SAM: Self-adaptive Dynamic Analysis for Multithreaded Programs*

Qichang Chen[1], Liqiang Wang[1], and Zijiang Yang[2]

[1] Dept. of Computer Science, University of Wyoming, WY, USA.
{qchen2, wang}@cs.uwyo.edu,
[2] Dept. of Computer Science, Western Michigan University, MI, USA.
zijiang.yang@wmich.edu

Abstract. Many dynamic analysis techniques have been proposed to detect incorrect program behaviors resulted from faulty code. However, the huge overhead incurred by such dynamic analysis prevents thorough testing of large-scale software systems. In this paper, we propose a novel framework using compile-time and run-time optimizations on instrumentation and monitoring that aim to significantly reduce the overhead of dynamic analysis on multithreaded programs. We implemented a tool called SAM (Self-Adaptive Monitoring) that can selectively turn off excessive monitoring on repeated code region invocations if the current program context has been determined to be redundant, which may assist many existing dynamic detection tools to improve their performance. Specifically, we approximate the program context for a code region invocation as a set of variables, which include path-critical variables and shared variables accessed in that region. The path-critical variables are inferred using a use-definition dataflow analysis, and the shared variables are identified using a hybrid thread-based escape analysis. We have implemented the tool in Java and evaluated it on a set of real-world programs. Our experimental results show that it can significantly reduce the runtime overhead of the baseline atomicity violation and data race analyses by an average of 50% and 20%, respectively, while roughly keeping the accuracy of the underlying runtime detection tools.

1 Introduction

Dynamic analysis often suffers from the expensive runtime overhead which prevents it from scaling up to large-scale enterprise software systems. Despite its superior accuracy on error reporting compared to static analysis, the dynamic monitoring overhead has been an issue that prevents many runtime approaches from being adopted practically. According to our and other researchers' experiences [4, 8, 6, 19, 11, 1, 2], the runtime overhead is largely due to repetitive monitoring on code blocks' executions with the same or similar context. For the programs with intensive memory accesses, the problem is more severe. In our

* The work was supported in part by ONR N000140910740 and NSF CAREER 1054834.

experiment with the benchmark `tsp` [10], the executions produce as many as 20 million memory access events even under a small input dataset with only 3 threads. With the further investigation, we found that most of these events are generated from a certain number of code blocks and involve many objects created from the same class.

Faced to these issues, we are motivated to design a more efficient and scalable approach to speed up the widely-used dynamic analysis on multithreaded programs. In this paper, we propose a novel framework using compile-time and run-time instrumentation and monitoring optimizations that aim to significantly reduce the overhead of dynamic analysis on multithreaded programs. We implemented a tool called SAM (Self-Adaptive Monitoring) in Java, which can selectively turn off excessive monitoring on a repeated code region invocation if the program context of the current thread has been determined to be redundant. For any code region that has been monitored before, we do not monitor it again as long as the code region is executed with the same thread context previously visited, because this usually will not contribute additional information to dynamic analysis of multithreaded programs. An important observation is that for concurrency error detection, where the primary goal is to reveal the bugs resulted from incorrectly using synchronization and accessing shared variables, we are concerned about the accessing order of shared memory locations rather than their contents.

To demonstrate its effectiveness, we evaluated SAM over a set of multithreaded Java programs in conjunction with two baseline analysis tools, the Eraser for detecting data race [15] and the DAVE for detecting atomicity violation [18]. The experimental results show that this approach is more effective in reducing the subsequent analyses' performance overhead and better in preserving the baseline tools' accuracy than the prior approach [8].

This paper makes the following contributions:

- Our approach uses a refined program context approximation to check program state equivalence and avoid repeated monitoring, which results in more precise context approximation. The program context for a code region invocation is approximated as a set of path-critical variables and shared variables accessed in that region. We extend use-definition dataflow analysis to infer the path-critical variables, and design a hybrid thread-based escape analysis to identify the shared variables.
- SAM is specially designed for multithreaded programs by taking into account the synchronization information and shared variables when approximating the program contexts for each thread. Specifically, SAM considers the lockset held by the current thread.
- SAM can be easily integrated with dynamic analysis tools, which allows developers to focus on the dynamic analysis design rather than be distracted on tuning overhead and optimizing performance.

The rest of this paper is organized as follows. Section 2 reviews the literature and discusses how SAM differs from the related work. Section 3 intro-

duces the motivations and key insights in SAM, and describes our extended use-def dataflow analysis. Section 4 covers SAM's implementation details. Section 5 presents our experimental evaluation on SAM and demonstrate its performance improvement. Section 6 concludes this paper and discusses the future work.

