The abc Group

Flow-sensitive static optimizations for runtime monitors

abc Technical Report No. abc-2007-3

Eric Bodden, Patrick Lam, Laurie Hendren

Sable Research Group
School of Computer Science
McGill University

July 18, 2007

Contents

List of Figures

List of Tables

Abstract

Runtime monitoring enables developers to specify code that executes whenever certain sequences of events occur during program execution. Tracematches, a Java language extension, permit developers to specify and execute runtime monitors. Tracematches consist of regular expressions over events, where each event may specify free variables that are bound to run-time objects. Naïve implementations of runtime monitoring are expensive and can cause prohibitive slowdowns. In previous work, we proposed optimizations based on flow-insensitive pointer analyses. While these optimizations worked well in most cases, more difficult cases with large overheads remained.

In this paper, we propose three novel intraprocedural optimizations with the goal of eliminating the overhead from runtime monitors. Our optimizations rely on flow-sensitivity and precise local may-alias and must-alias information. The first two optimizations identify and remove unnecessary instrumentation, while the third one hoists instrumentation out of loop bodies.

We applied our transformations to seven difficult combinations of tracematches with programs from the DaCapo benchmark suite which defeated our earlier analyses. Our results show that our three optimizations, in combination, can remove much of the instrumentation in this benchmark set. For two of the seven cases, we can remove all instrumentation: our analysis successfully shows that the benchmark programs will always satisfy the verification properties stated in the tracematches. Our results furthermore suggest that our analysis can detect hidden method preconditions which ought to be documented and visible to the developers.

After our optimizations, only three cases (out of an original 90 cases) still have noticeable runtime overheads. One of these cases cannot possibly be optimized, because the runtime monitors actually trigger. While our optimizations ought to be able to handle the remaining two cases, only an imprecision in our underlying global points-to analysis currently prevents us from removing the overhead in those cases as well.

1 Introduction

A software system's sequence of actions over an execution is a rich source of information about the system's behaviour on that execution and often gives insight into the system's behaviour on other executions. Certain sequences of runtime events indicate defects in the system. Runtime monitoring can detect such sequences of events, enabling developers to handle the sequences with code that reports errors or enables the system to recover from faults.

Tracematches [1] are a Java language extension which enable programmers to specify traces via regular expressions of symbols with free variables, along with some code to execute if the trace occurs in an execution. A symbol's free variables bind heap objects at runtime. A tracematch executes its associated code if a suffix of the symbols in the current execution trace contains 1) the right symbols with 2) a consistent variable binding (*i.e.* symbols' free variables match up) in 3) an order which matches the regular expression. At the implementation level, the compiler and runtime system implement tracematches using runtime monitors based on finite-state machines. Compiler-generated instrumentation code updates the monitor's internal state each time an event of the execution trace matches a declared symbol from the tracematch. When the monitor finds a consistent match in the program's execution trace, it triggers the code associated with the tracematch.

Unfortunately, naïve implementations of runtime monitoring can be impractical due to the run-time expense: as expected, instrumented code runs more slowly than uninstrumented code. There are therefore two basic approaches for reducing the overhead due to runtime monitoring: 1) run each instrumentation point faster (corresponding to dynamic improvements); or 2) reduce the number of instrumentation points (static improvements).

Avgustinov et al. have developed optimized runtime monitor implementations to make runtime monitoring usable, at least at development time [3,4]. For instance, they use a special encoding for variable bindings at runtime. However, even after such optimizations, 5x slowdowns over the uninstrumented code were not uncommon, and some cases were even more expensive.

In [7], we explored the second alternative by proposing some static optimizations for tracematches. These optimizations successfully eliminated overheads for all but 9 out of our 90 benchmark/tracematch combinations. The key idea was to identify instrumentation points which could not trigger a complete match because (1) the program did not contain enough symbols to give a complete match, (2) the variable bindings among the symbols that the program did contain were inconsistent, or (3) the symbol never executed in an order which would be matched by the regular expression. We used a flow-insensitive pointer-based analysis to remove tracematches which were unnecessary because they satisfied properties (1) and (2). This analysis proved very effective, reducing overheads to below 10% in most of our benchmarks. However, a significant number of pathological cases with much larger overheads—from 18% to 260%—still remained. In the same work, we also proposed a flow-sensitive whole-program analysis which attempted to address property (3), but that analysis did not manage to identify any additional unnecessary instrumentation points.

We therefore set out to optimize the important cases that were not susceptible to improvement by either more efficient monitor implementations or previously developed static analyses. Our approach was to identify the weaknesses of the previous static analysis and to design new analyses targetted towards solving the remaining—hard—problems. We found that, in many cases, much of the overhead came from a few hot instrumentation points, which we studied in detail. We observed that an intraprocedural analysis ought be able to conclude that the hot shadows from our benchmarks would never trigger the tracematches, if the analysis was flow-sensitive and used both may-alias and must-alias information. In other words, exploiting property (3) would indeed allow us to eliminate the hot shadows, given sufficiently strong alias information. We therefore set out to develop a precise and accurate intraprocedural analysis that would enable us to reduce the overhead of runtime monitoring.

We found that accurately estimating the possible tracematch configurations at each instrumentation point enabled a number of optimizations based on property (3). We therefore describe a static analysis which abstractly models the possible runtime configurations of tracematches. Based on our abstraction,

- we can remove an instrumentation point if it will never modify the tracematch configuration at runtime;
- we can remove an instrumentation point if it will never be on a path that reaches a final configuration; and
- we can either move instrumentation points within a loop body outside the loop body or execute the instrumentation points only once per loop.

We applied our optimizations to the those cases from [7] with remaining overhead. Our results show that using all three optimizations in conjunction with the flow-insensitive optimization from [7] reduces the runtime overhead in all but three cases to below 10%. In two cases, we were able to remove all instrumentation. Because our tracematches detect error conditions, our analysis for those benchmarks therefore guarantees that those error conditions can never occur.

Of the three cases with remaining overheads, one cannot be statically optimized because its monitor triggers at runtime. While our optimizations ought to be able to handle the remaining two cases, only an imprecision in our underlying global points-to analysis currently prevents us from removing the overhead in those cases as well.

Contributions. This paper makes the following contributions:

- a novel intraprocedural flow-sensitive static analysis for statically estimating possible states of a tracematch automaton, based on may-alias and must-alias information;
- three optimizations for eliminating overhead due to tracematches, all of which are based on our static analysis; and
- an experimental evaluation of our optimizations on a suite of sizeable benchmark applications.

We believe that our results generalize beyond the immediate context of optimizing tracematches. Section 6 discusses the applicability of our analysis to other runtime monitoring frameworks such as PQL [11]. Furthermore, our results suggest that our analysis can help detect hidden—currently undocumented—method preconditions which ought to be visible to the developer.

The remainder of the paper is organized as follows. Section 2 introduces the syntax and runtime behaviour of tracematches. It also points out some situations where unnecessary updates to the tracematch monitor occur. Section 3 introduces an abstraction that mimics the dynamic tracematch evaluation statically. Section 4 describes the tracematch optimizations, while Section 5 evaluates the results of our optimizations. Finally, Section 6 discusses related work and Section 7 concludes.

2 Tracematches: Definition and examples

In this section, we describe tracematches, the mechanism for runtime monitoring that we seek to optimize, and explain some of the key concepts behind how a compiler for tracematches creates code that implements tracematches at runtime. We also include two examples which explain some of the reasoning behind our static analysis and optimizations.

In this work, we focus on verification tracematches. Our tracematches typically encode API usage rules; in our examples, the tracematch bodies report errors, but they could equally well contain error-recovery code which would enable the program to continue running.

2.1 `HasNext` example tracematch

Figure 1 presents the `HasNext` verification tracematch. This tracematch captures the fact that, given an `Iterator` object i, it is unsafe to call `i.next()` twice in a row without a call to `i.hasNext()` in between. Each tracematch may declare formal variables that bind to objects at runtime. Here, line 1 declares the formal variable `i` of type `Iterator`. Tracematches also declare a set of symbols establishing the alphabet for the tracematch's regular expression. These symbols define events on the runtime execution trace using AspectJ pointcuts. In the example, lines 2–5 declare symbols `hasNext` and `next`. These symbols capture method calls to the `hasNext()` and `next()` methods of our iterator i. Finally, a tracematch declares a regular expression over this alphabet and some code to execute when the regular expression matches a suffix of the execution trace with a consistent variable binding. Here, line 7 declares the tracematch's regular expression, `next next`, and states the code to execute if `next next` occurs in some execution with both `next` symbols binding the same iterator i.

