Optimizing Nop-shadows Typestate Analysis by Filtering Interferential Configurations

Chengsong Wang[1], Zhenbang Chen[1,2⋆], Xiaoguang Mao[1]

[1] College of Computer, National University of Defense Technology, Changsha, China
[2] National Laboratory for Parallel and Distributed Processing, Changsha, China
jameschen186@gmail.com,{zbchen,xgmao}@nudt.edu.cn

Abstract. Nop-shadows Analysis (NSA) is an efficient static typestate analysis, which can be used to eliminate unnecessary monitoring instrumentations for runtime monitors. In this paper, we propose two optimizations to improve the precision of NSA. Both of the optimizations filter interferential configurations when determining whether a monitoring instrumentation is necessary. We have implemented our optimization methods in Clara and conducted extensive experiments on the DaCapo benchmark. The experimental results indicate that the optimized NSA can further remove unnecessary instrumentations after the original NSA in more than half of the cases, without a significant overhead. In addition, for two cases, all the instrumentations are removed, which implies the program is proved to satisfy the typestate property.

Keywords: Typestate Analysis, Runtime Monitoring, Static Analysis, Nop-shadows Analysis

1 Introduction

A typestate property [23] describes the acceptable operations on a single object or a group of inter-related objects, according to the current state (i.e., the typestate) of the object or the group [7, 10]. For example, usually, programmers cannot call the method *write* until the method *open* is called on a same File object. Lots of large-scale software system errors are caused by the violations of typestate properties. What is worse, it is very difficult and time-consuming to find out and fix these errors [6, 22]. The static analysis of a program with respect to a typestate property is generally undecidable. The existing static typestate checking tools [3, 19] suffer from the scalability and the false-alarm problems. Dynamic typestate checking methods complement the static methods with runtime monitoring to improve the scalability and the accuracy of the analysis, but sacrifice the completeness.

Usually, dynamic typestate analysis approaches, such as runtime verification [5, 11, 15, 16], automatically convert typestate properties into runtime monitors that can detect the property violations at runtime. Implementing runtime monitors needs to instrument the monitored programs. The instrumentation can

⋆ Corresponding author.

be done manually or automatically based on existing techniques, such as AOP [14]. However, the programs instrumented with runtime monitors usually contain many redundant instrumentations, which result in a significant monitoring overhead. Therefore, some approaches [6, 12] exploit static analysis information to remove provable unnecessary instrumentations for reducing the overhead of runtime monitoring. These methods are often called hybrid typestate analysis.

Theoretically, hybrid typestate analysis is equivalent to the static analysis of typestate properties. If all the instrumentations of a runtime monitor can be removed, the program is proved to satisfy the typestate property. Nop-shadows analysis (NSA) [6, 7] is one of the existing hybrid analysis methods. NSA is implemented in Clara [1] to optimize the runtime monitoring of large-scale Java programs. NSA uses intra-procedural flow-sensitive and partially context-sensitive data-flow analysis to identify the redundant instrumentations generated for monitors.

Although NSA is effective [6, 7], there are some cases in which unnecessary instrumentations still remain after NSA. One of the main reasons is that NSA is only an intra-procedural flow-sensitive static analysis. The overly conservative approximations of inter-procedural cases in NSA reduce the accuracy of the analysis. In this paper, we propose two optimizations to improve the precision of NSA. Both of the optimizations can filter interferential configurations when determining whether a monitoring instrumentation is necessary. An interferential configuration refers to the configuration that lowers the precision of identifying "nop shadows" in NSA. One optimization identifies changeless configurations produced by the backward data-flow analysis of NSA; the other one utilizes local object information to refine the iterations of data-flow analysis. Using the two optimizations, more unnecessary instrumentations can be removed.

To evaluate our optimizations, we have integrated our optimizations into Clara, and applied them to the DaCapo benchmark suite [4]. In more than half of the cases, the optimized NSA can further remove unnecessary instrumentations after the original NSA. In two cases, we get a perfect result, *i.e.*, all the monitoring instrumentations are removed, entirely obviating the need for monitoring at runtime.

To summarize, our paper has the following contributions:

- Propose two optimizations for NSA to improve the precision of the analysis. Both of the optimizations filter interferential configurations by identifying changeless configurations and exploiting local object information.
- Propose and implement an approximate, but sound, intra-procedural flow-sensitive algorithm to determine whether a variable points to a local object.
- Implement the two optimizations and integrate them into Clara.
- Conduct extensive experiments on the DaCapo benchmark suite to show the effectiveness of our optimizations.

The remainder of this paper is organized as follows. We begin with an overview of NSA in Section 2. In Section 3, we give two motivating examples to illustrate the two different optimization methods, respectively. Section 4 formulates the details of our proposed optimizations. Our experiments, described in

Section 5, justify that our optimizations are effective in the majority of cases. Section 6 describes the related work and the paper is concluded in Section 7.

