Journal of Logic and Computation Advance Access published November 27, 2008

Collaborative Runtime Verification with Tracematches

ERIC BODDEN and LAURIE HENDREN, *Sable Research Group, McGill University, Montréal Québec, Canada.*
E-mail: ebodde@sable.mcgill.ca; hendren@sable.mcgill.ca

PATRICK LAM, *Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, Canada.*
E-mail: p.lam@ece.uwaterloo.ca

ONDŘEJ LHOTÁK and NOMAIR A. NAEEM, *Programming Languages Group, University of Waterloo, Waterloo, Ontario, Canada.*
E-mail: olhotak@uwaterloo.ca; nanaeem@plg.uwaterloo.ca

Abstract

Perfect pre-deployment test coverage is notoriously difficult to achieve for large applications. Given enough end users, however, 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 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 can distribute the burden of runtime verification among multiple users and over multiple runs. 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.

Keywords: Runtime monitoring, verification, randomization, whole-program analysis, aspect-oriented programming.

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. If deployed code carried instrumentation for runtime verification, developers could track down the causes of observed failures more easily. However, instrumentation 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.

doi:10.1093/logcom/exn077

Tracematches [1] are one mechanism for specifying runtime monitors that can be used to conduct runtime verification. Tracematches enable developers to state sequences of program events and actions to take if the program execution generates sequences of interest.

According to researchers in industry [14], companies would likely be willing to accept runtime verification in deployed code if the verification overhead was below 5%. In previous work on tracematches, researchers have shown that, in many cases, static analysis can enable efficient runtime monitoring. While Avgustinov *et al*. [2] focused on statically optimizing the implementation of the runtime monitor, we ourselves used a static whole-program analysis to remove unnecessary instrumentation from the program under test [4, 5, 11].

Most often, the combination of both techniques can reduce runtime overhead to under 10%. However, our evaluation also showed that unreasonably large overheads—sometimes >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 (PQL) [9] causes up to 37% overhead on its benchmark applications, and JavaMOP [6] incurs up to 176% overhead when only automated optimizations are applied.

In this work, we attack the problem of runtime-verification induced overhead by using methods from remote sampling [8]. Because companies that produce large pieces of software (which are usually hard to analyse) often have access to a large user base, one can leverage the size of the user base to deploy different partial instrumentations ('probes') for each user. A centralized server can then combine runtime verification results from runs with different probes. Although sampling-based approaches have many different applications, we are most 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 set of instrumentation points into different subsets. We call each subset of instrumentation points a *probe*. Each user is given a program instrumented with only a few probes. 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, limiting the total overhead. This method works even if there 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 violations of monitored runtime verification properties may go unnoticed while the probes are disabled.

In both spatial and temporal partitioning, any still-enabled instrumentation must operate correctly. To avoid additional burden on the developers and maintainers, we further demand that the partitionings must never cause false positives. False positives could easily arise under a naive partitioning scheme: some events discard partial matches, resetting runtime monitors to their initial states. If the discarding events do not execute, the monitor might trigger, even if it would not have triggered under the complete instrumentation. It might therefore seem that we must always retain all discarding events. However, we found that, under some conditions, the discarding events would have no further effect and we implemented an optimization to disable discarding events when these conditions apply.

We explored the feasibility of our approach by applying our modified tracematch compiler to benchmarks whose overheads persisted after the static analysis in [4]. 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 <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% of the time. We found that the overhead increased surprisingly steadily with the proportion of time that the probes were enabled, so that one can gain limited but efficient runtime monitoring by running probes only some of the time.

The remainder of this article 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. Developers provide definitions for symbols using AspectJ [7] 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 object and may refer to these objects when the tracematch matches. If a tracematch does not bind any variables, then the verification of that tracematch reduces to verifying finite-state properties of the program as a whole. If a tracematch binds a single variable then the verification of that tracematch is essentially equivalent to statically determining the possible typestates [15] of each bound object.

2.1 HasNext tracematch

Figure 1 presents the `HasNext` verification tracematch, matching suspicious traces where a program calls `i.next()` twice in a row without any intervening call to `i.hasNext()`.

