Collaborative runtime verification with tracematches

Eric Bodden[1], Laurie Hendren[1], Patrick Lam[1],
Ondřej Lhoták[2] and Nomair A. Naeem[2]

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

Abstract. Perfect pre-deployment test coverage is notoriously difficult to achieve for large applications. With enough end users, many more test cases will be encountered during an application's deployment than during testing. The use of runtime verification after deployment would enable developers to detect and report on unexpected situations. Unfortunately, the prohibitive performance cost of runtime monitors prevents their use in deployed code.
In this work we study the feasibility of collaborative runtime verification, a verification approach which distributes the burden of runtime verification onto multiple users. Each user executes a partially instrumented program and therefore suffers only a fraction of the instrumentation overhead.
We focus on runtime verification using tracematches. Tracematches are a specification formalism that allows users to specify runtime verification properties via regular expressions with free variables over the dynamic execution trace. We propose two techniques for soundly partitioning the instrumentation required for tracematches: spatial partitioning, where different copies of a program monitor different program points for violations, and temporal partitioning, where monitoring is switched on and off over time. We evaluate the relative impact of partitioning on a user's runtime overhead by applying each partitioning technique to a collection of benchmarks that would otherwise incur significant instrumentation overhead.
Our results show that spatial partitioning almost completely eliminates runtime overhead (for any particular benchmark copy) on many of our test cases, and that temporal partitioning scales well and provides runtime verification on a "pay as you go" basis.

1 Introduction

In the verification community it is now widely accepted that, especially for large programs, verification is often incomplete and hence bugs still arise in deployed code on the machines of end users. However, verification code is rarely deployed, due to large performance penalties induced by current runtime verification approaches. Consequently, when errors do arise in production environments, their causes are often hard to diagnose: the available debugging information is very limited.

Tracematches [1] are one mechanism for specifying runtime monitors. Tracematches enable developers to state sequences of program events and actions to take if the execution matches the sequence. Events bind objects in the heap; a tracematch only triggers if all of the events occur on a consistent set of objects.

According to researchers in industry [13], larger industrial companies would likely be willing to accept runtime verification in deployed code if the overhead is below 5%. In previous work on tracematches, we have shown that, in many cases, static analysis can enable efficient runtime monitoring by improving both the specification [3] and program under test [6]. Most often, our techniques can reduce runtime overhead to under 10%. However, our evaluation also showed that unreasonably large overheads—sometimes more than 100%—remained for some classes of specifications and programs. Other techniques for runtime monitoring also incur similar runtime overheads; for instance, the Program Query Language [10] causes up to 37% overhead on its benchmark applications (although it is intended to be a debugging tool rather than a tool for monitoring deployed programs), and JavaMOP [7] incurs up to 13% overhead on non-pathological test cases for runtime monitoring.

In this work, we attack the problem of runtime verification-induced overhead by using methods from remote sampling [9]. Because companies which produce large pieces of software (which are usually hard to analyze) often have access to a large user base, one can generate different kinds of partial instrumentation ("probes") for each user. A centralized server can then combine runtime verification results from runs with different probes. Although there are many advantages to a sampling-based approach, we are interested in using sampling to reduce instrumentation overhead for individual end users. We have developed two approaches for partitioning the overhead, *spatial partitioning* and *temporal partitioning*.

Spatial partitioning works by partitioning the instrumentation points into different subsets. We call each subset of instrumentation points a *probe* and each user is given a program instrumented with only one probe. This works very well in many cases, but in some cases a probe may contain a very hot—that is, expensive—instrumentation point. In those cases, the unlucky user who gets the hot probe will experience most of the overhead.

Temporal partitioning works by turning the instrumentation on and off periodically, reducing the total overhead. This method works even if there are are very hot probes, because even those probes are only enabled some of the time. However, since probes are disabled some of the time, any runtime verification properties of interest may be ignored while the probes are disabled.

In both spatial and temporal partitioning, the remaining instrumentation must operate correctly and, in particular, must never report false positives. The key point is that our transformations must never remove instrumentation points that can remove candidate bindings; identifying such instrumentation points can be difficult for tracematches, which may bind one or more objects and require each event to match the same objects. We have found a simple mechanism for reducing the number of these instrumentation points that appears to work well on our benchmarks.