Note that the sequence `next hasNext next` is not matched by our tracematch: no suffix of this sequence is matched by `next next`.

```
1 tracematch(Iterator i) {
2   sym hasNext
3     before: call(* java.util.Iterator+.hasNext()) && target(i);
4   sym next
5     before: call(* java.util.Iterator+.next()) && target(i);
6
7   next next { System.err.println("Trouble with "+i); }
8 }
```

Figure 1: Tracematch definition for the `HasNext` tracematch.

Tracematch implementation. The AspectBench compiler [2] (`abc`) implements tracematches by compiling Java source or bytecode, together with any desired tracematches, into instrumented Java programs augmented with runtime monitoring. The `abc` compiler first creates a *tracematch automaton* from the tracematch's regular expression. It then identifies a set of instrumentation points, or *shadows* [12], corresponding to the points in the code where symbols will potentially execute (and thereby update the tracematch state). Note that these shadows bind a subset of the tracematch's variables, as specified by the symbol's definition.

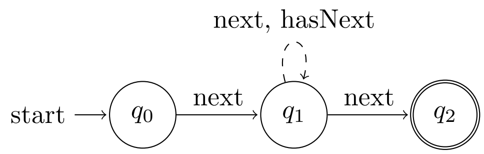


Figure 2: Automaton for the `HasNext` tracematch from Figure 1.

While the tracematch automaton resembles the standard finite automaton induced by the tracematch's regular expression, the tracematch machinery uses the automaton in an unusual way. Normally, an automaton is in one state at a time. But recall that a tracematch binds a set of heap objects to its variables. The tracematch automaton must therefore track possible states for each set of bindings of the tracematch variables. If a set of bindings reaches the final state of the automaton, then the runtime system executes the body of the tracematch.

Figure 2 presents the tracematch automaton for the `HasNext` tracematch. Solid lines represent state transitions, while dashed lines represent special skip loops.

State transitions are fairly standard: Whenever a shadow with label ℓ executes, the tracematch runtime processes all transitions $s \xrightarrow{\ell} t$, for each possible pair of states s and t. If state s holds a variable binding that is consistent with the binding induced by the shadow, the runtime propagates this binding to state t. This propagation ensures that the tracematch automaton reaches the final state whenever the regular expression matches with a consistent set of bindings.

On the other hand, the tracematch runtime machinery must also discard candidate matches as they become invalidated. For instance, in the `HasNext` tracematch, the runtime must discard any candidate match binding an iterator `i` in the event of a call to `i.hasNext()`, because the tracematch should only trigger if two adjacent calls to `i.next()` occur with *no* call to `i.hasNext()` in between. Hence, at any call to `i.hasNext()`, for `i`, the match has to start all over again. Skip loops instruct the runtime to discard invalidated candidate matches.

2.2 Dynamic tracematch configurations

A configuration for a tracematch automaton $\mathcal{A}$ is a function mapping states of $\mathcal{A}$ to constraints. Figure 3 presents the grammar for these constraints. Constraints are stored in disjunctive normal form. A constraint can be a disjunction of disjuncts D, or one of the boolean literals `true` and `false`. Each disjunct is a conjunction of bindings B, each of which is either a positive binding $v = o$ or a negative binding $v \neq o$. The left hand side v is a tracematch variable, as declared in the tracematch. At runtime, the right hand side o is a heap object. The runtime also maintains the invariant that each disjunct only contains one positive binding for each tracematch variable. One way to think of a disjunct is as a partial function from tracematch variables to heap objects; when the function is partial, the negative bindings give additional information about the objects that may be bound to a tracematch variable.

$$
\begin{array}{rcl}
C & ::= & \mathsf{true} \mid \mathsf{false} \mid \bigvee D \\
D & ::= & \bigwedge B \\
B & ::= & v = o \mid v \neq o
\end{array}
$$

Figure 3: Grammar for configuration formulas.

Example. We next present an example of dynamic tracematch configurations. The `abc` compiler generates instrumentation that manipulates such tracematch configurations at runtime. Dynamic configurations are especially useful to understand because our static abstraction of tracematch configurations mimics the dynamic configuration information.

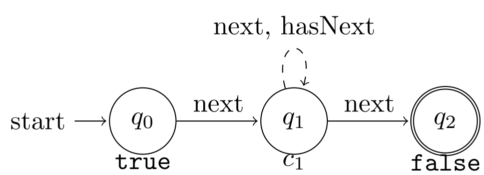
We will present the evolution of dynamic configurations for the `HasNext` tracematch from Figure 1. Recall that this tracematch detects the case where the `next` method of some `Iterator` object is called twice without an intervening call to `hasNext`. Consider the tracematch's behaviour on the following method:

```
1 void m(Iterator it) {
2   it.hasNext();
3   it.next();
4   it.hasNext();
5   it.hasNext();
6 }
```

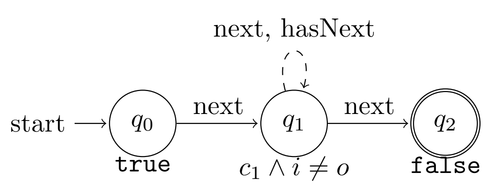
This method clearly does not trigger the `HasNext` tracematch. Furthermore, it is possible to deduce the state of the object pointed to by `it` after each statement in the method.

To explain dynamic configurations, we describe the dynamic configuration at each program point of method `m`. Note that, throughout the execution of method m, there is only one object bound to variable `it`. We denote this object by o.

Initial configuration when entering m. Our automaton diagrams show constraints for automaton states below the states themselves. Because the tracematch semantics state that a tracematch triggers whenever its regular expression matches a suffix of the current execution trace, any variable binding can start a new candidate match at any time. We represent this fact with the constraint `true` at the initial state. No objects are bound in the final state: hitting the final state triggers the tracematch body, so objects are immediately consumed as soon as they reach the final state. At runtime, when entering m, the constraint at q_1 is known; we symbolically represent this known constraint by c_1.

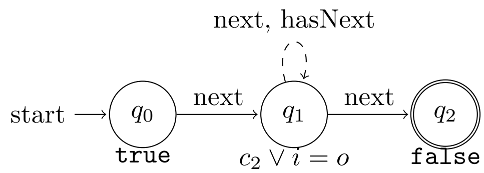


After line 2 (`hasNext` shadow). We learn that o is not in state q_1, due to the skip loop on q_1, so we conjoin the negative binding $i \neq o$ at q_1:



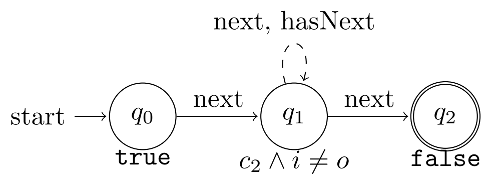
This conjunction models the fact that o can certainly not be in state q_1. The runtime engine optimizes the constraints: if c_1 contained a disjunct d with a positive binding $x = o$, and we conjoin c_1 with the negative binding $x \neq o$, then we get `false`, which means that disjunct d can simply be dropped from c_1.

After line 3 (`next` shadow). We label the resulting constraint at q_1 with c_2. Now o is in the intermediate state q_1, giving the binding $i = o$ at q_1:



Here, object o is definitely bound to i at state q_1.

After line 4 (`hasNext` shadow). After line 4 we compute the constraint $c_2 \wedge i \neq o$, which is equal to $c_1 \wedge i \neq o$. Hence, we effectively return to the same configuration as after line 2.



After line 5 (`hasNext` shadow). The shadow at line 5 does not have any effect on the configuration, as o is already known not to be in q_1.

Discussion. Our example has presented the evolution of runtime configurations through a simple method. We have seen how the `abc` runtime maintains a constraint for each tracematch configuration; this constraint tracks states of various runtime objects.

We can deduce several properties of `m` and its interaction with the `HasNext` tracematch. First of all, `m` never triggers the final state of this tracematch on any object: the only object `m` can affect is o, and we know what `m` does to o. Secondly, observe that, despite knowing nothing about o at method entry, we can deduce precise information about the state of o at each program point: after executing the `hasNext` shadow, o can only be in state q_0. Finally, note that the shadow at line 5 is unnecessary because it does not change the automaton configuration. This type of observation has inspired the optimizations that we present in this paper.

