Dynamic Optimization of Bytecode Instrumentation

Yudi Zheng, Lubomír Bulej, Cheng Zhang, Stephen Kell, Danilo Ansaloni, Walter Binder
University of Lugano, Switzerland
firstname.lastname@usi.ch

Abstract

Accuracy, completeness, and performance are all major concerns in the context of dynamic program analysis. Emphasizing one of these factors may compromise the other factors. For example, improving completeness of an analysis may seriously impair performance. In this paper, we present an analysis model and a framework that enables reducing analysis overhead at runtime through adaptive instrumentation of the base program. Our approach targets analyses implemented with code instrumentation techniques on the Java platform. Overhead reduction is achieved by removing instrumentation from code locations that are considered unimportant for the analysis results, thereby avoiding execution of analysis code for those locations. For some analyses, our approach preserves result accuracy and completeness. For other analyses, accuracy and completeness may be traded for a major performance improvement. In this paper, we explore accuracy, completeness, and performance of our approach with two concrete analyses as case studies.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors - Code generation, Optimization, Run-time environments

General Terms Measurement, Performance

Keywords JVM bytecode instrumentation, runtime adaptation, performance

1. Introduction

Dynamic program analyses proceed by collecting and processing information about a program's execution. Usually, this information is collected using program instrumentation. In general, denser and more fine-grained instrumentation allows greater accuracy and greater insight to be obtained, but this comes at the cost of higher performance overheads. Therefore, dynamic analyses must find a trade-off between accuracy and detail of the results and the runtime cost of obtaining them.

A body of existing work has considered optimizing this trade-off by careful selection (and omission) of instrumentation sites, both spatially across the program text and temporally over the execution. Arnold and Ryder [3] use sampling to reduce the overhead of profiling. Liblit et al. [11] also rely on sampling to make the slowdown of instrumented programs tolerable to end users. Meanwhile, static analysis is often used to identify useful instrumentation points. Ball and Larus [4] propose to update special integer values at selected control flow graph edges for path profiling, and Bodden [5] avoids redundant monitors at program points that are equivalent in terms of type-state. Moreover, Bodden and Havelund use a static whole-program analysis to soundly reduce the statements matched by the `maybeShared()` pointcut in their AspectJ extension [6], resulting in substantial reduction of runtime overhead.

Existing systems have devised and implemented specific optimizations for specific purposes. However, currently it is difficult to generalize and reuse this work in the context of new analyses. In this paper, we present a general approach to the problem of reducing runtime overhead of dynamic analyses, where we characterize this problem over three dimensions: accuracy, completeness, and performance. We begin by defining this three-dimensional model. Based on this characterization, we present a framework for reducing analysis overhead via adaptive runtime instrumentation.

Building on top of our dynamic analysis framework FRANC [2], our approach targets dynamic program analyses implemented by bytecode instrumentation techniques on the Java platform. Our approach builds on a mechanism for dynamic undeployment and redeployment of instrumentation, and focuses on the scenario where as the analysis executes, code locations may be determined to be unimportant, or no longer important, regarding the accuracy and/or completeness of the results, allowing the extent of instrumentation to be gradually reduced.

We evaluated our approach with two case studies. For one analysis, namely *Stationary Field Analysis* (SFA), our approach significantly improves performance, while preserving result accuracy and completeness. For the other analysis, namely *method execution time analysis*, our approach again significantly improves performance, while improving the accuracy of the results with a certain sacrifice of completeness. The observation that the performance and accurarcy of the analysis are improved at the same time is a little surprising, as it is contrary to the conventional trade-off in dynamic analyses. However, it is reasonable, because the reduction of instrumentation can actually reduce the runtime overhead

as well as the intrusiveness of the analysis. In summary, the main contributions of this paper are as follows.

1. Using a table-based model of analysis-collected data, we articulate the key concepts related to accuracy, completeness, and performance of dynamic analyses. We describe the design and architecture of a framework for dynamically changing the extent of active instrumentation at runtime. We show how this framework allows program instrumentation to be dynamically undeployed and redeployed in a highly flexible fashion, adaptable to the needs of a wide range of analyses.
2. We show the efficacy of our framework in reducing analysis overhead on two case studies with different sensitivity to overhead introduced by instrumentation. Our approach generally gains significant performance improvement in various dynamic analyses while preserving accuracy and completeness for some of them.