2 Nop-shadows Analysis

As in the literature [6, 7, 17], we also use the term "shadow" to represent an instrumentation point created for runtime monitoring. NSA is a static typestate analysis method proposed and implemented in the Clara framework [6], which extends tracematch [2] with static analysis to remove "nop shadows". Here a "nop shadow" means that the shadow does not influence the results of runtime monitoring, *i.e.*, it can neither trigger nor suppress a property violation [6, 7].

Clara consists of three static analysis stages, in which NSA is the most expensive and precise one. Given a typestate property (usually a finite-state machine (FSM)) and an instrumented Java program, NSA uses an intra-procedural data-flow analysis to check whether a shadow in a method of the program can be removed. The basic idea of NSA is to compute the reachable states of each statement in a program according to the semantics of the program and the monitored typestate property. Given an FSM typestate property M and its state set S, for each statement st, there are two types of reachable states: $source(st)$ and $futures(st)$, which are calculated respectively by a *forward* data-flow analysis (forward analysis) and a *backward* data-flow analysis (backward analysis). The source set $source(st) \subseteq S$ contains all the states that can be reached before executing st from the beginning of the program; $futures(st) \subseteq \mathbb{P}(S)$ is the *future* set, and each element of $futures(st)$ contains the states from which the remainder program execution after st can reach a final state (usually the error state) of M. Therefore, for a given shadow s, which is usually a method call statement in the program, NSA identifies s as a "nop shadow" if the execution of the shadow has no impact on the monitoring result, which can be formalized by following two conditions:

- $target(s) \cap F = \emptyset$, where $target(s) = \{q_2 \mid \exists q_1 \in sources(s) \bullet q_2 = \delta(q_1, s)\}$ is the resulting state set after executing s, $\delta(q_1, s)$ is the resulting state after executing s from the state q_1 according to the FSM property M, and F is the final state set of M. This condition means the execution of s does not directly lead to an error state.
- $\forall q_1 \in source(s), \forall Q \in futures(s) \bullet q_1 \in Q \Leftrightarrow \delta(q_1, s) \in Q$. It means the execution of s does not influence whether or not a final state will be reached.

The shadow s can be removed if both conditions are valid.

Figure 1 gives an example for NSA. The left part is an FSM for "ConnectionClosed" [7] typestate property, which requires the "write" operation should not be called after a connection is closed. The right part displays a program annotated with the state information of each statement. The elements in the *source* set and the $futures$ set of each statement are next to the downward and upward arrows, respectively. For instance, for the shadow s_3 at line 3, we have:

$$source(s_3) = \{0\}$$

$$target(s_3) = \{1\}$$
$$futures(s_3) = \emptyset$$

$futures(s_3) = \emptyset$ means that there is no state from which the property state machine can reach the final state via the execution after line 3. According to the preceding two conditions, s_3 is a "nop shadow" that can be removed.

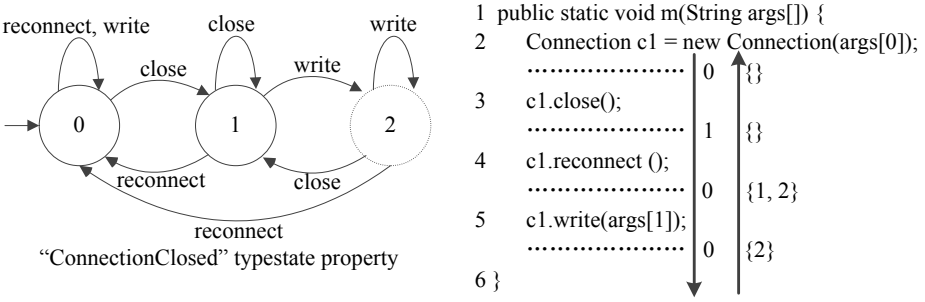

Fig. 1: An example for Nop-shadows Analysis

After removing a "nop shadow", the $source(st)$ and $futures(st)$ of each statement will be calculated again, until no "nop shadow" exists. If there is no shadow after NSA, the program is proved to satisfy the typestate property. For example, all the shadows of the program in Figure 1 will be removed finally. For the inter-procedural cases, the method calls are soundly approximated by using the transitive closure of the shadows in the called methods.

3 Motivating Examples

We motivate our optimizations of NSA through two examples. We also use the "ConnectionClosed" property in Figure 1 as the typestate property. Figure 2 shows an example that invokes the "close" and "write" methods of the class Connection. The shadows at line 7 and 10 violate the typestate property, because they can both drive the state machine into the final state. The "close" operation at line 8 is between these two violating shadows. Hence, from the semantics of the program and the property FSM (*c.f.* Figure 1), the runtime monitor does not need to monitor the shadow at line 8. Whereas, the original NSA cannot identify the shadow at line 8 as a "nop shadow" at compile-time. The reason is explained as follows.