We explored the feasibility of our approach by applying our modified tracematch compiler to benchmarks whose overheads persisted after the static analysis in [6]. We first experimented with spatial partitioning. We found that some benchmarks were very suited to spatial partitioning. In these cases, each probe produced lower overhead than the complete instrumentation, and many probes carried less than 5% overhead. However, in other cases, some probes were so hot that they accounted for almost all of the overhead; spatial partitioning did not help much in those cases. We also experimented with temporal partitioning and examined runtimes when probes were enabled for 10, 30, 50, 70, 90 and 100 percent of the time. As expected, we found that the overhead increased steadily with the proportion of time that the probes were enabled, so that one can gain limited runtime monitoring by running probes only some of the time.

The remainder of this paper is structured as follows. In Section 2, we give background information on tracematches and describe the instrumentation for evaluating tracematches at runtime. In Section 3, we explain the spatial and temporal partitioning schemes. We evaluate our work in Section 4, discuss related work in Section 5 and finally conclude in Section 6.

2 Background

The goal of our research is to monitor executions of programs and ensure that programs never execute pathological sequences of events. In this project, we monitor executions using tracematches. A tracematch defines a runtime monitor using a regular expression over an alphabet of user-defined events in program executions. The developer is responsible for providing a tracematch to be verified and definitions for each event, or symbol, used in the tracematch. He provides definitions for symbols using AspectJ [8] pointcuts. Pointcuts often specify patterns which match names of currently executing methods or types of currently executing objects. Pointcuts may also bind parts of the execution context. For instance, at a method-call pointcut, the developer may bind the method parameters, the caller object, and the callee objects, and may refer to these objects when the tracematch matches. If a tracematch does not bind any variables, then it reduces to verifying finite-state properties of the program as a whole.

```
1 tracematch(Iterator i) {
2     sym next before:
3       call(* java.util.Iterator+.next()) && target(i);
4     sym hasNext before:
5       call(* java.util.Iterator+.hasNext()) && target(i);
6
7     next next { /* emit error message; may access variable i */ }
8 }
```

Figure 1. Tracematch checking that hasNext() is always called before next()

Figure 1 presents an example tracematch. The tracematch header, in line 1, declares a tracematch variable `i`. Lines 2–5 declare two symbols, `next` and `hasNext`, which establish the alphabet for this tracematch's regular expression. The `next` symbol matches calls to an `Iterator`'s `next()` method and binds the target object of the method call to `i`. The `hasNext` symbol matches calls to `Iterator.hasNext()`, on the same iterator `i`. Line 7 declares the tracematch's pattern (regular expression) and body. The pattern, `next next`, states that the tracematch body must execute after two consecutive calls to `next()`, as long as no `hasNext()` call intervenes.

A crucial point about the semantics of tracematches' regular expressions is that intermediate events matching an explicitly-declared symbol *cannot be ignored*; that is, any occurrence of a non-matching symbol in an execution invalidates related partial matches. In our example, a sequence `next hasNext next` (all on the same iterator, of course) would *not* match. (Avgustinov et al. discuss the semantics of tracematches in detail in [2].)

The implementation of tracematches uses finite state machines to track the states of active partial matches. The compiler tracks variable-to-object bindings with *constraints*; each state q in the finite state machine has an associated constraint that stores information about groups of bound heap objects that must or must not be in state q. Constraints are stored in Disjunctive Normal Form as a set of *disjuncts*. Each disjunct maps from tracematch variables to objects. Note that the runtime cost of this approach comes from the large number of simultaneously-bound heap objects, and that the number of tracematch variables does not contribute to the runtime cost.

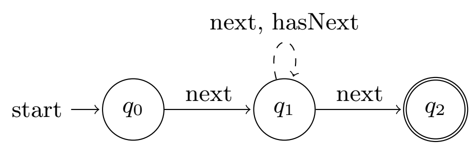


Figure 2. Finite state machine for the tracematch of Figure 1

Figure 2 presents the automaton for the `HasNext` pattern; we can observe that two calls to `next` (on the same `i`) will cause the automaton to hit its final state q_2. Note that state q_1 carries a dashed self-loop. We call this loop a *skip-loop*. Skip loops remove partial matches that cannot be extended to complete matches: they delete a partial match whenever an observed event invalidates that partial match.

As an example, assume that state q_1 is associated with the constraint $\{[i \mapsto i_1], [i \mapsto i_2]\}$; that is, the program has executed `next()` once, and only once, on each of the iterators i_1 and i_2, following the most recent call to `hasNext()` on each of i_1 and i_2. If the program then executes `hasNext()` on i_2, then another call to `next()` on i_2 can no longer trigger an immediate match. Hence the skip-loop labelled `hasNext` will reduce the constraint on the intermediate state q_1 to $\{[i \mapsto i_1]\}$; the implementation discards the disjunct for i_2 at q_1. (In the tracematch semantics, the skip-loop implements a conjunction of the constraint at q_1 with the binding $i \neq i_2$.)