The rest of the paper is organized as follows. Section 2 describes our table-based characterization of the main factors in dynamic program analyses. Sections 3 and 4 present an evaluation of our approach to stationary field analysis and for method execution time analysis, respectively. Section 5 discusses related work and Section 6 concludes the paper.

2. Reducing Analysis Overhead

The value of a dynamic analysis depends on the quality and utility of its result and the cost of obtaining it. While the cost is typically associated with performance (in the form of runtime overhead), the quality of the result is largely determined by its accuracy and completeness. We begin by discussing a model of these, then continue by outlining how this model can support instrumentation adaptation decisions at runtime.

2.1 Characterizing accuracy and completeness

Distinguishing completeness from accuracy in a precise way requires some assumptions over the structure of the analysis' outputs.[1] We assume that the analysis output can be structured as a table, or mapping from keys to values. Then the keys in the table typically represent some programmer-recognizable program elements of interest to the analysis (e.g., statement, basic block, method, program path, interleaving, etc.), whereas the values in the table associated with the keys represent the data being collected (e.g., execution count, execution time, etc.).

Result accuracy and completeness can then be defined as properties of the (sets of) keys and values in the table. Imagine that the analysis generates imperfect results—perhaps slightly wrong, or perhaps completely fictitious. To define accuracy and completeness, we suppose that an oracle has supplied us the perfect results, which by definition are entirely complete and accurate. Accuracy and completeness can be then defined as edit-distance metrics between these two tables. Any edit which deletes a key reduces completeness, while an edit which modifies a value reduces accuracy. If values are in some domain on which orderings are defined, we can talk about inaccurate values over- or under-approximating the true value.

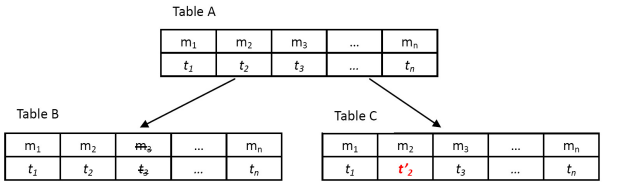

Figure 1. Example results of a method execution time analysis. Table A is the perfect result, while Table B leaves out the key m_3 and its value t_3 (the result is incomplete but accurate), and Table C is over-approximated ($t'_2 > t_2$) owing to perturbation introduced by runtime monitoring (the result is complete but inaccurate).

Fig. 1 shows example results of a method execution time profiling, where Table A is the supposed perfect (i.e., both accurate and complete) result. In contrast to Table A, Table B shows result obtained when instrumentation of an uninteresting method (say m_3) has been disabled. Although Table B shows incomplete results, the results are nevertheless accurate, and may still be useful (if the analysis' user is not interested in m_3). Table C shows a different but common case, where the perturbation introduced by instrumentation entails that some values are over-approximated. Specifically, the recorded execution time of method m_2 is larger than that in the perfect result. In this case, the results in Table C have an entry that is inaccurate, but are still complete. These simple examples show that accuracy and completeness, according to our definitions, are distinct properties yet are intuitively familiar to any dynamic analysis user.

The distinction between keys and values creates some subtleties. Consider an approximation whose effect is to generate a result with the wrong key (say, to attribute some execution time to the wrong method). This can do one of two things: create a spurious additional entry for a key already present in the result, or create a spurious entry for a key which does not exist in the perfect result. In the first case, we assume the duplicate entry is merged with the colliding (key-identical) entry and becomes an accuracy flaw. In the second case, we have a result that is "over-complete", in that it contains spurious keys. In another case our results might be under-complete, meaning some key is missing (but no key is spurious). Just as values may be accurate, under-approximate, over-approximate or "neither" (neither over- nor under-approximate, e.g., when only a partial order is defined over the values), so the set of keys may be accurate,

[1] If we assume no structure—say, by considering the output as an opaque bit-pattern—it is impossible to distinguish which edits to the bit-pattern (additions, removals, or bit-flips) would affect completeness versus which would affect accuracy.

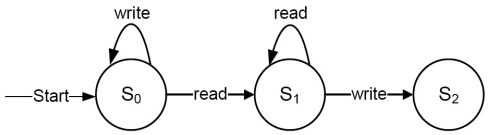

Figure 2. State machine tracking an observed field's stationarity. If the final state is S_0 or S_1, the field is considered stationary. Once the state becomes S_2, the field is determined non-stationary and the instrumentation tracking its status can be removed.

under-complete, over-complete, or otherwise inaccurate (both missing some true keys, and containing some spurious ones).