2.3 `FailSafeIter` example tracematch

We present an additional example illustrating the case where a tracematch binds two variables. Figure 4 presents the `FailSafeIter` tracematch, which reports cases where the program modifies a `Collection` while an `Iterator` is active on that `Collection`. Figure 5 shows the corresponding automaton.

Consider the `findVariableHere` method from one of our benchmarks, `pmd`, a static analysis tool which detects potentially problematic patterns in Java source code.

```
public int findVariableHere(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); ) {
        Object o = i.next();
        if (o == null)
            return 0;
    }
    return 1;
}
```

This method simply creates an iterator i and iterates over it. We can observe that i never hits the final state, because it does not escape the `findVariableHere` method, so that all shadows on `i` can safely be disabled.

Our static analysis computes possible configurations for the tracematch automaton after every program point. We observe that `findVariableHere` certainly never updates the collection bound to c. Furthermore,

```
public aspect TMFailSafeIter {
  pointcut collection_update(Collection c):
      (   call(* java. util . Collection+.add*(..))  ||   ...   ||
          call(* java. util . Collection+.remove*(..))  )  && target(c);

  tracematch(Collection c, Iterator i) {
    sym create_iter after returning(i):
              call(* java. util . Collection+. iterator ())  && target(c);
    sym call_next before:
              call(* java. util . Iterator+.next()) && target(i);
    sym update_source after: collection_update(c);

     create_iter   call_next * update_source+ call_next {  ...  }
  }
}
```

Figure 4: `FailSafeIter` tracematch: detect updates to a `Collection` which is being iterated over.

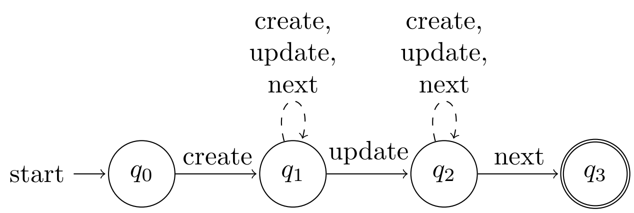


Figure 5: Automaton for `FailSafeIter` from Figure 4.

the iterator bound to i is only live within `findVariableHere`. We can therefore conclude that the combination of i and c can never reach a final state in the automaton. Our static analysis will be able to remove all shadows in this method.

Flow-sensitivity is crucial here: The collection is certainly updated somewhere in the program. We can only safely remove the shadows in `findVariableHere` because the collection is not being updated *while the iterator is in use*. We designed our analysis to use a flow-sensitive abstraction so that it would be able to optimize situations like this one.

Note that tracematches bind multiple variables simultaneously, which enables them to express relationships between multiple program objects. This feature complicates our analysis—we are forced to track sets of bindings to objects, rather than tracking states of objects, one object at a time—but increases the expressive power of the specification language: in particular, tracematches can express more sophisticated properties than approaches based on typestate [8]. Section 6.2 describes the relationship between tracematches and typestate verification in more detail.

3 Analysis abstraction

Our static abstraction of tracematch configurations enables us to 1) reason about the state of tracematch automata throughout a program and 2) perform optimizations based on information that we collect about possible tracematch configurations. This section presents our analysis abstraction and the update rules for our abstraction.

Our static abstraction closely models the runtime tracematch configuration information, but substitutes local variable names for runtime objects. We next present an example of our static analysis.

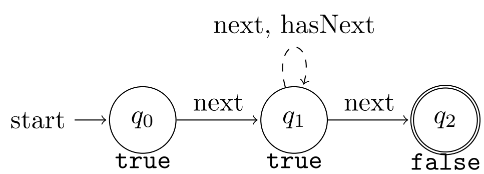
3.1 Example of static analysis

We return to our `HasNext` example from Section 2.2 and explain the result of our static analysis on that example. Note that the analysis actually operates on the instrumented code, with explicit shadows. A simplified version of this instrumented code follows[1]:

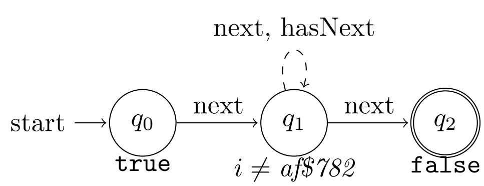
```
1  void m(Iterator it) {
2    uniqueArgLocal7 = it;
3    adviceformal$782 = uniqueArgLocal7;
4    theAspect$TMReader.beforeAfter$14(adviceformal$782);
5    it.hasNext();
6    adviceformal$783 = uniqueArgLocal7;
7    theAspect$TMReader.beforeAfter$15(adviceformal$783);
8    it.next();
9    adviceformal$784 = uniqueArgLocal7;
10   theAspect$TMReader.beforeAfter$14(adviceformal$784);
11   it.hasNext();
12   adviceformal$785 = uniqueArgLocal7;
13   theAspect$TMReader.beforeAfter$14(adviceformal$785);
14   it.hasNext();
15 }
```

This intermediate code clearly shows the need for pointer information. Without pointer information, it would be impossible to keep track of which shadows apply to which heap objects. While simple transformations would be sufficient to push `it` throughout the method in this case, they are not enough in general. Our pointer analyses will determine that all of the local variables in this method must-alias each other.

Initial approximation. We will present the result of the static analysis when we initially approximate the value of q_1 with `true`.

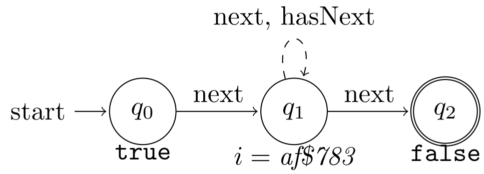


After line 4 (`hasNext` shadow). Statically, we know that variable `adviceformal$782` is not in state q_1, due to the skip loop on q_1, so we create the negative binding $i \neq$ `adviceformal$782` at q_1:

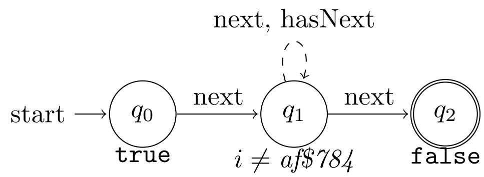


After line 8 (`next` shadow). We must add the binding $i = af\$783$ at q_1, since we know that $af\$783$ is now in state q_1. Because $af\$782$ must-aliases $af\$783$, we can drop the $i \neq af\$782$ binding due to the `next` skip loop.

[1] In this text, we refer to the variables used in advice applications as *advice actuals*, but the `abc` compiler calls these variables *advice formals*.



After line 12 (`hasNext` shadow). We generate the negative binding $i \neq af\$784$. Furthermore, because $af\$784$ must-aliases $af\$783$, we drop the positive binding $i = af\$783$.



After line 15 (`hasNext` shadow). We now generate the negative binding $i \neq af\$785$. Because $af\$785$ must-aliases $af\$784$, we consider both values equal and store only one of them. Hence this configuration is equal to the previous configuration; we discuss equality of configurations in more detail below. Our unnecessary shadow elimination optimization (Section 4.2.1) would eliminate this shadow.

3.2 How our static analysis works

We continue by describing our static analysis in detail. To compute our static abstraction of the tracematch state for a method m, we perform a fixed-point iteration on m, starting with the initial approximation at the start of the method.

Soundness properties. We have designed our approximation to be sound in the following sense:

1. if a shadow can trigger a final state at some program point, then our approximation at that point must also flag the fact that the tracematch may hit the final state;
2. if two tracematch configurations may be different, then our approximation identifies that these configurations are different.

Property 1 supports transformations which estimate when tracematches cannot possibly reach their final state, enabling such transformations to eliminate shadows that cannot contribute to a match. Property 2 supports transformations which recognize and eliminate shadows that do not have any effect at runtime.

Contents of bindings. Formally, our grammar for static configurations replaces runtime objects from Figure 3 with local variables; the definition for bindings therefore becomes

$$B ::= x = v \mid x \neq v.$$

Instead of stating that tracematch variable x is bound to runtime object o ($x = o$), we state that tracematch variable x is bound to the contents of local variable v ($x = v$), and we keep in mind that v could point to a number of different objects.

3.3 Initial approximation

At runtime, the tracematch automaton may be in an arbitrary configuration upon method entry. We model this arbitrary configuration by running the static analysis with a set of configurations: one configuration has `true` at the initial state only and `false` at other states; other configurations have `true` at the initial state and at each of the non-final states in turn, and `false` elsewhere. This initial approximation enables us to detect all cases where a binding may potentially be propagated to a new state. (Note that starting with `true` at all non-final states would mask some updates.)

