computes SDGs that represent nested parameter objects precisely (Hammer and Snelting 2004). Their algorithm is the foundation of our SDG generator. An overview of fundamental slicing techniques can be found in Tip's survey (Tip 1995), empirical results are surveyed in (Binkley and Harman 2004).

A program dependence graph (PDG) for a single procedure P consists of one node for each statement or predicate expression of P and two kinds of edges: *data dependence edges* and *control dependence edges*. A node n is *control dependent* on node m, denoted by a control dependence edge $m \rightarrow_{cd} n$, if there is a path from m to n in the procedure's control flow graph (CFG) and every node on that path is post-dominated by n, but m is not post-dominated by n. A node n is *data dependent* on node m, denoted by a data dependence edge $m \rightarrow_{dd} n$, if m defines a variable v that n uses and if there is a path in the CFG on which n executes after m and there is no other node m' between m and n on that path that redefines v.

A *system dependence graph* (SDG) for a program consists of the PDGs of its procedures, connected at *call-sites* (Horwitz et al. 1990). A call-site consists of a call node c that is connected with the entry node e of the called procedure via a *call edge* $c \rightarrow_c e$. Parameter passing and result returning is modeled by *parameter nodes*. For every passed parameter there exists an *actual-in node* a_i and a *formal-in node* f_i that are connected via a *parameter-in edge* $a_i \rightarrow_{pi} f_i$. For every modified parameter and returned value there exists an *actual-out node* a_o and a *formal-out node* f_o that are connected via a *parameter-out edge* $f_o \rightarrow_{po} a_o$. Formal-in and formal-out nodes are control dependent on entry node e, actual-in and actual-out nodes are control dependent on call node c by definition. So-called *summary edges* represent transitive flow in the called method between actual-in and actual-out nodes of one call site. Figure 3 shows an example SDG[2].

Summary edges permit an efficient computation of context-sensitive slices in two phases. As an example we compute the slice for `print j` in Figure 3, consisting of the shaded nodes. Phase 1 starts at the slicing criterion and traverses backwards all edges but parameter-out edges, while the source nodes of encountered parameter-out edges are saved in a list L. In our example this phase visits the light gray nodes. Phase 2 starts at all nodes in list L and traverses backwards all edges but call and parameter-in edges. Because of the summary edges, there is no need to return from a called procedure back to the callee. In our example phase 2 visits the dark gray nodes. The resulting slice consists of all nodes visited in phases 1 and 2.

A *statement-minimal* slice for a slicing criterion s is a slice that only contains statements that are guaranteed to influence s in some execution. Weiser has shown that the computation of statement-minimal slices is undecidable due to evaluation of conditional branches (Weiser 1984). Therefore, conditional branching is modeled as non-deterministic branching. Context-sensitive slicing of sequential programs that ignores conditional branching is therefore commonly called *precise slicing*. A slice that contains additional (i.e. unnecessary) statements is called *imprecise*. A slice that lacks a statement which might influence the slicing criterion in some execution is called *incorrect*.

[2] For better readability, we omit some control dependences in our figures if they do not influence the result of the demonstrated slices.

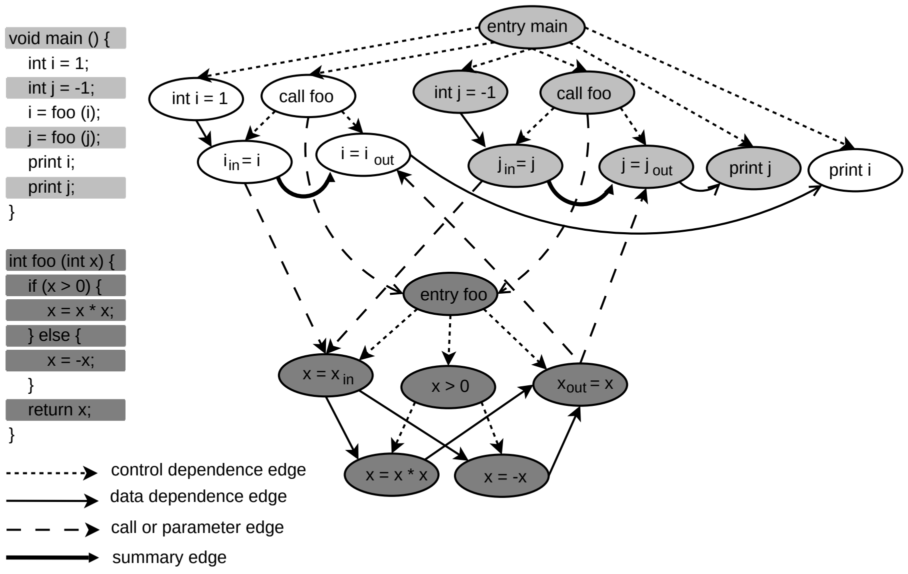


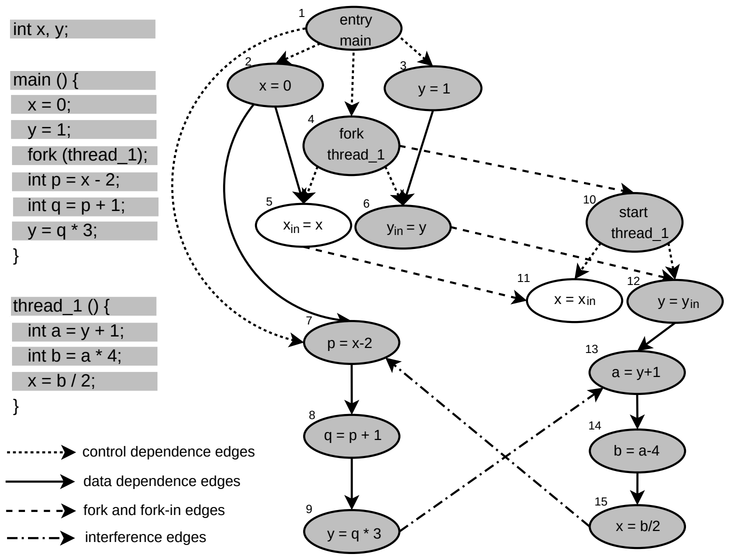
Fig. 3 An example SDG

3.1 Slicing of concurrent programs

SDGs can be extended to *concurrent system dependence graphs* (cSDG) to represent concurrent programs where threads communicate via shared variables. Such concurrent programs exhibit a special kind of data dependence called *interference dependence* (Krinke 1998). A node n is interference dependent on node m, denoted by an interference edge $m \rightarrow_{id} n$, if m defines a variable that n uses and m and n can execute concurrently. Thread invocation is modeled similar to procedure calls using *fork sites*, where shared variables are passed as parameters. To this end, *fork edges* and *fork-in edges* are defined in analogy to call and parameter-in edges. We currently do not model join points of threads, because in many languages like Java or C$\sharp$ this would require must-aliasing between the target objects of fork and join: Threads are conservatively assumed to run until the last thread terminates. Hence an equivalence to parameter-out edges is not needed, because changes in parameters (the shared variables) are propagated immediately via interference edges. Figure 4 shows an example cSDG.

Several authors define further dependences in concurrent programs based on synchronization like *synchronization dependence* or *ready dependence* (Chen and Xu 2001; Hatcliff et al. 1999; Zhao 1999). Both Nanda and Krinke suggest to analyze synchronization-related constructs to prune interference dependences. Since we do not consider data flow computation in this paper, we omit these details.

Unfortunately, the two-phase slicing algorithm from sequential programs may not be used to slice cSDGs, because summary edges do not capture interprocedural effects of interference dependences (Nanda and Ramesh 2006): The resulting slices

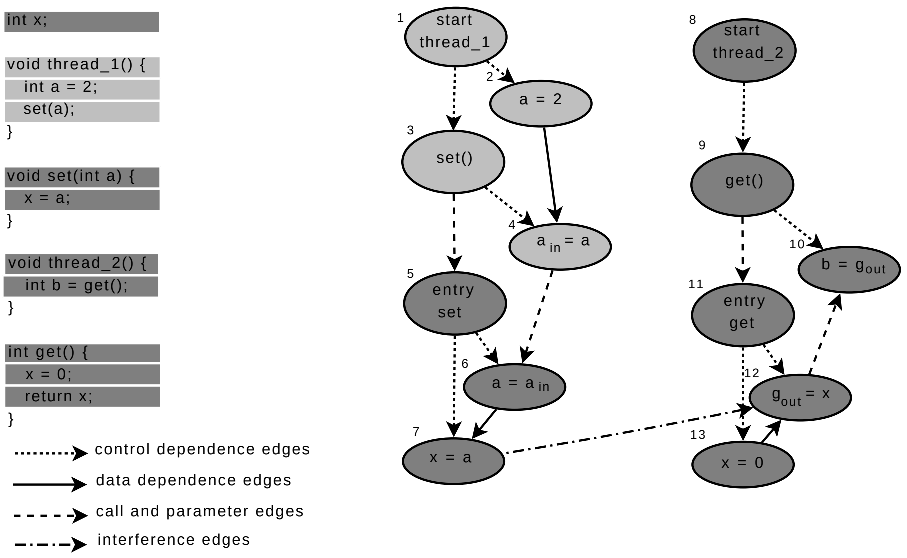


```
int x, y;

main () {
  x = 0;
  y = 1;
  fork (thread_1);
  int p = x - 2;
  int q = p + 1;
  y = q * 3;
}

thread_1 () {
  int a = y + 1;
  int b = a * 4;
  x = b / 2;
}
```

Fig. 4 An example cSDG



```
int x;

void thread_1() {
  int a = 2;
  set(a);
}

void set(int a) {
  x = a;
}

void thread_2() {
  int b = get();
}

int get() {
  x = 0;
  return x;
}
```

Fig. 5 A two-phase slice on a cSDG, omitting nodes that belong to the slice

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
                          //( true represents phase 1, false phase 2)

repeat
  $W = W \setminus \{n\}$  // remove next node n from W

  foreach $m \rightarrow_e n$  // handle all incoming edges of n
    // If m wasn't visited yet or we are in phase 1 and m was visited in phase 2,
    if $m \notin \text{dom}\, M \vee (\neg M(m) \wedge (M(n) \vee e = id))$
      // if we are in phase 1 or if e is not a call or param-in edge, add m to W
      if $M(n) \vee e \notin \{pi, c\}$
        $W = W \cup \{m\}$