For the dimension of performance, we do not prescribe any particular definition on its own. Rather, we assume that performance overhead is quantified in some standard way, such as an overhead factor as measured across the whole run of the instrumented program.

2.2 Reasoning about dynamically varying instrumentation

We assume that instrumentation exists to collect data, and this data is used to update the analysis state. Let us assume that our table-based view of analysis results—the "output subset" of the state of a completed analysis—can also be used to view the analysis' state at any point in its execution. This is not unreasonable; for any given analysis run, we can imagine cutting it short at any point in the observed execution trace, and asking what the results would be for the truncated execution. For example, a method-based profiler can be considered as maintaining a table of per-method counts or accumulated time values which are continually updated during execution. In this view, each value in the table is now a *state variable* with a defined state machine. For example, our method profiler has one state variable for each distinct method executed, and each variable's state machine is that of a counter, i.e., the ascending chain of natural numbers starting at zero. Similarly, a simple coverage tool might define one state variable per basic block executed so far, and its state machine is a "ratcheted boolean": it starts at false, may progress to true, and once true, remains in that state.

Suppose that an instrumentation site J exists to update a state variable v. If J can no longer advance the state machine of v into a different state, the instrumentation site J can be removed. For example, once a basic block has been covered, its boolean state variable can never change to a different state, so the instrumentation covering the basic black can be removed. In practice, we often need to consider *sets* of state variables $v \in V$ affected by a given instrumentation site J (e.g., whenever state variables are *per-object*, such as lock sets in a data race detector, since a given instrumentation site can affect many objects).

2.3 Framework architecture overview

Our adaptive instrumentation framework is built on top of the FRANC, a framework for expressing instrumentation-based dynamic analyses on the Java platform. Our system's architecture extends that of FRANC, and its overall architecture is shown in Fig. 3. For details of FRANC, we refer the reader to a previous publication [2]. We note, however, that some of FRANC's unique features are advantageous when extending analyses to use adaptive instrumentation. In particular, FRANC offers specific abstractions for pieces of analysis state. These abstractions are called *mappers* and *updaters*. Updaters can be used naturally to capture state machines of the kind required by SFA. (Meanwhile mappers are used to specify, for example, that these machines are maintained on a per-field classwise basis.)

The architecture is based around two processes, as with FRANC, since instrumentation occurs in a separate process from the observed program (as designed in DiSL[2] for robustness reasons [12]). A key addition in our system is a user-defined controller module in the observed JVM, responsible for monitoring the analysis result and adapting the instrumentation according to some pre-defined strategy. The controller module can be either integrated into the analysis or run in a separate thread with full access to the analysis result. The controller can instruct FRANC to change the scope of the instrumentation and trigger retransformation of existing classes, which is then carried out by FRANC.

2.4 Running example

Having introduced our approach in general terms, we now step through a specific example based on *Stationary Field Analysis* (SFA). Stationary fields represent a generalization of final fields [15] found, e.g., in the Java language. However, unlike final fields, stationary fields can be initialized by multiple writes spanning multiple methods—as long as all the writes to the field happen before all the reads. Knowing what fields are stationary simplifies reasoning about object aliases and opens up opportunities for aggressive compiler optimizations.

SFA associates a per-class and per-instance state variable with every instance field of every class loaded during execution. Each variable represents the state of a simple state machine (see Fig. 2), which tracks the per-class and per-instance status of each field during execution. Once any of the per-instance state variables determine a field to be non-stationary, the per-class status of the field, which is the one we are interested in, cannot change for the rest of the execution.

