abc Technical Report No. abc-2006-4

Eric Bodden[1], Laurie Hendren[1], Ondřej Lhoták[2]

[1] McGill University, Montréal, Québec, Canada
[2] University of Waterloo, Waterloo, Ontario, Canada

December 21, 2006

---

aspectbench.org

Contents

List of Figures

Abstract

In runtime monitoring a programmer specifies a piece of code to execute when a trace of events occurs during program execution. Our work is based on tracematches, an extension to AspectJ, which allows programmers to specify traces via regular expressions with free variables. In this paper we present a staged static analysis which speeds up trace matching by reducing the required runtime instrumentation.

The first stage is a simple analysis that rules out entire tracematches, just based on the names of symbols. In the second stage, a points-to analysis is used, along with a flow-insensitive analysis that eliminates instrumentation points with inconsistent variable bindings. In the third stage the points-to analysis is combined with a flow-sensitive analysis that also takes into consideration the order in which the symbols may execute.

To examine the effectiveness of each stage, we experimented with a set of nine tracematches applied to the DaCapo benchmark suite. We found that about 25% of the tracematch/benchmark combinations had instrumentation overheads greater than 10%. In these cases the first two stages work well for certain classes of tracematches, often leading to significant performance improvements. Somewhat surprisingly, we found the third, flow-sensitive, stage did not add any improvements.

1 Introduction

Various mechanisms have been proposed for monitoring programs as they run. Aspect-oriented programming (AOP) is one approach where a programmer specifies which events should be intercepted and what actions should be taken at those interception points. More recently, this concept of event matching has been further expanded to include matching of *traces of events* [1, 8, 10, 12]. While this expanded notion of matching on traces is much more powerful, it can also lead to larger runtime overheads since some information about the runtime history must be maintained in order detect matching traces.

In this paper, we examine the problem of improving runtime performance of *tracematches*. Tracematches are an extension to AspectJ which allows programmers to specify traces via regular expressions of symbols with free variables [1]. Those variables can bind objects at runtime, a crucial feature for reasoning about object-oriented programs. When a trace is matched by a tracematch, with consistent variable bindings, the action associated with the tracematch executes. Trace matching is implemented via a finite-state-based runtime monitor. Each event of the execution trace that matches a declared symbol in a tracematch causes the runtime monitor to update its internal state. When the monitor finds a consistent match for a trace, it executes its associated action.

There are two complementary approaches to reducing the overhead for this kind of runtime monitoring. The first line of attack is to optimize the monitor itself so that each update to the monitor is as inexpensive as possible and so that unnecessary state history is eliminated. These approaches were presented by Avgustinov et al. and have been shown to greatly reduce overheads in many cases [3]. However, as our experimental results show, there remain a number of cases where the overhead is still quite large.

Our work is the second line of attack, to be used when significant overheads remain. Our approach is based on analysis of both the tracematch specification and the whole program being monitored. The analysis determines which events do not need to be monitored, i.e. which instrumentation points can be eliminated. In the best case, we can determine that a tracematch never matches and all overhead can be removed. In other cases, our objective is to minimize the number of instrumentation points required, thus reducing the overhead.

In developing our analyses, we decided to take a staged approach, applying a sequence of analyses, starting with the simplest and fastest methods and progressing to more expensive and more precise analyses. An important aspect of our research is to determine if the later stages are worth implementing, or if the earlier stages can achieve most of the benefit. We have developed three stages where each stage adds precision to our abstraction. The first stage, called the *quick check* is a simple method for ruling out entire tracematches, just using the names of symbols. Our second stage uses a demand-driven points-to analysis [9], along with a flow-insensitive analysis of the program, to eliminate instrumentation points with inconsistent variable bindings. The third stage combines the points-to analysis with a flow-sensitive analysis that takes into consideration the order in which events may occur during runtime.

We have evaluated our approach using the DaCapo benchmark suite [4] and a set of 9 tracematches. We found that even though previous techniques often kept the runtime overhead reasonable, there were a significant number of benchmark/tracematch combinations which led to a runtime overhead greater than 10%. We focused on these cases and we found that our first two stages worked well for certain classes of tracematches. We were somewhat surprised to find that our third stage did not add any further accuracy, even though it was a flow-sensitive analysis, and we provide some discussion of why this is so.

This paper is organized as follows. Section 2 introduces tracematches, explains how they apply to Java programs, and gives some examples of where monitoring instrumentation can statically be shown to be unnecessary. In Section 3 we present our staged static analysis which performs such detection automatically. We carefully evaluate our work in Section 4, showing which problem cases our analysis can handle well, but also which cases might need more work or will probably never be statically analyzable. In Section 5 we discuss related analyses, finally concluding in Section 6. There we also briefly discuss our intended scope for future work on the topic.

2 Background

A tracematch defines a runtime monitor using a declarative specification in the form of a regular expression. An example is shown in Figure 1. This tracematch checks for illegal program executions where a vector is updated while an enumeration is iterating over the same vector. First, in lines 2-5 it defines a plain AspectJ pointcut capturing all possible ways in which a vector could be updated. The actual tracematch follows in lines 7-14. In its header (line 7) it declares that it will bind a Vector v and an Enumeration e. Then in lines 8-10 it defines the alphabet of its regular expression by stating the symbols create, next and update. The first one, create, is declared to match whenever any enumeration e for v is created, while next matches when the program advances e and update on any modification of v.

Line 12 declares a regular expression that states when the tracematch body (also line 12) should execute. This should be the case whenever an enumeration was created, then possibly advanced multiple times and then at least one update to the vector occurs, lastly followed by another call to Enumeration.nextElement().

```
1  aspect FailSafeEnum {
2    pointcut vector_update() :
3      call(* Vector.add*(..))  ||  call(* Vector.clear())  ||
4      call(* Vector.insertElementAt(..))  ||  call(* Vector.remove*(..))  ||
5      call(* Vector.retainAll(..))  ||   call(* Vector.set*(..));
6
7    tracematch(Vector v, Enumeration e) {
8         sym create after returning(e) : call(* Vector+.elements()) && target(v);
9         sym next before : call(Object Enumeration.nextElement()) && target(e);
10        sym update after : vector_update() && target(v);
11
12       create  next* update+ next { /* handle error */ }
13    }
14 }
```

Figure 1: Safe enumeration tracematch

The declarative semantics of tracematches state that the tracematch body should be executed for any sub-sequence of the program execution trace that is matched by the regular expression with a consistent variable binding. A variable binding is consistent when at every joinpoint in the sub-sequence each variable is bound to the same object. Those declarative semantics demand that, conceptually, one evaluates a tracematch by running an unbounded number of state machines at the same time, each state machine treating another variable binding.

Internally, any tracematch is represented by a single non-deterministic automaton that evaluates all of those sub-sequences at the same time. In order to simulate an unbounded number of different state machines, each state holds a *constraint*, which is a set of *disjuncts* in Disjunctive Normal Form. Each disjunct is a mapping from variables to objects and so represents a single partial match. Whenever a disjunct in the state machine reaches a final state, the tracematch body is executed with the variable binding held in this disjunct.

When compiling a program that contains a tracematch, the compiler firstly generates program code for this state machine and secondly instruments the program such that it notifies the state machine about any joinpoint of interest, i.e. any joinpoint that matches any of the declared symbols of any declared tracematch. When such a notification occurs, the related state machine updates its internal state accordingly, i.e. propagates disjuncts from one state to another, generates possibly new disjuncts or discards disjuncts.

Figure 2 shows the automaton for the safe enumeration tracematch. As one can see, it looks very much like the most intuitive automaton for this pattern but augmented with additional loops (here dashed) on each non-initial and non-final state. Those loops here appear dashed, because they are of a special kind and have different semantics from usual edges. They are called *skip loops*.