The tracematch compiler weaves code to monitor tracematches into programs at appropriate event locations. For every static code location corresponding to a potential event execution, the compiler therefore includes instrumentation code that also updates the appropriate disjuncts. This instrumentation code is called a *shadow*. In this paper, we use a previously-published static analysis that removes shadows if they can be shown to never contribute to complete matches [6]; for instance, a program which calls `hasNext()` but never `next()` would never trigger the final state of the `HasNext` automaton, so the `hasNext` shadows can removed.

3 Shadow partitionings

Collaborative runtime verification leverages the fact that many users will execute the same application many times to reduce the runtime verification overhead for each user. The two basic options are to (1) reduce the number of active shadows for any particular run; or (2) reduce the (amortized) amount of work per active shadow. To explore these options, we devised two partitioning schemes, *spatial* and *temporal* partitioning. Spatial partitioning (Section 3.1) reduces the number of active shadows per run, while temporal partitioning (Section 3.2) reduces the amortized workload per active shadow over any particular execution.

Our partitioning schemes are designed to produce false negatives but no false positives. Our monitoring may miss some pattern matches (which will be caught eventually given enough executions), but any reported match must actually occur.

3.1 Spatial partitioning

Spatial partitioning reduces the overhead of runtime verification by only leaving in a subset of a program's shadows. However, choosing an arbitrary subset of shadows does not work; in particular, arbitrarily disabling skip shadows may lead to false positives. Consider the following code with the `HasNext` pattern.

```
1 for(Iterator i = c.iterator(); i.hasNext();)
2         Object o = i.next();
```

In this case, if the iterator `i` only exists in this loop, one safe spatial partitioning would be to disable all shadows in the program except for those in the loop. However, disabling the `hasNext` skip shadow on line 1 and enabling the `next` shadow on line 2 on a collection with two or more objects gives a false positive, since the monitor "sees" two calls to `next()` and not the call to `hasNext()` which prevents the match.

Enabling arbitrary subsets of shadows can also lead to wasted work. Disabling the `next` shadow in the above example and keeping the `hasNext` shadow would, of course, lead to overhead from the `hasNext` shadow. But, on their own, `hasNext` shadows can never lead to a complete match without any `next` shadows.

We therefore need a more principled way of determining sensible groups of shadows to enable or disable. In previous work, we have described the notion of a *shadow group*, which approximates 1) the shadows needed to keep tracematches triggerable and 2) the skip-shadows which must remain enabled to avoid false positives. We will now summarize the relevant points; the complete details are given in [6]. We start by defining the notion of a static joinpoint shadow.

Definition 1 (Shadow). A shadow s of a tracematch tm is a pair $(lab_s, bind_s)$, where lab_s is the label of a declared symbol of tm and $bind_s$ is a variable binding, modelled as a mapping from variables to points-to sets. A points-to set is a set of object-creation sites in the program. The points-to set $pts(v)$ for a variable v contains the creation sites of all objects which could possibly be created at runtime and assigned to v.

In the example code above, the `hasNext` shadow in line 1 would be denoted by $(\texttt{hasNext}, \{i \mapsto \{i_1\}\})$, assuming that we denote the creation site of iterator objects that might be bound by this shadow by i_1.

Definition 2 (Shadow group). A *shadow group* is a pair of 1) a multi-set of shadows called *label-shadows* and 2) a set of shadows called *skip-shadows*. All shadows in *label-shadows* are labelled with labels of non-skip edges on some path to a final state, while all shadows in *skip-shadows* are labelled with a label of a skip-loop.

We use a multi-set for *label-shadows* to record the fact that the automaton might not reach its final state unless two or more shadows with the same label execute. For instance, the *HasNext* pattern only triggers after two `next` shadows execute; the multiplicities in the multi-set encode the number of times that a particular symbol needs to execute before the tracematch could possibly trigger.

Definition 3 (Consistent shadow group). A *consistent* shadow group g is a shadow group for which all variable bindings of all shadows in the group have points-to sets with a non-empty intersection for each variable.