For the sake of brevity, Figure 2 only shows partial critical state information calculated by the forward and backward analysis. In order to distinguish the typestates of multiple different objects or groups of related objects, the data-flow analysis of NSA propagates "configurations" instead of only state sets [7]. A configuration specifies the state information of some specific objects. A configuration $C = (Q, b)$ is composed by a state set Q and a variable binding b.

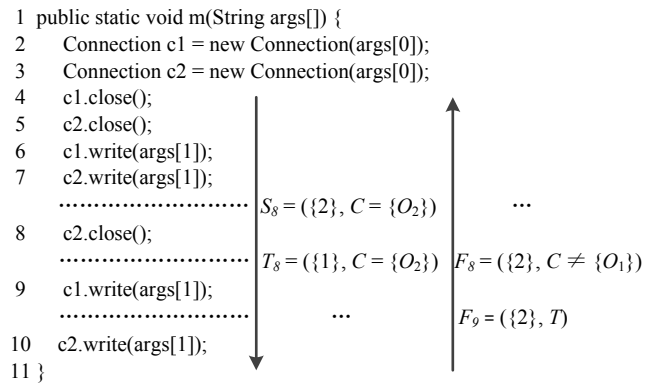

Fig. 2: The example for motivating the first optimization

The variable binding specifies the static objects [9] which represent the concrete runtime objects. Actually, for a shadow s, it also has a variable binding [8] specifying the objects whose typestates can be changed by s. Two variable bindings are *compatible* if they can be bound to a same static object or a same group of related static objects. A configuration and a shadow are *compatible* if their variable bindings are compatible. For a statement st associated with a configuration (Q, b), in forward analysis, the elements in set Q represent all the possible states which the static objects specified by the variable binding b can reach just before st; in backward analysis, they are the states from which the static objects specified by b can reach a final state via the execution after st. For example, the configuration S_8 in Figure 2 represents that the static object O_2 can reach state 2 before executing line 8. The configuration F_{12} in Figure 3 represents that the static object O can reach the final state from state 0 or 1 via the execution after line 3.

Providing that the program creates the compile-time static objects O_1 and O_2 at line 2 and line 3, respectively. The variable binding of the shadow at line 8 is $C = \{O_2\}$, and the shadow at line 8 changes the configuration from $S_8 = (\{2\}, C = \{O_2\})$ to $T_8 = (\{1\}, C = \{O_2\})$ in the forward analysis, with respect to the typestate property. $F_8 = (\{2\}, C \neq \{O_1\})$ associated to the shadow at line 8 is one of the resulting configurations produced by the backward analysis starting at line 9, which means the typestate of the object does not change if the object is not O_1. The variable bindings of S_8 and F_8 are both compatible to that of the shadow at line 8. According to the "nop shadow" conditions, because the states of the state transition caused by the shadow at line 8, *i.e.*, state 2 in S_8 and state 1 in T_8, are not both contained in the state set $\{2\}$ of F_8, NSA fails to identify this shadow as a "nop shadow".

Actually, the configuration F_8 is induced by the "final shadow"[3] at line 9, in which the variable c_1 is totally unrelated to the variable c_2 at line 8, *i.e.*,

[3] "final shadow", which can drive the FSM of the property into a final state [6].

they must not alias. Hence, in principle, we should filter this type of interferential configurations generated from backward analysis when checking whether a shadow can be removed. Based on this insight, our optimized NSA can successfully identify the shadows, similar to the shadow at line 8, as "nop shadows".

Figure 3 shows another example to motivate the second optimization approach. Different from the former one, the typestate property "Connection-Closed" is not violated by the method m. Therefore, all the shadows in method m can be safely removed. However, by using the original NSA, all the shadows will remain.

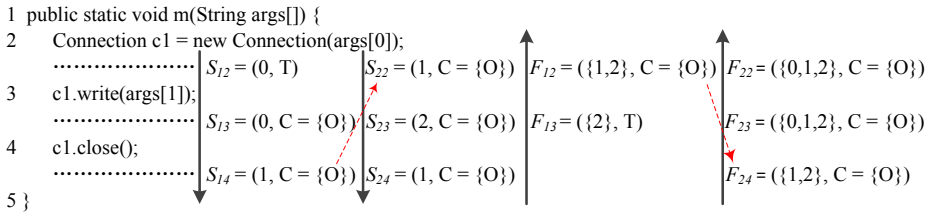

Fig. 3: The example for motivating the second optimization

The problem is mainly resulted by the approximated inter-procedure analysis. Figure 3 shows partial forward and backward analysis results that are next to the two downward arrows and two upward arrows, respectively. Because there may be several consecutive method calls to a method in a program, for ensuring the soundness, the forward analysis needs to propagate the configuration at the end of a method to the entry of the method until a fixed-point is reached. For example, S_{14} is propagated to the entry configuration S_{22} of the next iteration (indicated by the red dotted line). The propagation also happens in backward analysis. After reaching the fixed-point, the shadow at line 3 can produce the configuration S_{23}, which contains an error state. Thus, this shadow cannot be removed. In addition, the shadow at line 4 changes the configuration S_{13} to S_{14}, but state 0 in S_{13} and state 1 in S_{14} are not both contained in the state set $\{1, 2\}$ of the configuration F_{24}. Therefore, the shadow at line 4 cannot be removed either.