3.4 Update rules

We next describe how our static analysis updates the abstraction at shadows. Statically, at each shadow, our analysis receives two inputs: (1) the symbol name, and (2) a partial function from tracematch variables to local variables. Our analysis updates the abstraction by (1) taking automaton transitions, and generating negative bindings at skip loops, for every local variable that may alias the advice actuals (which represent the objects bound at that shadow) and (2) dropping disjuncts when the disjuncts contain local variables that are must-aliased with the advice actuals.

Need for aliasing information. Recall that our dynamic configuration example used a single object o as the object whose tracematch state was being tracked. Unfortunately, as we have seen, the `abc` compiler creates a number of temporary local variables and uses these temporary variables as advice actuals. Only some of these variables point to o. Our static analysis must determine which local variables must point to o and which local variables may not point to o. Our use of aliasing information enables our analysis to properly handle cases where different objects are aliased and shadows occur on some of these aliases.

Our analysis uses must-alias and not-may-alias pointer information. We gather this information using naïve intraprocedural analyses that estimate whether local variable ℓ_1 at program point p_1 must-aliases, or may not alias, local variable ℓ_2 at program point p_2; the key idea is that if the value of ℓ_1 and ℓ_2 originate at the same expression and flow to ℓ_1 and ℓ_2 by a sequence of copy statements, then they are must-aliased, and if ℓ_1 and ℓ_2 contain heap objects known to be disjoint (for instance, they are allocated on the heap at different `new` expressions), then they may not alias. Our must-alias analysis is modelled on Extended SSA Numbering [10].

Weak updates. When the runtime encounters a shadow s on a non-skip tracematch automaton edge, it updates the state of the heap object o bound to a tracematch variable; because the runtime knows the precise identity of o, it only needs to update the state of o. However, at static analysis time, our compiler only has an estimate of the set of variables which may point to o. Because our analysis attempts to find out all variable bindings which may trigger a final state, it must update the state of all local variables that may potentially point to o with the effect of s. (In fact, we omit updates at shadows only for those local variables which may-not alias local variables pointing to o).

Our analysis also handles skip loops by creating negative bindings $x \neq v$ if there are no positive disjuncts which must-alias v, reflecting the fact that we know that v is not in state s after shadow s executes.

We next describe a crucial optimization for the weak update rule. Consider the following code with the `FailSafeIter` tracematch from Figure 4. Method `m(..)` iterates over a collection `s`. Assume that this collection has been populated elsewhere.

```
1 m(Collection s) {
2   Iterator it = s.iterator();
3   while(it.hasNext()) {
4     it.next();
5   }
6 }
```

We next consider the analysis of `m(..)`, referring to the tracematch automaton in Figure 5. When processing the `next` shadow at line 4 with an initial assumption of `true` in state q_2, we infer that the `next` shadow can actually lead to a final state, a false positive.

Consider the set of states at which `it` can possibly be bound at the `next` shadow. The `create` shadow ensures that `s` and `it` are bound at state q_1, and also ensures that `s` and `it` are not in q_2. Therefore, q_2

could only contain a binding `i=it` if some collection besides `s'` $\neq$ `s` was associated with `it`. While we as programmers know that there can only always be one single collection per iterator, our analysis has no way of determining this. We have therefore implemented the following approximation.

The structure of the tracematch automaton ensures that at state q_2, tracematch variables `c` and `i` must be both bound. Therefore, if no shadows outside method m which share a shadow group with the `next` shadow bind `c` and `i` (possibly separately), then it is sound to omit the weak update on `i` at the `next` shadow: because `i` could not have reached q_2, it cannot advance to q_2's successor state.

Strong updates. At static analysis time, our compiler handles a skip loop with variable v bound by discarding positive bindings in the configuration which must-alias v.

Loops and redefinitions. When a local variable ℓ is redefined within a loop, it no longer must-aliases its value from previous iterations (old ℓ). However, our must-alias analysis cannot tell which local variable we are asking about: are we asking about the current value of ℓ or about old ℓ? Because our static analysis uses local variables in the analysis abstraction, it cannot use the results of the must-alias analysis directly. We therefore add an additional step to our static analysis rules for tracematches: if a local variable appears on the right-hand side of a binding and this local variable is redefined, we replace the variable and all of its must-aliases with a special `UNKNOWN` value at variable bindings. This `UNKNOWN` value never must-aliases any value. Note that this special rule gives us exact information within the first iteration of the loop, while distinguishing values of local variables between different iterations. Our loop optimizations use information about the first iteration to determine when it is safe to hoist shadows out of loops.

Formula transformations. We found that our analysis sometimes generated formulas which are equivalent to `true`; for instance, we found that our analysis generates

$$x = v \vee x \neq v$$

in one particular example. We apply an optimization to fix up formulas which are trivially seen to be `true`.

Method calls. Because our analysis is intraprocedural, we conservatively assume the worst of any calls to methods that contain shadows. After any such method call, we *taint* the configuration (effectively marking the configuration unknown) and propagate taintedness to all of the method call's successors. Our optimizations refrain from program transformations which would be based on tainted information.

Hit counters. The final state keeps no bindings, but we need to know when the final state may be triggered. Recall that when a set of bindings hits a final state, the runtime executes the tracematch body and immediately throws out the bindings, leaving the final state empty. Even if the tracematch configuration stays the same, we need to record the fact that something has changed in the program configuration (*i.e.* the tracematch body executed) to ensure Soundness Property 2.

To record this information, we keep a *hit counter* as part of our abstraction. The hit counter is an integer that we increment each time we potentially hit the final state. Clients of our static analysis can read the hit counter and know when the final state may potentially be hit.

Equality of configurations. Two configurations c_1 and c_2 are equal if: 1) they have equal hit counters and 2) the constraints for each state s are equal up to must-aliasing. Note that the effectiveness of our optimizations therefore depends on the accuracy of the must-aliasing information.

Termination. To conservatively approximate control flow, our static analysis performs a fixed point iteration over the control flow graph for each shadow-bearing method. Note that Soundness Property 2 is incompatible with reaching the fixed point in certain situations. Consider:

```
while (...) { v = c.iterator (); v.next(); }
```

Because v is redefined within the loop body, we must assume that it is bound to a different object within each iteration, preventing our analysis from reaching the fixed point. The iterations that did terminate did so in no more than 10 iterations. We therefore aborted our fixed point iteration after a maximum of 20 iterations, to be on the safe side.

Sources of inaccuracy. The sources of inaccuracy in our analysis are: 1) as with any static analysis, we must estimate the control flow—we guess that each branch of a conditional might be taken; 2) because our analysis is intraprocedural, we conservatively estimate the initial state at the beginning of each method; and 3) we identify variables with the set of heap objects that it points to. (Note that our Jimple intermediate representation splits local variables [16], so that disjoint lifetimes of local variables are split into different variables for our analysis. SSA form gives a similar splitting.) Our approximation was designed to soundly overestimate the set of objects that bind to tracematch variables in any given tracematch state.

Concurrency. As described, our static analysis does not handle concurrent programs; we verify that our benchmark applications are single-threaded before optimizing them. A number of straightforward extensions would enable our analysis to handle concurrency. First, we can conservatively assume that any shadow in another thread may occur at any point in our methods. We could then improve our results by considering only shadows that may actually occur in parallel with our shadows, for instance by determining thread-local objects as in [9].

4 Optimizations

The AspectBench Compiler [2] (`abc`) implements tracematches. It weaves together Java or AspectJ code with tracematches and emits instrumented Java bytecode. Figure 6 presents the entire weaving process, including our three optimizations. To weave a program with a tracematch, the compiler matches the symbol definitions of the input tracematch against the given program, giving a weaving plan. The weaving plan contains a complete description of the instrumentation needed to implement the runtime monitoring specified in the tracematches. Next, the compiler weaves together the program and the tracematch according to the weaving plan. All subsequent analyses are conducted on the woven program.

We optimize our input program as follows. First, we apply two optimizations from [7], the quick check and the flow-insensitive optimization. We then apply the three optimizations proposed in this paper: unnecessary shadow elimination, cannot-reach-final elimination and shadow motion. Because each optimization may, in principle, enable other optimizations, we iterate the optimizations, as illustrated by the back edge in the figure.