A notable SFA implementation is the rprof tool developed by Nelson et al. [13]. The tool intercepts field accesses to identify fields that can be considered final or stationary, i.e., fields that are modified once before the constructor method returns and fields that are not modified after they have been read, respectively. To handle potentially billions of events, the original rprof makes use of a map/reduce framework to speed up the analysis and to avoid out-of-memory situations when performed on real-world programs. We have reimplemented

[2] http://disl.ow2.org/

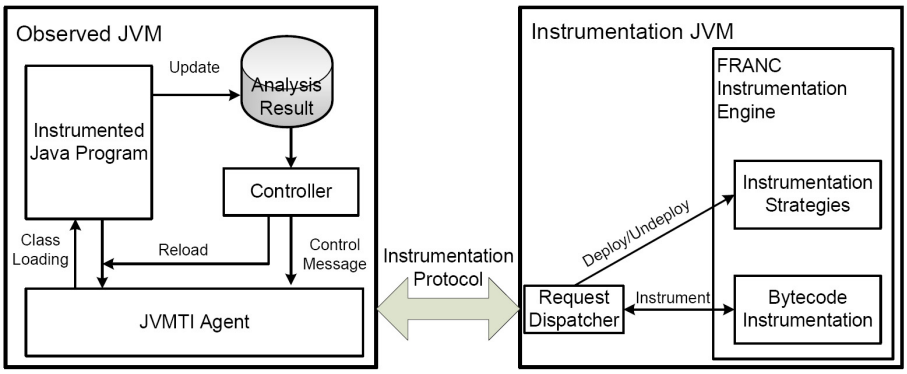

Figure 3. Architecture of our adaptive instrumentation framework

rprof tool using our framework. Instead of using map/reduce to mitigate performance problems, we use a simple in-process analysis and rely on adaptive instrumentation to mitigate the overhead. (We explore the performance of the resulting system in Sect. 3; here we focus on illustrating how the analysis is built.)

Our implementation of SFA can be therefore factored into two parts: a straightforward implementation of SFA using the FRANC framework, and additional logic to control the adaptation of instrumentation over the course of the analysis.

The control logic in this example is straightforward: once a field is determined non-stationary, we can notify the controller to remove all instrumentation intercepting accesses to that field. In the case of instrumentation on hot paths, this will significantly reduce the overhead being imposed on the observed program.

We expect the performance of our system to be much better than a comparable implementation without adaptive instrumentation, since we can gradually eliminate large amounts of instrumentation as the status of more and more fields becomes known. Reduced instrumentation also improves opportunities for dynamic compiler optimizations (reducing method sizes, thus enabling method inlining), further improving performance. To experimentally validate these claims, we examine the performance of our system quantitatively in Sect. 3.

3. Case Study: Stationary Field Analysis (SFA)

We described the application of our approach to SFA in Sect. 2.4. Here, we discuss a quantitative experimental evaluation of our SFA case study. Our goal is to assess the influence of adaptive instrumentation on SFA result completeness and accuracy (as per the table-based model defined in Section 2.1), as well as on analysis performance.

3.1 Experimental setup

We evaluate two variants of our FRANC-based reimplementation of rprof. One is designed to mimic the original rprof tool (referred to as *Fixed SFA*) and only uses fixed[3] instrumentation—this variant serves as the baseline. The other uses adaptive instrumentation (referred to as *Adaptive SFA*), with a controller thread that periodically (every 100 milliseconds) triggers removal of instrumentation for fields that are known to be non-stationary.

To compare the two variants, we run them on the benchmarks from the DaCapo[4] suite. Of the 14 benchmarks, we excluded tradesoap, tradebeans, and tomcat due to well known issues[5] unrelated to our framework, which may cause the benchmarks to fail under expensive instrumentation.

All the benchmarks were run with the largest workload available[6] except for jython, where we used the *standard* workload, because it would not finish within 24 hours with the *huge* workload. All experiments were conducted on a 64-bit multicore platform with Oracle Hotspot Server VM.[7]

3.2 Evaluation results

Like the original rprof tool, our FRANC-based SFA implementations report the stationary status of each field accessed during program execution. We now analyze the results to evaluate the influence of adaptive instrumentation on completeness, accuracy, and performance of SFA. Given the nature of SFA, we expect the adaptive instrumentation to preserve completeness and accuracy, while improving performance. We also expect the adaptive instrumentation to be more effective with long-running analyses, which provide more time for amortizing the adaptation overhead.

[3] FRANC performs load-time instrumentation on all classes.

[4] Release 9.12-bach, `http://www.dacapobench.org/`.

[5] See bug ID 2955469 (hardcoded timeout in tradesoap and tradebeans) and bug ID 2934521 (StackOverflowError in tomcat) in the DaCapo bug tracker at http://sourceforge.net/tracker/?group_id=172498&atid=861957.