After carefully analyzing the example program, we find the reason is that the configuration propagation disregards the *local object* information. In this paper, we call a static object, which is created by a "*new*" statement within the method currently being analyzed, a *local object*. For example, the static object O created by the statement at line 2 is a local object. Obviously, at runtime, each local object will be assigned with a different runtime object each time when the method is invoked and the "*new*" statement is executed. Therefore, for the example in Figure 3, the configuration S_{22} should not have a same variable binding as S_{14}. If we have the local object information of the example program, *i.e.*, no need to do the second forward iteration and the second backward iteration, then both of the "nop shadows" at line 3 and line 4 can be removed. Based on the obser-

vations motivated by this example, we optimize NSA by exploiting local object information.

For simplicity, the motivating examples do not contain complex programming language features, such as recursion, exception handling and aliasing. In Section 4, we will give the details of our optimization methods that can be applied in general.

4 Optimization Methods

This section presents the details of our optimization methods for filtering interferential configurations when checking whether a shadow is a "nop shadow". The first subsection explains how to identify changeless configurations generated from backward analysis. The second subsection proposes an algorithm for determining whether a static object is a local object, and describes how to propagate configurations along the inter-procedural control-flow of a analyzed method. The two optimizations are complementary to each other. They separately address different issues that can potentially lead NSA to lose precision. Therefore, these two optimizations can be combined together to further improve the accuracy of NSA.

4.1 Identifying Changeless Configurations

How can we identify changeless configurations, like F_8 in Figure 2, from the results produced by backward analysis? Basically, if the states of a configuration have never been through a state transition during backward analysis, then we consider the configuration as a *changeless* configuration. For a changeless configuration $C_i = (Q_i, b_i)$ that is induced by a "final shadow" s_f and associated to a shadow s_i, even if C_i is compatible with s_i, there is no need to consider C_i when checking whether the shadow s_i is a "nop shadow". The reason is: the states in set Q_i of the objects specified by the variable bindings b_i will definitely not change anymore before program execution passes the "final shadow" s_f, and the execution of the "final shadow" s_f would not trigger an error because of the incompatibility of s_f and C_i.

Hence, we extend the original configuration tuple from (Q, b) to (Q, b, T), where Q is the state set, b is the variable bindings and T indicates whether the states of this configuration have ever been through a state transition before. Therefore, we need to record the information of T during the configuration transitions in backward analysis. The new configuration transition algorithm is displayed in Algorithm 1.

The algorithm is basically the same as that in [6]. The different parts are enclosed in boxes. Line 4 computes the state set of the successor configuration. If the shadow s can drive the state set Q_c to Q_t (*c.f.* line 4), and the shadow is compatible to the configuration (determined by $\beta^+ \neq \perp$ at line 7), the value of T in the successor configurations is assigned with *true*, indicated by Lines 7-9; otherwise, the value of T remains the same during the configuration transition

Algorithm 1 $transition((Q_c, b_c, T_c), s, \delta)$

1: $cs := \emptyset$; // initialize result set
2: $l := label(s), \beta_s := shadowBinding(s)$; //extract label and bindings from s
3: //compute target states
4: $Q_t := \delta(Q_c, l)$;
5: //compute configurations for objects moving to Q_t;
6: $\beta^+ := \underline{and}(b_c, \beta_s)$;
7: **if** $\beta^+ \neq \perp$ **then**
8: $\quad cs := cs \cup (Q_t, \beta^+, true)$;
9: **end if**
10: //compute configurations for objects staying in Q_c;
11: $B^- := \bigcup_{v \in dom(\beta_s)} andNot(b_c, \beta_s, v) \setminus \{\perp\}$;
12: $cs := cs \cup \{(Q_c, \beta^-, T_c) \mid \beta^- \in B^-\}$;
13: **return** cs;

(Lines 11-12). Moreover, for the backward analysis, when we create an initial configuration, the value of T in the initial configuration is set to *false*.

Based on the extended configuration definition and the transition algorithm, we can identify changeless configurations produced by backward analysis. For a given shadow s and a configuration (Q, b, T) in $futures(s)$, if T is *false*, the configuration will be considered to be interferential, and should be filtered when checking whether the shadow s is a "nop shadow".

4.2 Exploiting Local Object Information

First, we present how to determine whether a static object is a *local object*. We have the following two observations: first, for a given static object inside in a method, if it is created by a "*new*" statement within the method, the object must be a *local object* to this method; second, for any two *strong must-alias* [9] static objects O_1 and O_2 inside a method, these two objects always refer to a same heap object, which implies that they always point to a same local object or a same non-local object. Based on these two insights, Algorithm 2 is designed to identify local objects.