Whenever any of the optimizations proves that a shadow can be removed, it updates the weaving plan accordingly. The iteration terminates when no optimization removes any shadows. We then re-weave the program according to the updated weaving plan.

4.1 Previous analyses

In previous work [7], we described some techniques for statically optimizing tracematches. Because this paper builds on some of our previous work, we briefly summarize some key points. The main idea in all of our work is to use static analyses to move or disable shadows which cannot possibly trigger a final state.

4.1.1 Quick check

The *quick check* technique uses the following insight. A tracematch automaton can only hit its final state if the program executes a sequence of shadows which lead to the final state. If a critical edge in the automaton has no corresponding shadows in the program (leaving the automaton disconnected), then the automaton

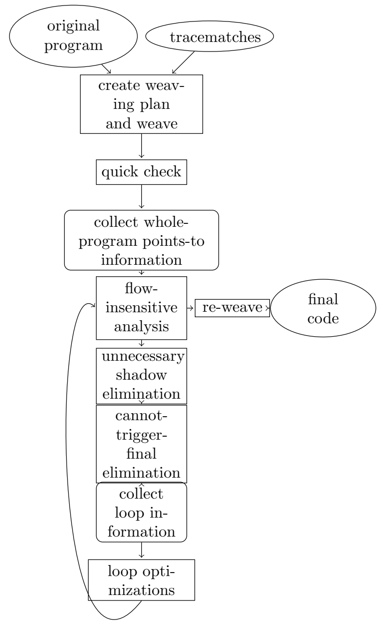


Figure 6: Weaving Process.

can never reach its final state. In that case, the quick check may remove all other shadows belonging to that tracematch.

For example, consider the `HasNext` tracematch and a target program that does not call the `next()` method. Since such a program can never trigger the `HasNext` tracematch, we remove the all instrumentation for this tracematch, including at calls to `hasNext()`.

4.1.2 Flow-insensitive analysis

The *flow-insensitive* analysis uses the following main idea: given a set of shadows which contain transitions reaching the final state, the automaton can only actually trigger on those shadows if each shadow's tracematch variables are potentially bound to the same objects. For the `HasNext` tracematch, if a program calls `i.hasNext()` on some iterator `i`, but never `i.next()` on the same `i`, then it is sound to remove the shadow at `i.hasNext()`.

Shadow groups. A shadow group consists of a collection of shadows that may drive the tracematch into a final state, along with the points-to sets for each shadow's bound objects. The shadow group can only lead to a match at runtime if there potentially exists at least one actual heap object for each bound variable. We call such shadow groups consistent. Our analysis determines whether a shadow group is consistent or not by testing whether the intersection of the points-to sets for bound variables is empty or not. Our flow-insensitive analysis disables all shadows that do not belong to at least one consistent shadow group.

In this work, we only use the following property:

> If two shadows might ever collaborate to drive a tracematch configuration into a final state at runtime, then there exists a shadow group that contains both of these shadows.

Shadow groups enable us to soundly handle method calls in our analysis: if method m transitively calls method n, and m and n have shadows in the same shadow groups, then we must taint the configuration after calls to n, since we are performing an intraprocedural analysis. If they do not have any common shadows, then n has no effect on the configurations we are tracking in m.

4.2 Novel optimizations

Our static analysis enables novel program optimizations that are based on an estimate of the possible tracematch states at various program points. In this section, we present three transformations that use the information collected by our static analysis. Two of the transformations disable shadows that cannot contribute to reaching a final state. Other shadows are not unnecessary—they might contribute to reaching a final state—but only need to execute once. Our other transformation therefore manipulates loops to ensure that such shadows are only executed once, either by hoisting shadows out of loops, or by guarding them with special Boolean flags.

4.2.1 Unnecessary Shadow Elimination

Recall that a program executes a shadow every time it encounters a pointcut corresponding to a symbol definition. Generally, a shadow triggers a change in the tracematch configuration. However, it may turn out that a particular shadow will never change the tracematch configuration, given a set of known possible input configurations to that shadow. Consider the *HasNext* tracematch (Figure 1) and the following typical example of printing the contents of a collection:

```
1    while(it.hasNext()) {
2        if( it .hasNext()) {
3            System.out.println(",");
4        }
5        System.out.println( it .next());
6    }
```

Observe that the inner call to `it.hasNext()`, on line 2, cannot possibly affect the tracematch automaton: the call to `it.hasNext()` on line 1 has already cleared all disjuncts binding `it` from state q_1, so that the call on line 2 is always a no-op. We can therefore safely disable the shadow at line 2.

Note that this transformation is only possible because the shadow at line 2 must execute immediately after the shadow at line 1, ensuring that the iterator `it` must be in the initial state. Flow-sensitivity is crucial for this transformation.

Implementation. We have implemented the unnecessary shadow elimination transformation as follows:

- Collect the static analysis results for method m.
- If the analysis did not reach the fixed point, abort.
- For each shadow-bearing statement s,
  - If, for every input state reaching s, s generates an identical output state, and s is not tainted, then disable shadows at s.

Note that we perform the verification state-by-state: that is, we split the input configuration into a collection of input tracematch states (one input per non-final automaton state) and verify that the output on that state is unchanged. This ensures that our over-estimation of the input configuration does not mask cases where one input state changes the configuration to a different state, but that new state is invisible because it is already in the input configuration.

4.2.2 Cannot-trigger-final Elimination

Our unnecessary shadow elimination handles shadows that do not change the tracematch configuration. Some shadows do change the tracematch configuration, but can still never lead to the final state. Consider, for instance, the following code with the `HasNext` tracematch:

```
Iterator  i = c.iterator ();
while (i.hasNext()) {
  Object o = i.next();
}
```

Clearly, this code can itself never trigger the final state, regardless of the input configuration. Furthermore, if `i` is a local variable and does not escape its defining method, no other shadow in the program can cause the tracematch to hit the final state on `i`. Alternately, if the shadows in the rest of the program cannot subsequently trigger a final state on the objects in our method of interest, then we can remove the shadows in our method of interest. In general, local variables that do not escape are candidates for removal by our "cannot-trigger-final" transformation, as are objects that do escape but whose potentially-dangerous shadows are confined to one method.

Our static analysis identifies all shadows that may reach the final state within a particular method m. However, even if the shadows of m do not trigger the final state while m is executing, they could leave the tracematch automaton in a state where a shadow in some subsequently-executed method will trigger the final state.

We therefore model future actions as follows. First, using our flow-insensitive whole-program analysis (as described in Section 4.1), we identify a set of *relevant* shadows. A shadow is relevant for method m if it is an active shadow, does not belong to m, and shares a shadow group with some shadow in m. Note that only the shadows which share a shadow group with shadows in m can possibly be affected by m; the definition of shadow groups ensures that all other shadows operate on a disjoint set of bound objects and are therefore unaffected by m.

To use the information about future actions, we create an extended control flow graph, augmented with the relevant shadows, and feed the extended CFG to our static analysis. We augment the control flow graph of m by replacing each exit statement s_e of the graph with a jump from s_e to a *synthetic* node. The synthetic node is a fresh node that we create and to which we add all relevant shadows.

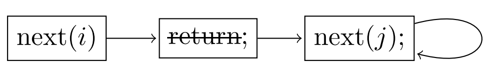


Figure 7: Extended control flow graph.

Figure 7 presents an example of an extended control flow graph for a method with one shadow, `next(i)`. Assume that the program contains one other shadow, `next(j)`, in some other method (a foreign shadow). We replace the `return;` statement with a jump to a synthetic node that triggers the foreign `next(j)` shadow.

If the `next(i)` shadow from the current method and `next(j)` from the foreign method belong to the same shadow group, then `i` and `j` may be aliased, *i.e.* may at runtime point to the same object. In that case, our analysis would have to keep `next(i)` alive. Otherwise, the analysis may safely remove the `next(i)` shadow.

In general, we interpret the results of the static analysis on the extended control-flow graph as follows. If m never hits any final states, we remove all shadows in m. Otherwise, we must make sure that we do not remove any shadow that can contribute to a final state. A shadow can contribute to a final state if any of its successors is either tainted (we therefore assume that the successor will potentially trigger the final state) or contains a shadow that leads into a final state.

