Space-for-Time Optimization Aspects

A Case for Harmless Aspects with `around` Advice

Ahmed Abdelmeged
Northeastern University
mohsen@ccs.neu.edu

Karl Lieberherr
Northeastern University
lieber@ccs.neu.edu

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Debugging aids; D.2.4 [*Software/Program Verification*]: Assertion checkers

General Terms Languages, Algorithms, Design, Verification, Performance

Keywords Optimization, Cache, Dynamization, Memoization, Incrementalization, Aspect, Harmless, Interaction

Abstract

The space-for-time trade-off is one of the earliest to be discovered in computer science. Taking program complexity into account, we can safely call it "space and elegance"-for-time trade-off. Contemporary algorithms text books often tackle the complexity problem by first presenting algorithms in an elegant, but inefficient form, followed by a few paragraphs informally describing optimizations. We used aspects to improve the presentation of several algorithms like Topological Ordering, Dijkstra's shortest path, Closest Pair, Stable Marriage, etc.

Harmlessness is a key safety requirement for optimization aspects. Existing models for harmless aspects disallow general `around` advice which is indispensable for expressing optimization aspects. We present a model for harmless aspects with `around` advice. We also present a domain specific model for space-for-time optimization aspects that rules out several situations when the optimization aspect is not improving the runtime efficiency and enables more informative error messages.

1. Introduction

The space-for-time trade-off is one of the earliest trade-offs to be discovered in computer science. There are automated approaches that trade off memory space to improve the run time efficiency of an easy to understand top-down algorithm (obtained through functional decomposition). These approaches include memoization and incrementalization [12], [13] (a.k.a. dynamization). It is also possible to execute these algorithms on a self-adjusting machine [1] that trades space for time.

However, there are application specific space-for-time optimizations for which it is highly unlikely that they can be implemented using an automated approach. The reason is that application specific space-for-time optimizations tend to be more liberal in:

1. Cache representation. The cache can be represented using a hash map or using "inter-type declaration". For example, it is possible to keep a hash map from a node to its predecessor count in a particular graph or inter-type-declare the predecessor count into the node (see Section 2.2 on topological ordering). Even when the cache is implemented as a hash map, the table size as well as the hashing function can make a difference. For example, when the cache can hold at most a single entry.
2. Adding entries to the cache. It might be more efficient to eagerly compute several values together than performing them individually on demand. For example we can compute the predecessor count of all nodes in a graph using a single walk where computing the predecessor count of a single node would also require walk of the entire graph (see Section 2.2 on topological ordering). It might be also more efficient to maintain the cache under functional updates. For example, if the cache for the method `sort` maps `list` to `v` and the statement `list' = filter(list, predicate)` is executed, then it is possible to add to the cache a mapping from `list'` to the value of `filter(v, predicate)` (see Section 2.3 on finding the closest pair of points).
3. Maintaining cache entries under destructive updates. For example, it might be more efficient to maintain the cache after several destructive updates rather than eagerly maintain the cache after each destructive update. For example, if the cache for the method `sort` maps `list` to `v`. It is possible to maintain the cache eagerly after `list.add(e)`. The maintenance involves an insertion of

`e` into `v`. Grouping several `list.add(e)` operations together, it is possible to sort all the newly added elements using an $O(n \cdot log(n))$ algorithm then merge them into `v`. It can also be beneficial from a software engineering perspective to abstract over several alternative destructive updates. For example, the fact that we maintain the cache for the method `sort` after `list.add(e)` abstracts over several alternative sequences of destructive updates of several `list.add(..)` implementations.

4. Overriding the computation to make it more amenable to cache-based optimizations [1]. For example, the method `Graph.sourceNode()` (Shown in Figure 2) that finds a node without predecessors in a graph, is not amenable to incrementalization under `Graph.remove(Node)` because the old source node is not useful in computing the new source node once the old source node is removed from the graph. It is possible to override `Graph.sourceNode()` with a less efficient implementation that computes all source nodes then returns one of them. The method that computes all source nodes can be incrementalized under `Graph.remove(Node)`.
5. Utilizing a single cache to speed up several computations. For example, it is possible to utilize a cache for the `sort` method to speed up the computation of order statistics.

Taking program complexity into account, the space-for-time trade-off can be safely called "space and elegance"-for-time trade-off. Contemporary algorithms text books often tackle the complexity problem by first presenting algorithms in an elegant, but inefficient form, followed by a few paragraphs informally describing optimizations.

Aspects can not only be used to restore the lost elegance of optimized programs, but also to better support the process of optimizing programs, enhance the safety of optimized programs, and to enable mixing and matching of optimizations.

Using aspects, an optimized program $OPTPROG$ can be structured as $PROG + OPT_1 + \ldots + OPT_n$ enabling developers to mix and match optimizations. But more importantly, each of $OPT_1, \ldots, OPT_n$ can be separately checked for harmlessness which enhances safety and provides more targeted feedback which supports the debugging of optimizations. The concept of harmless aspects is a well known concept that has been used in the context of enabling modular reasoning about the base program [8], [5], [6], [16], [7].