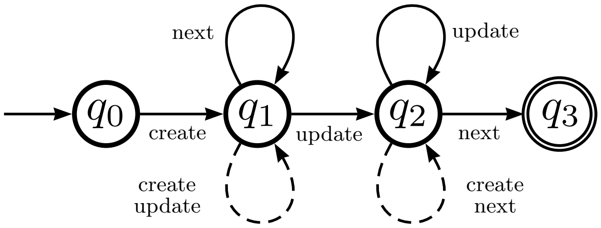

Figure 2: Finite automaton for safe enumeration tracematch of Figure 1

The purpose of skip loops is to discard partial matches. The safe enumeration pattern is unfortunately one of the few where their relevance is somewhat hidden. Hence, in order to explain the purpose of skip loops, consider Figure 3. This figure shows the automaton for the tracematch *HasNext* which uses a pattern "next next" over a symbol alphabet {next,hasNext}.

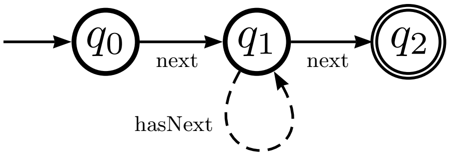

Figure 3: Finite automaton for tracematch pattern *HasNext*

The intent of this tracematch is to find all cases where there are two calls to Iterator.next(), with no call to hasNext() in between. Since the tracematch alphabet contains both the next and hasNext symbols, matching on the pattern "next next" implies that there was no call to hasNext() between the two next events. This implicit negation is formulated in tracematches by including a symbol in the alphabet but not in the pattern, just like it is done with hasNext here. During runtime, when next() is called on a particular iterator i_1, a disjunct $\{i \mapsto i_1\}$ is generated on state q_1. Now, if there is a call to next() *is* followed by a call to hasNext(), this binding can be discarded, because at least for the moment for this particular iterator i_1 the requirement is fulfilled. This is exactly what the skip loop on state q_1 achieves. When hasNext() is called on i_1, it *discards* the partial match for i_1 by deleting its disjunct from q_1.

Running example To get a better feeling for the semantics of tracematches and the implications of our optimization, let us look at the following running example. Assume that we want to evaluate the safe enumeration tracematch over the code shown in Figure 4. The code does not do anything meaningful but it allows us to explain how tracematches work and which cases the different stages of our analysis can handle. In lines 5-10, the program modifies and iterates over the vector vector and does so in a safe way. In lines 12-15 it modifies and iterates over another vector globalVec. It also calls doEvil (..) , modifying globalVec while the enumeration is used. This is a case which the tracematch should capture. In lines 17-18 a third vector and an enumeration over this vector are created.

The comments on the right-hand side of the figure label *allocation sites*, i.e. places where vectors or enumerations are allocated. We use those labels to denote objects. An object is labeled with the site at which it was allocated.

```
 1 class Main {
 2    Vector globalVector = new Vector();                                          //v2
 3
 4    void someMethod() {
 5       Vector vector = new Vector();                                             //v1
 6       vector.add(”something”);
 7       for (Enumeration iter = vector.elements(); iter .hasMoreElements();) {   //e1
 8          Object o = iter.nextElement();
 9          doSomething(o);
10       }
11
12       globalVector.add(”something_else”);
13       Enumeration it2 = globalVector.elements();                                //e2
14       doEvil(o);
15       it2 .nextElement();
16
17       Vector copyVector = new Vector(globalVec);                                //v3
18       Enumeration it3 = copyVector.elements();                                  //e3
19    }
20
21    void doSomething(Object o)
22    { /* does not touch globalVector */ }
23
24    void doEvil(Object o)
25    { globalVector.remove(o); }
26 }
```

Figure 4: An example program

In our static analysis, we attempt to remove unnecessary *shadows*, i.e. unnecessary instrumentation points in the base program that trigger the tracematch at a point where it can statically be decided that the particular event can never be part of a complete match. Let us first manually find such places in the code for our running example.

Shadows occur wherever a tracematch symbol matches a part of the program. In our example, this means we have shadows at each creation of an enumeration, each update of a vector and each call to Enumeration.nextElement(). However, when looking at the code more carefully, it should become clear that not all of the shadows are necessary for the example program.

In particular, the first sequence of statements in the lines 5-10 is safe in the sense that the pair of vector and enumeration is used correctly and the tracematch will not be triggered. Consequently, no shadows need to be inserted for this part of the program. Lines 12 to 15 and line 25 show an unsafe enumeration that should trigger the tracematch. So generally, shadows here need to stay in place. However, looking at the code more carefully, one can see that actually the shadow at line 12 is also superfluous, because the match that triggers the tracematch does not start before line 13, where the enumeration is actually created. In lines 17 to 18 we have a pair of vector and enumeration where the vector is never even updated. For this piece of code it should be obvious that no shadows are required.

In the next section we describe our static program analyses which automatically identify the unnecessary shadows.

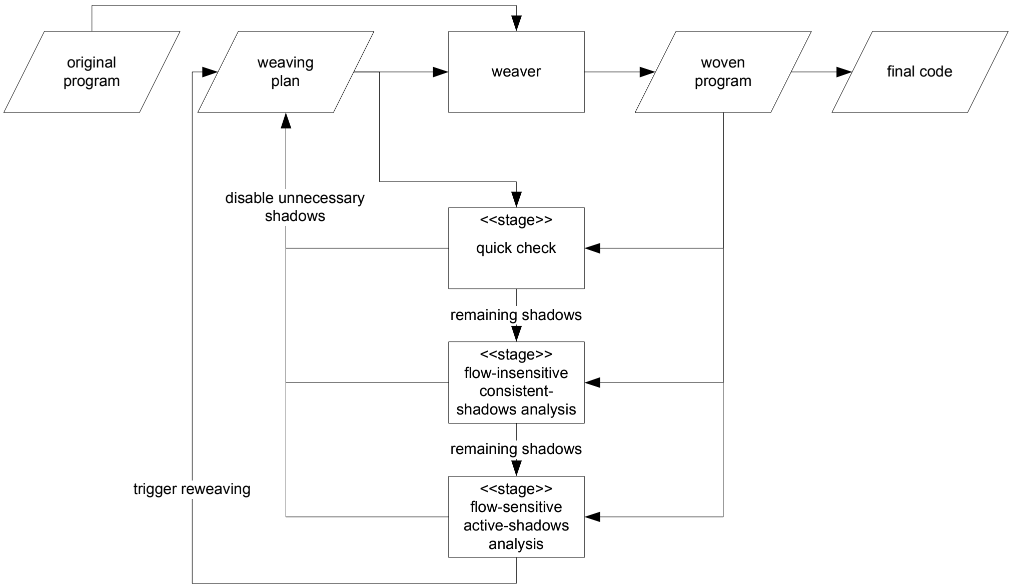

Figure 5: Outline of the staged analysis

3 Staged analysis

Our analysis is implemented using the *reweaving* framework [2] in the AspectBench Compiler (`abc`), which implements tracematches. The basic idea is that the compiler first determines which points in the program correspond to places where instrumentation should be woven. These instrumentation points are commonly called *shadows* and the list of shadows and what to weave at those shadows is called the *weaving plan*. In order to determine which shadows are unnecessary, a first weaving is done according to the original weaving plan. This results in a woven program on which our proposed staged analyses are performed. The analysis determines which shadows are unnecessary and removes them from the weaving plan. The program is then rewoven according to this new plan, resulting in a more efficient woven program.

The analyses are performed on the Jimple[1] representation of the woven program. In this representation, all instructions corresponding to tracematch shadows are specially marked so that they can be recognized.

An outline of the staged analyses is shown in Figure 5. Each stage uses its own abstract representation of the program and applies an analysis to this representation in order to find unnecessary shadows. After each stage, those shadows are removed so that subsequent stages do not have to consider them any more in their analyses.