For a given method m and a static object O, the algorithm returns *true* if O is a local object in m; otherwise returns *false*. Algorithm 2 first declares a set *newObjects*, and then adds all the local objects created by the "*new*" statements in m to *newObjects* (Lines 1-7). Then, the algorithm checks whether there exists an element in *newObjects* that is *strong must-alias* to O (Lines 8-12). Currently, the must-alias analysis is only intra-procedural flow-sensitive, and makes a conservative assumption that any two static objects coming from different methods may alias. Therefore, Algorithm 2 is an approximated, but sound, evaluation. In order to gain more efficiency, we extract Lines 2-7 from Algorithm 2 and compute the *newObjects* set before the optimized NSA.

Besides identifying *local objects*, for a configuration, we also need to know whether it is gotten by statically modeling the multiple consecutive invocations

Algorithm 2 isLocalObject(m, O)

```
1: Set⟨staticObject⟩  newObjects;
2: for all stmt ∈ m do
3:     if stmt is a new statement then
4:         create a new static object O_i;
5:         newObjects := newObjects ∪ {O_i};
6:     end if
7: end for
8: for all O_j ∈ newObjects do
9:     if O_j must-alias O then
10:        return true;
11:    end if
12: end for
13: return false;
```

of the analyzed method in forward or backward analysis. Same as the first optimization, we also extend the original configuration tuple to a triple (Q, b, R), where R is *true* if the current configuration is indeed gotten by statically modeling the multiple consecutive invocations of the analyzed method.

For a given method m, the intra-procedural control-flows cannot lead to the multiple consecutive invocations of m. Hence, the shadows in methods cannot change the value of R. Figure 4 visualizes four types of possible inter-procedural control-flows (*solid* arrows) of m [7]. Solid arrows (1) and (2) are used to model the transitively recursive method calls to m. We cannot determine that a method call must be transitively recursive at compile-time. Hence, both of the arrows (3a) and (3b) are used to model the non-recursive method calls within m. Additionally, method m can re-executes again after its returning. Arrows (4) is used to model this case. Obviously, there are only three types of inter-procedural control flows (*solid* arrows (1), (2) and (4)) in Figure 4, which can lead to the multiple consecutive invocations of the method m. Therefore, in forward analysis, for all the configurations that reach the entry statements of m or the recursive call sites within m along these inter-procedural control flows, we should assign *true* to R in these configurations. Based on the same argument, in backward analysis, the value of R in configurations, which reach the exit statements of m or the recursive call sites within m along these reverse inter-procedural control flows, should be assigned *true*. Furthermore, for each initial configuration, the value of R is set to *false* in both forward and backward analysis.

Based on Algorithm 2 and the extended configuration, we can filter interferential configurations as follows: for a given shadow s, which is usually a method call statement, inside a method, if there exists a variable v in the variable bindings of s pointing to a *local object*, a configuration (Q, b, R) in $source(s)$ or $futures(s)$ can be safely eliminated if R is *true*. The reason is: even if the shadow s and the configuration have a same static variable binding with respect to the variable v, v will definitely point to a different object during each method invocation at runtime, which means that the shadow s and the configuration are actually

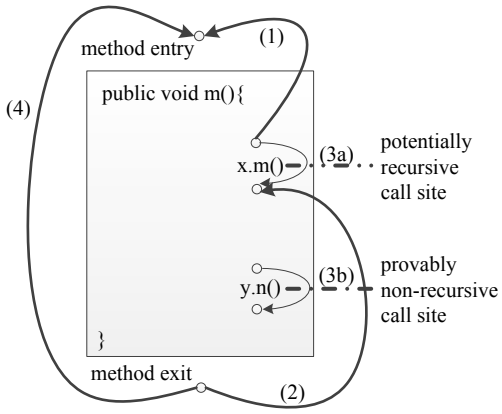

Fig. 4: Possible inter-procedural control-flows of a method [7]

not compatible at runtime. After eliminating all the interferential configurations from $sources(s)$ and $futures(s)$, we use the remaining configurations to determine whether the shadow s is a "nop shadow" according to the conditions in Section 2.

5 Experiments and Discussion

We have implemented our optimizations on the Clara framework [6] and conducted experiments on the DaCapo benchmark suite [4]. NSA cannot support multi-threaded programs. Hence, we ignore the multi-threaded programs *hsqldb*, *lusearch* and *xalan* in the benchmark. Our experiments are based on the experiments of the original NSA in [7]. We are only interested in 23 property/program combinations for each of which the original NSA cannot remove all the shadows. These 23 combinations involve 8 typestate properties and 7 programs. Table 1 lists the typestate properties used in the experiments.

In order to make our experimental results more convincible, we also limit the maximum number of configurations to be 15000, which is the same as that of the original NSA in [7]. Once the number of configurations computed by our optimized analysis is above the threshold, it will abort the analysis of the current method and process the next one.