        /* Now we determine how to mark m: */

        // If we are in phase 1 and e is a param-out edge, mark m with phase 2
        if $M(n) \wedge e = po$
          $M = M \cup \{m \mapsto false\}$
        // If we are in phase 2 and e is interference edge, mark m with phase 1
        elseif $\neg M(n) \wedge e = id$
          $M = M \cup \{m \mapsto true\}$
        // Else mark m with the same phase as n
        else
          $M = M \cup \{m \mapsto M(n)\}$

until $W = \emptyset$
return dom $M$
```

Fig. 6 The iterated two-phase slicer

could be incorrect. Figure 5 shows a minimalist producer-consumer-style example with interference between the producer and the consumer. The shaded nodes highlight the correct slice for node 10, the darker nodes mark the slice computed by the two-phase slicer. In the first phase, nodes 10, 9 and 8 are visited. The second phase starts at the omitted parameter-out edge to node 12 and visits the nodes 12, 13, and 11. Because the algorithm traverses interference as if it were a standard dependence edge, it also marks the nodes in the `set` method in the second phase. However, since the second phase must not ascend into calling methods, the invocation of `set` will not be included, even though it clearly belongs to the slice. But a simple modification allows slicing of cSDGs: One computes a two-phase slice for the slicing criterion and each time an interference dependence edge is traversed, the reached node becomes a new slicing criterion if it has not yet been visited in phase 1. This technique is repeated until no new node can be added. This *iterated two-phase slicer* was first described by Nanda (Nanda and Ramesh 2006) as a two-phase slicer nested in an outer while loop and has a runtime complexity of $\mathcal{O}(2 * |edges|)$ (an edge might be traversed at most once in phase 1 and once in phase 2). We use a more compact implementation based on a single map (Figure 6).

Still, the computed slices are imprecise, because interference dependence is not transitive. Consider the example in Figure 4, where the gray nodes are the slice for node 14 computed by the iterated two-phase slicer of Figure 6. The computation leaves thread 1 at node 13 towards node 9 and later returns to node 15 via the interference edge from node 7. Obviously, node 15 cannot influence the slicing criterion 14, because it cannot execute before node 14 (Krinke calls this effect *time travel* (Krinke 1998)). Unlike data or control dependence, interference dependence is not based on a specific execution order of the interfering statements, as the execution order is in general undecidable. Therefore, traversing interference edges transitively can result in time travel similar to the example above.

One approach to analyze if traversing an interference edge corresponds to a valid execution is to keep track of control flow that is necessary for a valid execution. To this end, every visited node is annotated with a *state tuple* Γ containing the last visited node for each thread with respect to the path taken from the slicing criterion to the currently visited node: Initially, the state tuple of the slicing criterion s contains s itself as the state of the thread of s, and all other threads are mapped to an initial (i.e. nonrestrictive) state $\perp$, as they have not been visited yet. Following each backward traversal of an edge $(m \to n)$, m is annotated with a copy of n's state tuple, where the entry for m's thread is replaced by m.

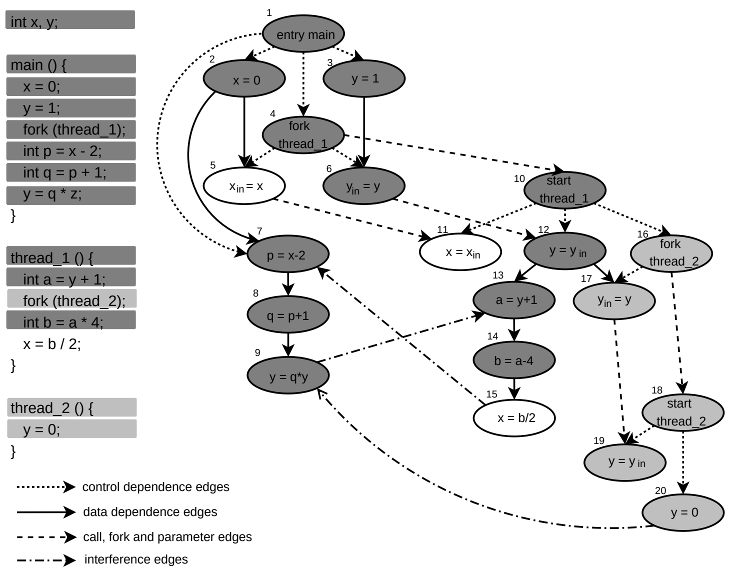
These annotations allow detection of invalid interference edge traversals: If the slicing algorithm is about to traverse an interference edge $q \to_{id} m$ towards q in thread t, and q_{old} is the state of t in m's state tuple, then it is compulsory that q may reach q_{old} in the CFG, or else the traversal forms an invalid execution and must therefore be rejected.

In our example of Figure 4, this situation arises when the algorithm traverses from node 7 to node 15. But `thread_1` had previously been left via interference dependence from node 13. Hence the algorithm needs to check whether it is possible that node 13 is reachable from node 15 in the CFG, which is not the case. Thus this traversal would result in an invalid execution order and is rejected.

This approach is still imprecise as the calling contexts of the previously visited nodes need to be taken into account. Both Nanda's and Krinke's algorithms improve precision with calling contexts. In the remainder, we use the term *context* for a node and its calling context.

To remain sound, the described algorithms need to model every thread that might exist at runtime in thread state tuples. For languages like Java, where threads are created dynamically, one needs to distinguish different instances of the same thread type that exist at runtime. For that purpose, every possible thread instance has an entry in the thread state tuples, which requires a way to cope with thread generation inside loops or recursive procedures. Krinke's algorithm assumes that the number of thread instances is finite, whereas Nanda's algorithm approximates infinite numbers of thread instances resulting from thread generation inside loops, but not inside recursive procedures. For such threads a user has to provide an upper bound of the number of instances.

Müller-Olm and Seidl have shown that precise slicing of concurrent interprocedural programs is undecidable (Müller-Olm and Seidl 2001). Basically, if two nodes n and m are interference dependent $n \to_{id} m$ due to some variable v, then it is not decid-



```
int x, y;

main () {
  x = 0;
  y = 1;
  fork (thread_1);
  int p = x - 2;
  int q = p + 1;
  y = q * z;
}

thread_1 () {
  int a = y + 1;
  fork (thread_2);
  int b = a * 4;
  x = b / 2;
}

thread_2 () {
  y = 0;
}
```

Fig. 7 Thread regions enable a more precise analysis

able whether another statement *s* that redefines *v* may execute between *n* and *m* (i.e. *s* is a *killing definition*). This results from the conservative assumption that scheduling is non-deterministic, which abstracts from the exact scheduling algorithm. Therefore slicing of concurrent interprocedural programs may only be precise up to killing definitions for interference dependences. We refer to this as precise slicing for concurrent programs.

3.2 The impact of concurrency models on precise slicing

Section 2 described the different models of concurrency used by Nanda and Krinke. They affect the precision of slicing concurrent programs in two distinct ways: The first and obvious way is that a more precise model results in less interference edges in the cSDG. Consider again the example program in Figure 1. If Krinke's model of concurrency is used, then `x = a` and `y = a` in thread 0 are interference dependent from `a = 0` in thread 1. But using Nanda's model of concurrency, the interference between `x = a` and `a = 0` can be removed, because `x = a` is executed before thread 1 is invoked. If synchronization information were consulted, like in the right hand graph of Figure 2, one could even prune the interference between `y = a` and `a = 0`, because `y = a` must be executed before `a = 0` due to synchronization.

Concurrency models affect precision also in a second, more subtle way during the computation of a slice. As introduced above, state tuples are used to keep track of the threads' execution states. In fact they work on the level of thread regions and contain one element per thread region. If the entry of a thread region *p* is to be updated to node *n*, then all entries of thread regions that are guaranteed to execute sequentially to *p* are assigned the same value n. Note that when using Nanda's model of concurrency this may result in a thread region r being mapped to a node of a different thread, if r and p are of different threads but execute sequentially to each other. In that case the mapping from r to n signals that the thread to which r belongs either has not been started yet, or it already has finished execution (because parallelism is computed on the level of fork and join points). We show at the example in Figure 7 that this information allows detection of more time travel situations. The set of shaded nodes marks the slice for node 14 with Krinke's model of concurrency, its slice with Nanda's model consists of the dark gray nodes only. Using Nanda's model, one can 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. Since thread 2 is started after node 13, this would require time travel. With Krinke's model of concurrency, where one thread is represented by a single thread region and all threads are entirely parallel, one cannot detect that time travel. Note, however, that the interference edge $20 \rightarrow_{id} 9$ cannot be removed from the cSDG! For slicing criteria other than node 14 its traversal might be valid.

3.3 The Algorithms of Nanda and Krinke

Both Nanda's and Krinke's algorithms can be viewed as extensions of the iterated two-phase slicer. They iterate a slicing algorithm which does not traverse interference edges while determining which interference edges are valid for traversal (fork and fork-in edges are treated like call and parameter-in edges). A precise concurrent slice is achieved due to keeping track of contexts for thread region states and contexts at which the slicing algorithm leaves and enters threads. To this end, their algorithms apply slicing based on contexts instead of nodes. The slicers that ignore interference are called with a context c and its state tuple Γ as slicing criterion and return its interference-free slice $\bar{S}(c)$ and the set I of visited pairs of contexts and state tuples where a thread can be left via interference edges. Similar to the iterated two-phase-slicer, these *interference-free slicers* are called iteratively for every pair of context and state tuple that is reached via a valid interference traversal.

Figure 8 shows the basic structure of both algorithms: First all possible contexts C of the slicing criterion node s are determined. Then they annotate each context $c \in C$ with an initial state tuple Γ, where the execution states of the thread region t of c and of all regions sequential to t are set to c and the states of the other regions are set to an initial context $\perp$: Every interference edge traversal towards a region in the initial execution state is valid by definition. These annotated contexts are inserted into a worklist W. Now the algorithms iterate over every element (c,Γ) of W and compute its interference-free slice $\bar{S}$ and the set I of visited pairs (i,Γ_i) of contexts i and state tuples Γ_i with incoming interference edges. Then they compute the valid interference edges: For each pair $(i,\Gamma_i) \in I$ they determine the set of valid contexts C_m of m for each incoming interference edge $m \rightarrow_{id} n$, where n is the node of context i. A context c_m of m is considered valid if c_m may reach the context that is saved as the state of c_m's thread region in Γ_i in the CFG. If c_m is valid, it is annotated with an updated state tuple Γ_m, where the states of c_m's thread region t_m and of all regions sequential to t_m

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

let $\bar{C}(n)$ return all possible contexts for node n
let $\theta(n)$ return the thread region of node n
let NewState$(\Gamma, c, t)$ return a new state tuple $\Gamma'$ by mapping thread region t and every region sequential
to t in state tuple $\Gamma$ to context c
let IFSlice$(c, \Gamma)$ return the interference-free slice $\bar{S}$ for context c and state tuple $\Gamma$
and the set I of visited pairs of contexts and state tuples with incoming interference edges

/* Initialize the worklist W with an initial state tuple and mark its contents */
$\Gamma_0 = (\bot, ..., \bot)$ // every thread region is in an initial state
$W = \{(c, \Gamma) \mid t = \theta(s) \wedge c \in \bar{C}(s) \wedge \Gamma =$ NewState$(\Gamma_0, c, t)\}$
$M = \{s\}$ // a list for marking the contents of W

repeat
    $W = W \setminus \{(c, \Gamma)\}$ // remove next element $(c, \Gamma)$ from W

    /* Compute a interference-free slice $(\bar{S})$ for $(c, \Gamma)$ and the set I of visited
    pairs of contexts and state tuples with incoming interference edges */
    $(\bar{S}, I) =$ IFSlice$(c, \Gamma)$
    $S = S \cup \bar{S}$

    /* Compute valid interference edges */
    foreach $(i, \Gamma_i) \in I$
        foreach $m \to_{id} n \mid n \text{ is node of context } i\}$
            $t_m = \theta(m)$    // the thread region we want to enter

            /* Compute the valid contexts of m */
            $C_m = \{c_m | c_m \in \bar{C}(m) \wedge c_m \text{ reaches the state of } t_m \text{ in } \Gamma_i\}$

            foreach $w \in \{(c_m, \Gamma_m) | c_m \in C_m \wedge \Gamma_m =$ NewState$(\Gamma_i, c_m, t_m)\}$
                if $w \notin M$
                    $W = W \cup \{w\}$
                    $M = M \cup \{w\}$

until $W = \emptyset$
return $S$
```