Unfortunately, AspectJ-like aspects can become harmful for a variety of subtle reasons that are hard discover. To make things worse, space-for-time optimization aspects are not expressible in AOP languages devised to exclusively express harmless aspects such as harmless advice [6] and pure aspects [16]. Because both systems disallow general `around` advice [2] which is indispensable for expressing optimization aspects.

We developed a model for harmless aspects with `around` advice. The model consists of a set of contracts that the aspect's behavior must satisfy to be considered harmless.

We also developed a domain specific model for space-for-time optimization aspects. In our model, each advice in a space-for-time optimization aspect must be declared to be one of three kinds (memoization, maintenance and management). Furthermore, the behavior of each kind of advice has a particular set of contracts to satisfy. For example, the cache must be consistent before a memoization advice executes. The purpose of this model is neither to ensure the correctness of optimization aspects nor to enable a more concise expression of optimization aspects, but to support the debugging of space-for-time optimization aspects through informative error messages.

1.1 Organization

The rest of this paper is organized as follows, In section 2 we present a recipe for developing space-for-time optimization aspects and illustrate our recipe with few examples that we also use as running examples. In section 3 we present our model for harmless aspects with around advice. In section 4 we present our model for space-for-time optimization aspects. In section 5 we describe the checkers we implemented for both models. In section 6 we discuss some of the related work. Section 7 concludes this paper.

2. Space-for-Time Optimization Aspects

We studied space-for-time optimizations described in [11] for algorithms like Topological Ordering, Dijkstra's shortest path, Closest Pair, Stable Marriage, etc. Eventually, we came up with a recipe for developing space-for-time optimization aspects. We start by outlining our recipe then we illustrate it by examples.

2.1 Developing Space-for-Time Optimization Aspects

The development of space-for-time optimization aspects has three key activities, identifying optimization opportunities, ranking them, and handling them.

2.1.1 Identifying Optimization Opportunities

We identify optimization opportunities by analyzing the trace of an execution of our program on a typical input. An optimization opportunity is a set of similar join points corresponding to the execution of referentially transparent methods. A set of join points are considered similar iff they have the same shadow and have at least one similar context element. For example, two join points with the same shadow such as the execution of some method `m` are considered sim-

[1] Related to the notion of trace stability in [1]

[2] In pure aspects a limited form of `around` advice is allowed where the advice must invoke `proceed()` on all execution paths and return what `proceed()` returns

ilar if they have similar implicit `this` arguments or similar first arguments and so on.

A set of objects are considered similar iff there is at least one object reachable from all of them. For example, an `ArrayList` object containing the object `o1` is considered similar to a `LinkedList` object containing the object `o1`.

Our notion of similarity between join points and objects is a weak one that is expected to apply to a large number of sets of join points. Therefore, it is important to rank them.

2.1.2 Ranking Optimization Opportunities

We rank optimization opportunities based on the average cost of join points in them, then by the number of join points in them. The cost of a join point is defined to be the number of join points it encloses.

2.2 Example I: Topological Ordering

Given a directed graph g it is desired to find an ordering of its nodes such that node a comes before node b in the computed ordering if g contains an edge from a to b, if such ordering exists.

Figures 1 and 2 show a simple directed graph structure and a simple algorithm for computing the topological ordering of a given directed graph. We chose to separate the structure from the algorithm to simplify the presentation of the latter. We also omitted an aspect that prevents the algorithm from destroying the graph by marking removed nodes and overriding the structure accessors to take those marks into account. To save space and to avoid distracting the reader, we elided visibility modifiers, some generics, some initializers, and some method implementations.

The algorithm keeps removing some arbitrary node with no predecessors from the graph appending it to the output list until there are no more nodes with no predecessors.

2.2.1 Optimization Opportunities

Considering the join points in the control flow of an invocation of `Graph.topord()`, lines 19- 32 in Figure 2 we identify the following optimization opportunities:

1. The set of join points corresponding to executing `Graph.getSourceNode()` (lines 10- 17 in Figure 2) with the same `Graph` object. The number of join points in this set is n, the number of nodes in the graph.
2. The set of join points corresponding to executing `Node.getPredCount()` (lines 2- 2 in Figure 2) with the same `Node` object enclosed in the same `Graph` object. The number of join points in this set is $n \cdot (n+1)/2$.
3. Several sets of join points corresponding to executing `Node.getPredCount()` (lines 2- 2 in Figure 2) with some `Node` object enclosed in the same `Graph` object. The average number of join points in each of these sets is $n/2$.

```
1  class Graph {
2   List<Node> nodes = ...
3   class Node{
4     List<Node> successors = ...
5     Node(){
6       getEnclosingGraph().add(this);
7     }
8     Graph getEnclosingGraph(){...}
9     Collection getSuccessors(){...}
10    boolean hasSuccessor(Node succ){
11      return successors.contains(succ);
12    }
13    void removeAllSuccessors(){
14      successors.clear();
15    }
16  }
17  void add(Node node){...}
18  List getNodes(){...}
19  int getNumNodes(){...}
20  void remove(Node node){
21    node.removeAllSuccessors();
22    nodes.remove(node);
23  }
24 }
```