For our HasNext example, a consistent shadow group could have this form:

$$\begin{aligned} \mathit{label\text{-}shadows} &= [(\mathit{next}, i \mapsto \{i_1, i_2\}), (\mathit{next}, i \mapsto \{i_1\})],\\ \mathit{skip\text{-}shadows} &= \{(\mathit{hasNext}, i \mapsto \{i_1\}), (\mathit{hasNext}, i \mapsto \{i_1, i_3\})\} \end{aligned}$$

This shadow group is consistent—it may lead to a match at runtime—because the variable bindings for `i` could potentially point to the same object, namely an object created at creation site i_1. The shadow group holds two label shadows (labelled with the non-skip labels `next`). If the label shadows had disjoint points-to sets, then no execution would bind the tracematch variables to consistent objects, and the shadow group would not correspond to a possible runtime match. In addition, the shadow group holds *all* skip-shadows that have points-to sets that overlap with the label-shadows in the shadow group.

Conceptually, a consistent shadow group is the static representation of a possibly complete match at runtime. Every consistent shadow group may potentially cause its associated tracematch to match, if the label shadows execute in the proper order. Furthermore, only the skip shadows in the shadow group can prevent a match based on the shadow group's label shadows.

Our definition of a shadow group is quite well-suited to yielding sets of shadows that can be enabled or disabled in different spatial partitions. We therefore define a *probe* to be the union of all label-shadows and skip-shadows of a given consistent shadow group. (In constructing probes from shadow groups, we discard the multi-set structure of the label shadows and combine the label-shadows and skip-shadows into a single set). Probes "make sense" because they contain a set of shadows that can lead to a complete match and they are sound because they also contain all of the skip-shadows that can prevent that match. (We will explain why skip-shadows are crucial for probes in Section 3.2). Note that different probes may overlap; indeed, as Section 4 shows, many similar probes share the same hot shadows.

We can now present our algorithm for spatial partitioning.

- Compute all probes (based on the flow-insensitive analysis from [6]).
- Generate bytecode with two arrays: one array mapping from probes to shadows and one array with one entry per shadow.

– When emitting code for shadows, guard each shadow's execution with appropriate array look-ups.

The arrays, along with some glue code in the AspectJ runtime, allow us to dynamically enable and disable probes as desired. In the context of spatial partitioning, we choose one probe to enable at the start of each execution; however, our infrastructure permits experimentation with more sophisticated partitioning schemes.

3.2 Temporal partitioning

We found that spatial partitioning was effective in distributing the workload of runtime verification in many cases. However, in some cases, we found that a single probe could still lead to large overheads for some unlucky users. Two potential reasons for large overheads are: 1) a shadow group may contain a large number of skip-shadows, if all those shadows have overlapping points-to sets, leading to large probes; or 2) if shadows belonging to a probe are repeatedly executed within a tight loop which would otherwise be quite cheap, any overhead due to such shadows would quickly accumulate. The `HasNext` pattern is especially prone to case 2), as calls to `next()` and `hasNext()` are cheap operations and almost always contained in loops.

In such situations, one way to further reduce the runtime overhead is by sampling: instead of monitoring a given probe all the time, we monitor it from time to time and hope that the program is executed long enough that any violations eventually get caught. However, it is unsound to disable an entire probe and then naïvely re-enable it again on the same run: missing a skip shadow can lead to a false positive.

Consider the following code and the `HasNext` pattern:

```
for(Iterator i = c.iterator(); i.hasNext();)
  Object o = i.next();
```

If we disabled monitoring during the call to `hasNext`, we could get a false positive after seeing two calls to `next`, since the intermediate call to `hasNext` went unnoticed.

Because false positives arise from disabling skip-shadows, one sound solution is to simply not disable skip-shadows at all. Unfortunately, the execution of skip-shadows can be quite expensive; we found that leaving skip-shadows enabled also leaves a lot of overhead, defeating the purpose of temporal partitioning.

However, we then observed that if a state s holds an empty constraint (i.e. no disjuncts), then skip-shadows originating at s no longer need to execute[1]. We implemented this optimization for our temporal partitioning and found it to be quite effective: Section 4 shows that our temporal partitioning, with this optimization, does not incur much partitioning-related overhead; most of the overhead is due only to the executing monitors. Intuitively, this optimization works because, while all non-skip shadows are disabled, no new disjuncts are being generated. Hence, the associated constraint will become empty after few—often, just one—iterations of the skip-shadow, practically degenerating the skip-shadow to a no-op.

[1] This optimization is only safe if all variables are known to be bound at s. However, for all patterns we used in this work, and for almost all patterns we know, this is the case for all states. Our implementation statically checks this property and only applies the optimization if it holds.

We implemented the temporal partitioning as follows.