Fig. 8 Slicing concurrent programs precisely

are mapped to c_m and the other regions are mapped to the same contexts as in Γ_i, and is inserted into worklist W. The resulting slice is the union of all slices $\bar{S}$.

Note that in the outlined algorithm, a context can be visited multiple times, as long as its annotated state tuples differ. Thus this approach has a worst-case runtime complexity exponential to the number of threads that exist at runtime.

Krinke describes his slicing algorithm in detail in (Krinke 2003). Nanda describes two versions of her algorithm in (Nanda and Ramesh 2006); a version with cobegin-coend parallelism and a version with fork-join parallelism suitable for Java. Our evaluation is based on dependence graphs for Java, so we only refer to the latter.

3.4 Differences between the algorithms

Krinke's algorithm represents a context of a node with a *call string* (Sharir and Pnueli 1981). A call string for a node n is a sequence of procedure calls leading to the procedure that contains n. Contexts are computed dynamically during slicing. Krinke leverages a slightly modified version of his *explicitly context-sensitive slicer* (ECSS) (Krinke 2002). This algorithm for sequential programs is not based on summary edges but call strings to gain context-sensitivity. In the evaluation of the explicitly context-sensitive slicer, Krinke concludes that it is significantly more expensive than two-phase slicing with summary edges due to combinatorial explosion of call strings (Krinke 2002). The modified version he uses for concurrent programs returns an interference-free slice for a given slicing criterion, and the set of visited contexts where a thread can be left via interference edges.

A similar approach determines if a context can reach another one in the CFG. It uses the contexts' call strings to traverse the control flow graph in a context-sensitive manner. To avoid infinite stacking of call strings, Krinke folds cycles in control flow graphs that result from loops and recursion. His folding algorithm preserves context-sensitivity and works in two phases (Krinke 2003): First, it folds all cycles consisting of call- and control flow edges, then it folds the remaining cycles that consist of return- and control flow edges. Thus it does not fold procedures that are called from within a cycle, i.e. loops are only folded intra-procedurally, and in recursive cycles only the recursive procedures are folded. Figure 9 shows an example. It consists of two procedures, `foo` and `bar`, where `foo` calls `bar` once outside a loop and once inside a loop, and `bar` calls itself recursively. The empty nodes are the folded nodes.

Nanda uses a special folding method for cycles in control flow graphs to create *interprocedural strongly connected regions* (ISCR) graphs, which are completely free of cycles. This allows enumeration of the remaining contexts topologically in reverse preorder, such that contexts are represented by single integers. For that purpose instances of procedures that are called (transitively) from within a recursive or loop-based cycle are included into the cycle (called *virtual inlining* by Nanda (Nanda and Ramesh 2006)). This results in stronger folding than in Krinke's algorithm and thus in a smaller number of contexts. Nanda's reachability analysis is a traversal of the ISCR graph that uses the context enumeration to preserve context-sensitivity. Figure 9 shows how Nanda and Krinke fold the same CFG. The roman numbers represent the topological enumeration. Note that Nanda's virtual inlining duplicates the nodes of the inlined procedures, whereas Krinke's folding algorithm preserves unique nodes. For example, Nodes III and VII in Figure 9 both contain the CFG nodes 10 and 11.

The interference-free slicer in Nanda's algorithm is a modified two-phase slicer based on contexts instead of nodes. Unlike Krinke's algorithm, it does not compute contexts itself but queries 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 retrieved from the ISCR graph. Then reachability analysis on the ISCR graph determines every context $c' \in C'$ that reaches c. The slicer proceeds with these contexts c'. Nanda's slicer performs that reachability analysis upon each edge traversal, which can be a bottleneck in graphs with many contexts (see section 5).

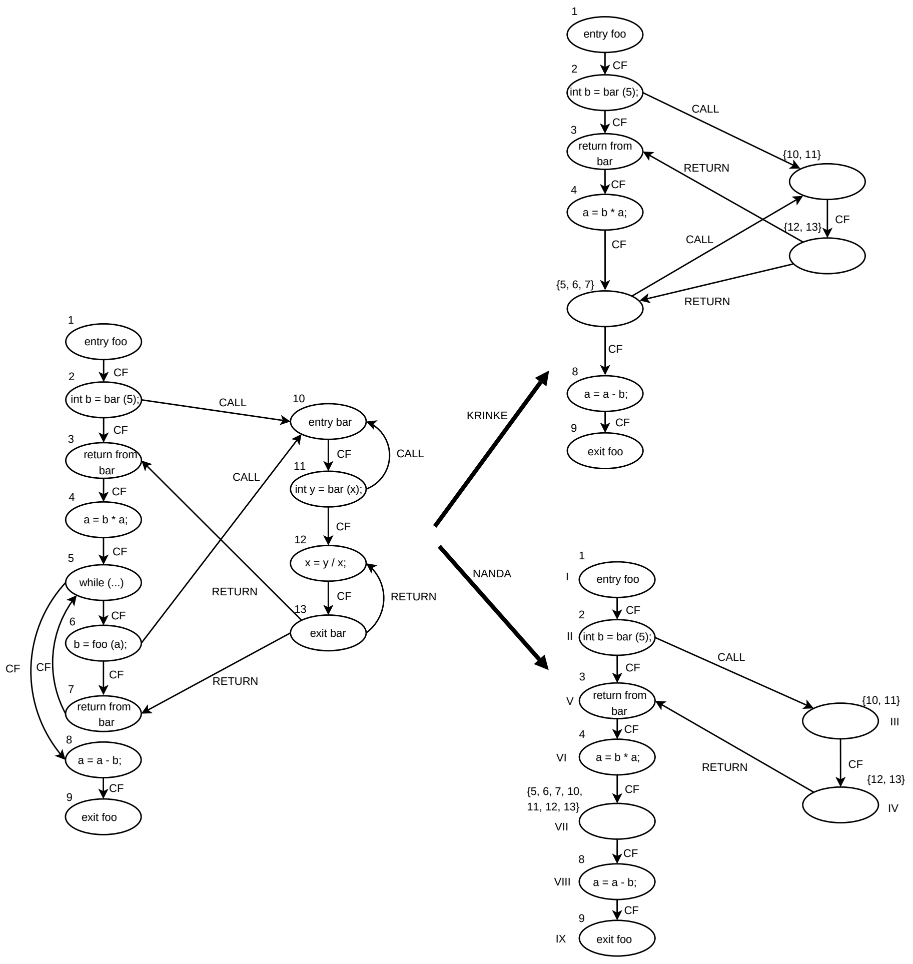


Fig. 9 Folding a CFG with Krinke's and Nanda's method

Nanda's 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. Since now every node of thread t has the same context, every interference edge traversal towards an instance of t during the slice computation is valid according to the reachability analysis. Her approach does not handle dynamic thread generation inside of recursive cycles. Krinke's algorithm does not handle dynamic generation of threads inside of loops or recursion.

Nanda identifies combinatorial explosion of thread state tuples to be a major performance problem and defines *restrictive state tuples* as a remedy. Let $[c_1, ..., c_n]$, $[c'_1, ..., c'_n]$ be two state tuples. If $\forall i \in 1, ..., n : c'_i$ *reaches* c_i, 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

```
int x;

main () {
  foo();
  x = 1;
  bar();
}

void foo() {
  int a = 1;
  a = m(a);
}

void bar() {
  int b = m(0);
  b = b + x
}

void m(int a) {
  x = x + a;
  a = a + x;
  x = 1;
  return a;
}

thread_1 () {
  int z = x;
  x = z + 1;
}
```

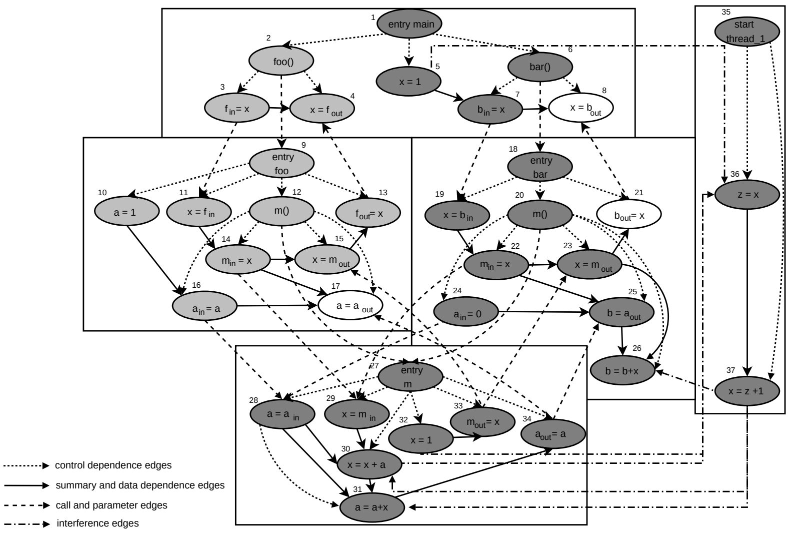


Fig. 10 Incorrect slice computed by Nanda's algorithm

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), because t' imposes more restrictions on the set of valid interference edges than t does. This property allows identification of redundant context pairs and state tuples: When a dependence edge e is traversed towards 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 each edge traversal in the cSDG.