[6] *Huge* for all benchmarks except fop and luindex, for which *standard* is the largest.

[7] Dell PowerEdge M620, 1 NUMA node with 64 GB of RAM, Intel Xeon E5-2680 CPU 2.7GHz with 8 cores, CPU frequency scaling and Turbo Mode disabled, Oracle JDK 1.6.0 b43 Hotspot Server VM (64-bit) running on Ubuntu Linux Server 64-bit version 12.04.2 64-bit with kernel 3.5.0-25-generic.

Completeness. With respect to completeness, we expect the *Adaptive SFA* variant of our tool to report the same fields as the *Fixed SFA* variant. Both tool variants were configured to instrument the same code locations in all classes loaded by the JVM. Any differences in the sets of fields reported by the two versions can be attributed either to non-determinism in the benchmarks, or perturbation caused by the framework. In both cases, different (or additional) code paths (or classes) may be executed (or loaded). We note that even in the case of the single-threaded fop benchmark, there are multiple threads executing in the JVM. The activity of these threads (usually in the Java Class Library or in JVM vendor's proprietary classes) will be observed by our tool, because it ensures full bytecode coverage.

To obtain the baseline for comparison, we first collected data from 10 executions (in different JVMs) of a single iteration of each benchmark with *Fixed SFA*. The results are summarized in Table 1. For the completeness evaluation, we are mainly interested in the column showing the total number[8] of fields reported. When analyzing the results, we programatically verified that the reports from all benchmark executions with *Fixed SFA* refer to the same fields. In case of xalan, additional fields[9] were identified, which we attribute to non-determinism in the execution.

The results for *Adaptive SFA* are summarized in Table 2. For clarity, the table only shows differences w.r.t. *Fixed SFA* (see Table 1). We observe minor differences in the number of reported fields compared to the *Fixed SFA*. Looking into the differences, we found that they stem from additional code paths executed in the Java Class Library due to perturbation (class retransformation[10], memory allocation, additional controller thread) caused by our framework. (These two factors affect all bytecode-based instrumentation systems, and mean that small divergences such as these are to be expected.) We thus conclude that our adaptive instrumentation preserves completeness of SFA results.

Accuracy. With respect to accuracy, we generally expect *Adaptive SFA* to classify identical fields the same as *Fixed SFA*. Since the state-machine tracking the field status is identical in both SFA variants, different classification for the same field can only occur when one of the analysis variants observes different execution paths that access the field in a different way. This is not uncommon, especially in the presence of multi-threading in the workloads. Together with the fact that SFA (as a dynamic analysis) tends to yield optimistic results, expecting completely identical results from both analysis variants is unrealistic. We therefore proceed similarly to the above completeness evaluation.

To assess the stability of *Fixed SFA* results, which we use as the baseline, we first collect data from 10 executions of a single iteration of each benchmark with *Fixed SFA*. The results are summarized in Table 1. For the accuracy evaluation, we are interested in the columns showing the numbers[8] of declared-final, undeclared-final, and non-final fields classified either as stationary or non-stationary. We again observe that even the baseline results exhibit minor fluctuations in the classification of some fields. Most notable are the cases of lusearch, pmd, and xalan, where stationary fields fluctuate between undeclared-final and non-final categories. This means that depending on the code path executed, the analysis finds the fields initialized either in the constructor (undeclared-final), or in another method. In several cases, we observed final fields (declared and undeclared) fluctuating between stationary and non-stationary. This is caused by reading a final field before it is first assigned[11], which is legal Java behavior. Overall, the slight fluctuations in the results are consistent with the expectation of SFA producing optimistic results that depend on the code paths executed.

The results for *Adaptive SFA* are again summarized in Table 2. Like with *Fixed SFA*, we observe minor fluctuations between undeclared-final and non-final stationary fields with the lusearch, pmd, and xalan benchmarks. Overall, considering the optimistic nature of SFA, and the comparable scale of fluctuations and their causes present in *Fixed SFA* and *Adaptive SFA* results, we conclude that adaptive instrumentation preserves accuracy of SFA.

Performance. With respect to performance, we generally expect *Adaptive SFA* to perform better than *Fixed SFA*. The adaptive variant of the analysis incurs additional overhead due to class retransformation, which will temporarily undo optimizations performed not only on the retransformed class, but also on methods in other classes containing, e.g., inlined code from the methods in the retransformed class. However, in the long run, the adaptive analysis should incur lower overhead.