- Generate a Boolean flag per tracematch.
- When emitting code for shadows, guard each non-skip shadow with the appropriate flag.
- Change the runtime to start up an additional instrumentation control thread.

The control thread switches the instrumentation on and off at various time intervals. Figure 3 presents the parameters that the instrumentation control thread accepts; non-skip edges are enabled and then disabled after t_{on} milliseconds. Next, after another t_{off} milliseconds, the non-skip edges are enabled again.

Note that the Boolean flag we generate is independent of the Boolean array we use for spatial partitioning. If both spatial and temporal partitioning are used, a non-skip shadow is only enabled if both the Boolean array flag (from spatial partitioning) for this particular shadow and the Boolean flag (from temporal partitioning) for its tracematch are enabled. A skip shadow will be enabled if the Boolean array flag for its tracematch is enabled.

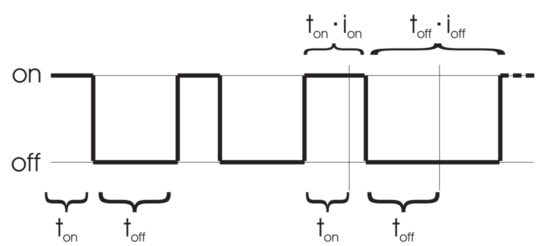


Figure 3. Parameters for temporal partitioning, with increase period of $n = 2$

The thread can also scale the activation periods: every n periods, it can scale t_{on} by a factor i_{on} and t_{off} by i_{off}. This technique—a well-known technique from adaptive systems such as just-in-time compilers—allows us to keep non-skip edges enabled for longer as the program runs longer, which gives our temporal partitioning a better chance of catching tracematches that require a long execution time to match. Because we increase the monitoring periods over time, the cost of monitoring scales with the total execution time of the program.

4 Benchmarks

To demonstrate the feasibility of our approach, we applied our modified tracematch compiler to five of the hardest benchmark/tracematch combinations from previous evaluations [6]. These benchmarks continue to exhibit more than 10% of runtime overhead, even after we applied all available static optimizations. They all consist of tracematches that verify properties of frequently used data structures, such as iterators and streams, in the applications of version 2006-10 of the DaCapo benchmark suite [5]. As usual, all our benchmarks are available on `http://www.aspectbench.org/`, along with a version of `abc` implementing our optimization. In the near future we also plan to integrate this implementation into the main `abc` build stream. Table 1 explains the tracematches that we used.

pattern name	description
FailSafeIter	do not update a collection while iterating over it
HasNextElem	always call hasNextElem before calling nextElement on an Enumeration
HasNext	always call hasNext before calling next on an Iterator
Reader	don't use a Reader after its InputStream was closed

Table 1. Tracematches applied to the DaCapo benchmarks

benchmark	classes	methods	complete overhead	# probes
antlr-Reader	307	3517	471.45%	4
chart-FailSafeIter	706	8972	25.08%	742
lucene-HasNextElem	309	3118	12.53%	6
pmd-FailSafeIter	619	6163	44.36%	426
pmd-HasNext	619	6163	66.53%	32

Table 2. Number of classes and methods per benchmark (taken from [5]), plus overhead of the fully instrumented benchmark, and number of probes generated for each benchmark

4.1 Spatial partitioning

We evaluated spatial partitioning by applying the algorithm from Section 3.1 to our five benchmark/tracematch combinations, after running the flow-insensitive static analysis described in [6]. Table 2 shows the runtime overheads with full instrumentation. All of these overheads exceed 10%, and the overhead for antlr-Reader is almost 500%. Table 2 also presents the number of probes generated for each benchmark.

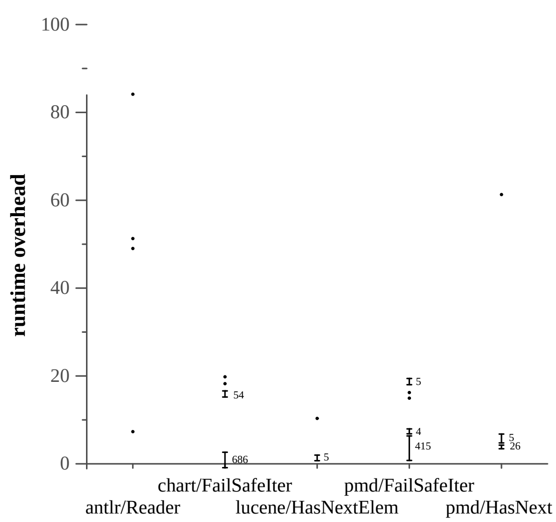


Figure 4. Runtime overheads per probe in spatial partitioning (in percent; bars indicate clumps of probes, labelled by size of clump)