Figure 1: Directed Graph Structure

```
1  aspect Topord {
2    int Node.getPredCount(){
3      int predCount = 0;
4      for (Node n : getEnclosingGraph().getNodes())
           {
5        if(n.hasSuccessor(this)) predCount++;
6      }
7      return predCount;
8    }
9
10   Maybe<Node> Graph.getSourceNode(){
11     for (Node n : getNodes()) {
12       if(n.getPredCount() == 0){
13         return new Some(n);
14       }
15     }
16     return new None();
17   }
18
19   Maybe<List<Node>> Graph.toporder(){
20     List<Node> orderedNodes = ...
21     while(true){
22       Maybe<Node> mbSource = getSourceNode();
23       if(!mbSource.isSome()) break;
24       Node n = mbSource.get();
25       orderedNodes.add(n);
26       remove(n);
27     }
28     if(getNumNodes() > 0 ){
29       return new None();
30     }
31     return new Some(orderedNodes);
32   }
33 }
```

Figure 2: Algorithm for Finding The Topological Ordering of a Directed Graph

4. Several other join points corresponding to getter methods.

It is worth noting that in a top-down program it is also possible to search for optimization opportunities in a top-down fashion as the average cost of join points corresponding to higher level methods is expected to be higher than that at the lower level because higher level methods invoke lower level ones. For example the average cost of join points corresponding to the execution of `Graph.getSourceNode()` is expected to be more than that of `Node.getPredCount()` because the former invokes the latter.

It is also worth noting that the second optimization opportunity have the same shadow, the execution of `Node.getPredCount()`, as any of third-ranked optimization opportunities. The difference is that join points in the second opportunity have less similar implicit this argument and therefore the second optimization opportunity contains all of the third-ranked optimization opportunities.

2.2.2 Handling The First Optimization Opportunity

We can start by memoizing `Graph.sourceNode()` method. We do so by developing an aspect that inter-type-declares a field `memoizedSorceNode` of type `Node` into `Graph`. We then develop a memoization advice for `Graph.sourceNode()`. Soon we discover that our optimization aspect is harmful because the memoization advice returns a different value than what `Graph.sourceNode()` returns. The root cause for this problem is that the implicit `this` argument was destructively updated by the `remove(..)` call on line 26 in Figure 2. To fix this problem we add a maintenance advice for the `memoizedSorceNode` field. Our maintenance advice catches the execution of `Graph.remove(Node)` and assigns `memoizedSourceNode` a correct value. Unfortunately, the maintenance advice can only compute a new value of `memoizedSourceNode` from scratch, making the whole optimization not beneficial.

At this point, we have two approaches to improve our optimization aspect:

1. to eagerly compute the list of all source nodes the first time `Graph.sourceNode()` is invoked. This requires a pass over all the nodes in the graph which is not necessary for the typical case. This approach is presented in Figure 3. We chosen this approach because we know that sooner or later all of the source nodes are going to be computed.
2. to save as much information as possible to speed up subsequent computations of source node. For example, the first time `Graph.sourceNode()` is invoked, it examines say the first 7 nodes in the graph before it returns the 8th node. This information should be made available to speedup subsequent computations of source node.

With this optimization aspect in place, the frequency with which the `Node.getPredCount()` is invoked in a typical run decreases. `Node.getPredCount()` is invoked with some `Node` object enclosed in the same `Graph` object as many as the sum of number of nodes and edges in the graph. Also, `Node.getPredCount()` is invoked with the same `Node` object as many times as the in-degree of that node plus one.

```
1  aspect SrcNode {
2    List<Node> Graph.sourceNodes = null;
3
4    //Memoization + Eager initialization
5    Maybe<Node> around(Graph g): execution(public
         Maybe<Node> getSourceNode()) && target(g){
6      if(g.sourceNodes == null){
7        g.sourceNodes = new ArrayList<Node>(); for
             (Node n : g.getNodes()) {
8          if(n.getPredCount() == 0){
9            g.sourceNodes.add(n);
10         }
11       }
12     }
13     if(g.sourceNodes.size() == 0){
14       return new None();
15     }else{
16       return new Some(g.sourceNodes.get(0));
17     }
18   }
19
20   //Maintenance
21   void around(Graph g, Node n): execution(public
         void remove(Node)) && target(g) && args(n){

22     List<Node> successorsOfToBeRemovedNode = new
           ArrayList<Node>(n.getSuccessors());
23     proceed(g,n);
24     for (Node succ : successorsOfToBeRemovedNode)
           {
25       if(succ.getPredCount() == 0){
26         g.sourceNodes.add(succ);
27       }
28     }
29     g.sourceNodes.remove(n);
30   }
31 }
```