2 Related Works

Different optimization techniques for dynamic analysis have been designed. Fei *et al.* [8] present a tool called Artemis, which is the most related work to our tool SAM. Artemis is a dynamic tool implemented in C and helps reduce the runtime overhead of the dynamic analysis tools. The code region analyzed by Artemis is based on the function level. Our current implementation of SAM follows this approach and also works on the method/function level. All prior observed contexts for a code region are recorded in a table. The currently observed context is compared with the previously preserved contexts to determine whether the monitoring on the function can be safely turned off. The context in Artemis at the entry point of each function invocation contains all global variables and function parameter variables. If a variable in the context is in primitive type, its value is checked when comparing contexts; if a variable in the context is in complex type (*e.g.*, pointer to a data structure), its type, instead of its value, is checked when comparing contexts. Note the context of Artemis is an approximation. While it reduces monitoring overhead, certainly, it will also cause the underlying tools to miss information, which further affects the accuracy of the baseline tools. In addition, Artemis does not consider synchronization operation and concurrent accessing, hence cannot handle multithreaded programs. Our tool SAM utilizes a more accurate context approximation approach and supports multithreaded programs.

Arnold *et al.* [1] design a similar framework that also duplicates code into two versions: original and instrumented, and inserts counter-based sampling code to allow statistically turning on/off the monitoring. SAM differs from it in that it focuses on multithreaded programs, and its context approximation is more precise.

Sampling is another popular technique to reduce the runtime overhead of dynamic analysis. This approach is suitable for the scenarios when multiple runs of the sampled program yield complementary information. However, it suffers from under-reporting problem hence may miss errors. Moreover, the need of multiple runs in the sampling-based monitoring further limits its applicability. Liblit *et al.* [11] use the sampling technique to reduce the frequency of code monitoring for long running programs. Hauswirth and Chilimbi [9] sample the code for possible memory leak error at a sampling rate that is inversely proportional to the frequency of code segment execution.

Many other runtime monitoring tools [16, 5] have resorted to static analysis to reduce the overhead of dynamic analysis. Yong *et al.* [19] proposes several techniques that rely on the results of static analysis to remove unnecessary instrumentation from the code, which in turns reduces the subsequent dynamic

analysis overhead. The techniques specifically designed to reduce the overhead of runtime type-checking can also be adopted for other similar dynamic analysis systems. Although static analysis can guide dynamic analysis to avoid monitoring some code executions, its effect on reducing overhead is usually limited and quite ad-hoc to applications.

There are also research work in the area of parallelizing runtime checking to reduce overhead. Patil *et al.* [14] suggest to use shadow process to check pointer and array accesses in C program. Oplinger *et al.* [13] spawn a speculative thread to execute the checking code. Although these techniques use parallelism to reduce the checking overhead, they also introduce additional communication overhead that is usually not negligible.

3 The Design of SAM

3.1 An Overview

The insight for the overhead of dynamic analysis is that monitoring and analyzing many repeated events inevitably slows down the program's execution. To avoid repeated monitoring, SAM relies on checking program context to direct the baseline tool to avoid monitoring repeated events. As we mentioned before, the execution of the benchmark `tsp` contains as many as 20 million access events on shared variables. Without optimization, such huge number of events will prevent any dynamic analysis from finishing within reasonable time.

The tool Artemis [8] is a step toward this goal. It adopts a method level context checking scheme. Artemis keeps track of the method contexts prior to the entrance of every method. However, besides its inaccurate context, Artemis is targeted for serial code and cannot handle multithreaded programs. For example, Artemis does not consider any synchronization state or the currently held locks when computing the context, which leads to under-approximation of the context. In contrast, our tool SAM is designed to assist the dynamic tools to analyze multithreaded programs. It considers the current synchronization state when computing the context and takes into account the path-critical variables and shared (escaped) objects accessed in the current method, while Artemis considers all global variables for the context. For example, if a method $f()$ is invoked by a thread that holds the locks l_1 and l_2, then the two acquired locks l_1 and l_2 will be included in the context.