Under the spatial partitioning approach, our compiler emits instrumented benchmarks which can enable or disable each probe dynamically. We tested the effect of each probe individually by executing each benchmark with one probe enabled at a time; this gave us 1210 benchmark configurations to test. For our experiments, we used the Sun Hotspot JVM version 1.4.2_12 with 2GB RAM on a machine with an AMD Athlon 64 X2 Dual Core Processor 3800+. We used the `-s large` option of the DaCapo suite to provide extra-large inputs, which made it easier for us to measure changes in runtimes. Figure 4 shows runtime overheads for the probes in our benchmarks. Dots indicate overheads for individual probes. For some benchmarks, many probes were almost identical, sharing the same hot shadows. These probes therefore also had almost identical overheads. We grouped these probes into clumps and present them as a bar, labelled with the number of probes in the clump.

Our results demonstrate that, in some cases, the different probes manage to spatially distribute the overhead quite well. However, spatial partitioning does not always suffice. For pmd-FailSafeIter, 9 probes out of 426 have overheads exceeding 5%, while for chart-FailSafeIter, 56 such cases exist, out of 742 probes in total. On the other hand, the lucene-HasNextElem and pmd-HasNext benchmarks contain only one hot probe each; spatial partitioning is unlikely to help in these cases.

Finally, antlr-Reader still shows high overheads, but these overheads are much lower than the original overhead of 471.45%. Interestingly, the four different overheads do not add up to 471.45%. Upon further investigation, we found that two probes generate many more disjuncts than others. In the fully instrumented program, each shadow in each probe has to look up all the disjuncts, even if they are generated by other probes, which might lead to overheads larger than the sum of the overheads for each individual probe. We are currently thinking about whether this observation could lead to an optimization of the tracematch implementation in general. (Disjunct lookup is described in greater detail in [4].)

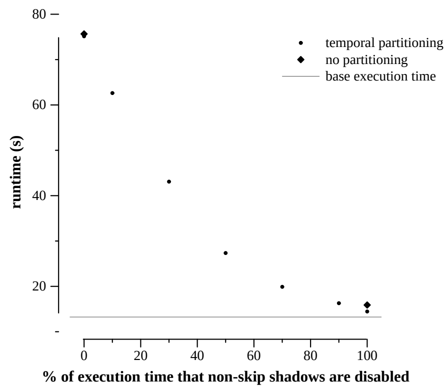
We conclude that spatial partitioning can sometimes be effective in spreading the overhead among different probes. However, in some cases, a small number of probes can account for a large fraction of the original total overhead. In those cases, spatial partitioning does not suffice for reducing overhead, and we next explore our temporal partitioning technique for improving runtime performance.

4.2 Temporal partitioning

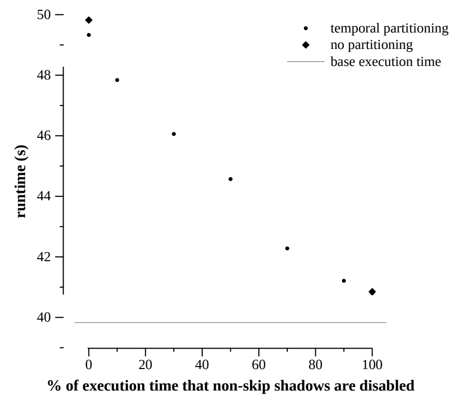
To evaluate the effectiveness of temporal partitioning, we measured ten different configurations for each of the five benchmark/tracematch combinations. Figure 5 presents runtimes for each of these configurations. The DaCapo framework collects these runtimes by repeatedly running each benchmark until the normalized standard deviation of the most recent runs is suitably small.

Diamond-shaped data points depict measurements of runtimes with no temporal partitioning; the left data point includes all probes (maximal overhead), while the right data point includes no probes (no overhead). The gap between the right diamond data point and the gray baseline, which denotes the runtime of the completely un-instrumented program, shows the cost of runtime checks. Note that spatial partitioning will always cost at least as much as the right diamond.