Figure 3: First Optimization Aspect for The Topological Ordering Algorithm

2.2.3 Handling The Second Optimization Opportunity

Join points in the second optimization opportunity do have the same `Graph` object but different `Node` objects, to handle this optimization opportunity we need to share all the work performed by `Node.getPredCount()` that is related to the `Graph` object. Again, we have the same two approaches to share that work. We choose the eager approach here because we know that the predecessor counts for all nodes will be eventually computed. Figure 4 shows an implementation of the second optimization opportunity.

```
1  aspect Preds {
2    int Node.predCount = -1;
3
4    //Memoization + Eager initialization
5    int around(Node node):
6      execution(public int getPredCount())
7        && target(node){
8      if(node.predCount == -1){
9        for (Node n  : node.getEnclosingGraph().
             getNodes()) {
10         n.predCount = 0;
11       }
12       for (Node n  : node.getEnclosingGraph().
             getNodes()) {
13         for (Node succ : n.getSuccessors()) {
14           succ.predCount++;
15         }
16       }
17     }
18     return node.predCount;
19   }
20
21   //Maintenance
22   before(Node n):
23     execution(private void removeAllSuccessors())
24       && target(n){
25     for (Node succ : n.getSuccessors()) {
26       succ.predCount--;
27     }
28   }
29 }
```

Figure 4: Second Optimization Aspect for The Topological Ordering Algorithm

2.3 Example II: Finding The Closest Pair of Points in a Plane

Given a set S of n points in a two dimensional plane, it is desired to find the closest pair of points. The computational complexity of the problem is known to have an asymptotic lower bound of $\Omega(n \cdot log\ n)$[3]. We describe a divide an conquer algorithm for solving this problem matching the lower bound.

In the divide phase, the points are split into two subsets $S_1,\ S_2$ with roughly the same number of points in each set. The points are split such that the x-coordinate of all the points in S_1 is smaller than x_{median} and all the points in S_2 are greater than it. Where x_{median} is the median value of all x-coordinates of the points in S. The linear time median finding algorithm is used.

After the split, the closest pair of points in each subset is recursively computed. Let $P_1,\ P_2$ be the closest pair of points in $S_1,\ S_2$ respectively. When the number of points is sufficiently small, the brute force procedure is used to bottom up from the recursion.

In the combine phase, it is not correct to return one of P_1 or P_2. It is necessary to also examine pairs with one point in S_1 and the other point in S_2. More precisely, it is enough to examine the set S' of points in either S_1 or S_2 with an x-coordinate within a distance δ from x_{median}, where $\delta = min(\delta_1,\ \delta_2)$, δ_1 is the minimum distance between the pair of points P_1, δ_2 is the minimum for P_2.

Since the set S' can be as large as S, applying the brute force procedure to the elements in the strip lead to an $\Omega(n^2)$ best case performance. Instead, we use a more efficient procedure that relies on the fact that the points in S' do have a special structure. Namely, they form a $2 \cdot \delta$ vertical strip and that they are δ-sparse meaning that there is a *fixed* number of points within a δ distance from any point in S'.

The solution for the δ-sparse vertical strip case involves sorting the points by their y-coordinate then going through the sorted list and for each point, a fixed number of its following points are examined. Figure 5 shows a Java implementation of the entire algorithm.

2.3.1 Optimization Opportunities

Considering the join points in the control flow of an invocation of `ClosestPair.DCCP(List)`, lines 2- 25 in Figure 5 we identify the following optimization opportunities:

1. Sets of join points corrresponding to executing `ClosestPair.DCCP(List)` with similar `List` objects.
2. Sets of join points corrresponding to executing `median(List, Comparator)` with similar `List` objects and the same `Comparator` object.
3. Sets of join points corrresponding to executing `partition(List, Predicate)` with similar `List` objects.
4. Sets of join points corrresponding to executing `filter(List, Predicate)` with similar `List` objects.
5. Sets of join points corrresponding to executing `ClosestPair.DCVSCP(List)` with similar `List` objects.
6. Sets of join points corrresponding to executing `sort(List, Comparator)` with similar `List` objects and the same `Comparator` object.
7. Several other sets join points corresponding to other methods such as `Result.combine(Result)`.

Any two join points in an optimization opportunity corresponding to `ClosestPair.DCCP(List)` have the property that the `List` argument in one of the join points must be a sublist of the `List` argument of the other. The algorithm already utilizes all these opportunities.

We also observe that join points corresponding to the execution of `partition(List, Predicate)` differ in their `Predicate` argument. By inspecting the implementation of `partition(List, Predicate)` we discover that it cannot be effectively incrementalized under updates to the `Point` argument. The same observation applies to `filter(List, Predicate)`.