The crucial point of this approach is that the earlier stages (on the top of the figure) are more coarse-grained than later ones. Hence they use a more lightweight abstract representation of the program and execute much faster. By applying stages in this order we make sure that at each stage only those shadows remain active which could not be proven unnecessary using an easier approach.

Figure 5 shows the three analysis stages we apply here as boxes. First we apply a quick check that determines if a tracematch can apply to a given program at all, just by looking at shadow counters, which are already computed during the initial weaving process. The second stage uses points-to information in order to find groups of shadows which could during runtime possibly lead to a complete match by possibly referring to a consistent variable binding. The third and final stage is flow-sensitive, meaning that here we look at all those groups of shadows and try to determine in which order their shadows could possibly be executed when the program is run. In many cases, all shadows might already be removed in an early stage. When this happens, later stages are not executed at all. In any case, however, the code is eventually rewoven

---

[1] Jimple is a fully typed three-address code representation of Java bytecode provided by Soot, which is an integral part of `abc`.

using the updated weaving plan, i.e. weaving only those shadows that have not been disabled before.

In the following subsections, we explain all three stages as well as their required program abstractions in more detail.

3.1 Quick check

One use of tracematches is to specify behavioural constraints for Java interfaces. When developing a library, for example, one could ship it together with a set of tracematches in order to enforce that objects of that library are used in a certain way or in certain combinations. Consequently, it might often be the case that certain tracematches might never match or that only some of their symbols match, simply because the client uses only parts of the library.

For example, imagine a program which uses vectors but no enumerations. In this case, when applying the safe enumeration tracematch, `abc` would normally instrument all locations where a vector is updated, although an analysis of the whole program would show that the tracematch can never match.

The abstract program representation used by the quick check is simply a mapping from tracematch symbols to number of shadows at which the tracematch symbol may match. Those numbers are obtained during the initial weaving phase. For our running example, we would obtain the following mapping because enumerations are created at three places, they are advanced at two places and vectors are updated at three places.

$$\{create \mapsto 3, next \mapsto 2, update \mapsto 3\}$$

We use these counts, plus the tracematch automaton to determine if the tracematch could ever match. The key idea is that if a symbol that is necessary to reach a final state in the automaton has a count of 0 (i.e. no instances in the program under analysis), then there is no possibility that the tracematch could match.

We implement this check as follows. For each tracematch, we remove edges from its automaton whose labels have shadow counters of 0. Then we check to see if a final state can still be reached. If the final state can't be reached, the entire tracematch is removed and all its associated shadows are disabled.

If the quick check fails for a tracematch, i.e. all necessary symbols were applied at least once, we have to change to a more detailed level of abstraction which leads us to the flow-insensitive analysis.

3.2 Flow-insensitive consistent-shadows analysis

Tracematches can only match a trace if the trace refers to symbols with *consistent variable bindings*. In the quick check we just used the names of the symbols and did not use any information about variable bindings. In contrast, the flow-insensitive *consistent-shadows analysis* uses points-to analysis results to determine when shadows cannot refer to the same object and thus cannot lead to consistent variable bindings. The analysis is flow-insensitive in the sense that we do not consider the order in which the shadows execute.

3.2.1 Preparation:

In order to prepare for this analysis, we first need points-to information for each variable involved in the tracematches. We compute the required points-to information as follows.

First we build a call graph using the Soot/`abc` internal Spark framework. Spark builds a call graph for the whole program on-the-fly, i.e. by computing points-to information at the same time as discovering new call edges due to new points-to relationships. This first phase results in a complete call graph and context-insensitive points-to information for the whole program.

In our preliminary experiments we found that the context-insensitive points-to analysis was not always precise enough, and so we added a second phase that computes context-sensitive points-to analysis for those variables bound by shadows. For this second phase we use Sridharan's demand-driven refinement analysis for points-to sets [9]. This algorithm starts with the call graph and context-insensitive results from the first

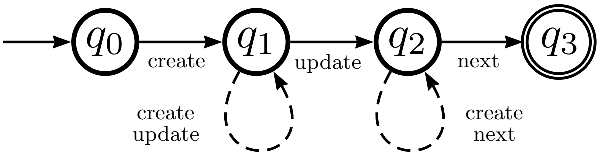

Figure 6: Automaton from Figure 2 with loops due to Kleene-* sub-expressions removed

phase and computes context information for *a given set of variables* often yielding more precise points-to information for these variables. The advantage of this approach is that we need to perform this rather expensive computation only for variables that are really bound by shadows. In all our benchmarks this was always fewer than 5% of the total number of variables. (For exact numbers, see Section 4.)

Our running example illustrates quite clearly why context-sensitive points-to analysis is required. In this case, context information is necessary to distinguish the different enumerations from each other. Since all are created within the factory method elements(), without such context-sensitivity, all enumerations would be modelled as the same abstract object — their common creation site inside the method elements(). Allocation sites e1, e2 and e3 would collapse, and so the analysis would have to assume that all three enumerations might actually be one and the same, penalizing the opportunities for shadow removal.

3.2.2 Building path infos:

At runtime, a tracematch matches when a sequence of events is executed which is matched by the given regular expression, however *only* if those events occurred with a *consistent variable binding.* The idea of the flow-insensitive analysis stage is to identify groups of shadows which could potentially lead to such a consistent variable binding at runtime.

At runtime, a final state in the tracematch automaton can be reached from any initial state, generally over multiple paths. A first observation is that edges which originate from symbols within a Kleene-* sub-expression are always optional. For example, in the safe enumeration tracematch (Figure 1), the initial iter $*$ *may*, but does not have to, match a joinpoint in order for a sequence to lead to a complete match. Hence, we first generate an automaton using a customized Thompson construction that omits "starred" sub-expressions, modelling them with an ϵ-edges (which are then later on inlined).

Figure 6 shows the fail safe enumeration automaton after this transformation. We call this representation the *reduced* automaton. Note that skip loops are preserved in this representation, however no other strongly-connected components remain. Hence, we can enumerate all paths through this automaton which do not lead through a skip loop.

Then, for each such path we compute a *path info.* A path info contains information about what symbols the edges on the path are labeled with, split into symbols of skip loops and "normal" edges. For the latter, we will later on also need the information of how often such a label occurs on the path. This yields the following definition.

Definition 1 (Path info)**.** Let *path* be a path from an initial to a final state. A path info *info(path)* consists of a set *skip-labels(info)* and a multi-set *labels(info)*, defined as follows. If $path = (p_1, l_1, q_1) \dots (p_n, l_n, q_n)$, then

$$\mathit{labels} := \biguplus_{1 \leq i \leq n} \{l_i\}$$

$$\mathit{skip\text{-}labels} := \bigcup_{1 \leq i \leq n} (\mathit{skip\text{-}labels}(p_i) \cup \mathit{skip\text{-}labels}(q_i))$$

where $\uplus$ denotes the union for multi-sets. (A multi-set or bag is a similar to a set but can hold the same object multiple times.) In the following, we denote multi-sets with square brackets of the form $[a, a, b]$.

For a tracematch *tm*, we denote the set of all its path infos by *infos(tm).* It is defined as the set of all path infos for all paths through the reduced automaton of *tm*.

For the fail safe enumeration tracematch in our example, only one path info exists, which means that the set *infos* has the following form.

$$\begin{aligned}&\mathit{infos}(\text{FailSafeEnum}) = \\ &\quad \{(\ \mathit{labels} = [\text{create}, \text{update}, \text{next}],\ \ \mathit{skip\text{-}labels} = \{\text{create}, \text{update}, \text{next}\}\ )\}\end{aligned}$$

The reader should not be misled by this example. In general, *labels* and *skip-labels* do not have to coincide. For example, for the automaton in Figure 3, we would have a single path info with $\mathit{labels} = [\text{next}, \text{next}]$ and $\mathit{skip\text{-}labels} = \{\text{hasNext}\}$.