Let m be a method, Θ_m be the program context prior to the execution of m. The context Θ_m consists of the following three kinds of variables:

$$\Theta_m : \begin{cases} R_m : \text{all references to shared (escaped) objects that that are accessed in} \\ \quad m\text{, including “}\texttt{this}\text{” and locking objects accessed in } m. \\ PCCV_m : \begin{cases} \text{If } R_m = \emptyset,\ PCCV_m \text{ is } \emptyset. \\ \text{If } R_m \neq \emptyset,\ PCCV_m \text{ is a set of path-critical context variables,} \\ \quad \textit{i.e.}\text{, the variables that are not defined inside } m \text{ (}\textit{e.g.}\text{,} \\ \quad \text{method parameters, fields of escaped objects) and could} \\ \quad \text{directly or indirectly affect the execution path of } m. \end{cases} \\ O_m^{lck} : \begin{cases} \text{If } R_m = \emptyset,\ O_m^{lck} \text{ is } \emptyset. \\ \text{If } R_m \neq \emptyset,\ O_m^{lck} \text{ is all locking objects held at the entrance of } m. \end{cases} \end{cases}$$

Thus, Θ_m is represented by $\langle R_m, PCCV_m, O_m^{lck} \rangle$. A variable v in Θ_m is denoted by $\langle v, val(v) \rangle$. If v is an object reference, $val(v)$ is its hash code obtained in runtime. If v is in a primitive type, $val(v)$ is its runtime value.

Figure 1 illustrates how the context check works and the difference between our tool SAM and Artemis. The code in the `then` branch of the `if` statement is the original code. The deposited value is added to account `a1` then `allBalance` is updated. In this example, Artemis's context contains `a1`, `a` (method arguments), and `allBalance` (global variable), whereas SAM's context contains only `a1` (an object accessed in the method) and `allBalance` (global variable), because `a` is not a path-critical variable, and `allBalance` is a global variable that is shared among multiple threads. Here, Artemis does not consider the address of object `a1` but only its class type. The approximation of the object type would cause the underlying baseline tool to miss some important information if the method `deposit` is invoked twice with two distinct objects of `Account`, as the second invocation would be deemed by Artemis as redundant and thus not monitored by the underlying error detection tool. Therefore, the underlying baseline tool's accuracy suffers in this case.

```
public class Account{
   public static int allBalance = 0;
   private int bal = 0;
   void deposit(Account a1, int a){
    if(artemisCheck(a1, a, allBalance)){
       // original version
       a1.bal += a;
       allBalance += a;
    }{
       // instrumented version
       ...
    }
   }
}
```

```
public class Account{
   public static int allBalance = 0;
   private int bal = 0;
   void deposit(Account a1, int a){
    if(samCheck(a1)){
       // original version
       a1.bal += a;
       allBalance += a;
    }{
       // instrumented version
       ...
    }
   }
}
```

Fig. 1. Code snippets that illustrate the context check in Artemis and SAM.

There are trade-offs between accuracy and efficiency for the approach of SAM. SAM is subject to the thread scheduling nondeterminism that might cause shared variables or path-critical context variables to be changed during the method invocation, which may invalidate the initial context checking. Intuitively, such scenarios happen very rarely. In addition, SAM's context does not include the newly created objects within the current method. Usually, the escaping of these objects, if happens, will occur in the following invocations of other methods, which will be analyzed when these methods are invoked. Certainly, an object may escape within the method where it is created (*e.g.*, is assigned to a static field). However, such possibility is rare. In our experiment shown in Section 5, all these scenarios affecting inaccuracy did not appear. Note that each thread has its own context profiles in SAM does not include any information from other threads to compute the method context for current thread because they will not affect monitoring events except for the two cases mentioned before.

Algorithm 1 shows the algorithm of SAM's context check. We conduct an intraprocedural analysis to generate a symbolic context for each method. The symbolic context is similar to the runtime context, except that the symbolic context contains all object references accessed in the method, and the values of the context variables are null. In the runtime context, all references to unescaped objects are removed, and the context variables are updated by the corresponding runtime values. At the same time, each method is expanded with two versions of the code: one version consists of the original code, and the other version is the instrumented code by the baseline tool. A context check is inserted at the beginning of each method. When a method is invoked, we first call "RemoveUnescapedObjects (C_m)" to get rid of all non-escaped objects in the symbolic context C_m. Then "RuntimeValueUpdate(C'_m, P_m)" is called to update all symbolic names with their runtime values. Then we check whether the current context has ever been contained in the context table CT_m that stores all previous visited contexts. If the current context Θ_m has been encountered before, we call the uninstrumented code directly; otherwise, we call the instrumented code and save Θ_m into CT_m.

3.2 Context Checking for Multithreaded Programs

Given a method m, a variable v is a *branching variable* if v is included in a branching expression of m(*e.g.*, the conditional expression in `if/for/while/switch` statements). A variable u is a *path-critical context variable* ($PCCV$) if u is not defined inside m (*e.g.*, method parameters or fields of escaped objects) could directly or indirectly affect the value of a branching variable.