Tracematch symbols may bind variables; line 1 of the tracematch declares that symbols in the `HasNext` tracematch may bind an `Iterator i`. Lines 2–5 define symbols `hasNext` and `next`, which capture calls to the `hasNext()` and `next()` methods of `i`. These two symbols establish the alphabet for the tracematch's regular expression '`next next`' at line 7. This expression specifies that the tracematch should execute after seeing `next` two times, as long as `hasNext` does not occur

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

FIGURE 1. HasNext tracematch: do not call `next()` twice with no intervening call to `hasNext()`

in between calls to `next`. Every time the regular expression matches, the tracematch will execute the attached body of code (also on line 7). In this work, we focus on verification tracematches, which typically encode API usage rules.

2.2 Tracematch matching example

The tracematch runtime matches the regular expression against each suffix of the abstract (symbol based) execution trace. For instance, consider the abstract event sequence

```
hasNext next next next.
```

The '`next next`' regular expression would match this sequence twice, executing the tracematch body at the second and third `next` events.

However, tracematches also contain variables, and matches require their symbols' variable bindings to be consistent. The above example would therefore only match if two calls to `next` occurred on the same iterator. Hence, with iterators `i1` and `i2`, the concrete events for the sequence we considered earlier could actually be

```
i1.hasNext() i2.next() i1.next() i2.next(),
```

giving an abstract event sequence of

```
hasNext(i=i1) next(i=i2) next(i=i1) next(i=i2).
```

Conceptually, tracematch matching projects the event sequence onto distinct subsequences as determined by variable bindings. Our example sequence contains two projections: (1) '`hasNext next`' for `i=i1`, and (2) '`next next`' for `i=i2`. Projection (1) is not matched by the tracematch's regular expression, but projection (2) is matched, and the tracematch body should execute once, at the last call to `next()`, with the binding `i=i2`.

2.3 Semantics of tracematches

We next present a summary of the semantics of tracematches. See [1] for a full semantics.

Fix a tracematch *tm* with free variables $V = \{v_1, \ldots, v_n\}$. Let the heap $\mathcal{H}$ consist of objects $\{o_1, \ldots, o_m\}$. Define bindings $\sigma : V \to \mathcal{H}$ which map the free variables of *tm* to heap objects. An event sequence s is a sequence of pairs $\langle \mathrm{sym}_i, \sigma_i \rangle$ where sym_i is a symbol of *tm* and σ_i is a (possibly partial) binding. The projection $s|_\sigma$ is the subsequence of s which preserves the events where $\sigma \supseteq \sigma_i$. Then the runtime must execute the tracematch body if there exists some total function σ such that the regular expression from *tm* matches a suffix of the symbols of $s|_\sigma$.

2.4 Tracematch implementation

The tracematch compiler handles tracematches by (i) creating finite-state machines to track the states of active partial matches and (ii) creating code that updates the finite-state machine whenever events of interest occur. The compiler uses *constraints* to track objects that have partial trace matches; 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

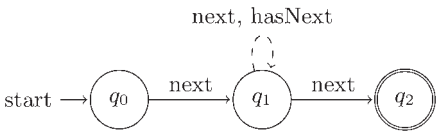


FIGURE 2. Finite-state machine for the tracematch of Figure 1

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 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. Allan *et al.*[1] call this loop a *skip loop*. Skip loops remove partial matches that cannot extend to complete matches: they discard a partial match whenever an observed event invalidates that partial match.

As an example, assume that state q_1 carries the constraint $i=i_1 \vee i=i_2$; 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, we know that i_2 has to leave state q_1 (because there is no `hasNext` self-loop on q_1). Skip loops cause the runtime to discard i_2 at q_1 as follows: at the skip loop, the runtime will conjoin the original constraint at q_1, $i=i_1 \vee i=i_2$, with the skip loop-generated negative binding $i \neq i_2$ ('i_2 is not at q_1'):

$$(i=i_1 \vee i=i_2) \wedge i \neq i_2 \equiv (i=i_1 \wedge i \neq i_2)$$

The runtime further simplifies this constraint to $i=i_1$, as $i=i_1$ implies $i \neq i_2$. The resulting constraint $i=i_1$ then states that only i_1 is in state q_1. To summarize, a skip loop discards a partial match by forcing a variable binding to leave the state that the loop is attached to.