[3] Via a straight forward reduction from the element uniqueness problem

```
1  class ClosestPair {
2    public static Result DCCP(List<Point> points){
3      int n = points.size();
4      if(n<=3){
5        return BruteForceCP(points);
6      }else{
7        final Point xMedian = Utils.median(points,
             Point.xComparator());
8        List<List<Point>> s = Utils.partition(
             points, new Utils.Predicate<Point>(){
9          @Override
10         public boolean holdsFor(Point point) {
11           return Point.xComparator().compare(
                 point, xMedian) <= 0;
12         }});
13       Result P1 = DCCP(s.get(0));
14       Result P2 = DCCP(s.get(1));
15
16       final Result cpInPartitions = P1.combine(P2
             );
17       List<Point> ptsInStrip = Utils.filter(
             points, new Utils.Predicate<Point>(){
18         @Override
19         public boolean holdsFor(Point point) {
20           return Math.abs(point.x - xMedian.x) <=
                 cpInPartitions.getDistance();
21         }});
22       Result cpInStrip = DCVSCP(ptsInStrip);
23       return cpInStrip.combine(cpInPartitions);
24     }
25   }
26
27   public static Result DCVSCP(List<Point> points)
         {
28     List<Point> ySortedPoints = Utils.sort(points
         , Point.yComparator());
29     Result result = new Result();
30     for (int i = 0; i < ySortedPoints.size(); i
         ++) {
31       Point pi = ySortedPoints.get(i);
32       for (int j = i+1; j < ySortedPoints.size()
           && j < i+7 ; j++) {
33         result = result.combine(new Result(pi,
               ySortedPoints.get(j)));
34       }
35     }
36     return result;
37   }
38
39   // ... BruteForceCP is elided
40 }
41 class Result {
42   final double distance;
43   final Point p1;
44   final Point p2;
45
46   public Result combine(Result other){
47     return (other.distance < distance)? other :
           this;
48   }
49   // ... rest is elided
50 }
```

Figure 5: Divide and Conquer Algorithm for Finding The Closest Pair of Points

Depending on the input, the vertical strips could intersect. In this case, there will be some similarity between the `List` arguments of some `ClosestPair.DCVSCP(List)` invocations as well as `sort(List, Comparator)`.

`ClosestPair.DCVSCP(List)` cannot be incrementalized under removing a `Point` from its input list because `Result` objects cannot be uncombined.

Therefore, we are left with two sets of optimization opportunities, those corresponding to `median(List, Comparator)` and those corresponding to `sort(List, Comparator)`.

2.3.2 Handling The First Optimization Opportunity

Figure 6 shows an aspect that handles optimization opportunities corresponding to `median(List, Comparator)`. Conceptually, the aspect is similar to the first optimization aspect for the topological ordering algorithm. The aspects stores a sorted version of the `List` to speed up subsequent median computations because the old median is not so useful in computing the new median once the `List` argument is updated.

This aspect differs from the first optimization aspect for the topological ordering algorithm in that the `List` argument is not destructively updated. Instead, `List` is functionally updated through `partition`. Failure to maintain the cache under functional updates makes the aspect ineffective.

2.3.3 Handling The Second Optimization Opportunity

Figure 7 shows an aspect that handles optimization opportunities corresponding to `sort(List, Comparator)`. Conceptually, the aspect memoizes `sort(List, Comparator)` invocations and maintains the cache under functional updates by `partition(List, Predicate)` and `filter(List, Predicate)`. However, the cache must be eagerly initialized otherwise the aspect will be ineffective.

3. Harmless Aspects with `around` Advice

An aspect is said to be harmless, to the base program, iff

1. the aspect does not influence the final value produced by the base program,
2. the aspect influences other aspects only indirectly through triggering or blocking their advice.

The aspect can change the termination behavior of the base program and perform I/O and still be considered harmless.

3.1 A Model for Harmless Aspects

An aspect is considered harmless if its behavior satisfies the following contracts:

3.1.1 Contract 1: No External Data or Control Flow Effects

Every `around` advice must return the *same* value or throw the *same* exception as the join point it advises. For primitive types, *same*ness is defined as equality. For reference types,

```
1  aspect Median {
2    Map<List,List> xSorted = new IdentityHashMap();
3
4    // memoization
5    Object around(List list, Comparator comparator)
         :
6      execution(public Object Utils.median(..))
7        && args(list, comparator)
8        && !cflow(within(Median||YSort)){
9      List value = xSorted.get(list);
10     if(value == null){
11       value = Utils.sort(list, Point.xComparator
             ());
12       xSorted.put(list, value);
13     }
14     return value.get(list.size()/2);
15   }
16
17   // management
18   after(List list, Predicate predicate)
19     returning (List partitions):
20       execution(public List Utils.partition(..))
21       && args(list, predicate)
22       && !cflow(within(Median||YSort)){
23     List value = xSorted.get(list);
24     if(value!=null){
25       List sortedPartitions = Utils.partition(
             value, predicate);
26       for(int i = 0 ; i< partitions.size(); i++){
27         xSorted.put((List) partitions.get(i), (
               List)sortedPartitions.get(i));
28       }
29     }
30   }
31 }
```

Figure 6: First Optimization for The Closest Pair Algorithm

*same*ness is defined as reference equality unless the object is constructed in the control flow of the join point. In this case, *same*ness is defined as *same*ness of corresponding subcomponents. Every `before` and `after` advice in the aspect must not throw any exceptions.

The aspect cannot have an external data flow effect because every `around` advice returns the *same* value or throw the *same* exception as the join point it advises.