To infer $PCCV$ according to branching variables, we use an extended use-def dataflow analysis. Use-def analysis [12] identifies and tracks a variable's definition and usage sites inside a function. The DU (*i.e.*, def-use) and UD (*i.e.*, use-def) chains are a concise representation of the dataflow information about a given variable. The DU chain of a variable starts from the definition site of the variable and connects it to all the variable use sites where the defined variable can flow

Input:
C_m: the symbolic context for method m generated from static analysis;
P_m: the program runtime state at the entrance of method m;
CT_m: the runtime context table storing previously visited contexts for m;

SAM-ContextCheck(C_m, P_m, CT_m){
C'_m = RemoveUnescapedObjects(C_m);
Θ_m = RuntimeValueUpdate(C'_m, P_m);
for *each* $ct_m \in CT_m$ **do**
    if $\Theta_m == ct_m$ **then**
        execute the uninstrumented version of code of m;
        return;
    end
    continue;
end
$CT_m = CT_m \cup \{\Theta_m\}$;
execute the instrumented version of code of m;
}

Algorithm 1: The SAM context check algorithm.

to. In contrast, the UD chain for a variable connects a variable's use site to all its definition sites.

Algorithm 2 illustrates how to compute the set of $PCCV$. We iteratively apply a use-def intraprocedural dataflow analysis (*i.e.*, UD chain) to identify and track the path-critical context variables that indirectly affect the program's execution path. Specifically, given a branching variable's use site, we track its definition statements. For any local variable on the assignment statement's right-hand-side (RHS), we continuously track its definition recursively. This search is repeated till we identify all non locally defined variables whose values directly or indirectly affect the branching variables. These non-local variables, which may be object references, fields, or parameter variables, are $PCCV$ for the given branching variables.

4 Implementation and Optimization

We use the Eclipse JDT [7] to instrument program source code. To simplify the instrumentation, SAM first duplicates each method in the original program source into two methods with different suffix names. The method with the suffix name "`_original`" represents the original method that will not be instrumented by the baseline analysis tool. The method with the suffix "`_instrumented`" is the method that will be instrumented by the baseline tool. This allows the baseline tool to easily instrument program source code without resorting to complex tagging or structure identification mechanism. At each method entrance, SAM inserts an `if` statement for the context check and places the function calls to the original or the instrumented methods into the `then` and `else` branches,

```
Input: AST_M: the abstract syntax tree of a method M.
Output: PCCV: the set of symbolic path-critical context variables.

ComputePCCV(M){
BV = ∅; // the set of branching variables.
PCCV = ∅;
for each statement S in AST_M do
    if S contains branching expression, say expr then
        for each variable v in expr do
            BV = BV ∪ {v};
        end
    end
end
for each v ∈ BV do
    ComputeUseDef(v);
end
}

ComputeUseDef(v) {
compute the definition sites DS_v of v;
for each definition site ds_v ∈ DS_v do
    if ds_v is on a field of locally created object then
        continue;
    end
    if ds_v is out of the scope of M then
        PCCV = PCCV ∪ {v};
    end
    Let RHS_V be the list of variables on the RHS of ds_v;
    for each variable rhs_v in RHS_V do
        ComputeUseDef(rhs_v);
    end
end
}
```

Algorithm 2: The algorithm for computing path-critical context variables.

respectively. If the context has been observed before, the original code is chosen. Otherwise, the instrumented version will be activated in the execution.

Escape analysis plays an important role in our context checking. Thread escape analysis is to determine whether and when a variable becomes shared by multiple threads. We utilize our thread-based escape analysis to identify escaped objects, which is based on our previous work [3]. When an object o is created, o is owned by its creating thread. Object o is said to be *thread-escaped* or *shared* when it becomes accessible by two or more threads. When an object o becomes accessible by multiple threads, its fields are vulnerable to concurrent accessing, hence may result in concurrency errors such as data races and atomicity violations. Thus, it is important to know whether (even when) an object escapes from its creating thread during the program's execution. During the program's execution, the dynamic thread escape analysis is complemented and refined with

the thread escape information from the context-sensitive flow-insensitive interprocedural static analysis for each unexecuted code block to produce the final hybrid thread escape analysis results. The thread-based escape analysis results are used to identify the shared objects.

To further reduce the context checking overhead incurred by SAM, we design the following optimization technique. To avoid maintaining a huge context table and reduce the memory usage in SAM, we insert an array field to store the contexts in different threads for each method in the class, which can be indexed by the thread ID in runtime. These context tables are initialized with empty content. When a context check is encountered in runtime, the accessing thread will use its thread ID as index to retrieve the current context table and compare the newly computed context against the ones saved in current table. This approach avoids maintaining a multi-level hierarchical context table for each thread and reduces the lookup frequency and overhead.