3.5 Correctness

Unfortunately, it seems that Nanda's algorithm may compute incorrect slices, which we show here at an example. The basic algorithm is correct, but it applies the restrictive state tuple optimization after each edge traversal, which might prune valid interference edges. As mentioned before, her algorithm uses a modified two-phase slicer, where a worklist W1 is used for the first phase and a worklist W2 for the second phase. This slicer is iterated inside a while-loop, which works on an outer worklist W0: W0 is equivalent to worklist W in figure 8. It further employs a map M, which maps every visited context to the thread state tuples it has been annotated with. To detect restrictive state tuples, it checks if the thread state state tuple in question is restrictive to any thread state tuple to which the according context is mapped to in M. Figure 10 contains our example program. The program consists of two threads - the main thread and thread_1. 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 denote every context of a node with the node itself, nodes of `m` will be appended 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: The outer 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.

Start: The first element of W0, (26, [26, $\perp$]), is taken out and inserted into W1, the worklist for phase 1 of her interference-free slicer.

First interference-free slice for (26, [26, $\perp$]): In phase 1, the algorithm visits 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}, $\perp$]), (33_{foo}, [33_{foo}, $\perp$]), (34_{bar}, [34_{bar}, $\perp$]) and (34_{foo}, [34_{foo}, $\perp$]) are visited and inserted in worklist W2 (the worklist for phase 2 of her interference-free slicer). 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}, $\perp$]) and (n_{foo}, [n_{foo}, $\perp$]) in W2. 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. The elements (37, [31_{foo}, 37]) and (37, [30_{foo}, 37]) are also visited, but are discarded because of restrictive state tuples (*foo* can reach *bar*).

The next element of W0, (37, [26, 37]), is taken out and inserted into W1.

Second interference-free slice for (37, [26, 37]): In phase 1, the algorithm visits nodes {37, 36, 35}. 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, since 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 elements (30_{foo}, [30_{foo}, $\perp$]) and (30_{bar}, [30_{bar}, $\perp$]), respectively, because 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 and thus the algorithm omits nodes that belong to the slice.

Although this problem is hard to detect, it can be fixed unproblematic. The optimization of restrictive state tuples is only applied when traversing interference edges, and only contexts and thread state tuples that are reached via an interference edge traversal are inserted into the map M.

4 Further development

During our work, we developed and applied several further optimizations to both algorithms. We extended both algorithms to handle dynamic thread invocation inside of loops and recursion. We applied an optimization to Nanda's algorithm that eliminates reachability analysis after each traversal of an intra-procedural dependence edge. During ISCR graph construction, we annotate every context with an ID of the method invocation it belongs to. Then, after traversing an intra-procedural 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 invocation ID as the current context. For Krinke's algorithm we adopted Nanda's more precise model of concurrency and the restrictive state tuple optimization. We further applied an optimized reachability analysis and identified and removed redundant invocations of the interference-free slicer. We now describe several of these optimizations in detail. At the end of this section we present a pseudo-code style description of our optimized version of Krinke's algorithm.

4.1 Applying Nanda's model of concurrency

Krinke's model of concurrency allows 'lazy' updates of state tuples, because a thread region corresponds with exactly one thread. It suffices to update state tuples only when traversing interference edges. Thus, his interference-free slicer totally ignores state tuples. When using Nanda's model of concurrency, thread states need to be updated as soon as a thread region is entered or left. For simplicity this is approximated by updating thread states after each edge traversal. To apply Nanda's model to Krinke's algorithm, his interference-free slicer is adjusted accordingly. Figure 11 shows a modified version of that algorithm (Krinke 2002) based on thread regions, and Figure 12 shows how to update state tuples for thread regions. Note that these pseudo-code algorithms represent contexts as pairs of nodes and call strings.

4.2 Thread invocation inside of loops and recursion

A simple way to handle threads that are dynamically invoked inside of loops and recursion is to give an upper bound for the number of invocations. A user of our system can do that by annotating threads with the number of instances of that thread that exist at runtime. But often such an upper bound is not known. We use the following conservative approximation to handle such threads: Both Nanda's and Krinke's algorithm initially give threads an initial execution state. A thread with an initial execution state is by definition always reachable via an interference edge. We now assume conservatively that threads that are invoked inside of loops or recursion have an infinite number of instances, so every traversal of an interference edge towards such a thread is able to find an instance that is in the initial execution state: The algorithm can simply omit the reachability analysis when it traverses towards such a thread. Furthermore, these threads do only need one entry in the thread state tuples that represents their (infinite number of) instances in the algorithm. Note that the iterated two-phase slicer uses this conservative approximation implicitly for all threads, because it treats every interference edge as valid, and is thus able to handle dynamic thread invocation as well. To determine conservatively the number of possible instances of a thread in a program we employ Ruf's *thread allocation analysis* for Java (Ruf 2000).

Input: The SDG G of a thread, a slicing criterion (s, c_s, Γ_s),
where s is a node, c_s a call string and Γ_s a state tuple for thread regions.
Output: The slice S for s and a set $I = \{(i, c_i, \Gamma_i)\}$ of tuples (node, call string, states) where a thread can be left via an interference edge.

```
$W = \{(s, c_s, \Gamma_s)\}$ // a worklist
repeat
    $W = W \setminus \{(n, c, \Gamma)\}$ // remove next element from W
    $S = S \cup \{n\}$
    foreach $m \to_e n$ // consider all incoming edges of n
        if $e \in \{id\}$
            $I = I \cup \{(n, c, \Gamma)\}$ // for interference edges, save (n, c, Γ) in I

        else if $e \in \{pi, c, fork, fork\text{-}in\}$ // ascend to a method callee
            Let $c_m$ be the call site to which m belongs
            if top(c) = $c_m$ // this test guarantees context-sensitivity
                $c'$ = tail(c) // tail(c) returns the substring of c behind top(c)
                if m has not been marked with $c'$
                    $\Gamma'$ = update(m, $c'$, $\Gamma$) // update thread states (Figure 12)
                    $W = W \cup \{(m, c', \Gamma')\}$
                    mark m with $c'$
                if $c_m$ is marked as recursive and m has not been marked with c
                    // For recursive calls, we additionally have to conserve call string c
                    $\Gamma''$ = update(m, c, $\Gamma$)
                    $W = W \cup \{(m, c, \Gamma'')\}$
                    mark m with c

        else if $e \in \{po\}$ // descend into a called method
            Let $c_n$ be the call site of n
            if $c_n$ is recursive and top(c) = $c_n$ and m has not been marked with c
                // We are in a recursive cycle – the old call string is the new one
                $\Gamma'$ = update(m, c, $\Gamma$)
                $W = W \cup \{(m, c, \Gamma')\}$
                mark m with c
            else
                $c'$ = push(c, $c_n$) // put call site $c_n$ on top of c
                if m has not been marked with $c'$
                    $\Gamma'$ = update(m, $c'$, $\Gamma$)
                    $W = W \cup \{(m, c', \Gamma')\}$
                    mark m with $c'$

        else // intra-procedural edges – call string c does not change
            if m has not been marked with c
                $\Gamma'$ = update(m, c, $\Gamma$)
                $W = W \cup \{(m, c, \Gamma')\}$
                mark m with c

until $W = \emptyset$
return $(S, I)$
```

Fig. 11 ecss: A Krinke-style interference-free slicer that works with thread regions

Input: A node n, a call string c and a state tuple Γ.
Output: A state tuple Γ'.

```
Let tr be the thread region of n
$\Gamma' = [(n,c)/tr]\Gamma$  // create a copy of $\Gamma$ and set tr's state to $(n,c)$
for all thread regions $tr'$ that do not execute in parallel to tr
    $\Gamma' = [(n,c)/tr']\Gamma'$  // set the value of $tr'$s entry in $\Gamma$ to $(n,c)$

return $\Gamma'$
```

Fig. 12 update: Updating thread state tuples

4.3 An optimized reachability analysis

Both Nanda and Krinke perform a thread-local reachability analysis based on the information provided by thread regions. When reaching a context c via an interference edge, one needs to check if c reaches the current state s of c's thread region r. There are three cases for s in the CFG due to the update mechanism described in Figure 12:

1. s is of the same thread as region r.
2. s is of another thread and lies before the fork point of r's thread.
3. s is of another thread and lies behind the join point of r's thread.

Only in the first case one has to commit a reachability analysis from c to s on the CFG. In the second case c cannot reach s, because r's thread has not been invoked. In the third case c reaches s, because c reaches the join point of its own thread.

Whereas Nanda describes her reachability analysis in detail, Krinke does not give a detailed description. He shows that a reachability analysis that uses call strings like in the algorithm of Figure 11 remains context-sensitive, but still there are different ways to implement it. We first present two intuitive ways to analyze reachability, a ‘brute-force’ and a guided traversal. Then we present a third approach that effectively reduces the reachability analysis to an intra-procedural traversal. In the following three descriptions we aim for checking if we can reach a context *target* from a context *source* in a CFG C.

4.3.1 A brute-force approach

This approach is a simple traversal of all context-sensitive paths in C starting from *start* to see if one finally reaches *target*. Call string information of *start* is used and modified when entering or leaving a method to remain context-sensitive, similar to the algorithm in Figure 11.

4.3.2 A guided traversal

The brute-force approach totally ignores that more information is available than just the call string of *start*. We can use the call string of *target* to check if entering a method brings us closer to *target*: When entering a method, the resulting call string must be a prefix of the call string of *target*, or else we do not need to enter that method.

4.3.3 An intra-procedural approach

The guided approach reduces the number of edge traversals significantly, compared to the brute-force approach. But it is not even necessary to leave and enter methods. Consider the following example:

```
void main() {
   foo();
   skip;
   bar();
}
void foo() {
   int i = 0;
}
void bar() {
   int j = 1;
}
```

Let *start* be the instance of `int i = 0;` and *target* be the instance of `int j = 1;`. The call string of *start* consists of the call $foo()$, the call string of *target* consists of $bar()$. It is easy to see that *start* reaches *target*. Both previously described approaches perform the following 3 steps to reach *target*: First, they traverse from *start* to the intra-procedural successor `skip` of call `foo()`. Then they traverse from `skip` to `bar()`. And in the last step they traverse from `bar()` to *target*.