3.2.3 Building groups of shadows with possibly consistent binding:

With the path infos computed, we have information about what combinations of shadows are *required* for a complete match. In the next step we try to find groups of shadows that fulfil this requirement. This means that we look for groups of shadows which contain the labels of the *labels* field of a path info and, in addition, share a possibly consistent binding. But before we define shadow groups, let us first formally define how a single shadow is modelled.

Definition 2 (Shadow)**.** A shadow s of a tracematch tm is a pair $(lab_s, bind_s)$ where lab_s is the label of a declared symbol of tm and $bind_s$ is a variable binding, modelled as a mapping from variables to points-to sets. In the following we assume that the mapping $bind_s$ is extended to a total function that maps each variable to the full points-to set $\top$ if no other binding is defined:

$$bind_s(v) := \begin{cases} bind_s(v), \text{if } bind_s(v) \text{ explicitly defined} \\ \top, \text{otherwise} \end{cases}$$

Here, $\top$ is defined as the points-to set for which holds that for all points-to sets $s : s \cap \top = s$.

In our running example, the update shadow in line 6 would be denoted by $(\text{update}, \{v \mapsto \{v1\}\})$ as the only objects v can point to are objects being created at creation site $v1$.

Definition 3 (Shadow group)**.** A *shadow group* is a pair of a multi-set of shadows called *label-shadows* and a set of shadows called *skip-shadows*. We call a shadow group *complete* if it holds that: (1) its set of labels of its *label-shadows* contains all labels of a path info of a given tracematch; and (2) its set of *skip-shadows* contains all shadows which have the label of a skip loop of this tracematch and a points-to set that overlaps with the one of a label shadow.

This definition implies that a complete shadow group has: (1) enough symbols in its *label-shadows* to drive a tracematch state machine into a final state; and (2) that all shadows that could interfere with a match via skip loops are contained in *skip-shadows*.

Definition 4 (Consistent shadow group)**.** A *consistent* shadow group g is a shadow group for which all variable bindings of all shadows in the group have overlapping points-to sets for each variable. More formally, if $vars$ is the set of all variables of all shadows in g, then it must hold that:

$$\forall s_1, s_2 \in (\mathit{label\text{-}shadows} \cup \mathit{skip\text{-}shadows})\ \forall v \in vars :\ \ bind_{s_1}(v) \cap bind_{s_2}(v) \neq \emptyset$$

Conceptually, a complete and consistent shadow group is the static representation of a possibly complete match at runtime. For such a shadow group, there is a possibility that if the label shadows in this group are executed in a particular order at runtime, the related tracematch could match. Skip shadows in the same group could prevent such a match when executed.

In particular, if a shadow group has a multi-set of label shadows which is *not* consistent this means that no matter in which order those shadows are executed at runtime, this group of shadows can *never* lead to a complete match. Consequently, we can safely disable all shadows which are not part of any consistent shadow group.

The algorithm we use in order to compute all complete and consistent shadow groups is shown as Algorithm 1. For each tracematch and for each path-info of this tracematch, the algorithm first computes a cross

Algorithm 1 complete and consistent shadow groups

```
$shadow\_groups := \emptyset$
for tracematch $tm$ do
  for each path info $info$ in $infos(tm)$ do
    /* cross product over all shadows for all labels in the path info */
    $cross\_product := \prod_{l \in labels(info)} [s \in shadows(tm) \mid label(s) = l]$
    /* filter out results with non-overlapping variable binding */
    for $shadow\_set \in cross\_product$ do
      for $s_1, s_2 \in shadow\_set$ do
        for $var \in \{v \mid v$ bound by $bind_{s_1} \wedge v$ bound by $bind_{s_2}\}$ do
          if $bind_{s_1}(var) \cap bind_{s_2}(var) = \emptyset$ then
            $cross\_product := cross\_product \setminus \{shadow\_set\}$
          end if
        end for
      end for
    end for
    /* find skip shadows */
    for $bag \in cross\_product$ do
      $skip\_shadows := \emptyset$
      for label $l$ in $skip\text{-}labels(info)$ do
        for skip shadow $s_s \in \{s \in shadows(tm) \mid label(s) = l\}$ do
          for label shadow $s_l \in bag$ do
            $overlaps := true$
            for variable $v$ bound by $s_s$ do
              if $bind(s_s) \cap bind(s_l) = \emptyset$ then
                $overlaps := false$
              end if
            end for
            if $overlaps = true$ then
              $skip\_shadows := skip\_shadows \cup \{s_s\}$
            end if
          end for
        end for
      end for
      /* add new shadow group as pair of label and skip shadows */
      $shadow\_groups = shadow\_groups \cup \{(bag, skip\_shadows)\}$
    end for
  end for
end for
```

product over the multi-set containing for each label $l \in \mathit{labels}(\mathit{info})$ the set of shadows with this label (line 5). The result is a set of multi-sets, where each multi-set at this stage can however still contain shadows with an inconsistent variable binding. Hence, next we filter out all multi-sets containing two or more shadows which have a non-overlapping points-to set bound to the same variable (lines 7-15). This leaves us with all possible multi-sets of (label) shadows which have a consistent variable binding.

Now for each such multi-set, we still have to construct the set of skip shadows with overlapping points-to sets. This is what is done in the following iteration in lines 19-33. The final combination of multi-set and set of skip shadows then becomes a consistent shadow group (line 35).

Our actual implementation is a little bit smarter, constructing the cross product in such a way that inconsistent results are even not at all computed in the first place.

Based on the consistent shadow groups, flow-insensitive shadow removal is then quite easy. For each shadow that exists in the program, we look up if it is member of at least one consistent shadow group (i.e. it is either a label-shadow or a skip shadow of that group). If this is *not* the case, the shadow can never be part of a complete, consistent match and can safely be removed.

In our running example, this is true for the shadow in line 18. Since for this create-shadow there exist neither an update shadow for the same vector nor a next-shadow for the same enumeration, there can no complete and consistent shadow set be computed that contains the create-shadow.

Here we can also see that context information for points-to sets is important. As noted earlier, without context information, all enumerations would be modelled by the same abstract object. Hence, in this case, the points-to sets for those shadows would overlap and the shadow in line 18 *could* be part of a complete and consistent match, in combination with one of the vectors globalVector or vector.

If after this stage there are still shadows remaining we know that there exist groups of shadows which have a possibly consistent variable binding. This means that if they are at runtime executed in a particular order, the related tracematch could indeed be triggered. Hence, it is only natural that in the next stage we compute information that tells us whether those shadows could can actually be executed in the required order or not. This leads us to the flow-sensitive consistent-shadows analysis stage.

3.3 Flow-sensitive active-shadows analysis

As input to this stage we expect a set of complete and consistent shadow groups as well as a complete call graph, both of which were already computed earlier. (In the following, when we refer to a shadow group, we always assume it is complete and consistent.)

In order to determine in which order shadows could be executed during runtime, we need a flow-sensitive representation of the entire program. It is a challenge to build such a representation efficiently. Since any Java program is potentially multi-threaded, we also have to take into account that shadows could be executed by multiple threads. This makes it more difficult to determine whether a shadow may happen before or after another.

A tracematch can be defined to be *per-thread* or *global*. For a per-thread tracematch, a separate automaton is executed for each thread, and only events from that one thread affect the automaton. A global tracematch is implemented using a single automaton which processes events from all threads. Hence, for global tracematches, our analysis must handle multi-threading soundly.

Also, a whole program abstraction may potentially be very large. There might potentially be thousands of shadows spread over hundreds of methods. Hence it is important that we keep our program abstraction concise at all times.

3.3.1 Handling of multi-threading

We handle the first problem of multi-threading conservatively. In the preparation phase for the flow-insensitive analysis stage, we already constructed a complete call graph. In this call graph, call edges that spawn a thread are already specially marked. Using this information, we can easily determine by which threads a given shadow can be executed.