5 Experiments

We present three sets of experimental results over the benchmarks including `Elevator`, `Tsp`, `Sor`, and `Hedc` from [17], and `Vector`, `Stack`, and `Hashtable` from JDK 1.6. For the `Elevator` benchmark, we tested it with 2 threads using the provided input data and instrumented a timer that forces the program to terminate after a wall-clock time of 10 seconds. For the benchmark `Tsp`, we tested it with 3 threads using the provided input datasets `map13`, `map14`, and `map15`. For the benchmark `Sor`, we tested it with 2 threads and 50 iterations. For the benchmarks `Vector, Stack`, and `HashTable`, we tested them using 2 threads to concurrently insert, update, and remove elements.

The first experiment measures the performance overhead and effectiveness of SAM's context checking without the baseline monitoring tool. The second and third experiments aim to illustrate the SAM's performance improvement and accuracy preservation on the dynamic analysis. Specifically, we evaluate SAM using our dynamic atomicity violation detector DAVE [18] and the Eraser race detector [15].

To compare SAM with Artemis, we also implement a revised version of Artemis [8] that can work for multithreaded Java programs context checking. Specifically, in the enhanced Artemis_C (C stands for Concurrent), each thread rather than the whole program maintains its own context profiles and the context of a method consists of the method parameters and global variables (static fields) that are accessed in the method.

The experiments are performed on a machine with 1.6 GHz Intel Core Duo dual-core CPU with 4 GB memory, Windows XP SP3, and Sun JDK 1.6.

5.1 Artemis_C and SAM's Context Checking Overhead

To measure the context check overhead and effectiveness of Artemis_C and SAM, we evaluate Artemis_C and SAM over the aforementioned benchmarks without

the baseline monitoring instrumentation. The experimental results are shown in Figure 2 and Figure 3. The runtime overhead introduced by the context check in SAM itself is around 270% on average and thus is not significant compared to the huge overhead (typically in the order of 10x or more) incurred by the baseline tool for the memory-intensive benchmarks. In addition, SAM filters 67% of all observed contexts in the benchmarks, which is very effective in filtering out the redundant monitoring for reducing the subsequent baseline tool's monitoring overhead. The runtime overhead introduced by the context check in Artemis_C is about 370% and 72% of the observed contexts are deemed redundant by Artemis_C. Although Artemis_C has a higher context filtering rate than SAM, it does not preserve baseline tool's accuracy as well as SAM, which is discussed in the following sections.

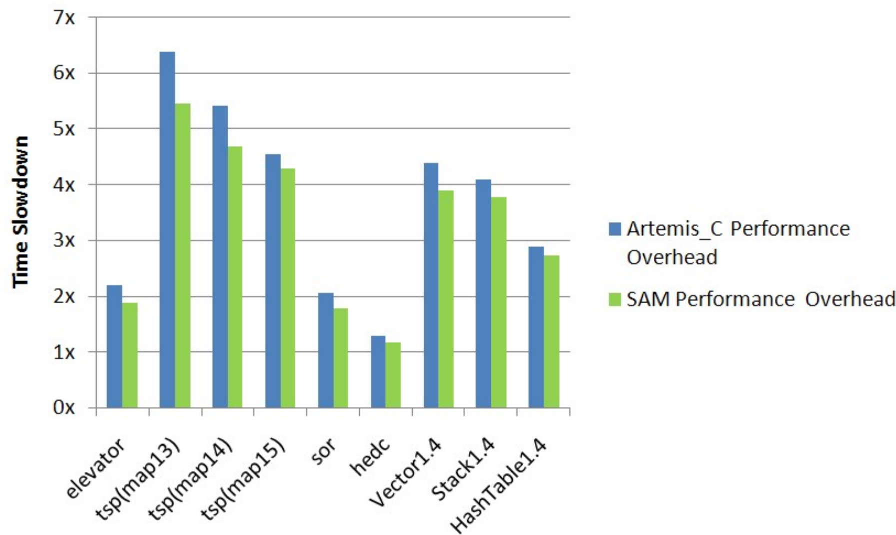

Fig. 2. The performance overheads of Artemis_C and SAM's context checking.

5.2 SAM + DAVE vs. Artemis_C + DAVE

In the second experiment, we evaluate SAM with our atomicity violation detector DAVE. DAVE is a dynamic analysis [18] that detects atomicity violations in multi-threaded Java programs.

The experimental results are shown in Figures 4 and 5. As we can see from Figure 5, our baseline tool DAVE with SAM has experienced an average 43% performance improvement, whereas the performance improvement of Artemis_C over DAVE is about 20%.