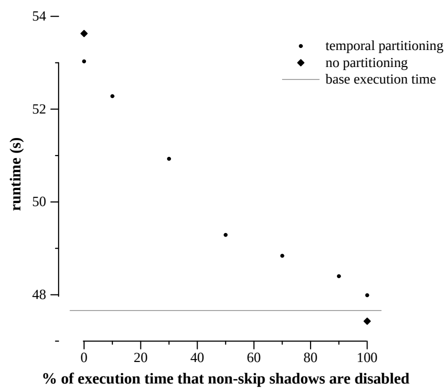
The circle-shaped data points present the effect of *temporal partitioning*. We measured the runtimes resulting from enabling non-skip edges 10, 30, 50, 70, 90



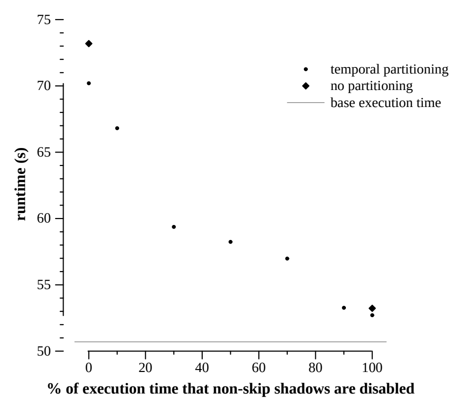
(a) antlr-Reader



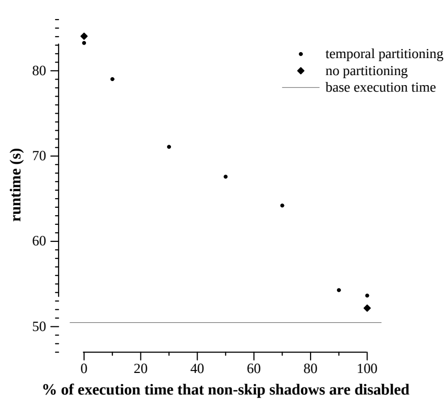
(b) chart-FailSafeIter



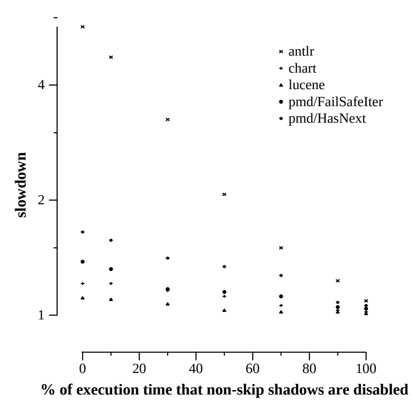
(c) lucene-HasNextElem



(d) pmd-FailSafeIter



(e) pmd-HasNext



(f) temporal partitioning for all benchmarks, log scale

Figure 5. Results of temporal partitioning for five benchmark/tracematch combinations

and 100 percent of the time. Our first experiment sought to determine the effect of changing the swapping interval for temporal partitioning.

At first, we executed four different runs for each of those seven configurations, with four different increase periods n. We doubled the duration of the on/off intervals every $n = 10, 40, 160$ and 640 periods. As expected, n has no measurable effect on runtime performance. We therefore plotted the arithmetic mean of the results over the different increase periods. The full set of numbers is available on our website.

Figure 5 (f) overlays the results from all of our benchmark/tracematch combinations. Note that the shape of the overhead curve is quite similar in all of the configurations. In all cases, temporal partitioning can properly scale down from 100% overhead, when all non-skip edges are always enabled, to just above 0%, when non-skip edges are never enabled. We were surprised to find that the decrease in runtime overhead did not scale linearly with a decrease in monitoring intervals. This data suggest that there might exist a "sweet spot" where the overhead is consistently lowest compared to the employed monitoring time.

The relationship between "no temporal partitioning" with all probes enabled and the 100% measurement with temporal partitioning enabled might seem surprising at first: we added additional runtime checks for temporal partitioning, and yet, in the cases of chart-FailSafeIter, lucene-HasNextElem and pmd-FailSafeIter, the code executes significantly faster. We believe that this speedup is due to the skip-loop optimization that we implemented for temporal partitioning: this optimization is applied even when non-skip edges are enabled, thereby improving overall performance.

The far right end of the graphs shows that the overhead of the runtime checks for spatial and temporal partitioning are virtually negligible. They are not zero but close enough to the baseline to not hinder the applicability of the approach.

5 Related work

Our work on collaborative runtime verification is most closely related to the work of Liblit et al. for automatic bug isolation. The key insight in automatic bug isolation is that a large user community can help isolate bugs in deployed software using statistical methods. The key idea behind *Cooperative Bug Isolation* is to use sparse random sampling of a large number of program executions to gather information. Hence, one can amortize the cost of executing assertion-dense code by distributing it to many users, each user only executing a small randomly selected number of assertions. This minimizes the overhead experienced by each user. Although each execution report in isolation gives only very limited information, the aggregate of all such reports provides a wealth of debugging information for analysis and a high chance of finding violations of an assertion, if they exist.