These three steps are performed in every successful reachability analysis: In the first step, the CFG is traversed from *start* back to that method where the call strings of *start* and *target* start to diverge. We call this method the *last shared method*. In the second step, the analysis traverses intra-procedurally from the successor of the call that leads to *start* to the call that leads to *target*. And in the third step it traverses from the latter call to *target*. But the first and third step are redundant: The first step always succeeds, since a call string can always be decomposed and since there always exists a last shared method (as all methods are reachable from the main method). The third step always succeeds, too: The call string of *target* shows that there is a valid path from its call in the last shared method to *target*. So only the second step requires actual checking. This leads to the following reachability algorithm:

1. Determine the longest call string *pre* that prefixes both *source* and *target*. The topmost element of *pre* is the call of the last shared method.
2. Determine the calls in the call strings of *source* and *target* that directly follow the prefix. These are the calls in the last shared method that lead to *source* and *target*. We call them *sourceCall* and *targetCall*, respectively.
3. Perform an intra-procedural reachability analysis from the direct intra-procedural successor of *sourceCall* to *targetCall* in the last shared method. If it is successful, then *source* reaches *target*, otherwise not.

To handle fold nodes that arise from Krinke's CFG folding, some modifications are required. Consider the following cases:

– The prefix *pre* contains a fold node.
  In this case the reachability check will always succeed: *start* and *target* are in

```
Input: The CFG $C$ and two contexts source and target, where source is a context
            that is reached via an interference edge, and target is the current state of
            source's thread region.
Output: 'true', if source reaches target in $C$, else 'false'.

let $t_s$ be the thread of source
let $t_t$ be the thread of target

if $t_s \neq t_t$ // their threads are different
  if target lies behind the join point of $t_s$
    return true

else // they have the same thread
  if the thread of target is created inside of a loop or recursion
    return true
  else if $target = \perp$ // the initial state
    return true
  else
    /* Else we do our reachability analysis. */
    pre = prefix(source, target) // compute the longest common prefix

    if pre contains a fold node
      return true

    // Determine the successors of pre in source and target
    sourceCall = source[pre.length]
    targetCall = target[pre.length]
    sourceSucc = the intra-procedural successor of sourceCall

    if sourceSucc reaches targetCall intra-procedurally in C
      return true

return false
```

Fig. 13 reach: Reachability analysis

methods that are called (transitively) from the same fold node, which means that *start* and *target* can reach each other.

– One of *sourceCall* or *targetCall* is a fold node (or even both).
  This case does not influence our algorithm, since *start* and *target* are not called from a common fold node. *source* reaches *target* iff *sourceCall* reaches *targetCall* in the last shared method.
– None of the previous cases applies.
  Our algorithm is not influenced, since *start* and *target* are not called from a common fold node.

Hence we have to modify the first step of our algorithm as follows:

1'. Determine the longest call string *pre* that prefixes both *source* and *target*. If *pre* contains a fold node, then *source* reaches *target*. Else proceed with step 1.

This leads to our reachability analysis shown in Figure 13.

Table 1 Average runtimes for 1000 reachability-analysis invocations (in seconds)

	SmallExample	Dijkstra	Matrix-Multip.	JavaCard Wallet
nodes	85	2927	3346	23340
edges	226	8837	20706	109360
Brute	.077	.322	.358	1049.194
Guided	.011	.077	.112	2.435
Intraproc	.017	.003	.014	.279

4.3.4 Case study

We have implemented these three approaches in Java and compared their runtime behavior. Table 1 shows a small case study containing four sequential programs. It shows the average execution times in seconds that our three approaches need for 1000 reachability analyses. Our intra-procedural approach is by far the fastest. More fine-grained measurements show that its runtime mainly depends on the structure and size of the program methods.

4.4 Avoiding redundant interference-free slices

In both Krinke's and Nanda's algorithms, contexts are visited multiple times when their thread state annotations differ. Thus the interference-free slicers are often called repeatedly for the same context c. But as can be seen in Figure 11, Krinke's interference-free slicer works independently of thread state tuples – they are just updated and propagated. Calling that algorithm with a context k and corresponding state tuple Γ_k returns an interference-free slice $\bar{S}(k)$ and a set $I(k)$ of tuples (i, Γ_i) of contexts i and their thread state tuples Γ_i with incoming interference edges. Of course Γ_i depends on Γ_k. But because the interference-free slicer itself is independent of the state tuples, $\bar{S}(k)$ and the changes $\Delta\Gamma_i$ that are done to the state tuple Γ_k when the slicing algorithm reaches i are always the same, regardless of Γ_k. If we are able to cache these changes, we can avoid calling that slicer more than once for the same context.

To get those $\Delta\Gamma_i$, the algorithm in Figure 11 is called with context k and a 'dummy' state tuple where all entries have the initial value $\perp$. During graph traversal, this state tuple is updated, thus for every element (i, Γ_i) in the resulting set $I(k)$ the state tuple Γ_i contains the state tuple changes: $\Delta\Gamma_i = \Gamma_i$. We obtain the real state tuple Γ_i^k of context i by merging Γ_k and $\Delta\Gamma_i$: First, Γ_k is copied to Γ_i^k and then all entries in $\Delta\Gamma_i$ that do not have the initial value $\perp$ are copied to Γ_i^k. Figure 14 shows the merging, Figure 15 contains this optimization.

4.5 Our resulting algorithm

Figure 15 shows pseudo-code for our variant of Krinke's algorithm that contains the described optimizations. It is basically an extended version of the algorithm in Figure 8. It calls the algorithms of Figures 11 – 14. We do not present pseudo-code for several functions which we just adopted. Among them are the computation of all

Input: Two thread state tuples, Γ_k and Γ_i.
Output: The merged thread state tuple Γ_i^k.

```
$\Gamma_i^k = \Gamma_k$ // initialize $\Gamma_i^k$ with $\Gamma_k$

// Copy every non-initial context to $\Gamma_i^k$
foreach $(c, tr) \in \Gamma_i$ // c is the context at thread region tr in $\Gamma_i$
    if $c \neq \perp$
        map tr to c in $\Gamma_i^k$

return $\Gamma_i^k$
```

Fig. 14 merge: Merging thread state tuples.

contexts of a node and the folding algorithms for CFG-cycles, which are described in Krinke's and Nanda's work (Krinke 2003; Nanda and Ramesh 2006), the computation of thread regions that is described by Nanda (Nanda and Ramesh 2006) and Ruf's thread allocation analysis (Ruf 2000).

5 Implementation and evaluation

We further examined two imprecise algorithms in our evaluation. The first algorithm is the iterated two-phase slicer, presented in Figure 6. The second algorithm trades precision for speed by using nodes instead of contexts to mark the thread execution states, which is the algorithm described at the end of section 3.1. It uses Krinke's model of concurrency and applies Nanda's optimization of restrictive state tuples after each interference edge traversal. Its code is similar to the iterated two-phase slicer, except for the code for the restrictive state tuple optimization and is shown in Figure 16. We are not aware of any previous work describing such an algorithm.

We used the following algorithms in 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 4, (ON), our modification of Krinke's algorithm using Nanda's model of concurrency (GK) and Krinke's model of concurrency (GK*), the imprecise slicer described in Figure 16 (S), and the iterated two-phase slicer (I2P). All these algorithms are implemented in Java and operate on dependence graphs computed by Hammer's dataflow analysis for Java programs (Hammer and Snelting 2004) extended with Ruf's thread allocation analysis (Ruf 2000) to analyze conservatively how many instances of every thread exist at runtime. For the tests, we used a uniprocessor 2.2Ghz AMD 3200+ workstation with 2GB of memory running Fedora 2.6.16 Linux. Table 2 summarizes the features of these algorithms.

5.1 Precision and runtime behavior

Our main focus in the evaluation was to examine the precision and runtime-behavior of our implemented algorithms. Table 3 shows the programs that we used in our case

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

```
let $\bar{C}(n)$ return all possible contexts $(n,c)$ for node n, where c is a call string
let $\theta(n,c)$ return the thread region of context $(n,c)$

/* Create worklist W and insert all possible contexts of s,
annotated with state tuples. */
$\Gamma = (\bot, ..., \bot)$ // an initial state tuple
$W = \{(s,c,\Gamma') \mid t = \theta(s) \wedge (s,c) \in \bar{C}(s) \wedge \Gamma' = update(s,c,\Gamma)\}$

$M = \{w \in W\}$ // a list for marking the contents of W
$M' = \{\}$ // a set for marking contexts
$Cache = \{\}$ // a map

repeat
    remove next element $w = (n,k,\Gamma_n)$ from W

    /* Compute interference-free slice $\bar{S}(n,k)$ and the visited contexts with incoming
     interference edges $I(n,k)$. Use the optimization of section 4.4. */
    if $(n,k) \notin M'$
        $(\bar{S}, \Delta_{(n,k)}) = ecss(n,k,(\bot,...,\bot))$ // call Figure 11 with a dummy state tuple
        $S = S \cup \bar{S}$ // update the slice
        $Cache = Cache \cup \{(n,k) \to \Delta_{(n,k)}\}$ // update the cache
        $M' = M' \cup \{(n,k)\}$ // mark context $(n,k)$
    else
        $\Delta_{(n,k)} = Cache.get((n,k))$ // get the state tuple changes for $(n,k)$
    $I =$ merge($\Delta_{(n,k)}$, $\Gamma_n$) // compute the real state tuples

    /* Compute valid interference edges */
    foreach $(i,k_i,\Gamma_i) \in I$
        foreach $m \to_{id} i$
            $t_m = \theta(m)$ // $t_m$ is the thread region of m

            /* Compute the contexts of m that reach the current state of $t_m$. */
            $C_m = \{(m,k_m) | (m,k_m) \in \bar{C}(m) \wedge$ reach($(m,k_m)$, $\Gamma[t_m]$)$\}$

            /* Update worklist W. */
            foreach $(m,k_m) \in C_m$
                $\Gamma_m =$ update($m$, $k_m$, $\Gamma_i$)

                /* Check if $(m,k_m,\Gamma_m)$ has not been visited yet,
                 and if $\Gamma_m$ is not a restrictive state tuple. */
                if $(m,k_m,\Gamma_m) \notin M \wedge \nexists (m,k_m,\Gamma'_m) \in M : \Gamma_m$ is restrictive to $\Gamma'_m$
                    $W = W \cup \{(m,k_m,\Gamma_m)\}$
                    $M = M \cup \{(m,k_m,\Gamma_m)\}$

until $W = \emptyset$
return $S$
```