We ignore, termination and nontermination control flow effects as they do not affect the harmlessness of an aspect by definition. Also, we assume that advice cannot directly manipulate the current continuation. Therefore, the only control flow effect that we need to consider is throwing exceptions[4]. `before` and `after` advice cannot throw exceptions and hence cannot have any external control effect. Also, `around` advice return the *same* value or throws the *same* exception as the join point it advises and hence cannot have any control effect observable by the base program.

[4] In AspectJ exceptions can declaratively softened. A harmless aspect is not allowed to soften exceptions.

```
1  aspect YSort {
2    Map<List,List> ySorted = new IdentityHashMap();
3
4    // memoization
5    List around(List list, Comparator comparator):
6      execution(public List Utils.sort(..))
7        && args(list, comparator)
8        && !cflow(within(Median||YSort)){
9      List value = ySorted.get(list);
10     if(value != null) return value;
11     value = proceed(list, comparator);
12     ySorted.put(list, value);
13     return value;
14   }
15
16   // management
17   after(List list, Predicate predicate)
18     returning (List partitions):
19       execution(public List Utils.partition(..))
20       && args(list, predicate)
21       && !cflow(within(Median||YSort)){
22     List value = ySorted.get(list);
23     if(value!=null){
24       List<List> sortedPartitions = Utils.
             partition(value, predicate);
25       for(int i = 0 ; i< partitions.size(); i++){
26         ySorted.put((List) partitions.get(i),
               sortedPartitions.get(i));
27       }
28     }
29   }
30
31   // management
32   after(List list, Predicate predicate)
33     returning (List filtered):
34       execution(public List Utils.filter(..))
35       && args(list, predicate)
36       && !cflow(within(Median||YSort)){
37       List value = ySorted.get(list);
38     if(value!=null){
39       List sortedFiltered = Utils.filter(value,
             predicate);
40       ySorted.put(filtered, sortedFiltered);
41     }
42   }
43
44   pointcut DCCP(List list):
45     execution(Result DCCP(..)) && args(list);
46   pointcut topLevelClosestPair(List list):
47     DCCP(list) && !cflowbelow(DCCP(List));
48
49   // init
50   before(List list) : topLevelClosestPair(list) {
51     ySorted.put(list, Utils.sort(list, Point.
         yComparator()));
52   }
53 }
```

Figure 7: Second Optimization for The Closest Pair Algorithm

3.1.2 Contract 2: No Blocked or Repeated External Memory Effects

Every `around` advice must only advise join points with no external memory effects. A join point is said to have no external memory effects iff none of the objects that existed before the join point executes is mutated in the control flow of the join point. For brevity, we use memory effects to denote external memory effects.

As a result, an `around` advice that does not invoke `proceed()` or invoke it several times cannot block or repeat any memory effect visible to the base program.

3.1.3 Contract 3: No Introduced Memory Effects

The fields of objects mutated in the control flow of the aspect cannot be referenced outside the control flow of the aspect unless reachable from one of the objects returned from the aspect. A join point J is said to be in the control flow of some aspect A iff J's static shadow is lexically in A or J is in the control flow of some join point K whose static shadow is lexically in A.

As a result, no advice can introduce a memory effect that is visible by the base program.

3.2 A Model for Crosscutting Harmless Aspects

The aforementioned harmlessness contracts are too strict to allow one harmless advice to crosscut another. For example, suppose that we have a harmless profiling aspect P that counts for every method the number of times it is invoked and stores this information in a file. P is likely to crosscut the base program as well as other aspects. The problem is that an otherwise harmless aspect A that has one of its advice advised by P will have a memory effect on some object (one of P's counters) that can be referenced outside the control flow of A and cannot be returned by one of A's advice.

Below we present a relaxed version of the harmlessness contracts that allows two harmless aspects to crosscut each other. But first we present a key definition *direct control flow*.

3.2.1 Direct Control Flow

We say that a join point J is in the *direct control flow* of another join point K iff K encloses J and there is no other `adviceexecution()` join point L that encloses J and is enclosed by K.

The intuition is that an `adviceexecution()` join point acts as a module boundary marker; an `adviceexecution()` join point is encountered before jumping into the execution of an independently developed module.

3.2.2 Contract 2': No Blocked or Repeated Memory Effects

Every `around` advice must only advise join points with either confined or no memory effects. A memory effect is said to be confined iff it is made in the control flow of some other harmless aspect C. The rationale is that if C is a harmless advice, then memory effects in its control flow are confined to C. Otherwise, C is blamed for violating the harmlessness conditions.

3.2.3 Contract 3': No Introduced Memory Effects

The fields of objects mutated in the direct control flow of the aspect cannot be referenced outside the direct control flow of the aspect unless reachable from one of the objects returned from the aspect.

4. A Model for Space-for-Time Optimization Aspects

Although the harmlessness rules are enough to guarantee the correctness of space-for-time optimization aspects, they can catch errors long after they occur when the root cause of the error is no longer on the stack.