We evaluate our optimizations as follows: for each optimization, we carry out original NSA first, then use the optimized NSA to further identify "nop shadows". This way of evaluation is *different* from using an optimized NSA directly. Actually, according to our experimental results, under the same configuration limit, using an optimized NSA after original NSA will have better results than that of using the optimized NSA directly. The reason is that using each optimized analysis directly generates more configurations than the original NSA.

We conducted all the experiments on a Server with 256GB memory and four 2.13GHz XEON CPUs.

Table 1: Typestates properties used in our experiments

Property Name	Description
FailSafeEnum	do not update a vector while iterating over it
FailSafeEnumHT	do not update a hash table while iterating over its elements or keys
FailSafeIter	do not update a collection while iterating over it
FailSafeIterMap	do not update a map while iterating over its keys or values
HasNextElem	always call hasMoreElements before calling nextElement on an Enumeration
HasNext	always call hasNext before calling next on an Iterator
Reader	do not use a Reader after its InputStream is closed
Writer	do not use a Writer after its OutputStream is closed

5.1 Experiment Results

To justify the effectiveness of our optimizations, we use original NSA as the baseline for our experiments. Table 2 shows the results of our optimizations, and the cases on which optimizations have no effect are not listed. The forth column (**Opt1**) shows the number of remained shadows after using the first optimization, *i.e.*, identifying changeless configurations generated from backward analysis. For 4 out of 23 combinations (17.4%), our optimized analysis can further identify removable shadows after the original NSA. In one case (**FailSafeIterMap + bloat**), the shadows removed by the optimized analysis are twice more than the shadows removed by the original NSA.

The fifth column (**Opt2**) of Tables 2 shows the number of the shadows that remain after using the optimization based on local object information. For 10 out of these 23 combinations (43.5%), the optimized NSA can further remove shadows after the original NSA. In two cases (**FailSafeEnum + fop** and **FailSafeIter + luindex**), the optimization can remove all the shadows that remain after the original NSA. Hence, the optimized NSA by local object information can give the static guarantee that the program satisfies the typestate property in each of these two cases. Furthermore, for 5 out of these 10 cases (50%), the optimized analysis can further remove more irrelevant shadows than the original one. Especially, in two cases (**FailSafeEnum + fop** and **FailSafeEnumHT + jython**) out of these 5, the original NSA cannot remove any shadow.

The last column of Tables 2 shows the results of the combination of two optimizations, *i.e.*, optimizing by local object information first and then by removing changeless configurations. For 13 out of these 23 combinations (56.5%), the combined optimized analysis can further identify "nop shadows" after the original NSA. In one case (**FailSafeIter + bloat**), the two optimizations both have positive effects and identify different "nop shadows" respectively. Interestingly, compared to perform these two optimizations after the original NSA individually, the combination can identify one more "nop shadow" in this case. The reason is: after the original NSA, the second optimization firstly removes some shadows from the instrumented program, so the first optimization generates less

Table 2: Results of the optimized NSA

Property + Program	BN	AN	Opt1	Opt2	Both
FailSafeEnum + fop	5	5	5	0	0
FailSafeEnum + jython	47	44	44	36	36
FailSafeEnumHT + jython	76	76	76	72	72
FailSafeIter + bloat	1010	916	911	905	899
FailSafeIter + chart	158	150	150	120	120
FailSafeIter + jython	119	115	115	105	105
FailSafeIter + luindex	30	15	15	0	0
FailSafeIter + pmd	305	290	290	262	262
FailSafeIterMap + bloat	481	479	476	479	476
FailSafeIterMap + jython	153	133	133	119	119
FailSafeIterMap + pmd	372	262	262	260	260
Writer + antlr	44	35	34	35	34
Writer + bloat	19	11	9	11	9

BN: The number of shadows that remain before the original NSA. **AN**: The number of shadows that remain after the original NSA. **Opt1**: The number of shadows that remain after the first optimization. **Opt2**: The number of shadows that remain after the second optimization. **Both**: The number of shadows that remain after the combination of two optimizations.

configurations and can identify one more "nop shadow" under the same configuration limit. In addition, there are three cases where local object optimization cannot remove any "nop shadow" but the other one can, which also justifies that the two optimizations complement each other.

5.2 Analysis Time

The analysis time of NSA is mainly dominated by the prior supporting analyses, such as constructing call graphs and computing points-to information. Because we evaluate each optimization by using NSA first and then the optimized NSA, the analysis time for evaluating each optimization is definitely longer than that of the original NSA. Table 3 displays the results of the analysis time of the cases on which our optimizations have effects.