The tracematch compiler weaves code to monitor tracematches into programs at appropriate event locations. In particular, the compiler includes instrumentation code that updates the appropriate disjuncts at every static code location corresponding to a potential event execution. This instrumentation code is called a *shadow* [10]. Figure 3 shows an example program that uses two iterators `entryIter` and `iterator`. For this program, the compiler would add `hasNext` shadows at lines 3, 10, 13 and 18, and `next` shadows at lines 4 and 11.

In this article, we use a previously published static analysis that removes shadows if they can be shown to never contribute to complete matches [4]; 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 be 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 (i) reduce the number of active shadows for any particular run; or (ii) 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.

```
1  private void mapToString(Map<String, List<String>> map, StringBuffer sb) {
2      for (Iterator<Map.Entry<String, List<String>>> entryIter =
3              map.entrySet().iterator(); entryIter.hasNext();) {
4          Map.Entry<String, List<String>> entry = entryIter.next();
5          sb.append(entry.getKey());
6          List<String> args = entry.getValue();
7          if(!args.isEmpty()) {
8              sb.append("(");
9              for (Iterator<String> iterator = args.iterator();
10                  iterator.hasNext();) {
11                 String varName = iterator.next();
12                 sb.append(varName);
13                 if(iterator.hasNext())
14                     sb.append(",");
15             }
16             sb.append(")");
17         }
18         if(entryIter.hasNext()) {
19             sb.append(",");
20         }
21         sb.append(" ");
22     }
23     sb.append("\n");
24 }
```

FIGURE 3. Example program using iterators

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 and long 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 is more than unsatisfactory; in particular, arbitrarily disabling skip shadows may lead to false positives. Consider again the example from Figure 3, in combination with the `HasNext` pattern. In this case, one safe spatial partitioning would be to disable all shadows in the program except for those referring to `entryIter` (lines 3, 4 and 18). However, many partitionings are unsafe; for instance, disabling the `hasNext` shadow on

line 3 (the shadow of a skip loop) but enabling the `next` shadow on line 4 on a map with two or more entries gives a false positive, since the monitor 'sees' two calls to `next()` and not the call to `hasNext()` which would prevent 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 [4], we have described the notion of a *shadow group*, which approximates (i) the shadows needed to keep tracematches triggerable and (ii) the skip shadows which must remain enabled to avoid false positives triggered by the former. We will now summarize the relevant points.

Note that we have moved from the dynamic view of the program's execution, as described in Section 2, to a static (compile time) view. In particular, instead of tracking runtime objects, we track their compile-time analogue, points-to sets. We continue by defining the notion of a static joinpoint shadow.

DEFINITION 1 (Shadow)
A shadow s of a tracematch tm is a pair (sym_s, σ_s), where sym_s is the label of a symbol of tm (e.g. `hasNext` or `next`) and σ_s is a variable binding, modelled as a mapping from tracematch variables (e.g. i) to points-to sets. A points-to set is a set of object-creation sites in the program. The points-to set $pts(\mathtt{v}) = \{v_1, \ldots, v_n\}$ for a program variable `v` contains the creation sites of all objects which could possibly be created at runtime and assigned to `v`. For a tracematch variable t, we write $t = \{v_1, \ldots, v_n\}$ if t was assigned the value of program variable `v` at a given shadow. This models the runtime situation where t is assigned an object that was created at any of the creation sides $\{v_1, \ldots, v_n\}$.

In our running example from Figure 3, the `hasNext` shadow in line 3 would be denoted by $(\mathtt{hasNext}, \{i = \{i_1\}\})$, assuming that we denote the creation site of `entryIter` by i_1.

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

In our running example, the `next` shadows are *progress-shadows* while the `hasNext` shadows are *skip-shadows*. Note that, in general, a symbol can induce both skip loops and normal edges (since an object might move from q to q' and yet always leave q); therefore, *progress-shadows* and *skip-shadows* are not necessarily disjoint.