Then, in an initial preprocessing step, we filter the list of all shadow groups in the following way. If a shadow group is associated with global tracematch and contains shadows which are possibly executed by multiple threads, we "lock" all its shadows (i.e. they will never be removed, not by this stage nor by subsequent stages) and remove the group from the list. The locking makes the analysis conservative with respect to threads. For the resulting list of shadow groups we then know that all shadows contained in a group are only executed by the same thread. Hence, no additional treatment of multi-threading is necessary.

3.3.2 A flow-sensitive whole-program representation

In the next step, we build a flow-sensitive representation of the whole program. Such a representation naturally has to depend on the static call graph of the program.

Call graph filtering In order to adhere to our principle of keeping our abstraction as small as possible at all times, we first filter this call graph in the following way. If in the call graph there is an outgoing call edge in whose transitive hull there is never any method of interest reachable (i.e. a method that contains a shadow), this edge and its entire transitive hull is removed.

Per-method state machines For each method that remains in this filtered call graph, we know that either it is "interesting" because it contains a shadow or it calls another interesting method. For those methods we do need flow-information, i.e. information about the order in which shadows may be executed during runtime and in which other methods may be called.

We encode such flow-information by a finite state machine for each such method. In order to generate this abstraction for a particular method, we first generate a control-flow graph for the method. This graph encodes the entire possible control flow, including exceptional one. Each node in this graph represents a statement.

We generate the per-method state machine by turning edges in the control-flow graph into transitions in the state machine. For each statement that is neither an invoke statement nor contains a shadow, we generate an ϵ transition. For each statement that contains a shadow s, we generate an edge labeled with s. For each statement that contains an invoke expression ie, we generate an *invoke-edge* labeled with all possible call targets of ie. Since our call graph is filtered, there may be invoke expressions for which there is no target. In this case, we label the edge with ϵ. If an entire method *body* contains a shadow s (e.g. an **execution** shadow), we generate an edge labeled with s which, in the case of a *before* symbol, precedes all other edges in this state machine or, in the case of an *after* symbol follows all other edges.

Before we proceed, we then inline all epsilon transitions in those state machines and minimize[2] them one by one by using standard algorithms known from automata theory. Then, in a last step, we equip each state machine with a unique start and end state. The start state is connected to all initial states via ϵ-transitions and so is any final state to the end state. (Since we do not use the state machine for language acceptance, the notion of a final state is for our purposes is not important in any other way.)

Inter-procedural combination In order to obtain an abstract representation of the entire program, we then combine all the per-method state machines inter-procedurally. For each invoke-edge we generate ϵ transitions from the source state of this edge to each start state of each possible call target. We generate a second transition from each end state of those call targets to the target state of the original invoke-edge. Finally, the original invoke-edge is removed.

Figure 7 shows this step for our running example. This figure shows three state machines for the methods doSomething(), someMethod() and doEvil() and how they become inter-procedurally combined.

Note that this way of combining automata is context-insensitive. In the resulting automaton there exist more paths than are actually realizable at runtime. One could branch out from a call statement c_1 to a

---

[2] The NFA is determinized and then minimized. We use the resulting DFA if it has fewer edges than the original NFA and the original NFA otherwise.

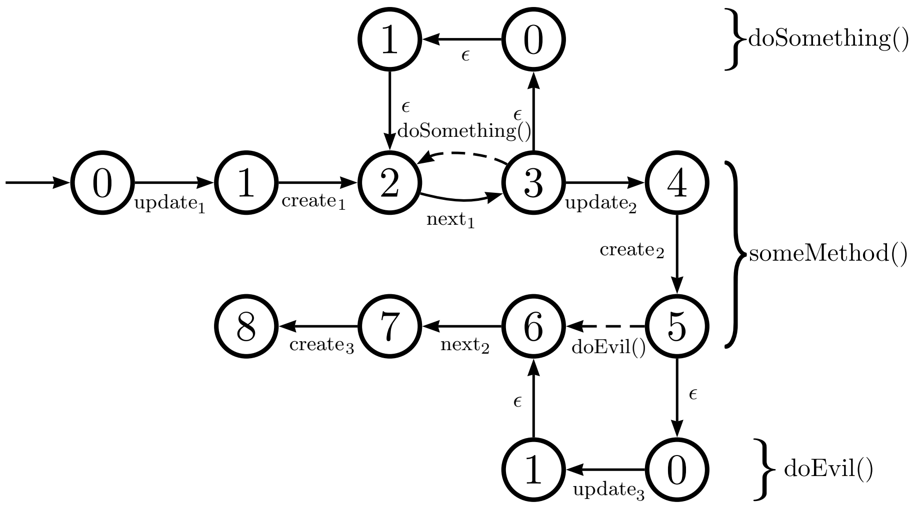

Figure 7: complete state machine for the running example during construction; the numbers at the shadow labels are for identification purposes only

possible call target t but then return to another caller c_2 of the same call target. This way of automaton construction is relatively cheap but gives away precision.

Abstract interpretation via fixed point iteration This whole-program state machine is the input to our actual flow-sensitive analysis. The task of this analysis is to compute if some part of this state machine contains such a path that when executing the program along this path at runtime, the tracematch could match. To us, it appeared that the most sensible way to do so is to perform a complete abstract interpretation of the actual tracematch machinery.

This abstract interpretation evaluates an *abstract counterpart* of the actual tracematch automaton (i.e. the one that is evaluated at runtime) over the whole-program state machine. Since the latter can have cycles, we employ, as is usually done in data-flow analysis, a fixed-point iteration.

The only two differences of the abstract interpretation in comparison to the evaluation at runtime are the following. Firstly, wherever the actual implementation binds variables to objects, the abstract interpretation binds them to points-to their sets. Consequently, where at runtime, the implementation checks for reference equality, the abstract interpretation checks for overlapping points-to sets. In the case of skip loops, variable bindings are not updated at all (due to the lack of must-alias information). Only the history is updated in this case.

The other difference is that while during runtime, the implementation exposes no explicit information about where partial matches occurred, the static abstraction needs to determine which shadows were visited on the way to a final state. Hence, in each disjunct, we store an additional *history* component: the set of shadows which this disjunct was propagated through. When such a disjunct reaches a final state, we can inspect its history and so determine which shadows need to be kept active in order to trigger the match for this disjunct at runtime.

We start off with an initial tracematch configuration in the unique initial state of this whole-program state machine, which represents that when the program starts, the tracematch is in its initial configuration. In terms of Figure 7, this would associate the following configuration with the initial node of the whole-program state machine.

$$(q_0 \mapsto \textbf{true}, q_1 \mapsto \textbf{false}, q_2 \mapsto \textbf{false}, q_3 \mapsto \textbf{false})\}$$

Here **true** is the constraint $\{(\emptyset, \emptyset)\}$ consisting of a single disjunct with empty variable binding and history while **false** is the empty constraint (modelled by the empty set of disjuncts).

This configuration is then driven through the whole-program state machine until a fixed-point is reached. Whenever a disjunct is propagated, its history component is updated with the shadow that triggered the propagation. Due to internals of the tracematch machinery, this is only the case if a constraint really moves to a new state. For example at node number 1 (of method someMethod()) in Figure 7, the configuration is still same the initial configuration as above. At node number 2, one would get

$$(q_0 \mapsto \mathbf{true}, q_1 \mapsto \{(\{v \mapsto v_1, e \mapsto e_1\}, \mathit{create}_1)\}, q_2 \mapsto \mathbf{false}, q_3 \mapsto \mathbf{false})\}$$

stating that the abstract tracematch automaton has one single partial match in state 1 with a variable mapping of $\{v \mapsto v_1, e \mapsto e_1\}$ which was produced by shadow create_1.