From the experimental results, it can be justified that the time for NSA is just a small part of the total compilation time. In all cases but two, the total compilation time including our optimizations is under 10 minutes. The average analysis time of the original NSA is under 1 minute, though in some cases it needs a few more minutes, such as **FailSafeIter + bloat**. In some cases, the analysis time for the optimized analysis is less than that of the original NSA. One of the key reasons is that the optimized NSA analyzes less shadows. For the optimized NSA with the combination of two optimizations, the analysis time introduced by optimization is under 2 minutes in the majority of cases. Overall, our optimization methods do not cause a significant overhead on the weaving

Table 3: The results of analysis time (*in seconds*)

Property + Program	The 1st optimization			The 2nd optimization		
	NSA	**Opt**	**Total**	**NSA**	**Opt**	**Total**
FailSafeEnum + fop	0.98	0.1	251.14	1.21	0.39	276.87
FailSafeEnum + jython	8.06	0.2	243.55	8.23	1.87	269.35
FailSafeEnumHT + jython	10.30	0.69	240.91	10.41	8.19	271.05
FailSafeIter + bloat	298.05	133.94	806.02	288.50	516.82	1214.12
FailSafeIter + chart	25.35	1.64	305.92	24.15	66.08	393.04
FailSafeIter + jython	16.9	0.74	270.23	17.69	14.87	303.80
FailSafeIter + luindex	2.21	0.07	99.1	2.53	0.47	110.64
FailSafeIter + pmd	46.01	2.51	352.67	46.97	112.01	490.54
FailSafeIterMap + bloat	58.78	30.17	433.4	65.92	85.19	521.37
FailSafeIterMap + jython	49.67	17.61	276.74	58.38	56	337.73
FailSafeIterMap + pmd	77.67	4.25	420.05	77.72	96.84	541.66
Writer + antlr	12.95	0.83	223.76	13.7	9.06	253.48
Writer + bloat	1.56	0.24	128.66	1.68	0.34	140.08

NSA: The analysis time that original NSA consumes. **Opt**: The analysis time of the optimized NSA after the original NSA. **Total**: the total compilation time of the case.

process in experiments. Considering the total compilation time, the overhead incurred by our optimizations is acceptable.

5.3 Discussions

According to the experimental results, the first optimization only has effects in 17.4% cases, which is not very impressive. The reason is that the optimization works well on the methods containing several *interleaved* relevant method invocations on different objects. For example, for the program in Figure 5(a), which is slightly different from that in Figure 2 (the method calls on c_1 and c_2 are not interleaved), the original NSA can identify the shadow at line 9 as a "nop shadow". Hence, the capability of the optimized NSA is the same as that of the original one in this case.

The optimization based on local objects also has limitations. For example, it has no effect on the local objects created within loop statements. In Figure 5(b), we show a method m that extends the method in Figure 3 by adding a loop. Obviously, the method satisfies the typestate property in Figure 1, but the optimized NSA by using local object information cannot remove the shadows at lines 5 and 6. The reason is the forward analysis will propagate the configurations at the end of a "*for*" statement to the entry of the "*for*" statement, and the backward analysis will propagate configurations in the inverse direction too. Therefore, we can further optimize NSA based on the local objects created in loop statements, which will be the future work.

Finally, we should note that even if the original NSA is inter-procedural flow-sensitive, it could not remove the shadows in Figure 2 and Figure 3 either. Hence,

```
1  public static void main(String args[]) {
2      Connection c1 = new Connection(args[0]);
3      Connection c2 = new Connection(args[0]);
4      c1.close();
5      c1.write(args[1]);
6      c1.write(args[1]);
7      c2.close();
8      c2.write(args[1]);
9      c2.close();
10     c2.write(args[1]);
11 }
```

(a) A program similar to the program in figure 3

```
1  public void m(String args[]) {
2      for(int i = 0; i < 10; i++)
3      {
4          Connection c1 = new Connection(args[0]);
5          c1.write(args[1]);
6          c1.close();
7      }
8  }
```

(b) An example similar to the example in figure 4

Fig. 5: Examples on which optimizations have no effect

the main ideas of our optimizations can also be used in the inter-procedural flow-sensitive static analysis.

6 Related Work

Recently, typestate analysis of large-scale programs attracts much attention, and several static and dynamic typestate analysis methods are proposed and implemented. In [13], Fink *et al.* propose a context-sensitive, flow-sensitive and integrated static typestate verifier. The verifier utilizes a combined abstract domain of typestate and pointer abstractions to improve the precision of alias analysis. Their static analysis framework is designed to be a staged system to improve the scalability and efficiency. However, their approach cannot verify the typestate specifications of multiple objects. In [18], a hybrid typestate analysis is proposed and implemented to be context-sensitive and inter-procedural flow-sensitive. The static analysis in [18] is based on a lattice-based operational semantics, which supports to track individual objects along control-flow paths and compute typestate information and points-to information simultaneously. However, their approach suffers from unsoundness problem [7]. Besides those work, Rahul Purandare presents in [20] a cost model for runtime monitoring. The model explains key factors of monitoring overhead and the relationship among them. The cost model guides the optimization of runtime monitoring. Different from the hybrid method in this paper, the approach in [20] also tries to remove instrumentations at runtime [21]. Furthermore, their optimization can reduce the runtime overhead by reclaiming unnecessary monitors. Whereas, their hybrid approach may easily lead to unacceptable overhead at runtime, especially for the typestate properties involving multiple interacting objects. In addition, when unchecked exceptions happen, the method may produce unsound results.