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 the HasNext tracematch, a consistent shadow group could have this form:

$$\begin{aligned} \textit{progress-shadows} &= \{\ (\textit{next}, i = \{i_1, i_2\})\ ,\ (\textit{next}, i = \{i_1\})\ \}, \\ \textit{skip-shadows} &= \{\ (\textit{hasNext}, i = \{i_1\})\ ,\ (\textit{hasNext}, i = \{i_1, i_3\})\ \} \end{aligned}$$

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 progress shadows (labelled with the progress label `next`). If the label shadows had disjoint points-to sets, then no execution would consistently bind the tracematch variables to 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 progress 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 progress 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 progress shadows. For further discussion on efficiently computing consistent shadow groups, please refer to [4].

Our definition of a shadow group is well-suited to finding sets of shadows that can be enabled or disabled in different spatial partitions. We therefore define a *probe* to be the union of all progress shadows and skip shadows of a given consistent shadow group. 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.)

We can now present our algorithm for spatial partitioning.

- Compute all probes (based on the flow-insensitive analysis from [4]).
- Generate bytecode with two arrays: a 'map' array mapping from probes to shadows and a 'flag' 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 (using array look-ups). 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.

3.1.1 Spatial partitioning example

In our running example, the 'map' array, which maps from probes to shadows, would look like this:

map	3	4	10	11	13	18
1	x	x				x
2			x	x	x	

This array encodes the information that the shadows at lines 3, 4 and 18 all belong to probe number 1, and the shadows at lines 10, 11 and 13 belong to probe number 2. To index into the array we assign each probe and each shadow a unique number, starting at 0. Note that different probes may overlap, i.e. there may be multiple rows with an 'x' entry in the same column; indeed, as Section 4 shows, many similar probes share the same hot shadows.

The 'flag' array, with one entry per shadow, would initially be initialized to all **false**:

flag	3	4	10	11	13	18

If we then assume that the user chooses to enable probe number p, the tracematch runtime sets the Boolean flags in the 'flag' array for all shadows of this probe (as determined by the 'map' array)

to **true**.

$$\forall i \in \{0, \ldots, |\text{shadows}| - 1\} : \ \text{flag}[i] := \text{flag}[i] \vee \text{map}[p, i]$$

For our example, with $p = 1$, this would result in:

flag	3	4	10	11	13	18
			x	x	x	

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: (i) a shadow group may contain a large number of shadows if all those shadows have overlapping points-to sets, leading to large probes; or (ii) 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 (ii), as calls to `next()` and `hasNext()` are cheap operations and are 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 re-enable it again on the same run: missing a skip shadow can lead to a false positive.

Consider again the example from Figure 3 in combination with the `HasNext` pattern. If we disabled monitoring during the call to `hasNext` at line 3, we could get a false positive after seeing two calls to `next` at line 4, 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 when a state q holds an empty constraint (i.e. no disjuncts), then transitions originating at q no longer need to execute, including skip loops. We therefore implemented a 'skip-disabling' optimization that disables shadows in the presence of empty constraints; this optimization is safe if all variables are known to be bound at q. That is, if all variables are bound, then whenever we generate a negative binding of the form $i \neq i_2$, there already has to be a positive binding of the form $i = i_1$, and as we discussed in Section 2 ($i = i_1 \wedge i \neq i_2$) reduces to $i = i_1$; the negative binding $i \neq i_2$ therefore need not be stored. Had i been unbound at q, then $i \neq i_2$ would have to be stored explicitly in the constraint, and the disabling optimization would not apply. However, for all patterns we used in this work, and for almost all patterns we know, all variables are bound on the first transition. Our implementation statically checks this property and only applies the optimization if it holds.

– At each state q for which it is guaranteed that all tracematch variables are bound to an object when reaching q, insert code that disables all skip loops on q for as long as q holds the empty constraint. Insert code that re-enables the skip loops as soon as a new disjunct is added to q again.

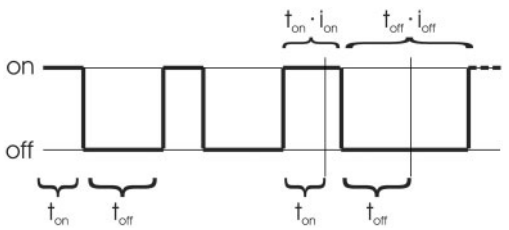


FIGURE 4. Parameters for temporal partitioning, with increase period of $n = 2$

We implemented this skip-disabling 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 progress shadows are disabled, no new disjuncts are being generated. Hence, the associated constraint will become empty after few—often just one—iterations of the shadow, degenerating the shadow to a costly no-op. Our optimization significantly reduces the runtime cost of this no-op.