As a remedy we designed a model for space-for-time optimization aspects. In our model, a space-for-time optimization aspect is valid iff

1. it is a harmless aspect,
2. it contains only three kinds of advice: memoization, maintenance, and management,
3. it has a method `boolean isConsistent()` method that checks the consistency of the aspect's confined memory which typically holds some sort of a memoization table or cache,
4. its behavior satisfies the contracts enumerated in section 4.1.

A memoization advice lazily populates the cache upon a cache miss. Upon a cache hit, a memoization advice returns a value from the cache instead of proceeding to the memoized method. It is also possible that a memoization advice eagerly computes the result of several possible invocations of the memoized method. A maintenance advice restores the cache consistency after destructive updates to objects reachable from the cache. A management advice lazily populates the cache upon functional updates to objects reachable from the cache.

4.1 A Behavioral Interface for Space-for-Time Optimization Aspects

4.1.1 Contract 1: Consistency After Every Advice

Every advice, whether it is memoization, maintenance, or management, must leave the aspect in a consistent state. Otherwise, that advice is incorrect. The purpose of this rule is to prevent errors from propagating from an incorrect advice eventually leading to the memoization advice returning an incorrect value.

4.1.2 Contract 2: Consistency Before Memoization and Management

The aspect must be consistent before memoization and management advice. Otherwise, there is a missing maintenance

advice. In this case, it is useful to report the *most recent deterioration join point*. That is the most recent join point at which the aspect ceased to be consistent.

4.1.3 Contract 3: Inconsistency Before Maintenance

For each join point shadow advised by a maintenance advice, there must be at least one join point belonging to that shadow where the cache was inconsistent before the maintenance advice executes. Otherwise, the shadow might be unnecessarily caught by the pointcut of the maintenance advice.

4.1.4 Contract 4: There is a Space-for-Time Optimization Opportunity

Every memoized method is invoked more than once with the same or related arguments. Otherwise, there is no chance that the optimization aspect improves the running time efficiency.

4.1.5 Contract 5: There are Cache Hits

For each memoization advice, there must be at least one cache hit. That is, execution does not affect the confined state of the aspect. Otherwise, either the memoization advice is incorrect or there is a missing or incorrect management advice.

5. Implementation

We implemented set of runtime checkers for the behavioral rules of our harmless aspects model and space-for-time optimization aspects model. The checkers recognize both kinds of aspects through the interfaces `Harmless` and `SpaceForTimeOpt` shown in Figure 8. The checkers recognize the advice kind in space-for-time optimization aspects through the annotations `@Memoization`, `@Maintenance`, and `@Management`.

```
1  interface Harmless{}
2  interface SpaceForTimeOpt extends harmless{
3    boolean isConsistent();
4  }
```

Figure 8: `Harmless` and `SpaceForTimeOpt` interfaces

One issue that we faced with checking that an `around` advice returns the *same* value as the join point it advises. The issue is computing the value returned by an `around` advised join point without the advice. At first, one might try to deactivate the aspect containing the `around` advice under check, `proceed()`, and then reactivate the aspect. Figure 9 shows a first attempt at writing such checker.

The activation/deactivation mechanism can be implemented using an `if()` clause added to the pointcut of every `around` advice in some `Harmless` aspect. The `if()` clause can be woven into the aforementioned pointcuts using ORATA [15].

```
1  aspect ObservationalEquivalence implements
       Checker{
2    Object around(Harmless theAspect):
         adviceexecution() && !within(Checker+) &&
         target(theAspect){
3      if(isAround(thisJoinPoint)){
4        deactivate(theAspect);
5        Object unadvised = proceed(theAspect);
6        activate(theAspect);
7        Object advised = proceed(theAspect);
8        if(!sameAs(advised, unadvised)) error();
9        return advised;
10     }
11   }
12 }
```

Figure 9: Naive Observational Equivalence Checker

The problem with this approach is that the checker is triggered by the execution of the `around` advice under check. Therefore, it is too late to deactivate the aspect under check from within the checker.

In our checker, we adopted the following workaround. We require every `around` advice in a `Harmless` aspect to:

1. only advise a single method call or method execution join point,
2. be annotated with the the method it advises.

The checker invokes the method specified in the annotation instead of the `proceed` on line 5 in Figure 9. Figure 10 shows the annotation type we use to annotate `around` advice in `Harmless` aspects.

```
1  @interface ObservationallyEquivalentTo {
2    Class<?> clazz();
3    String methodName();
4    Class<?>[] argumentTypes();
5    String thisArgument();
6    String[] arguments();
7  }
```

Figure 10: `ObservationallyEquivalentTo` Annotation Type

5.1 A Special `proceed` Form

A better solution is to introduce a pointcut designator `aroundadviceexecution()` that matches the execution of `around` advice. A special form `proceedwithoutadviceexecution()` should also be available in the body of `around()` advice with `aroundadviceexecution()` pointcut.

In the body of `around` advice with `aroundadviceexecution()`, the available `around` closure executes some inner `around` advice which in turn has an inner `around` closure. `proceedwithoutadviceexecution()` executes the inner `around` closure.