Program	LOC	threads	number of redundant contexts (Artemis_C)	number of total context checks performed (Artemis_C)	effectiveness	number of redundant contexts (SAM)	number of total context checks performed (SAM)	effectiveness
elevator	339	3	9513	9829	96.79%	9615	9723	98.89%
tsp(map13)	519	3	151689	151875	99.88%	151363	151875	99.66%
tsp(map14)	519	3	226687	227097	99.82%	210643	211869	99.42%
tsp(map15)	519	3	145664	146035	99.75%	144025	145028	99.31%
sor	8253	3	396	402	98.51%	198	402	49.25%
hedc	4267	3	9903	9921	99.82%	9898	9921	99.77%
Vector1.4	383	2	4	27	14.81%	4	27	14.81%
Stack1.4	418	2	0	33	0.00%	0	33	0.00%
HashTable1.4	597	2	4	11	36.36%	4	11	36.36%

Fig. 3. The effectiveness of Artemis_C and SAM's context checking.

In addition, we can see from Figure 4 that SAM outperforms Artemis_C in preserving the baseline tool DAVE's atomicity violation coverage in the benchmarks `elevator, tsp(map13), tsp(map14)` and `hedc`. Note that the atomicity violation coverage is not the number of atomicity violation errors revealed by the tool in the source program. For example, a program might have only 2 locations that are involved in an atomicity violation which may occur repeatedly for 1,000 times in the execution. If the dynamic analysis observes the repeated 900 atomicity violations occurred in the execution, we say it has a 90% violation coverage. If the dynamic analysis identifies all the two locations in the source code that are involved in the atomicity violations, it has no accuracy loss. Figure 8 compares the DAVE's accuracy loss when using Artemis_C and SAM, respectively. It can be seen that SAM keeps the accuracy on all benchmarks except for `Tsp(map14)`, whereas Artemis_C loses accuracy on both `Elevator` and `Tsp(map14)`. In addition, SAM also outperforms Artemis_C on accuracy on `Elevator`. One of the main reasons incurs loss of accuracy for SAM and Artemis_C is that they failed to feed some non-redundant events to the baseline analysis tools due to the context appromixation and filtering. Certainly, in order to improve accuracy, we can incorporate more information when computing the context, but it will lead to a much higher context checking overhead which might offset the performance benefits brought about by turning off the repeated monitoring.

5.3 SAM + Eraser vs. Artemis_C + Eraser

The Eraser [15] is a dynamic analysis tool for detecting data races. Eraser checks race conditions based on a simple locking policy, *i.e.*, all accesses to a shared variable should be protected by a common lock. To simplify experimental setup, our Eraser implementation does not classify the benign data races and false positives from the real data race bugs.

Figures 6 and 7 show the violation coverage and performance slowdown of Eraser using Artemis_C and SAM, respectively. As we can see, Eraser+SAM has about 93% violation coverage on average while Eraser+Artemis_C has only about

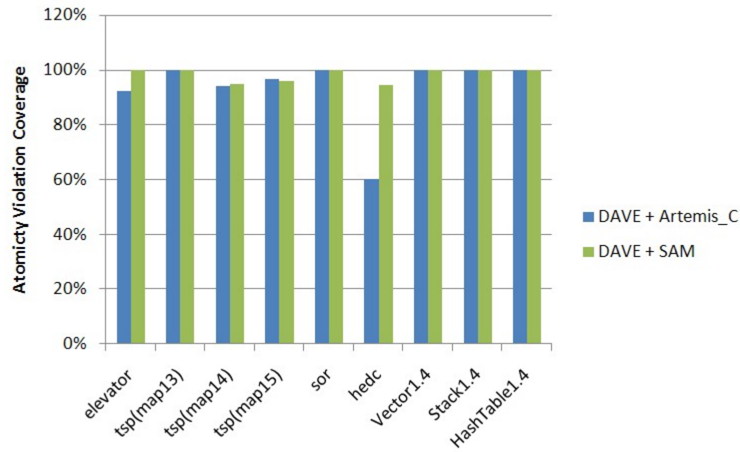

Fig. 4. DAVE's atomicity violation coverage when it is integrated with Artemis and SAM.