Let us again consider our running example from Figure 3, denoting the object stored in a program variable `v` with $o(\texttt{v})$. If we disable progress transitions with a constraint $i = o(\texttt{entryIter}) \vee i = o(\texttt{iterator})$ on state q_1, and then monitor a call to `iterator.hasNext()` (at line 10), then the constraint on q_1 reduces to $i = o(\texttt{entryIter})$ (cf. Section 2). When monitoring a subsequent call to `entryIter.hasNext()` (at line 18), this constraint reduces to **false**, the empty constraint, which enables our optimization.

We implemented temporal partitioning as follows.

- Generate a Boolean flag per tracematch.
- When emitting code for shadows, guard each progress 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 4 presents the parameters that the instrumentation control thread accepts; progress edges are enabled and then disabled after t_{on} milliseconds. Next, after another t_{off} milliseconds, the progress 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 progress shadow is only enabled if both the Boolean array flag (from spatial partitioning) for that particular shadow and the Boolean flag (from temporal partitioning) for its tracematch are enabled. A skip shadow will be enabled all the time if the Boolean array flag (from spatial partitioning) for its tracematch is enabled at program start-up.

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 progress 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.

TABLE 1. Tracematches applied to the DaCapo benchmarks

Pattern name	Description
FailSafeIter	do not update a collection while iterating over it
HasNextElem	do not call `Enumeration.nextElement` twice without calling `hasNextElement` in between
HasNext	do not call `Iterator.next` twice without calling `hasNext` in between
Reader	do not use a `Reader` after closing its `InputStream`

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 [4]. These benchmarks continue to exhibit >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 [3]. All our benchmarks are available on `http://www.aspectbench.org/`, along with a version of the AspectBench Compiler implementing our optimization. Table 1 explains the tracematches that we used.

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 [4]. 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. Recall that the number of probes depends heavily on the precision of the underlying points-to analysis, as well as intrinsic properties of the benchmark, for instance the lifetimes of bound objects: if objects are longer lived then they tend to be referenced at more program places than the short-lived objects. Therefore, in these cases many variables share the same points-to sets, yielding larger probes. If the points-to analysis is imprecise (e.g. because it soundly over-approximates dynamic class loading) this may lead to larger points-to sets that overlap a lot with among another, which in turn causes probes to be merged that would otherwise remain separate had the points-to analysis been more precise. Consequently the number of probes varies a lot from case to case depending on these properties.

Under the spatial partitioning approach, our compiler emits instrumented benchmarks which can enable or disable each probe at program start-up. 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. (Note that it is possible to enable multiple probes at the same time without jeopardizing soundness.) 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 5 shows runtime overheads for the probes in our benchmarks.

TABLE 2. Number of classes and methods per benchmark (taken from [3]), plus overhead of the fully instrumented benchmark and number of probes generated for each benchmark

Benchmark	Classes	Methods	Complete overhead	Number of 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

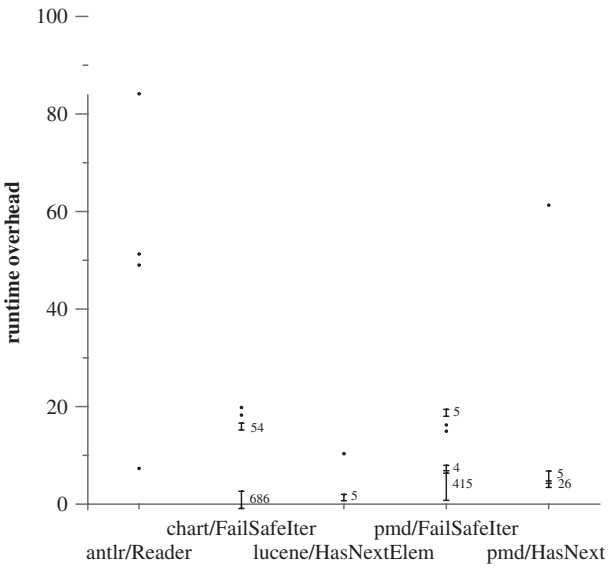


FIGURE 5. Runtime overheads per probe in spatial partitioning (in percent; bars indicate clumps of probes, labelled by size of clump)

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, 42 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 clearly insufficient 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. (Avgustinov *et al.* [2] describe the current implementation of disjunct look-up—indexing—in greater detail.)