Fig. 15 An optimized version of Krinke's algorithm

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

```
Let $\theta(n)$ return the thread of node n

$\Gamma_0 = (\bot, ..., \bot)$    // initial state tuple
$\Gamma_s = [s/\theta(s)]\Gamma_0$   // set the state of $\theta(s)$ to s
$W = \{(s, \Gamma_s)\}$    // a worklist
$R = \{(s, \Gamma_s)\}$    // a set for marking state tuples
$M = \{s \mapsto true\}$  // a map for marking the contents of W
                          // (true represents phase 1, false phase 2)

repeat
    $W = W \setminus \{(n, \Gamma)\}$ // remove next element from W
    foreach $m \rightarrow_e n$ // handle all incoming edges of n
        if $e = id$ // e is an interference edge
            $\Gamma_n = [n/\theta(n)]\Gamma$ // save where n's thread is left
            if $m$ reaches node to which $\theta(m)$ is mapped in $\Gamma'$ // reachability analysis
                $\Gamma_m = [m/\theta(m)]\Gamma_n$ // save where m's thread is entered
                if $\nexists (m, \Gamma'_m) \in R : \Gamma_m$ is restrictive to $\Gamma'_m$
                    // If the state tuple is not restrictive, update the sets and maps
                    $W = W \cup \{(m, \Gamma_m)\}$
                    $R = R \cup \{(m, \Gamma_m)\}$
                    $M = M \cup \{m \mapsto true\}$

        // If m wasn't visited yet or we are in phase 1 and m was visited in phase 2,
        elseif $m \notin \text{dom}\, M \vee M(n) \wedge \neg M(m)$
            // if we are in phase 1 or if e is not a call or param-in edge, add m to W
            if $M(n) \vee e \notin \{pi, c\}$
                $W = W \cup \{(m, \Gamma\}$ // inside of one thread, we can propagate the old states

                // If we are in phase 1 and e is a param-out edge, mark m with phase 2
                if $M(n) \wedge e = po$
                    $M = M \cup \{m \mapsto false\}$
                // Else mark m with the same phase as n
                else
                    $M = M \cup \{m \mapsto M(n)\}$

until $W = \emptyset$
return dom $M$
```

Fig. 16 A slicer that uses nodes as contexts

Table 2 Table of features

Name	Precise	Conc. Model	Dynamic Threads	Description
N	yes	Nanda	loops only	Nanda's original algorithm with remedy described in section 3.5
ON	yes	Nanda	yes	N with optimization of section 4
K	yes	Krinke	no	Krinke's original algorithm
GK*	yes	Krinke	yes	algorithm of Figure 15
GK	yes	Nanda	yes	algorithm of Figure 15
I2P	no	Krinke	yes	ignores time travels, alg. of Fig. 6
S	no	Krinke	yes	uses context-insensitive state tuples, algorithm of Figure 16

Table 3 Our tested concurrent programs

Name	Nodes	Edges	Classes	Methods	Thread Types
PrecisionTest	328	904	6	10	2
TimeTravel	413	1136	7	14	2
ProducerConsumer	420	1159	6	10	2
BoundedBuffer	1324	3900	14	25	3
Primes	2906	9693	18	36	2
AlarmClock	4085	13842	17	74	2
LaplaceGrid	10022	100730	22	95	1
SharedQueue	17998	139480	23	122	1

Table 4 Average execution times per slice for concurrent programs (in seconds)

Name, 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
	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
	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
	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
	3	.001	.127	–	.411	.879	.126	.088
Primes	1	.001	.023	3.883	.181	.258	.124	.050
	2	.001	.178	–	5.411	7.916	.692	.416
AlarmClock	1	.003	.099	–	3.641	3.358	.832	.202
	2	.003	.895	–	346.413	281.477	4.430	1.467
LaplaceGrid	1	.007	.323	111.900	1.945	.428	.476	.126
	2	.007	1.868	–	27.836	4.775	2.141	.315
SharedQueue	1	.034	.385	–	49.198	5.052	.566	.309
	2	.034	31.930	–	–	–	15.370	11.334

study. The values for ‘Nodes’ and ‘Edges’ show the number of nodes and edges, respectively, of the dependence graphs, the value for ‘Thread Types’ shows the number of different thread types in the programs (i.e. subclasses of `java.lang.Thread`), and the values for ‘Classes’ and ‘Methods’ show the number of classes and methods that are used in the programs. PrecisionTest and TimeTravel are small programs that model nested thread invocation and potential time travel situations. Producer-Consumer 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 (`http://www.cis.ksu.edu/santos`).

To measure the precise algorithms’ gain, we deactivated the handling of dynamically invoked threads and annotated the threads of our test programs with the number

Table 5 Average number of elements inserted into the worklist per slice

Name, Instances		I2P	S	K	GK*	GK	N	ON
	1	2.4	1.0	25.9	14.6	9.1	4.0	4.0
PrecisionTest	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
	1	1.9	1.0	5.9	5.7	5.4	4.9	4.9
TimeTravel	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
	1	3.2	1.1	24.7	11.1	11.1	10.0	10.0
ProducerConsumer	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
	1	15.3	1.0	360.5	96.7	96.7	90.2	90.2
BoundedBuffer	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

of 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[3]. This created several versions of our eight test programs, resulting in a total of 20 programs. For each program, we computed 100 slices, which were chosen randomly. Table 4 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. Table 5 shows the average number of elements inserted into the outer worklists due to interference edge traversals (worklist W in figure 8). Table 6 shows the average size of the computed slices in number of nodes. The column ‘Instances’ in tables 4, 5 and 6 shows the number of instances of every thread that exist at runtime. For example: BoundedBuffer contains 3 different types of threads, plus the main thread. The column with ‘Instances = 2’ means that the running program contains each two instances of every thread, resulting in 6 threads plus the main thread at runtime.

Table 6 shows that 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 that use Nanda’s model of concurrency, N, ON and GK, are the most precise. The algorithms that use 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 num-

[3] In our cSDGs, a thread entry node is annotated with the number of instances that may be created during program execution. These numbers are automatically determined by our thread allocation analysis mentioned in section 4.2. We manually raised these numbers in our experiments; no modification of the benchmarks was required.

Table 6 Average size per slice (number of nodes)

Name, Instances		I2P	S	K	GK*	GK	N	ON
	1	31.2	27.0	26.6	26.6	24.7	24.7	24.7
PrecisionTest	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

ber of thread instances decreases the benefit of the precise algorithms, whereas the computation times rise significantly: The more thread instances exist, the more interference edge traversals find a thread instance that is in a suitable execution state.

We identify two major issues that influence the performance of the precise algorithms: combinatorial explosion of state tuples and of the context computation and representation in the interference-free slicers. The combinatorial explosion of state tuples directly influences the number of elements inserted into the outer worklist (Table 5). Krinke's original algorithm K suffers from both issues and could only slice our smaller test programs in reasonable time. Table 5 shows that the size of its outer worklists grows very fast when the number of thread instances is raised. Several algorithms (S, N, ON, GK* and GK) use Nanda's restrictive state tuple optimization to ease this combinatorial explosion, which is very effective (Table 5). In Nanda's algorithm, the ISCR graph construction creates fewer contexts than Krinke's cycle-folding algorithm (see section 3.4), further reducing the possible combinations. Another advantage of Nanda's algorithm is its representation of contexts as single integers instead of call strings in Krinke's algorithm. The call string representation is likely to decline performance in bigger programs, because 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 larger programs their performance declines, because they use the weaker folding method and the call site representation for contexts. The S algorithm is less affected by the combinatorial explosion of thread state tuples, because it does not use contexts as thread states. The I2P algorithm is not affected at all since it does not consider thread states nor contexts.

Table 7 summarizes the speed-up gained by our optimizations and the effect of the two models of concurrency, and it compares our two optimized algorithms, GK and ON. It shows the minimum, maximum and the overall speed-up. Our optimizations for Krinke's algorithm provided an overall speed-up of 142.2 times compared to the

Table 7 Speed-up through optimizations and different models of concurrency

Name	Compared to	Min	Max	Overall
GK	K	1	288.9	142.2
GK	GK*	0.5	9.7	1.4
ON	N	1	6.8	1.7
ON	GK	0.8	148.9	21.1

Table 8 Average execution time per slice for sequential programs (in seconds)

	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

original algorithm. In the best case the optimized version was 288.9 times faster (for BoundedBuffer with 2 thread instances). Comparing the runtimes of GK and GK* shows the impact of concurrency models on computation costs. Nanda's more complex concurrency model may increase computation costs if it is not able to gain more precision than Krinke's model, e.g. GK* needs more than twice as much time than GK for BoundedBuffer with 2 thread instances. If it is able to increase precision it may also speed up execution, up to 9.7 times in our evaluation (for SharedQueue). It caused an overall speed up of 1.4. Of all precise algorithms our optimized version of Nanda's algorithm, ON, performed best. The optimization we found for Nanda's algorithm sped up execution 6.8 times in the best case (for LaplaceGrid with 2 thread instances) and 1.7 times for the whole test suite. It computed the slices of our test suite 21.1 times faster than our best version of Krinke's algorithm, GK.

Although Nanda's algorithm performed best in that test suite, it might perform poorly for programs with deep call-chains or high usage of libraries. Both factors affect the cost of her reachability analysis which is computed after every edge traversal in algorithm N and after every traversal of an interprocedural edge in algorithm ON. We made a small case study to observe these factors, which it is shown in Table 8. To eliminate the effects of time travel detection, the case study consists only of sequential programs. We used the two-phase slicer (2P), the iterated two-phase slicer (I2P) and the interference-free 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, because recursive cycles and all procedure calls within a cycle are collapsed into one single node (section 3.4). In highly recursive programs, this can reduce the number of contexts significantly.

Nanda provides an evaluation for her algorithm (Nanda and Ramesh 2006); however, it is difficult to compare its results with ours, because her original algorithm may compute incorrect slices by pruning valid interference edges. We fixed the algorithm according to section 3.5, which avoids such pruning but raises execution times.

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.