At merge-points (here only the same node number 2), configurations are merged by joining their constraints per state, i.e. two constraints with mappings $q_i \mapsto \{d_1, d_2\}$ and $q_i \mapsto \{d_2, d_3\}$ (for disjuncts d_1, d_2, d_3) is merged to a constraint with mapping $q_i \mapsto \{d_1, d_2, d_3\}$.

During the computation of the fixed point, whenever a disjunct reaches a final state of a configuration, we copy its history to a *global set* of *active shadows*. When the fixed point is reached, we know that all shadows in this set may lead to a complete match, with the binding that is stored in the disjunct, and hence have to be retained. All shadows which are never added to this set during the fixed point computation can safely be discarded.

Performance improvements The aforementioned fixed point computation generally works but it might not be very efficient. Hence, we apply two different performance optimizations, one of which does not sacrifice precision and one of which does.

The general problem is that the set of possible different disjuncts is quite large. Consequently it might take a long time and a lot of memory before the fixed point is reached. In the example, the whole-program state machine has only 8 non-ϵ edges. In realistic benchmarks, this number can however well be around 1000.

If this happens, two different cases can occur. In the first case (the good case), points-to sets are relatively precise. Then, we can apply the following trick to "shrink" the whole-program state machine without losing precision. We simply do not perform the fixed point iteration once but rather once for each shadow group. In each such iteration, all edges with shadows which are not part of the group are treated as ϵ transitions, simply copying configurations without modification. (This leads back to the idea of the declarative semantics of tracematches where multiple automata are executed simultaneously, one for each binding.)

In the running example, this would mean that first we evaluate only the sub-automaton on the left-hand side of node 3, then the one between nodes 3 and 7 and then the one between nodes 7 and 8. As a consequence, disjuncts "travel" smaller distances through the graph and so we gain disjuncts with smaller history components, hence as a consequence less disjuncts in general.

In the second case (the hard one), points-to sets overlap a lot, either because indeed the same objects trigger shadows over and over again or because the computed points-to sets are imprecise. This means that each shadow group is relatively large. In particular this is the case for the *skip-shadows* component of the shadow group; the *label-shadows* component has bounded size by construction. Hence, the above trick does not help because still in each iteration a lot of edges would be enabled.

In this case, we apply the same trick as for multi-threading. We lock all shadows in the *skip-shadows* set, i.e. they will not be removed from the weaving plan. Because this is the case, we can then disable edges labeled with those shadows during fixed point iteration. In our benchmarks, we switched to this evaluation mode if $|\mathit{skip\text{-}shadows}| > 2 * |\mathit{label\text{-}shadows}|$.

The combination of those two techniques allowed us to compute the fixed point in all our benchmarks within a few minutes. However, it often gives away crucial precision, as we will see in Section 4.

Handling of skip loops One important issue that has not yet been explained is the handling of skip loops. As explained earlier, the purpose of a skip-loop is to discard partial matches under certain circumstances. In the example we gave in Section 2, this is the case when a disjunct of the form $\{i \mapsto i_1\}$ exists and then hasNext is invoked on the iterator *object* i_1.

pattern name	description
ASyncIteration	only iterate a synchronized collection c when owning a lock on c
FailSafeEnum	do not update a vector while iterating over it
FailSafeIter	do not update a collection while iterating over it
HashMap	do not change an object's hash code while it is in a hash map
HasNextElem	always call hasNextElem before calling nextElement on an Enumeration
HasNext	always call hasNext before calling next on an Iterator
LeakingSync	only access a synchronized collection using its synchronized wrapper
Reader	don't use a Reader after it's InputStream was closed
Writer	don't use a Writer after it's OutputStream was closed

Table I: description of tracematch patterns

Note that here, i_1 is a dynamic object. We can remove the partial match because we know that for i_1 the requirement "always call hasNext() before next()" is fulfilled. In the static case, however we have to model i_1 by a points-to set. Generally, points-to information is of the form that a variable i *may* point to the *abstract* object i_1. In particular, i_1 is usually a creation site, not an actual object. If the creation site i_1 is part of a loop, it could be executed multiple times during runtime. Hence, i_1 could model multiple actual objects. This causes generally problems when considering *liveness* properties of the form something *must* happen.

Without some must points-to information or the use of *strong updates* (as done in [5]), we cannot safely remove partial matches. Hence, our analysis mostly ignores skip loops which makes flow-sensitive recognition of patterns like the one mentioned above impossible. Our next phase of work will investigate the kinds of must analyses we need to handle skip loops more precisely.

4 Benchmarks

In order to evaluate the feasibility and effectiveness of our approach we applied our analysis to a combination of nine different tracematches, applied to version 2006-10 of the DaCapo benchmark suite [4]. The tracematches validate generic safety and liveness properties over common data structures in the Java runtime library. They are briefly described in Table I. As usual, all our benchmarks are available on `http://www.aspectbench.org/`, along with a version of `abc` implementing our optimization. In the near future we also plan to integrate the analysis into the main `abc` build stream.

The reader should note that we chose some of our tracematches because we knew they would be particularly challenging. For example, the *HashMap* tracematch binds a hash code which is an integer (and hence, not a pointer) value. The *ASyncIteration* benchmark uses an `if`-pointcut accessing the native method Thread.holdsLock(Object).

The tracematches *HasNext* and *HasNextElem* specify *liveness* properties, i.e. that something good must *always* happen. As mentioned in Section 3.3.2, the flow-sensitive analysis can't remove shadows for such properties without using must-alias information. The flow-insensitive analysis would also perform badly on such properties, simply because on almost every iterator hasNext() is called if and only if next() is called. Hence, for those benchmarks we expected a very low shadow removal ratio. Yet, the benchmarks helped us to validate the completeness of our implementation because we knew that in those cases neither the flow-insensitive nor the flow-sensitive stage should remove any shadows.

For our experiments we used the IBM J9 JVM version 1.5.0 SR3 (64bit) with 2GB RAM on a machine with AMD Athlon 64 X2 Dual Core Processor 3800+. We used the `-converge` option of the DaCapo suite which runs each benchmark multiple times to assure that the reported execution times are within a confidence interval of 3%.

Table II shows the run times of the benchmarks without our optimizations, but with the optimizations mentioned in [3] already enabled. The leftmost column shows the raw runtime, in milliseconds, when no tracematch is present. The other columns show the runtime in percent over this baseline. We marked all cells

	no tracematch	ASyncIteration	FailSafeEnum	FailSafeIter	HashMap	HasNextElem	HasNext	LeakingSync	Reader	Writer
antlr	4098	101.42	102.20	100.93	100.44	106.54	99.85	**125.28**	**1066.98**	**208.76**
bloat	9348	**199.17**	100.75	**>8h**	**239.08**	100.58	**3972.66**	**597.35**	97.05	**192.52**
chart	13646	100.39	100.01	**120.73**	100.15	100.13	100.99	**445.30**	100.32	100.29
eclipse	50003	102.36	101.10	103.44	102.36	100.53	104.81	102.61	100.28	98.79
fop	3102	90.04	91.33	101.06	105.35	95.87	90.07	**689.30**	100.71	106.74
hsqldb	12322	100.00	99.68	100.03	100.19	100.07	99.84	100.79	100.32	99.94
jython	11133	101.47	102.04	106.57	101.05	98.93	102.67	88.83	99.11	100.50
lucene	17068	101.29	**126.34**	109.57	103.40	**134.05**	102.22	**204.50**	101.78	101.12
pmd	12977	102.96	99.89	**257.61**	99.17	98.15	**258.23**	**131.26**	102.43	99.79
xalan	13083	101.86	100.20	100.71	101.35	99.59	102.34	104.20	101.70	100.47

Table II: Runtimes of the benchmarks before applying our optimizations

with overheads of more than 10% as boldface. The benchmark *bloat/FailSafeIter* was stopped after around 8 hours of benchmarking time. This benchmark is very hard to handle, dynamically as well as statically, because it makes extraordinarily heavy use of long-lived iterators and collections. We shall return to this benchmark later, when we discuss the performance of our analysis.