7 Conclusion

In this paper, we present two optimization approaches for NSA to improve the precision. One optimization identifies changeless configurations during the back-

ward analysis; the other one use local object information to refine the forward analysis and backward analysis of NSA. According to the experiments on the DaCapo benchmark suite, in more than half of the studied cases, the optimized NSA can further remove unnecessary instrumentations, without a significant overhead. Additionally, we dissect the experimental results and the situations in which our optimizations have no effect. Furthermore, the main ideas of our optimizations can also be used in inter-procedural flow-sensitive static analysis.

Acknowledgement

This research is supported in part by grants from the National 973 project 2011CB302603, the National NSFC projects (Nos. 91118007 and 61103013), the National 863 projects (Nos. 2011AA010106 and 2012AA011201), the Specialized Research Fund for the Doctoral Program of Higher Education 20114307120015, and the Program for New Century Excellent Talents in University.

References

1. Clara. `http://www.bodden.de/clara/`
2. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., De Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to aspectj. ACM SIGPLAN Notices 40(10), 345–364 (2005)
3. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA). pp. 301–320. ACM, New York, NY, USA (2007)
4. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The dacapo benchmarks: Java benchmarking development and analysis. In: 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA). pp. 169–190. ACM, New York, NY, USA (2006)
5. Bodden, E.: J-lo-a tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen university (2005)
6. Bodden, E.: Verifying Finite-state Properties of Large-scale Programs. Ph.D. thesis, McGill University (2009)
7. Bodden, E.: Efficient hybrid typestate analysis by determining continuation-equivalent states. In: 32nd International Conference on Software Engineerin (ICSE). pp. 5–14. IEEE, ACM, New York, NY, USA (2010)
8. Bodden, E., Lam, P., Hendren, L.: Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In: 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). pp. 36–47. ACM, New York, NY, USA (2008)
9. Bodden, E., Lam, P., Hendren, L.: Object representatives: A uniform abstraction for pointer information. In: International Conference on Visions of Computer Science: BCS International Academic Conference (VoCS). pp. 391–405. British Computer Society, Swinton, UK, UK (2008)

10. Bodden, E., Lam, P., Hendren, L.: Clara: a framework for partially evaluating finite-state runtime monitors ahead of time. In: 1st International Conference on Runtime Verification (RV). pp. 183–197. Springer-Verlag, Berlin, Heidelberg (2010)
11. Chen, F., Roşu, G.: Mop: an efficient and generic runtime verification framework. In: 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA). pp. 569–588. ACM, New York, NY, USA (2007)
12. Dwyer, M.B., Purandare, R.: Residual dynamic typestate analysis exploiting static analysis: Results to reformulate and reduce the cost of dynamic analysis. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 124–133. ACM, New York, NY, USA (2007)
13. Fink, S., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. In: 15th International Symposium on Software Testing and Analysis (ISSTA). pp. 133–144. ACM, New York, NY, USA (2006)
14. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented Programming. Springer (1997)
15. Krüger, I.H., Lee, G., Meisinger, M.: Automating software architecture exploration with m2aspects. In: International Workshop on Scenarios and state machines: models, algorithms, and tools (SCESM). pp. 51–58. ACM, New York, NY, USA (2006)
16. Maoz, S., Harel, D.: From multi-modal scenarios to code: Compiling lscs into aspectj. In: 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT/FSE). pp. 219–230. ACM, New York, NY, USA (2006)
17. Masuhara, H., Kiczales, G., Dutchyn, C.: A compilation and optimization model for aspect-oriented programs. In: 12th International Conference on Compiler Construction (CC). pp. 46–60. Springer-Verlag, Berlin, Heidelberg (2003)
18. Naeem, N.A., Lhotak, O.: Typestate-like analysis of multiple interacting objects. In: 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA). pp. 347–366. ACM, New York, NY, USA (2008)
19. Pradel, M., Jaspan, C., Aldrich, J., Gross, T.R.: Statically checking api protocol conformance with mined multi-object specifications. In: 34th International Conference on Software Engineering (ICSE). IEEE Press, Piscataway, NJ, USA (2012)
20. Purandare, R.: Exploiting Program and Property Structure for Efficient Runtime Monitoring. Ph.D. thesis, University of Nebraska-Lincoln (2011)
21. Purandare, R., Dwyer, M.B., Elbaum, S.: Monitor optimization via stutter-equivalent loop transformation. In: 25th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA). pp. 270–285. ACM, New York, NY, USA (2010)
22. Slaughter, S.A., Harter, D.E., Krishnan, M.S.: Evaluating the cost of software quality. Communications of the ACM 41(8), 67–73 (1998)
23. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Transactions on Software Engineering 12(1), 157–171 (1986)