`proceedwithoutadviceexecution()` is designed be used to on line 5 in Figure 9 without deactivation and activation.

6. Related work

6.1 Harmless Aspects

Our model for harmless aspects is different from both harmless advice [6] and pure aspects [16] in the support for `around` advice. Also, our model differs from both systems in that it is dynamically checked. Harmless advice `HarmlessAdvice` can directly or indirectly influence other advice. Pure Aspects `PureAspects` are only checked for purity on a specific set of classes. In our model, harmless aspects cannot affect the base program neither can they directly affect other aspects. Harmless aspects can indirectly affect other aspects through triggering or blocking advice belonging to other aspects.

6.2 Domain Specific Aspect Languages

The earliest work on AOP [14], [10] was domain specific. An experimental domain specific aspect language for caching is implemented in [9]. Maintenance is limited to clearing the cache and there is no notion of management advice. Finally, there is no checking that the provided cache invalidation pointcuts are complete.

6.3 Contracts and Behavioral Interfaces

Computational contracts [4] assert properties of the computations other than the relation between its inputs and outputs. PIPA [17] is a behavioral interface specification for AspectJ. PIPA can assert certain control effects, for example that the advice `proceed`s in specific contexts. Translucid Contracts [3] are abstract algorithms that advice must be structurally similar to. Tracemonitors [2] can be used to assert that events during the program execution satisfy certain regular expression.

7. Conclusion and Future Work

We presented a model for harmless aspects with `around` advice and a model for space-for-time optimization aspects along with a recipe for developing space-for-time optimization aspects. We implemented a checker for both models.

In future, we plan to develop a tool that analyzes program traces to identify optimization opportunities. We also plan to use static analysis to reduce the runtime overhead of our checkers. Finally, we plan to implement `proceedwithoutadviceexecution`.

References

[1] U. A. Acar. *Self-adjusting computation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2005. AAI3166271.

[2] P. Avgustinov, E. Bodden, E. Hajiyev, L. Hendren, O. Lhoták, O. de Moor, N. Ongkingco, D. Sereni, G. Sittampalam, J. Tibble, and M. Verbaere. Aspects for trace monitoring. In K. Havelund, M. Nunez, G. Rosu, and B. Wolff, editors, *Formal Approaches to Testing Systems and Runtime Verification (FATES/RV)*, volume 4262 of *Lecture Notes in Computer Science*, pages 20–39. Springer, 2006. URL `http://www.bodden.de/pubs/abh+06aspects-for.pdf`.

[3] M. Bagherzadeh, H. Rajan, G. T. Leavens, and S. Mooney. Translucid contracts for modular reasoning about aspect-oriented programs. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 245–246, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0240-1. doi: http://doi.acm.org/10.1145/1869542.1869596. URL `http://doi.acm.org/10.1145/1869542.1869596`.

[4] W. D. M. Christophe Scholliers, Éric Tanter. Computational contracts. 2011.

[5] C. Clifton and G. T. Leavens. Spectators and assistants: Enabling modular aspect-oriented reasoning, 2002.

[6] D. S. Dantas and D. Walker. Harmless advice. *SIGPLAN Not.*, 41:383–396, January 2006. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1111320.1111071. URL `http://doi.acm.org/10.1145/1111320.1111071`.

[7] S. D. Djoko, R. Douence, and P. Fradet. Aspects preserving properties. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '08, pages 135–145, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-977-7. URL `http://doi.acm.org/10.1145/1328408.1328429`.

[8] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, 2000.

[9] W. Havinga, L. Bergmans, and M. Aksit. Prototyping and composing aspect languages. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 180–206, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70591-8. URL `http://dx.doi.org/10.1007/978-3-540-70592-5`$_9$.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[11] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005. ISBN 0321295358.

[12] Y. A. Liu. Efficiency by incrementalization: An introduction. *Higher Order Symbol. Comput.*, 13:289–313, December 2000. ISSN 1388-3690. doi: 10.1023/A:1026547031739. URL `http://portal.acm.org/citation.cfm?id=369129.369135`.

[13] Y. A. Liu, S. D. Stoller, M. Gorbovitski, T. Rothamel, and Y. E. Liu. Incrementalization across object abstraction. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 473–486, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. URL `http://doi.acm.org/10.1145/1094811.1094848`.

[14] C. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, Northeastern University, Boston, MA, USA, 1997.

[15] A. Marot and R. Wuyts. Composing aspects with aspects. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 157–168, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-958-9. doi: http://doi.acm.org/10.1145/1739230.1739249. URL `http://doi.acm.org/10.1145/1739230.1739249`.

[16] E. Recebli. Pure aspects. Technical report, Wolfson College, University of Oxford, 2005.

[17] J. Zhao and M. Rinard. Pipa: a behavioral interface specification language for aspectj. In *Proceedings of the 6th international conference on Fundamental approaches to software engineering*, FASE'03, pages 150–165, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-00899-3. URL `http://dl.acm.org/citation.cfm?id=1762980.1762995`.