5.2 Hot Spots

During our evaluation we observed that when we run a test case of 100 slices, the individual slices did not have the same execution times. There were certain slices which needed significantly more computation time than the others. We have analyzed the impact of such *hot spots* to see if there is only a handful of slicing criteria in a program that consumes the main part of the needed computation time. If so, it could be useful to analyze how such hot spots arise, i.e. if there are certain patterns of dependences in a cSDG that cause them. Then an analysis that detects hot spots could help to trade precision for speed: For a hot spot, one could apply the fast iterated two-phase slicer instead of the expensive precise algorithms.

To this end, we measured the execution times of every single slice in our test cases of section 5.1 and compared them with the overall needed computation time. Figure 17 shows the results for AlarmClock, BoundedBuffer, LaplaceGrid and SharedQueue, computed with our optimized versions of Nanda's and Krinke's algorithms. It shows for each of the 100 slices the percentage of the overall execution time of those 100 slices, sorted by execution time. One can see that there are in fact slices that need definitely more execution time than the average (= 1%), but they still range between 1% – 10%. If we classify a hot spot as a slicing criterion that needs more that 5% of the average execution time, Table 9 shows the number of hot spots in these programs and their portion of the computation costs, which range between 0% and 44%. In our opinion, hot spots do exist, but they do not dominate the overall execution time. Even if we were able to detect hot spots in advance and to reduce their relative costs to nearly 0% using imprecise slicing algorithms, this would not relieve the explosion of computation costs shown in Table 4.

A question arising from the hot spots analysis is whether the computation costs and the gain of precision of a slice correlate. In that case a treatment of hot spots via imprecise slicers would risk to decrease precision significantly. Figure 18 compares the gain of precision and the computation costs for each taken slice of AlarmClock and LaplaceGrid, where a gain of precision of *x%* means that the precise slice is *x%* smaller than the imprecise slice. In the charts shown in that Figure the slices are sorted in order of their execution, thus the same x-Coordinate in the charts for the same program corresponds to the same slice. Whereas the three curves for AlarmClock suggest that hot spots and gain of precision correlate, as the run of the curve measuring the gain of precision roughly follows those measuring the computation costs, the curves for LaplaceGrid show the opposite behaviour. Here the gain of precision is highest where the computation costs are lowest. This means that treating hot spots with imprecise slicing algorithms would not necessarily decrease precision.

What can we conclude from this analysis? It is clear that the high computation costs stem from time travel detection. This detection may be expensive even if it

Table 9 Number of hot spots and their portion of the computation costs

Name	Krinke		Nanda	
	hot spots	costs	hot spots	costs
AlarmClock	4	34.8%	5	33.1%
BoundedBuffer	2	10.6%	2	14.6%
LaplaceGrid	5	33.6%	0	0%
SharedQueue	6	43.8%	0	0%

Table 10 Slicing of dynamically invoked threads: Average execution times per slice in seconds (left part), average size per slice in number of nodes (right part)

Name	I2P	GK	ON	I2P	GK	ON
PrecisionTest	.001	.003	.001	31.2	24.7	24.7
TimeTravel	.001	.001	.001	24.1	23.5	23.5
ProducerConsumer	.001	.002	.002	38.8	38.8	38.8
BoundedBuffer	.001	.033	.052	211.9	211.1	211.1
Primes	.001	.228	.207	353.4	342.3	342.3
AlarmClock	.003	27.647	9.225	918.5	757.7	757.7
LaplaceGrid	.007	1.685	.290	1534.6	1528.6	1528.6
SharedQueue	.034	47.575	.676	2174.2	2086.9	2086.9

does not raise precision due to absence of time travels. Instead of detecting hot spots it could be useful to search for patterns of interference dependences in cSDGs that guarantee absence of time travels and to deactivate time travel detection for these interference dependences.

5.3 Precision and Runtime-Behavior in the Presence of Dynamic Thread Invocation

Our optimized versions of Nanda's and Krinke's algorithms can handle threads that are invoked inside of loops and recursion. In the case study of chapter 5.1 we disabled those approaches and instead annotated the cSDGs with the number of existing threads. In a further case study, we have analyzed our concurrent test programs with handling of dynamic threads enabled. Table 10 shows the results, consisting of the average execution times per slice in seconds and the average size per slice in number of nodes. The runtime costs for each program are noticeably lower than for its most expensive version in Table 4, the only exception being AlarmClock for Nanda's algorithm. But the gain of precision also decreases, especially for LaplaceGrid and SharedQueue: It only ranges between 0% and 20% (for PrecisionTest and AlarmClock). Thus, to gain as precise slices as possible, a user should give upper bounds for the number of threads if possible.

5.4 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 at a high price: The execution times rise dramatically and are dependent on the numbers of threads in the

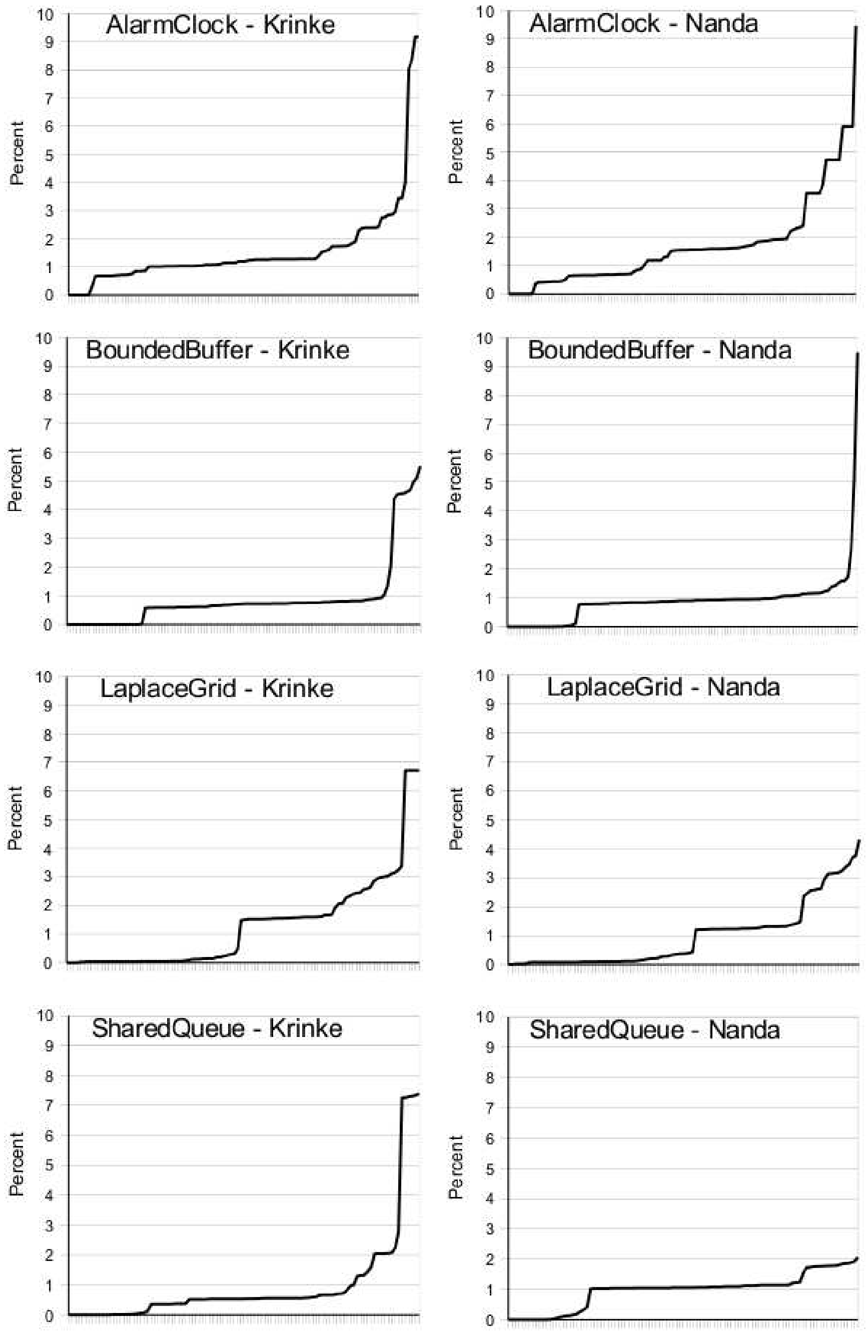


Fig. 17 Hotspot analysis for the optimized algorithms of Nanda and Krinke

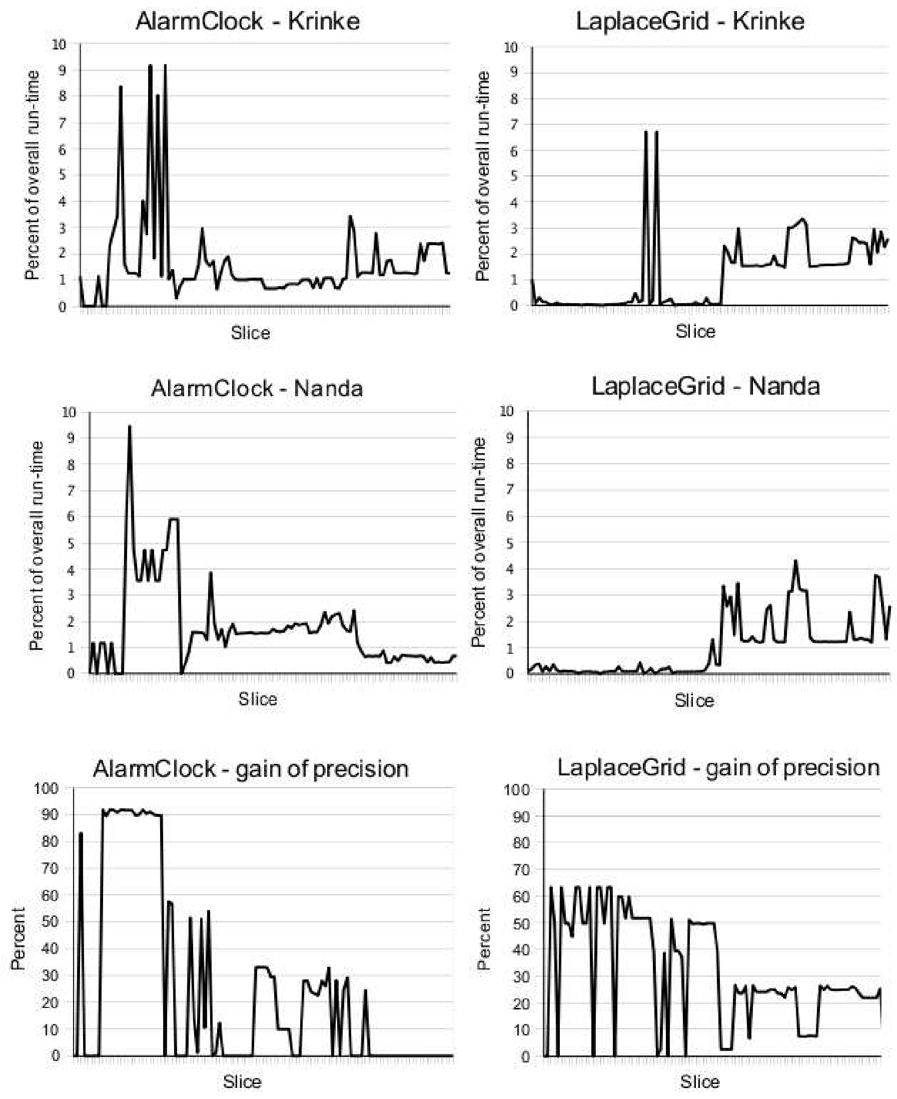


Fig. 18 Hotspots for the optimized algorithms of Nanda and Krinke (upper part) and the gain of precision compared to time-travel ignoring slicing (lower part)