Pavlopoulou et al. [12] describe a *residual test coverage* monitoring tool which starts off by instrumenting all the code. As different parts of the program are executed, the code is periodically re-instrumented, with probes added only in places which have not been covered by the testing criteria. Probes from frequently executed regions are therefore removed in the first few re-instrumentation cycles, reducing the overhead in the long term since the program spends more and more time in code regions without any probes. Such an adaptive instrumentation should be applicable to our setting, too. To avoid false positives, one would have to disable entire shadow groups at a time.

Patil *et al.* [11] propose two different approaches to minimize overhead due to runtime checking of pointer and array accesses in C programs. *Customization* uses program slicing to decouple the runtime checking from the original program execution. The second approach, *shadow processing*, uses idle processors in multiprocessor workstations to perform runtime checking in the background. The shadow processing approach uses two processes: a main process, which executes the original user program, *i.e.* without any run-time checking, and a shadow process which follows the main process and performs the intended dynamic analysis. The main process has minimal overhead (5%-10%), mostly arising from the need for synchronization and sharing of values between the two processes. Such an approach would not work for arbitrary tracematches, which might arbitrarily modify the program state, but could work for the verification-oriented tracematches we are investigating.

Recently, Microsoft, Mozilla, GNOME, KDE and others have all developed opt-in services for reporting crash data. When a program crashes, recovery code generates and transmits a report summarizing the state of the program. Recently, Microsoft's system has been extended to gather data about abnormal program behaviour in the background; reports are then automatically sent every few days (subject to user permission). Reports from all users can then be aggregated and analyzed for information about causes of crashes.

We briefly mention a number of alternative approaches for specifying properties for runtime verification. The Program Query Language [10] 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. Monitoring-Oriented Programming [7] is a generic framework for specifying properties for runtime monitoring; developers use MOP logic plugins to state properties of interest. PQL, MOP, and related approaches, can all benefit from collaborative runtime verification techniques.

6 Conclusion and future work

In this paper we have presented two techniques for implementing collaborative runtime verification with tracematches. The main idea is to share the instrumentation over many users, so that any one user pays only part of the cost of the runtime verification. Our paper has described the spatial and temporal partitioning techniques and demonstrated their applicability to a collection of benchmarks which exhibit high instrumentation overheads.

Spatial partitioning allocates different probes—consistent subsets of instrumentation points—to different users; probes generally have lower overheads than the entire instrumentation. Our experimental evaluation showed that spatial partitioning works well when there are no particularly hot probes.

Temporal partitioning works by periodically enabling and disabling instrumentation. We demonstrated a good correspondence between the proportion of time that probes were enabled and the runtime overhead.

We are continuing our work on making tracematches more efficient on many fronts, including further static analyses. We are also continuing to build up our benchmark library of base programs and interesting tracematches.

References

1. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Object-Oriented Programming, Systems, Languages and Applications*, pages 345–364. ACM Press, 2005.
2. Pavel Avgustinov, Oege de Moor, and Julian Tibble. On the semantics of matching trace monitoring patterns. In *Seventh Workshop on Runtime Verification, Vancouver, Canada*, March 2007. To appear in *Lecture Notes on Computer Science*.
3. Pavel Avgustinov, Julian Tibble, Eric Bodden, Ondřej Lhoták, Laurie Hendren, Oege de Moor, Neil Ongkingco, and Ganesh Sittampalam. Efficient trace monitoring. Technical Report abc-2006-1, http://www.aspectbench.org/, March 2006.
4. Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making trace monitors feasible. In *ACM Conference 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. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.
6. Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *21st European Conference on Object-Oriented Programming, July 30th-August 3rd 2007, Berlin, Germany*, volume 4609 of *Lecture Notes in Computer Science*, pages 525–549. Springer, 2007.
7. Feng Chen and Grigore Rosu. MOP: An efficient and generic runtime verification framework. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2007. To appear.
8. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *European Conference on Object-oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.
9. Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 141–154, San Diego, California, June 2003.
10. Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 365–383, 2005.
11. Harish Patil and Charles Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Softw. Pract. Exper.*, 27(1):87–110, 1997.
12. Christina Pavlopoulou and Michal Young. Residual test coverage monitoring. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 277–284, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
13. Wolfgang Grieskamp (Microsoft Research), January 2007. Personal communication.