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 10 different configurations for each of the five benchmark/tracematch combinations. Figure 6 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 grey baseline, which denotes the runtime of the completely uninstrumented program, shows the cost of checking the Boolean flags that we inserted. Note that spatial partitioning will always cost at least as much as the right diamond.

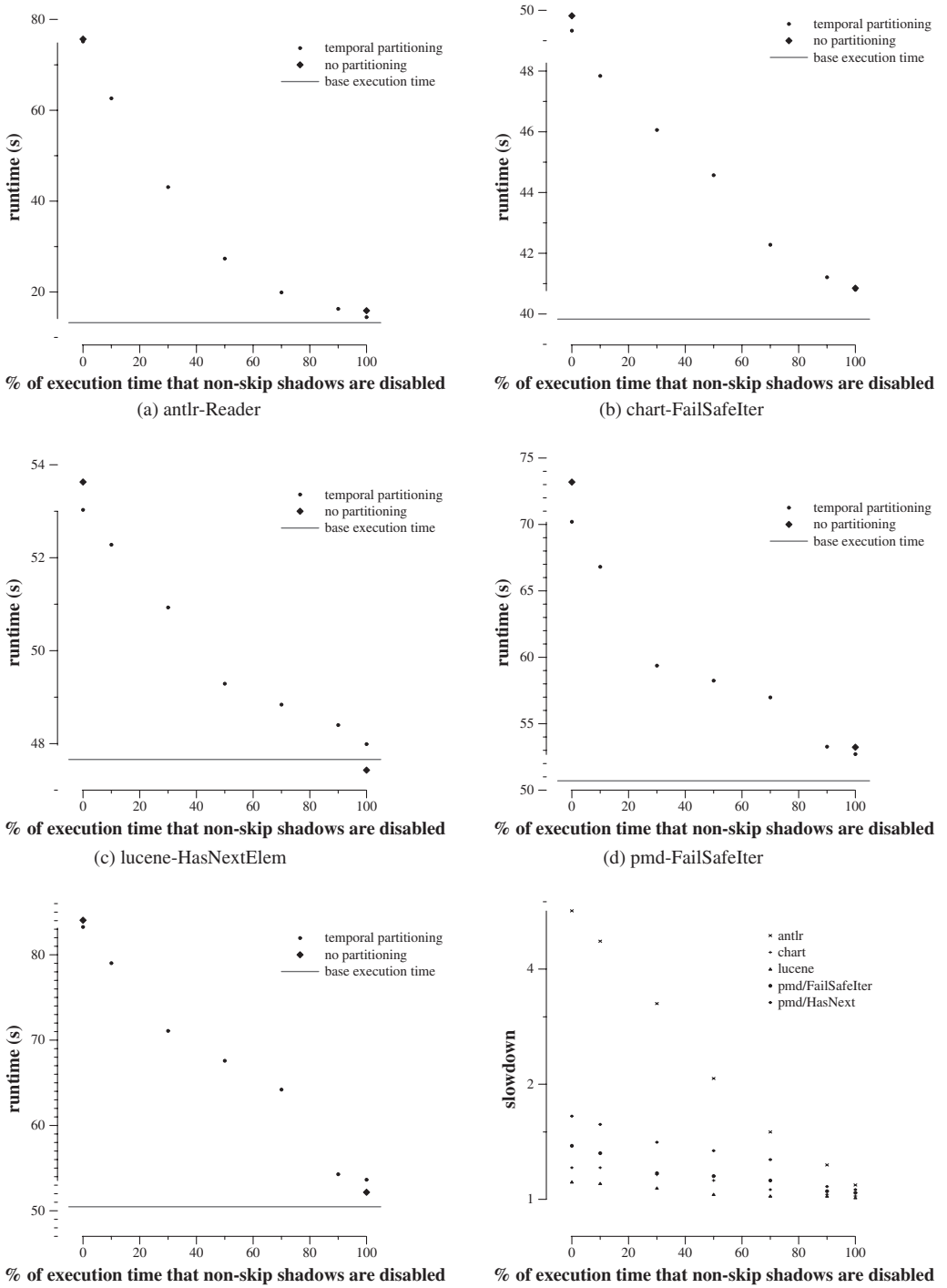
The circle-shaped data points present the effect of *temporal partitioning*. We measured the runtimes resulting from enabling progress edges 10, 30, 50, 70, 90 and 100% of the time.