Initial approximation. Unlike all of the other analyses, where we care about whether a shadow may change the tracematch configuration or not, in this transformation we only care about whether the tracematch may hit its final state. A sound initial overapproximation for this analysis is simply to assign `true` to each non-final tracematch state and `false` to final states. This approximation represents the state where all objects are potentially bound to all tracematch variables in all non-final states. (Recall that the final state is always empty because bindings immediately leave the final state after triggering the tracematch.)

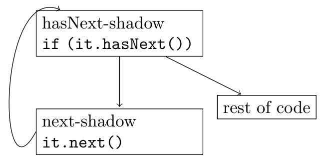


Figure 8: Candidate for shadow motion.

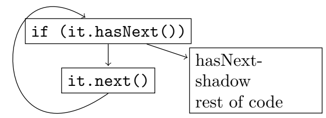


Figure 9: After shadow motion.

Implementation. We use the following algorithm for the cannot-hit-final transformation on method m:

- Augment method m with a synthetic node which contains all relevant shadows from other methods.
- Collect the static analysis results for augmented method m.
- Remove all shadows that do not reach final states or statements with tainted configurations.

Omitting the current method's shadows in the summary of the future. It is sound to omit the shadows of m when computing the set of shadows that may execute in the future, even if m may be called again in the future. Recall that our analysis makes the worst-case assumption of an arbitrary initial configuration, which includes all possible effects of method m. Our static analysis will therefore not miss any final states reached in future calls of m; they would have already been detected at this call to m.

Alias analyses. We comment briefly on the must-alias analysis with respect to the synthetic node. Each shadow on the synthetic node has a set of variable bindings with variables from other methods. Our must-alias analysis therefore states that the variables never must-alias the variables in method m. Note that our may-alias analysis can still give meaningful answers even for foreign variables. Observe also that the result of importing foreign shadows is that they are treated as weak updates (they may execute) but not as strong updates (there is no guarantee that they will execute).

4.2.3 Shadow Motion

Some shadows, of course, are not amenable to optimization with either the unnecessary-shadow or the cannot-trigger-final elimination. For instance, the optimizations that we have presented so far would not help for a shadow that may potentially contribute to a final state in some potential execution; our optimizations so far only eliminate shadows that provably do nothing. Our next insight is that many shadows occur in loops; if we can somehow reduce the number of times that these shadows are executed, then we will speed up the program.

Shadow motion optimizes loops by hoisting shadows out of loops. If the set of shadows in a loop body collectively leave the tracematch state unchanged (except, possibly, for the first time that these shadows execute), and if these shadows can never trigger a final state, then the shadows can be hoisted out of the loop. Note that shadows can be hoisted whether or not they contribute to reaching a final state later on. Consider the following example code with the `HasNext` tracematch:

```
1 while(it.hasNext()) {
2     it.next();
3 }
4 it.next();
5 it.next();
```

Figure 8 graphically illustrates the situation. No shadow in the code can be eliminated by the "unnecessary shadow" analysis. Further, line 5 can trigger the final state on `it`, so the "can't-trigger-final" optimization will not remove any of the shadows either, because line 5 can be reached from all those shadows.

Observe, however, the following two properties of the loop at lines 1-3: (1) the loop can never trigger a final state; (2) the state upon exit from the loop is known, and in particular, it is equivalent to the state obtained by calling `it.hasNext()` exactly once.

Note that the `hasNext` shadow must execute at least once, as it may change the tracematch state. We move the `hasNext` shadow to run after the loop exit, as shown in Figure 9.

Because we know that the loop never triggers the final state, it is sound to move the shadows around. In particular, any effect of these shadows will not be visible until after the loop exits.

Hitting the final state: Execute Shadows Once. If the code in a loop body may actually trigger the final state, it is unsafe to hoist the shadows out of the loop body: if the final state is actually hit at runtime, the tracematch body must execute right away. Here is an example of such a case. Consider the `FailSafeIter` tracematch from Figure 4, which detects collections that are modified while they are iterated on, with the following program:

```
1 Iterator it = c.iterator();
2 if (...)
3     c.add(new Object());
4 while(it.hasNext()) {
5     it.next();
6 }
```

Clearly, the shadow on line 5 may trigger the final state of the `FailSafeIter` tracematch, so that we cannot remove, or even move, that shadow: it must stay where it is. A naïve compiler, of course, will execute the shadow on line 5 every time through the loop. However, our static analysis allows us to conclude that subsequent iterations of the loop do not affect the tracematch automaton; the shadow at line 6 is dead on subsequent iterations (both the hit counter and configuration stays the same after that shadow). In such cases our transformation does not move shadows out of the loop. Instead, it copies shadows to loop exits and then guards the shadows in the loop so that they execute exactly once after the loop header executes.

Our execute-shadows-once transformation is equivalent to unrolling the loop once, leaving the shadows in the unrolled code, and disabling the shadows in the loop itself. We chose to implement this transformation using a boolean flag for practical reasons.

Implementation. We have implemented our "shadow motion" transformation as follows.

- Find all (reducible [13]) loops in method m.
- For each loop ℓ (inner loops first):
  - Apply our static analysis to only the statements of ℓ.

    During this analysis, store (for each statement) the output configurations after the first iteration of the analysis.
  - If, at any loop exist e, the outgoing configurations after the first iteration differ from the outgoing configurations at the fixed point, continue with the next loop.

    Having reached this point, we know that the method's static configurations reach their fixed points in one iteration: no subsequent execution of the loop body has any effect on the tracematch configuration. Because of our second soundness property, this also holds for all dynamic configurations. Executing the shadows in subsequent iterations is therefore unnecessary. We only need to execute the set of shadows in the loop once.
  - If some statement in ℓ potentially triggers a final state, guard the shadows in ℓ with conditionals so that they only execute once.
  - Else, disable all shadows in ℓ, and for each loop exit e:

∗ Determine the set of shadows S which lie on some path from the loop header to e. (Note that the choice of path does not matter: if it did matter, we would not have a stable configuration at e after one iteration. Given a stable configuration, we do not need to worry about infinite paths through inner loops, nor do we have to worry about possible branches within the loop.)

∗ Determine the unique post-loop successor s of e. Add a `nop` statement n before s, and make n the new successor of e. Annotate n with all of the shadows S, in order; they will be woven into the code at the next reweaving.

5 Results

To validate the effectiveness of our optimizations, we applied them to several combinations of tracematches and benchmarks from version 2006-10-MR2 of the DaCapo benchmark suite [5]. We have previously identified [7] a number[2] of benchmark/tracematch combinations as being resistant to our flow-insensitive analysis: even after the flow-insensitive analyses, these benchmark/tracematch combinations still had relatively large overheads.

pattern name	description
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 `hasNextElement` before calling `nextElement` on an `Enumeration`
HasNext	always call `hasNext` before calling `next` on an `Iterator`
Reader	don't use a `Reader` after closing it or its underlying `InputStream`

Table I: Tracematches applied to our benchmarks

benchmark	reachable	flow-insensitive	unnecessary-shadows	cannot-trigger-final	shadow motion	flow-insensitive-2	# moved
antlr/Reader	43	15	15	0	0	0	0
bloat/HashMap	29	28	28	28	28	28	0
bloat/HasNext	640	640	640	630	438	417	359
chart/FailSafeIter	110	110	107	107	107	107	0
luindex/HasNextElem	16	16	15	3	0	0	1
pmd/FailSafeIter	130	91	90	90	90	90	0
pmd/HasNext	88	87	87	73	47	42	43

Table II: Number of shadows remaining after each analysis stage, plus number of shadows moved by shadow motion

[2]We reported nine pathological combinations in [7]. However, Avgustinov et al. have recently further optimized the implementation of the generated runtime monitor [4] so that only seven of those combinations still showed overhead at submission time. We hence conducted experiments on those seven cases.

The tracematches that we use all validate safety properties. We assume that our benchmark applications are mature systems which will not trigger our tracematches. We should therefore—in principle—be able to remove all instrumentation from our benchmarks.

Our tracematches specify usage constraints for frequently used Java Runtime Environment data structures, including collections, iterators, readers and writers. These tracematches tend to induce reasonably hot instrumentation points and can therefore incur substantial instrumentation overheads at runtime without static optimization. Table I summarizes our example tracematches and the properties that they ensure.

Tested configurations. We performed our experiments on four different versions of the benchmark programs:

raw	no tracematch present (raw benchmark program)
no-opt	tracematch present, no whole-program optimizations
flowins	tracematch present, flow-insensitive analysis [7] only
full	tracematch present, flow-insensitive analysis and new intra-procedural analyses enabled

Our new intra-procedural analyses took several minutes to execute in the worst case, and usually substantially less.