As we can see from the table, some benchmarks expose a significant overhead.[3] In these cases the whole program optimizations presented in this paper are worth applying. In particular, given the sometimes large runtime overhead, the programmer might well want to trade some of this overhead for compile time.

We applied our analysis to all 90 benchmarks, and in Table III we report on the improvements for the 18 interesting cases with an overhead of more than 10%. We captured the optimized program after each stage in order to see how many shadows were removed and are still remaining and in order to evaluate the runtime impact of the shadow removal for that stage. Table III shows on the right hand side the runtimes and number of remaining shadows for all the 18 cases in question.

The first column shows runtimes without any tracematches present. Those benchmarks obviously have 0 shadows. The second column shows the runtime of the fully instrumented benchmark. The shadow quantity given here is the number of shadows, which are reachable from the DaCapo main class. This information is available only if a call graph was computed. The number of all shadows is shown in brackets. The next three columns show the runtimes and shadow quantities after applying each of our three stages.

The table is split vertically into multiple parts. For the benchmarks in the first part (rows 1-7), the quick check was able to eliminate all shadows. For row 8, the flow-insensitive analysis removed all 294 reachable shadows. In the benchmarks in rows 9-13, the flow-insensitive analysis removed at least some shadows, most often all but a few. In the benchmarks in row 14-16, the flow-insensitive analysis was ineffective. In benchmarks 17 and 18, the analysis failed to complete in a reasonable time or aborted due to insufficient memory.

Looking at the runtimes of the optimized benchmarks, we observe that there is most often a very direct relation between the number of shadows removed and the speedup gained by the optimization. After applying all three optimization stages, all but the benchmarks in rows 13 and 16-18 execute almost as fast as the uninstrumented program.

---

[3] The speedups for fop and jython apparently originate from the fact that those benchmarks are *bistable*. Depending on scheduling order they settle down in one of two different highly predictable states. Additional instrumentation can sometimes affect this order and make the benchmark settle into a more favourable state, i.e. make the benchmark execute faster. This interpretation was suggested by Robin Garner, one of the developers of the DaCapo benchmark suite.

#	benchmark	analysis	no tm	reachable (all)	quick	flowins	flowsens	unit
1	antlr/	12	4098	5134	4104	4095	4057	ms
	LeakingSync		0	n/a (170)	0	0	0	shadows
2	antlr/	11	4098	8555	4239	4262	4211	ms
	Writer		0	n/a (56)	0	0	0	shadows
3	bloat/	19	9348	18618	11072	11094	11086	ms
	ASyncIteration		0	n/a (419)	0	0	0	shadows
4	bloat/	39	9348	55840	10908	10847	10917	ms
	LeakingSync		0	n/a (2145)	0	0	0	shadows
5	chart/	28	13646	60766	13894	13896	13865	ms
	LeakingSync		0	n/a (920)	0	0	0	shadows
6	fop/	38	3102	21382	2818	2884	2799	ms
	LeakingSync		0	n/a (2347)	0	0	0	shadows
7	pmd/	22	12977	17034	12882	12892	13118	ms
	LeakingSync		0	n/a (986)	0	0	0	shadows
8	lucene/	137526	32564	170155	178676	31248	30957	ms
	LeakingSync		0	294 (653)	653	0	0	shadows
9	antlr/	123229	4098	43725	20856	4942	4934	ms
	Reader		0	46 (53)	53	15	15	shadows
10	bloat/	422056	9348	22349	21885	9588	9685	ms
	HashMap		0	16 (57)	57	2	2	shadows
11	bloat/	454699	9348	17997	35525	9732	9684	ms
	Writer		0	87 (206)	206	8	8	shadows
12	lucene/	116463	32564	42452	41629	31977	31633	ms
	FailSafeEnum		0	41 (61)	61	5	5	shadows
13	pmd/	1247864	12977	33430	33905	23120	23234	ms
	FailSafeIter		0	129 (529)	529	90	90	shadows
14	chart/	472313	13646	16475	16446	16697	16452	ms
	FailSafeIter		0	105 (469)	469	105	105	shadows
15	lucene/	112109	32564	38155	36855	36607	36445	ms
	HasNextElem		0	14 (22)	22	14	14	shadows
16	pmd/	260551	12977	33510	34743	34918	33498	ms
	HasNext		0	87 (346)	346	86	86	shadows
17	bloat/	aborted	9348	28800000	28800000	n/a	n/a	ms
	FailSafeIter		0	1015 (1500)	1500	1015	1015	shadows
18	bloat/	aborted	9348	371364	373469	384533	n/a	ms
	HasNext		0	639 (947)	947	639	639	shadows

Table III: Runtimes of the analysis (left) and benchmarks (right, both in ms) and shadow quantities after each stage. Grouped by effectiveness of analysis.

With respect to the effectiveness of the different analysis stages, we see that the quick-check is very effective, removing all shadows in seven benchmarks. The flow-insensitive stage is generally very effective too, reducing the instrumentation and runtime overhead in another seven cases. We wish to point out that even in the case of *bloat/HashMap*, where primitive values are bound, the flow-insensitive analysis can still rule out many shadows by relating those remaining variables which bind objects. In one case (number 8), it is even able to prove the program correct, i.e. that all synchronized collections are only accessed via their synchronized wrapper.

The reader should note that optimizations as we propose here would be hopeless to perform on a a handcoded monitor in plain AspectJ. Consequently at least in cases 1-8 where we remove all shadows, the optimized benchmark runs faster than it could ever be achieved using not tracematches but AspectJ only.

Looking at the flow-sensitive stage, we were very disappointed to see that it did not manage to remove more instrumentation over the flow-insensitive stage. While in some microbenchmarks which we used for testing, it yielded significant improvements, in the DaCapo benchmark suite it was not even able to remove a single additional shadow. We were able to identify three different factors that lead to this behaviour. We hope that these observations will lead to better analyses which can find further improvements.

Firstly, if a lot of shadows remain after the flow-insensitive analysis, this often indicates that for some reason there is a large overlap between points-to sets. When this is the case, it is however equally hard for the flow-sensitive analysis to tell different objects apart and hence to relate events on those objects temporally. As noted in Section 3.3, in such situations we often only perform a lightweight fixed point computation which treats skip shadows conservatively. In cases like *pmd/FailSafeIter* unfortunately, this seems to give away a lot of crucial precision.

Secondly, as we explained in Section 3.3.2, our whole-program state machine is context-insensitive, meaning that we over-approximate the set of actually realizable paths by not explicitly outgoing with returning call edges. This seems to lose precision in those cases where overlapping points-to sets are actually not the problem.

Thirdly, we handle multi-threading in a very conservative way. In benchmarks like lucene, the program does not trigger the tracematch only because it uses explicit wait/notify. Without analyzing such lock patterns explicitly, there is little hope for any more precision in those cases.

Case 14, *chart/FailSafeIter*, could also not be improved upon because of multi-threading. In addition, points-to sets largely overlapped due to the use of reflection which caused a safe over-approximation of points-to sets.

In the cases of *bloat/FailSafeIter* and *bloat/HasNext*, the analysis ran out of memory. The problem with bloat is that it uses an extraordinarily many iterator accesses and modifications of collections. In addition, almost all iterators and collections are very long-lived, so that points-to sets vastly overlap. The construction of the whole-program state machine suffers even more from the fact that bloat defines its own collections which delegate to collections of the Java runtime (JRE). Usually, collection classes are defined inside the JRE and thus not weavable and produce no shadows. Hence, due to the call graph abstraction, calls to hasNext() or updates to collections produce no edges in the whole-program state machine. In bloat, all those optimizations fail, making the problem of efficient model construction very hard to solve.