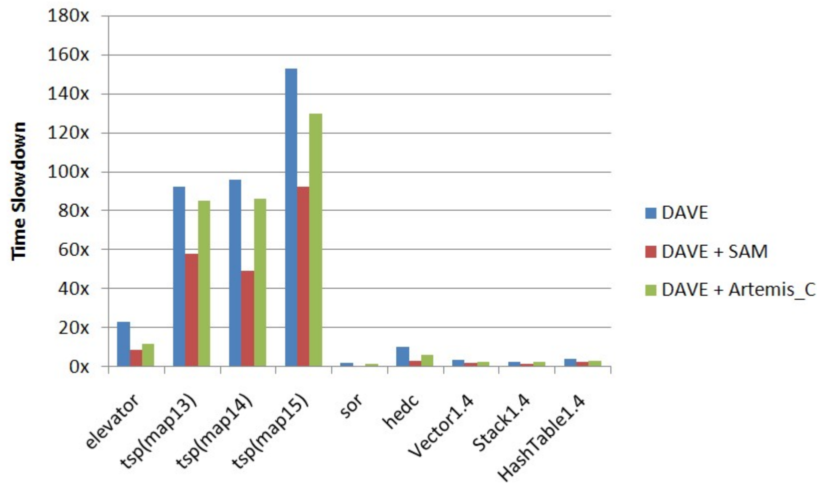

Fig. 5. DAVE's monitoring overhead using Artemis and SAM.

66%. In addition, Eraser+SAM has better violation coverage than Eraser+Artemis_C (90% violation coverage) on most benchmarks except for `sor`, `Vector`, `Stack`, and `Hashtable`, on which the baseline tool Eraser does not report any data race warnings. However, SAM and Artemis_C does not reduce the Eraser's monitoring time by a large portion, which is due to the simple on-the-fly analysis algorithm used in Eraser. As shown in Figure 7, Eraser+SAM has less per-

formance slowdown than Eraser+Artemis_C on most benchmarks except for `tsp(map13), hedc` and `Vector 1.4`. Figure 8 compares the Eraser's data race accuracy when using Artemis_C and SAM, respectively. It can be seen that SAM also outperforms Artemis_C in preserving the Eraser's accuracy in the benchmarks `elevator, tsp(map13), tsp(map14)` and `hedc` with only an average accuracy loss of 5%.

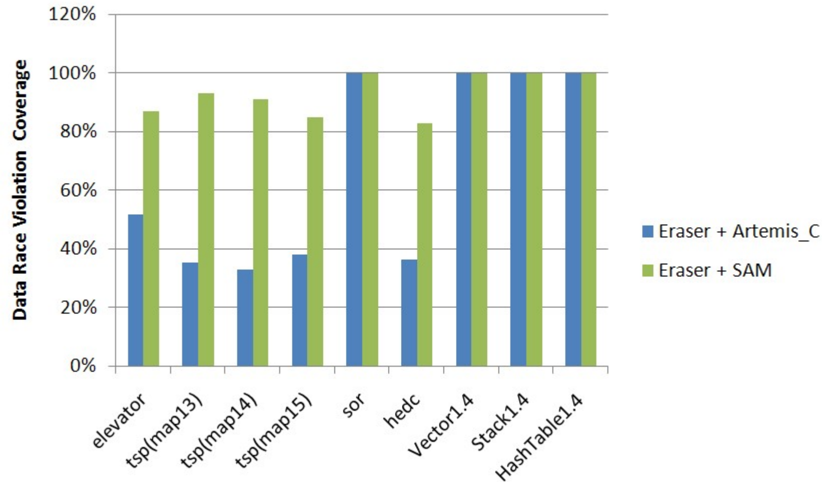

Fig. 6. Eraser's violation coverage when it is integrated with Artemis and SAM.

6 Conclusions and Future Work

In this paper, we propose a self-adaptive monitoring scheme for reducing the runtime overhead of dynamic program analysis. Our experiments show that this approach significantly reduces the overhead of the baseline dynamic analysis tools with slight accuracy loss. It can be utilized by the general dynamic analyses to improve their runtime performance, reduce the analysis turnaround time, and scale up to large memory-intensive programs.

Our future work includes integrating it and evaluating its effectiveness with other dynamic analysis tools, establishing context checking's cost model, and investigating more fine-grained block-level context checking scheme.

References

1. M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. *SIGPLAN Not.*, 36(5):168–179, 2001.

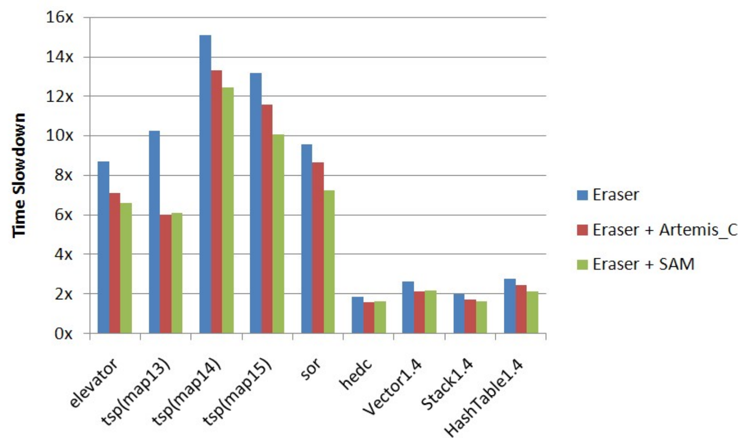

Fig. 7. Eraser's monitoring overhead using Artemis and SAM.