analyzed program. In our opinion, a vital requirement for the application of one of these algorithms is 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. Our optimizations have shown to be very effective and should be included into implementations of these algorithms. Our optimized version of Nanda's algorithm, ON, is the most performant and precise algorithm. Our version of Krinke's algorithm, GK, has equal precision; however, ON is faster than GK because it represents contexts by single integers and uses a stronger folding method for CFG cycles.

Our hot spot analysis shows that avoiding hot spots would not ease the exponential growth of computation costs, because the dimension of the cost reduction would be too small. Moreover, avoiding hot spots might lower precision, although hot spots and gain of precision do not necessarily correspond. An alternative could be to identify patterns of dependences in cSDGs that guarantee absence of time travels, in which case time travel detection could be deactivated. Such an optimization would not lower precision.

As expected, handling threads generated dynamically inside loops and recursion with our proposed technique decreases precision, but in return it speeds up computation. For maximal precision a user should provide upper bounds for the number of thread instances, if possible.

The application area of the investigated precise slicing algorithms is bound to concurrent programs with a low number of threads, since increasing numbers of threads decrease the precision benefits while at the same time raising 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 irrelevant.

5.5 Threats to validity

Since evaluations depend on the quality of the benchmark, we want to discuss possible flaws of our program selection.

Our case study lacks big programs. Because 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.

We have only computed 100 slices per program of our case study, because several of our algorithms were not able to cope with our bigger test programs, as can be seen in Table 4. Computing all possible slices for these programs would not have been possible in reasonable time. Of course, for a different set of slicing criteria the resulting numbers might look different than in our case study. We argue that our sample slices are sufficient to compare Nanda's and Krinke's algorithms in terms of precision and runtime behavior. However, a different set of slicing criteria might show a different result for our hot spot analysis and the average gain of precision.

Further threats to validity are possible bugs in our implementations, because these algorithms are extremely complicated.

6 Future work

This section suggests several topics for future research.

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 a few instances of every thread. Future work could therefore investigate how the algorithms cope with big programs with few threads or few instances of threads, e.g. graphical user interfaces written in Java.

In programs with many thread instances the computational costs of the algorithms may rise extremely, whereas their precision benefit decreases. This seems to restrict their usability to programs with a small number of thread instances. At the cost of reduced precision, this can be tackled by approximating threads whose instance exceed a certain number, as it is done for threads generated in loops or recursion: Every traversal of an interference edge towards such a thread is considered to be valid.

The precise algorithms do not scale well due to combinatorial explosion of state tuples. Further optimizations could focus on identifying redundant state tuples, similar to Nanda's restrictive state tuples, or on reducing the number of contexts, e.g. through stronger graph folding. Another possible optimization could be identifying exploitable patterns in cSDGs, e.g. turning time travel detection off when such a pattern guarantees absence of time travels.

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)(Gosling et al. 2005) or based on MHP (may-happen-in-parallel) analysis (Naumovich et al. 1999) would allow pruning of redundant interference dependence edges and detection of 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.

Another problem we have encountered is the fact that just-in-time compilers may dynamically alter the execution order of statements. The Java just-in-time compiler, for example, is allowed to reorder the instructions in a thread, as long as the reordering do not affect the semantics of that thread in isolation (Gosling et al. 2005). Since the precise slicing algorithms use CFGs to determine valid execution orders, is it necessary that the used CFG does in fact represent the execution order of the program execution. A dynamic reordering of statements might turn a supposed time travel, detected by the algorithms using a CFG, into a valid execution order. In that case, the slice computed by the algorithms would be incorrect due to spuriously rejected interference edge traversals. We are currently investigating different approaches to solve that problem.

7 Related work

We have only given a summary of Krinke's and Nanda's algorithms. Both have described their algorithms in detail in several publications (Krinke 2003; Nanda and Ramesh 2006). Also, there exist earlier, intra-procedural variants of both algorithms (Krinke 1998; Nanda and Ramesh 2000).

Chen presents a different approach to handle the intransitivity of interference dependence (Chen and Xu 2001). He uses execution orders, MHP analysis and synchronization information to detect time-travel situations during slicing. Since his approach needs to inline methods that use synchronization, it cannot completely handle recursion.

Probably the first author who addressed slicing of concurrent programs was Cheng (Cheng 1993, 1997). He uses a *Program Dependence Net* (PDN) to represent dependences in parallel or distributed programs without procedures, where the concurrent tasks communicate via channels. Slicing on PDNs is performed using simple graph reachability.

Zhao (Zhao 1999) introduced the *Multithreaded Dependence Graph* (MDG) for Java which is similar to the cSDG and additionally contains *synchronization dependences* arising from Java's operations for synchronization. To slice MDGs, he adapts the two-phase slicer such that it additionally traverses interference and synchronization dependences in both phases. Nanda has shown that such a simple inclusion of interference dependence results in incorrect slices (Nanda and Ramesh 2006).

Hatcliff et al. (Hatcliff et al. 1999) 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 dependences similar to that of the cSDG and define further dependences to represent synchronization and infinite 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 treat interference dependence as being transitive.

Ramalingam shows that synchronization-sensitive context-sensitive slicing of concurrent programs is undecidable (Ramalingam 2000). The proof consists of reducing Post's Correspondence Problem to the synchronization-sensitive context-sensitive reachability problem.

Binkley et al. (Binkley et al. 2007) conduct an empirical study on how to improve the performance of graph-based slicing for large sequential programs, and yield a maximum reduction of 71% in runtime for a certain combination of techniques. It would be interesting for future work to evaluate if these results carry over to slicing of concurrent programs, where both slicers are based on iteratively calling sequential slicers.

8 Conclusion

We presented the first realistic evaluation and comparison of Nanda's and Krinke's algorithms for precise slicing of concurrent programs. These algorithms are significant achievements in slicing technology, being the only algorithms for pruning time travel situations in programs written in contemporary languages. Nanda developed the restrictive state tuple optimization, which is essential for applying these algorithms in practice. Unfortunately, her algorithm applies that optimization at one point where it might prune valid dependences. We detected and explained that problem and presented a correction. We have further applied several optimizations to both algorithms, which provide a significant speed up, and have extended these algorithms to handle dynamic thread generation inside loops and recursion. For most programs in our test suite, our optimized version of Nanda's algorithm performed best.

In programs with many thread instances the computational costs may rise extremely, whereas the precision benefit decreases. While several options for further optimizations have been discussed, it seems that the high costs require a selective employment of these algorithms. A pragmatic approach e.g. for information flow control would employ a less precise algorithm first, and examine cases of possibly illicit flow further, if that flow can be excluded with one of the precise algorithms. Similar ideas are applicable for other areas of application. That way, one can greatly reduce analysis overhead, and still benefit from the precision of Nanda's and Krinke's algorithms.

Acknowledgements The authors would like to thank Gregor Snelting and the anonymous reviewers for their helpful comments.

References

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

Binkley, D. and Harman, M. A survey of empirical results on program slicing. In: M. Zelkowitz (ed.): *Advances in Computers*, Vol. 62. San Diego, CA: Academic Press, pp. 105–178, 2004.

Binkley, D., Harman M. and Krinke, J. Empirical study of optimization techniques for massive slicing. *ACM Trans. Program. Lang. Syst.*, 30(1):3, 2007.

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

Chen, Z., Xu, B., Yang, H., Liu, K. and Zhang, J. An approach to analyzing dependency of concurrent programs. In *APAQS '00: Proceedings of the The First Asia-Pacific Conference on Quality Software*, pp. 39–43, 2000.

Cheng, J. Slicing concurrent programs. *Automated and Algorithmic Debugging, LNCS*, 749, pp. 223–240, Springer, 1993.

Cheng, J. Dependence analysis of parallel and distributed programs and its applications. *International Conference on Advances in Parallel and Distributed Computing*, pp. 395–404, 1997.

Giffhorn, D. and Hammer, C. An evaluation of slicing algorithms for concurrent programs. In *7th IEEE Int. Work. Conf. on Source Code Analysis and Manipulation*, pp. 17–26, 2007.

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

Hammer, C. and Snelting, G. An improved slicer for Java. *Workshop on Program analysis for software tools and engineering (PASTE'04)*, pp. 17–22, 2004.

Hammer, C. and Snelting, G. Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs. Technical Report 2008-16, November 2008, Fakultät fr Informatik, Universität Karlsruhe (TH), Germany

Hatcliff, J., Corbett, J.C., Dwyer, M.B., Sokolowski, S. and Zheng, H. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. *Static Analysis Symposium, LNCS*, 1694, pp. 1–18, Springer, 1999.

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

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

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

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

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

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

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

Nanda, M.G. and Ramesh, S. Slicing concurrent programs. *ISSTA 2000*, pp. 180–190, 2000.

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

Ottenstein, K.J. and Ottenstein, L.M. The program dependence graph in a software development environment. *ACM Symposium on Practical Software Development Environments*, pp. 177–184, 1984.

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

Ruf, E. Effective synchronization removal for Java. *Programming Language Design and Implementation (PLDI)*, pp. 208–218, 2000.

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

Sridharan, M., Fink S. and Bodik, R. Thin slicing. *ACM SIGPLAN Notices*, 42(6):112–122, 2007.

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

Weiser, M. Program slicing. *IEEE TSE*, 10(4):352–357, 1984.

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