4.1 Execution time of the analysis

The analysis was run on the same machine configuration as the benchmarks but with 3GB of RAM. Total runtimes of the analysis are shown on the left hand side of Table III. The longest successful analysis run we had was pmd/FailSafeIter with a total analysis time of around 21 minutes. The different stages of this run are distributed as follows.

18ms - quick check

146,602ms - call graph construction, points-to set creation

127ms - call graph abstraction

2700ms - flow-insensitive analysis

20,176ms - creation of per-method state machines

108,406ms - creation of whole-program state machine

950,899ms - flow-sensitive fixed-point iteration

As we can see, the most expensive phase is the flow-sensitive fixed point iteration, followed by the time spent in the construction of the call graph and points-to sets. The quick-check is so fast that it is always worthwhile. The flow-insensitive analysis, in combination with its preparation phase, still runs in reasonable time. As Table III shows, usually the runtime is between 3 and 10 times lower.

It proved very sensible to make use of the demand-driven refinement-based points-to analysis. For example, in pmd/FailSafeIter, we queried points-to sets for 691 variables only, where a full context-sensitive points-to analysis would have had to compute context-sensitive points-to sets for all 33993 locals in the pmd benchmark.

The flow-sensitive analysis generally adds a large computational burden and our results show that it does not find any improvements over the cheaper flow-insensitive stage. We plan to further refine that phase in future work to see if it's really worthwhile pursuing.

5 Related work

While a lot of related work has been done in static program analysis and verification (model checking), there has been little previous work on using those techniques to speed up runtime monitoring. We list notable exceptions here.

Typestate properties Typestate properties [11] have been widely studied in the past. Very recently, Fink et al. presented a static optimization for runtime checking of typestate properties [5]. The analysis they present is in flavour similar to ours. In particular it is also implemented in multiple stages, one of which is flow-sensitive. However typestate properties allow one to express temporal constraints about *one single* object only, making their flow-insensitive stage simpler than ours. The authors paid special attention to the handling of strong updates, using must-alias information, which we leave to future work. The analysis Fink et al. present did not address the issue of multi-threading.

PQL. In [8], Martin et al. present their Program Query Language, PQL. They experimented with a flow-insensitive analysis similar to our consistent-shadows analysis. However, their analysis is still not integrated within the PQL tool, making effective comparisons impossible at the current time. We suspect, that their flow-insensitive analysis performs very similarly to ours since they made similar design decisions. In particular they also do not take must-alias information into account. However, our analysis should in the general case be much faster, because unlike the analysis for PQL ours is staged, employing a very effective quick check first. Also we compute context for points-to sets for certain variables only while they apply this very expensive computation for all program variables. While their analysis consists of a single block, ours is an open framework which can easily be enhanced with new stages as research progresses.

Bandera. Many model checking approaches use techniques to generate a slick abstract model of an actual executable program. One example is Bandera, a tool set for model checking concurrent Java software. Part of this tool set is a Slicer [6] which computes for the Jimple representation of the program under test a slice under a given specification, i.e. that part of the program which could possibly affect values which are referred to by the specification. While such techniques pursue the same goal of keeping abstractions small, they reside on a lower level of abstraction. Slicing is performed directly on the program while our abstraction is a much smaller and more lightweight set of shadows.

6 Discussion and future work

In this work we have proposed a staged static analysis for reducing the overhead of finite-state monitors. We have presented three stages including a very coarse-grain and inexpensive quick check based only on shadows matching symbol names, a flow-insensitive consistent-shadows analysis that finds all shadows with consistent points-to sets, and a flow-sensitive active-shadows analysis that also takes into consideration the order in which shadows execute.

As is often the case in program analysis, we were somewhat surprised that the first two simpler stages were the most effective. The quick check analysis is very simple and also quite effective in eliminating tracematches that can never match a base program. We believe that this test will be very useful in situations where whole libraries of tracematches are routinely applied to software as it is developed. For example, libraries can be associated with a collection of tracematches specifying constraints on how the library should be used. In these cases we expect that only some tracematches will actually apply to the program under analysis, and the quick check is a sound and simple way to eliminate those that don't apply. We expect this check to become a standard part of the `abc` compiler and it will be enabled by default at the -O level of optimization.

The second stage, flow-insensitive analysis to find consistent shadows, was also effective in some cases, and is also not a very complex analysis once one has a good points-to analysis available. We did find that a context-sensitive points-to analysis was necessary and this turned out to be an ideal use case for demand-driven context-sensitive analysis since we were only interested in the points-to information of variables involved in shadows. Based on our results, we think that this consistent-shadows analysis should be available at a higher-level optimization level (-O3), to be used when run-time overheads are high. In many cases we expect the overheads of a program optimized that way to be lower than those of programs using a handcoded AspectJ monitor which is not analyzable.

Although we expected that the third stage, the flow-sensitive active-shadows analysis, would give us even more improvement, we found that it did not. To implement and test this analysis we developed a lot of machinery to represent the inter-procedural abstraction of the matching automata, and techniques to soundly approximate even in the presence of threads. To our surprise, the end result is that this extra machinery did not lead to more precise shadow removal. However, this exercise did provide an analysis basis and some new insight into the problem and we think that further refinements to this approach are worth further investigations. We plan to work on this by experimenting with new kinds of must and hybrid points-to abstractions and by improving upon the treatment of multi-threading, perhaps by using the *May Happen in Parallel* (MHP) analysis which is currently being integrated into Soot [7].

Acknowledgements We are very grateful to Manu Sridharan for providing his points-to analysis to us and helping us integrate it into Soot. Steve Blackburn and Robin Garner helped with setting up the DaCapo benchmark suite and interpreting our benchmark results. Stephen Fink helped to clarify our interpretation of his related work. Nomair Naeem contributed with clarifying discussions and by commenting on a draft of this article. We also wish to thank the whole development group of the `abc` compiler, and in particular the tracematch guys, Pavel Avgustinov and Julian Tibble, for their enduring support and motivation.

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Object-Oriented Programming, Systems, Languages and Applications*, pages 345–364. ACM Press, 2005.

[2] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In *Programming Language Design and Implementation (PLDI)*, pages 117–128. ACM Press, 2005.

[3] Pavel Avgustinov, Julian Tibble, Eric Bodden, Ondřej Lhoták, Laurie Hendren, Oege de Moor, Neil Ongkingco, and Ganesh Sittampalam. Efficient trace monitoring. Technical Report abc-2006-1, http://www.aspectbench.org/, 03 2006.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *P LA Proceedings of the st annual ACM I PLAN conference on Object-Oriented Programing Systems Languages and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[5] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *I TA Proceedings of the international symposium on Software testing and analysis*, pages 133–144, New York, NY, USA, 2006. ACM Press.

[6] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.

[7] Lin Li and Clark Verbrugge. A Practical MHP Information Analysis for Concurrent Java Programs. In Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff, editors, *LCPC*, volume 3602 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2004.

[8] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors using PQL: a program query language. In *Proceedings of the th Annual ACM I PLAN Conference on Object-Oriented Programming Systems Languages and Applications*, pages 365–383, 2005.

[9] Manu Sridharan and Rastislav Bodik. Refinement-based context-sensitive points-to analysis for java. In *PL I Proceedings of the ACM I PLAN conference on Programming language design and implementation*, pages 387–400, New York, NY, USA, 2006. ACM Press.

[10] Volker Stolz and Eric Bodden. Temporal Assertions using AspectJ. *Electronic Notes in Theoretical Computer Science*, 144(4):109–124, 2006.

[11] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *I Transactions on Software Engineering*, 12(1):157–171, 1986.

[12] Robert Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *ACM Sigsoft International Symposium on Foundations of Software Engineering (-)*, pages 159–169, 2004.