2. S. Callanan. Flexible debugging with controllable overhead. In *Ph.D. Dissertation, Stony Brook University*, 2009.
3. Q. Chen, L. Wang, and Z. Yang. HEAT: A Combined Static and Dynamic Approach for Escape Analysis. In *33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC2009)*, Seattle, USA, July 2009. IEEE Press.
4. Q. Chen, L. Wang, Z. Yang, and S. Stoller. HAVE: Detecting Atomicity Violations via Integrated Dynamic and Static Analysis. In *International Conference on Fundamental Approaches to Software Engineering (FASE), European Joint Conferences on Theory and Practice of Software (ETAPS)*, York, UK, March 2009. Springer-Verlag.
5. J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 258–269, New York, NY, USA, 2002. ACM.
6. M. B. Dwyer, A. Kinneer, and S. Elbaum. Adaptive online program analysis. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 220–229, Washington, DC, USA, 2007. IEEE Computer Society.
7. Eclipse. Available from http://www.eclipse.org/.
8. L. Fei and S. P. Midkiff. Artemis: practical runtime monitoring of applications for execution anomalies. *SIGPLAN Not.*, 41(6):84–95, 2006.
9. M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. *SIGPLAN Not.*, 39(11):156–164, 2004.
10. Java Grande Forum. Java Grande Multi-threaded Benchmark Suite. version 1.0. Available from http://www.javagrande.org/.
11. B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. *SIGPLAN Not.*, 38(5):141–154, 2003.
12. S. S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

Program	Base(s)	Dummy(s)	DAVE		DAVE with SAM		DAVE with Artemis_C		Eraser		Eraser with SAM		Eraser with Artemis_C	
			time(s)	nAV	time(s)	nAV	time(s)	nAV	time(s)	nDR	time(s)	nDR	time(s)	nDR
elevator	0.25	0.95	5.73	18	2.16	18	2.96	16	2.18	43	1.65	41	1.78	3
tsp(map13)	0.33	4.63	30.57	0	19.18	0	28.13	0	3.38	11	2.02	11	1.98	5
tsp(map14)	0.43	5.78	41.46	14	21.13	12	37.13	12	6.48	12	5.35	10	5.72	5
tsp(map15)	0.41	5.62	52.12	26	31.42	26	44.21	26	4.48	13	3.43	11	3.94	7
sor	0.6	0.92	2.20	0	0.72	0	1.54	0	10.35	0	7.83	0	9.37	0
hedc	1.64	3.65	8.50	7	2.79	7	5.21	7	1.57	27	1.37	25	1.34	15
Vector1.4	0.1	0.2	0.80	6	0.4	6	0.61	6	0.58	0	0.48	0	0.47	0
Stack1.4	0.1	0.2	0.82	6	0.51	6	0.72	6	0.6	0	0.49	0	0.52	0
HashTable1.4	0.2	0.3	0.92	2	0.55	2	0.71	2	0.61	0	0.47	0	0.54	0

Fig. 8. Comparison of the performance and accuracy of the DAVE, DAVE+SAM, DAVE+Artemis_C, Eraser, Eraser+SAM, and Eraser+Artemis_C. "Base" is the original program's running time without instrumentation. "Dummy" is the instrumented program's running time without any analysis (intercepting events only). The column "nAV" denotes the number of atomicity violations, which are counted based on the places in source code involved in atomicity violations. The column "nDR" denotes the number of data races, which are counted based on the source code locations involved in data races. All execution times are measured in seconds.

13. J. Oplinger and M. S. Lam. Enhancing software reliability with speculative threads. *SIGARCH Comput. Archit. News*, 30:184–196, October 2002.
14. H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *Softw. Pract. Exper.*, 27:87–110, January 1997.
15. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.
16. S. Vakkalanka, G. Szubzda, A. Vo, G. Gopalakrishnan, R. Kirby, and R. Thakur. Static-analysis assisted dynamic verification of mpi waitany programs (poster abstract). In M. Ropo, J. Westerholm, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5759 of *Lecture Notes in Computer Science*, pages 329–330. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-03770-2_43.
17. C. von Praun and T. R. Gross. Object race detection. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 70–82. ACM Press, Oct. 2001.
18. L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proc. ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, March 2006.
19. S. H. Yong and S. Horwitz. Using static analysis to reduce dynamic analysis overhead. *Form. Methods Syst. Des.*, 27:313–334, November 2005.