Compilation was performed on Sun's 64-Bit Server VM (build 1.6.0-rc-b104), with 3GB maximal heap space and a default stack size of 2MB (-Xmx3072m -Xss2048k) on a machine with AMD Athlon 64 X2 Dual Core Processor 3800+ running Ubuntu with kernel version 2.6.15-28.

5.1 Shadow removal

Table II presents measurements of the effectiveness of our optimizations. In particular, it reports on the number of shadows that each of our optimizations removes. The first column contains the benchmark/tracematch combination. The second number is the total number of reachable shadows in the program. The third column contains the number of shadows remaining after performing the flow-insensitive optimizations from [7]. This column constitutes the baseline; the techniques in this paper seek to show improvements over this baseline number.

Although our three transformations could, in principle, synergistically work together to enable optimizations upon iteration, we found that, in practice, one iteration of each optimization was sufficient to ensure maximal results. In terms of Figure 6, we only needed to run through the optimization loop once; a second iteration had no effect. The next three columns of Table II show the number of remaining shadows after the "unnecessary-shadow", "cannot-trigger-final" and "shadow motion" optimizations.

After these optimizations, we found that a second iteration of the flow-insensitive analysis did sometimes remove additional shadows. The second-to-last column therefore contains the number of shadows remaining after the second application of the flow-insensitive analysis. This is the number of shadows remaining in the fully optimized program.

Because shadow motion does not necessarily reduce the number of shadows in the program, we evaluate its static effectiveness by presenting the number of shadows moved during shadow motion in the rightmost column.

Complete success: Static verification. We were happy to find that our optimizations removed all shadows for the `antlr/Reader` and `luindex/HasNextElem` benchmarks. Such a result has two benefits: the runtime overhead will obviously be nil, and better yet, the benchmarks are statically safe with respect to the verification property encapsulated in the tracematch.

No improvement for `bloat/HashMap`. The `HashMap` tracematch differs from the other tracematches: it does hit its final state. For this tracematch, part of the verification occurs in the tracematch body, which compares objects' current hash codes with previously-stored hash codes. We therefore did not expect to improve this case; instead, this benchmark helped ensure soundness for our analyses.

Other improvements. In the remaining four cases, our optimizations removed more shadows than the first flow-insensitive pass did. For `bloat/HasNext`, we removed 223 shadows, but 417 shadows remained. We believe that this is because `bloat` uses a number of highly non-local data structures, and iterators are frequently passed across method boundaries. `bloat` therefore does not lend itself very well to the optimizations described in this paper. In `pmd/HasNext`, some of the examples cannot be verified with any intraprocedural analysis; consider the following:

```
if (!c.isEmpty()) { foo(c.iterator().next()); }
```

Although this use of an iterator is innocuous, a conservative initial approximation for the `Iterator` returned by `c.iterator()` must assume that this (fresh) `Iterator` may already have had `next()` called once on it.

We found situations which were more worrisome from a maintenance perspective. Consider the following code sketch:

```
void bar() { if (!c.isEmpty()) { foo(c.iterator()); } }

private void foo(Iterator i) { doIt(i.next()); }
```

This code is safe as long as `foo(..)` is only called by `bar(..)`. However, `foo(..)` does rely on the fact that its input `Iterator` still has more elements. Of course, there is no documentation of this assumption (except in the `private` scope of `foo`). An unwary developer could easily call `foo` and crash the program.

Remaining shadows. Our optimizations were not able to improve on the flow-insensitive analysis in the `chart/FailSafeIter` and `pmd/FailSafeIter` benchmarks. We carefully investigated these cases and found the following idiom: method m iterates over a collection c with a fresh iterator i, and method m' updates this collection. We described an optimization for weak updates in Section 3.4. This optimization was designed to handle such cases. If we can determine that no method but m creates any iterator aliased to i (as is always the case), then our analysis of m will be able to omit the weak update at the call to `i.next()`. This weak update is currently preventing us from applying both cannot-trigger-final elimination and shadow motion to m. Unfortunately, our global points-to analysis currently does not provide enough precision to determine that no object in other methods may possibly alias i, due to a lack of context information.

Applicability of optimizations. Our results show that different transformations help differently on different benchmarks. While shadow motion removed by far the most shadows for `bloat`, cannot-trigger-final was very effective for `lucene`. Unnecessary shadow elimination did not remove many cases, but was easy to implement, and we hope that generalizations of unnecessary shadow elimination will be useful.

5.2 Runtime improvements

We executed all of the relevant benchmark/tracematch combinations to measure the runtime overhead. Because some of the benchmarks require a Java 1.4 Virtual Machine, we executed all of our benchmarks on Sun's HotSpot 32-Bit Client VM (build 1.4.2_12-b03), with 2GB of maximal heap space on a machine with a AMD Athlon 64 X2 Dual Core Processor 3800+ running Ubuntu 6.06 with kernel version 2.6.15-28. We used the standard workload size for the benchmark and enabled the `-converge` option, which tries to assure timing within a confidence interval of 3%. The benchmark chart renders objects to the screen. In order to minimize distortion by that fact, we opened a VNC server with a virtual screen that is not actually displayed. Screen output of chart was then redirected to that server. The two benchmarks luindex/lusearch use the same binaries (the program lucene) which are compiled/optimized only once. Just their run configuration differs.

Table III shows the results of those measurements. The "raw" column shows the running time of the raw benchmark without any tracematches. The "no-opt" column shows the slowdown with tracematches but without any whole-program program optimizations. Next, the "flowins" column presents the slowdown with the flow-insensitive analysis from [7] has been applied. Finally, the "full" column contains the slowdown after applying the optimizations in this paper.

benchmark	raw	no-opt	flowins	full
antlr/Reader	4.1s	5.50x	1.07x	1.00x
bloat/HashMap	9.6s	1.88x	1.89x	1.89x
bloat/HasNext	9.6s	15.55x	14.75x	13.78x
chart/FailSafeIter	14.7s	1.09x	1.09x	1.09x
luindex/HasNextElem	17.3s	1.12x	1.10x	1.02x
pmd/FailSafeIter	12.8s	2.12x	1.87x	1.84x
pmd/HasNext	12.8s	1.62x	1.63x	1.08x

Table III: Runtime overheads after different analysis stages

As we would expect, the two benchmarks where we removed all shadows suffer no overhead. For `pmd/HasNext`, even though we removed just over half of the original shadows, the runtime overhead shrinks from 63% to just 8%. This is by design: our analysis was targetted towards typical situations where hot shadows would be likely to occur. `pmd` has only two pairs (each containing `next` and `hasNext`) of such shadows, which account for 90% of the overhead. Our cannot-trigger-final optimization removes one of these pairs, while shadow motion improves the other pair. For the other benchmarks, speedups are in line with the proportion of shadows removed. The final result leaves us with only three benchmarks (one unoptimizable), out of an initial set of 90 benchmarks, that carry a runtime overhead of more than ten percent.

Our benchmark set and our current version of `abc` are available at `http://www.aspectbench.org`, along with an extended technical report version of this paper (`abc-2007-2`). As usual, our optimizations will be part of the upcoming release of `abc`.

6 Related Work

We next discuss a number of areas of related work. We first describe the relationship between our work and the ASTREE project, which uses static analysis to ensure that programs never trigger runtime errors. We compare our work on tracematches to research on the alternate specification languages encapsulated in typestate-based approaches and PQL. Finally, we explain the relationship between the standard loop hoisting compiler optimization and our loop optimizations for shadows.

6.1 Eliminating runtime errors

The ASTREE static analyzer [6] has successfully verified millions of lines of automatically genrated C code for the absence of runtime errors. ASTREE verifies that programs never trigger the runtime errors defined in the C language specification. Examples of such errors include out-of-bounds array accesses and arithmetic overflow. It combines a number of different static analyses to statically verify program properties, and in general, ASTREE uses abstract interpretation over a number of specialized abstract domains.

While, like ASTREE, we use static analyses to detect cases where error conditions might occur, our goals differ substantially from those of ASTREE. ASTREE attempts to remove all possible runtime errors from the code; we instead flag possible matching tracematches and to evaluate them at runtime. Furthermore, instead of detecting a set of runtime errors that is fixed in the C language specification, our specification language is flexible: we enable developers to choose the properties that are important to them, by supporting any property that can be specified as a regular expression over symbols. Finally, ASTREE generally verifies that integers and floating-point numbers fall within acceptable ranges, while we verify relationships between events on heap objects.