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 the website of the AspectBench Compiler: `http://www.aspectbench.org/`

Figure 6f 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 smoothly scale down from 100% overhead, when all progress edges are always enabled, to just above 0%, when progress 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 (note that the figure uses a log scale). 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 progress 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.



(a) antlr-Reader

(b) chart-FailSafeIter

(c) lucene-HasNextElem

(d) pmd-FailSafeIter

(e) pmd-HasNext

(f) temporal partitioning for all benchmarks, log scale

FIGURE 6. Results of temporal partitioning for five benchmark/tracematch combinations

5 Related work

Our work on collaborative runtime verification is most closely related to the work of Liblit *et al*. [8] 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 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 such violations exist.

Pavlopoulou *et al*. [13] 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 also be applicable to our setting. To avoid false positives, one would have to disable entire shadow groups at a time.

Patil *et al*. [12] 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 that executes the original user program, i.e. without any run-time checking, and a shadow process that 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 analysed for information about causes of crashes.

We briefly mention a number of alternative approaches for specifying properties for runtime verification. The PQL [9] 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 (MOP) [6] is a generic framework for specifying properties for runtime monitoring; developers use MOP logic plug-ins 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 article, 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 article has described the spatial and temporal

partitioning techniques and demonstrated their applicability to a collection of benchmarks, which exhibit high instrumentation overheads. Both partitioning approaches are sound, i.e. they produce no false positives.

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 handles the situation where some probes are disproportionately hot, 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 [5, 11]. We are also continuing to build up our benchmark library of base programs and interesting tracematches.

Funding

Natural Sciences and Engineering Research Council of Canada.

References

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 345–364. ACM Press, 2005.

[2] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. Steele Jr, eds, pp. 589–608. ACM Press, 2007.

[3] 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 *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 169–190. ACM Press, New York, NY, USA, 2006.

[4] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *21st European Conference on Object-Oriented Programming (ECOOP), July 30th-August 3rd 2007, Berlin, Germany*. Vol. 4609 of *Lecture Notes in Computer Science*, pp. 525–549. Springer, 2007.

[5] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM Press, 2008.

[6] F. Chen and G. Roşu. MOP: an efficient and generic runtime verification framework. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 569–588. ACM Press, New York, NY, USA, 2007.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *European Conference on Object-oriented Programming (ECOOP)*, J. Lindskov Knudsen, ed. Vol. 2072 of *Lecture Notes in Computer Science*, pp. 327–353. Springer, 2001.

[8] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pp. 141–154, San Diego, California, 2003.

[9] M. Martin, B. Livshits, and M. S. Lam. Finding application errors using PQL: a program query language. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 365–383, 2005.

[10] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *International Conference on Compiler Construction (CC)*, G. Hedin, ed. Vol. 2622 of *Lecture Notes in Computer Science*, pp. 46–60. Springer, 2003.

[11] N. A. Naeem and O. Lhoták. Typestate-like analysis of multiple interacting objects. In *OOPSLA '08: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 347–366. ACM Press, 2008.

[12] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software - Practice & Experience*, **27**, 87–110, 1997.

[13] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *21st International Conference on Software Engineering (ICSE)*, pp. 277–284. IEEE Computer Society Press, Los Alamitos, CA, USA, 1999.

[14] Wolfgang Grieskamp (Microsoft Research), January 2007. Personal communication.

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

Received 1 February 2007