6.2 Typestate

Typestate properties [14] have been enjoying renewed interest; recently, Fink et al. presented a static analysis for the runtime checking of typestate properties [8]. Their approach, like ours, uses a staged analysis which starts with a flow-insensitive pointer-based analysis, followed by flow-sensitive checkers. Note that Fink et al. aim to verify properties fully statically, and emit a warning or error message if they fail to verify the property, rather than inserting instrumentation code as we do. Also, Fink et al. do not discuss how developers might specify properties to be verified. Tracematches enable developers to specify the properties that are to be verified.

Tracematches are more expressive than typestate. Typestate describes the state of heap objects one-at-a-time: for instance, heap object i is in state s at program point p. Tracematches enable developers to relate the state of multiple heap objects. Tracematches are therefore more difficult to optimize than typestate properties: they change the worldview from one where it is sufficient to focus on a particular object to one where arbitrary objects can affect the property of interest. Consider the following two concrete cases.

Our `HasNext` example binds one free variable, i, for the `Iterator` object being considered. After any call to `i.hasNext()`, we know that the object bound to i must be in its initial state; in principle, we could track i from its creation site throughout its lifetime. Because tracematches bind multiple objects simultaneously, it is no longer clear where to start tracking the tracematch state for tracematches with multiple variables.

Next, consider the `FailSafeIter` example, which ensures that an `Iterator` does not suffer from changes in its underlying `Collection` during iteration. To analyze a call to `c.add()`, we need to know about all of the iterators that might be active over `c`; in principle, any `Iterator` in the program might be based on the `Collection` object `c`. Because typestate properties only constrain one object at a time, an analysis for typestate properties can always derive enough context information just from looking at the actions on the object itself. These two issues—where do we start the analysis? which objects are bound simultaneously?—drove us to use constraints in our static abstraction. These constraints cannot specify precisely which objects are related, but they encapsulate the information that we need for our analyses.

6.3 Program Query Language

The Program Query Language [11] is similar to tracematches in that it enables developers to specify properties of Java programs, where each property may bind free variables to runtime heap objects. PQL supports a richer specification language than tracematches, since it is based on stack automata rather than finite state machines. Runtime overhead is a problem for both PQL and tracematches, and the authors show that a flow-insensitive, pointer-based analysis can eliminate much of the overhead incurred by using PQL. Their approach inspired the flow-insensitive optimizations in our earlier work [7].

Because PQL uses a flow-insensitive approach to static analysis, we believe that it would suffer comparable overheads to those in [7]; no flow-insensitive analysis can remove any more instrumentation in the cases that we consider in the present paper. We have shown that our novel flow-sensitive approach can successfully handle more cases than older approaches, and we believe that our approach would apply to PQL as well. A static abstraction of PQL stack automaton configurations which satisfies the properties from Section 3 would therefore enable the use of our optimizations.

6.4 Loop optimizations

One of our optimizations is modelled on loop hoisting: we move redundant code out of loop bodies. Loop hoisting (also known as code motion) is a well-known standard compiler optimization technique (e.g. [13], chapter 14). In the standard setting, loop hoisting attempts to move redundant computations out of loops as follows. If an expression computed within a loop depends only on values defined before entering the loop, and this expression has no side-effects, then the computation of the expression can be moved out of the loop. The general code motion problem is difficult: any computations to be moved out of a loop must be free of side effects and dependences on the heap. Xu [17] and Sălcianu [15] report on some purity analyses for Java which would enable such code motion.

In any case, we are interested only in moving shadows out of loops. Fortunately, executing a tracematch shadow can only have two side-effects: the shadow may (1) modify the current configuration, or (2) execute the tracematch body. Our abstraction captures all possible instances of both of these side effects, enabling us to move shadows out of loops. Note that the finite-state model of tracematches enables us to quickly determine if two configurations are identical or not; it is, of course, much harder to determine whether two general program configurations are identical.

7 Conclusions

In this paper we have presented a novel approach to optimize the instrumentation required for runtime monitoring. We have proposed three different intra-procedural optimizations that identify unnecessary instrumentation using local flow-sensitive state and alias information. All three optimizations use a sound static abstraction of finite state machines for tracematches, the particular runtime monitoring system we developed. We see no reason that our approach should be restricted to tracematches. In general, our optimizations could use any sound static abstraction of runtime configurations.

Our results show that our three optimizations can remove most of the instrumentation in our benchmark set. We greatly improved the performance of four benchmarks which each had perceivable runtime overheads before our optimizations. For two of these benchmarks, we showed that the given tracematches never apply, making instrumentation unnecessary: for tracematches which capture error situations, our analysis proves that the error situations can never arise, and the program is proven safe with respect to that error. Another benchmark is not amenable to improvement by any static instrumentation-elimination technique, as it actually executes its body. For the remaining two benchmarks, we removed some instrumentation but did not reduce the observed runtime overhead. We believe that an improvement in the global points-to analysis that we use will enable our optimizations to remove most of the overhead in these cases.

Implications. We discuss broader implications of our research.

Many static analyses attempt to discover properties of the program being analyzed. Our analysis instead focusses on the behaviour of the runtime monitor—something external to the program. Because our analysis verifies properties that are specified apart from the program itself, each developer is free to specify the properties that are most useful to him or her.

Our static analysis is intraprocedural; we were somewhat surprised to find that it worked as well as it did. We believe that an intraprocedural analysis is powerful enough to handle many useful cases for the following reason. Recall that our use of tracematches aims to guarantee certain constraints on the program's behaviour; we want to ensure that the program never executes certain pathological sequences of events. When implementing a method, defensive programming on the developer's part will often (but not always) ensure that the program state is appropriate before proceeding with the method's actions. Because such defensive programming ensures that the program is in a desirable state, the static analysis can also use the fact that the defensive code has successfully completed to conclude that the tracematch will never be triggered.

Currently, runtime monitoring is usable during development but not for deployed programs: the runtime overhead is tolerable but noticeable. A goal of our research is to make runtime monitoring feasible in deployed code. This paper contributes to our goal by significantly reducing the runtime overheads for our benchmark applications; most of our benchmark programs suffer no performance loss at all under our benchmark runtime monitors.

Acknowledgements. We owe thanks to Manu Sridharan for help with using his demand-driven points-to analysis. We also thank the entire `abc` group for their continuing support.

References

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Object-Oriented Programming, Systems, Languages and Applications*, pages 345–364. ACM Press, 2005.

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising AspectJ. In *Programming Language Design and Implementation (PLDI)*, pages 117–128. ACM Press, 2005.

[3] P. Avgustinov, J. Tibble, E. Bodden, O. Lhoták, L. Hendren, O. de Moor, N. Ongkingco, and G. Sittampalam. Efficient trace monitoring. Technical Report abc-2006-1, abc, March 2006.

[4] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2007. To appear.

[5] 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. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA06*, pages 169–190, 2006.

[6] B. Blanchet, P. Cousot, R. Cousot, J. me Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *ACM PLDI*, San Diego, California, June 2003. ACM.

[7] E. Bodden, L. J. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP07*, 2007.

[8] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *Proc. 2006 International Symposium on Software Testing and Analysis*, 2006.

[9] R. Halpert, C. Pickett, and C. Verbrugge. Component-based lock allocation. In *PACT*, September 2007. To appear.

[10] C. Lapkowski and L. J. Hendren. Extended SSA numbering: Introducing SSA properties to languages with multi-level pointers. In *Proc. 1998 International Conference on Compiler Construction*, volume 1383 of *Springer LNCS*, pages 128–143, March 1998.

[11] M. Martin, B. Livshits, and M. S. Lam. Finding application errors using PQL: a program query language. In *Proc. 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 365–383, 2005.

[12] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Compiler Construction*, volume 2622 of *Springer Lecture Notes in Computer Science*, pages 46–60, 2003.

[13] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[14] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[15] A. Sălcianu. *Pointer Analysis for Java Programs: Novel Techniques and Applications*. PhD thesis, Massachussetts Institute of Technology, September 2006.

[16] R. Vallée-Rai. Soot: A Java optimization framework. Master's thesis, McGill University, July 2000.

[17] H. Xu, C. J. F. Pickett, and C. Verbrugge. Dynamic purity analysis for Java programs. In *PASTE'07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, New York, NY, USA, June 2007. ACM Press.