To evaluate the performance of both analysis variants, we execute them on the benchmarks from the DaCapo suite, and calculate their overhead compared to uninstrumented execution of the benchmarks. We then calculate the speedup of *Adaptive SFA* with respect to *Fixed SFA*. Since a developer would typically run our tool on her code once, our evaluation targets startup (instead of steady-state) performance. We therefore collect data from a single iteration of each benchmark.

The results of the performance evaluation are summarized in Table 3. As expected, the results for *Adaptive SFA* show a significant speedup compared to *Fixed SFA*—we observe the highest speedup of a factor of 6.36 with xalan, average speedup of a factor of 2.54 (calculated from total execution

[8] The numerical results are meant to identify and convey the extent of fluctuations observed between benchmark executions. However, our conclusions are based on manual analysis of the differences.

[9] All in java.util.concurrent.locks.AbstractQueueSynchronizer class.

[10] Fields such as classRedefinedCount, lastRedefinedCount in java.lang.Class.

[11] A virtual method in a derived class invoked from the constructor of a super class can observe a *final* field in its default state before it is assigned a value in the constructor.

	Total		Stationary Field						Non Stationary Field					
			dF		uF		¬F		dF		uF		¬F	
	Min	Range	Min	Range	Min	Range	Min	Range	Min	Range	Min	Range	Min	Range
avrora	1827	0	725	0	531	0	241	0	4	0	6	0	320	0
batik	3460	0	391	0	1781	0	498	0	0	0	15	0	775	0
eclipse	7262	0	1001	+4	2585	+5	1609	+6	3	+4	16	+1	2037	+3
fop	3507	0	400	0	1497	+2	912	0	0	0	13	0	683	+2
h2	2159	0	363	+2	812	+7	453	+7	6	+2	10	+1	505	+3
jython	2629	0	460	0	1162	0	429	0	0	0	10	0	568	0
luindex	1702	0	328	0	654	+2	246	0	0	0	9	0	463	+2
lusearch	1378	0	213	+5	505	+29	278	+21	6	+5	6	+1	331	+14
pmd	2099	0	361	+5	716	+44	469	+35	11	+5	7	+1	472	+26
sunflow	1741	0	309	0	729	0	287	+11	0	0	7	0	398	+11
xalan	2233	+6	342	+6	658	+82	610	+88	7	+6	9	+2	507	+24

Table 1. Minimal and maximal (reported as range, relative to minimum) counts of fields observed and classified (dF: declared final; uF: undeclared final; ¬F: not final) by *Fixed SFA* in 10 execution of a single iteration of each benchmark, started in separate JVM processes.

	Total		Stationary Field						Non Stationary Field					
			dF		uF		¬F		dF		uF		¬F	
	Min	Range	Min	Range	Min	Range	Min	Range	Min	Range	Min	Range	Min	Range
avrora	0	+6	0	0	-1	+1	0	+6	0	0	0	0	0	+1
batik	0	0	0	0	-1	0	0	0	0	0	0	0	+1	0
eclipse	+5	0	-1	-1	-9	+6	+6	+4	+2	-1	0	-1	+3	0
fop	0	+5	0	0	-1	+1	0	+5	0	0	0	0	0	+1
h2	0	0	+1	0	-7	+8	0	+5	-1	0	0	-1	0	0
jython	0	0	0	0	-1	0	0	0	0	0	0	0	+1	0
luindex	0	0	0	0	0	0	0	0	0	0	0	0	0	0
lusearch	0	0	-2	0	-27	+25	+1	+18	+2	0	0	+1	-2	+11
pmd	+5	0	0	+1	-1	-4	+17	-4	-1	+1	+1	+1	+2	-6
sunflow	-5	0	-2	0	-3	+5	+1	-2	0	0	0	0	-1	-1
xalan	+6	0	-1	0	-12	+13	+14	-6	+1	0	-1	0	-1	+5

Table 2. Difference (w.r.t. Table 1) in minimal and maximal (reported as range, relative to minimum) counts of fields observed and classified (dF: declared final; uF: undeclared final; ¬F: not final) by *Adaptive SFA* in 10 executions of a single iteration of each benchmark, started in separate JVM processes.

	Uninstrumented	*Fixed SFA*		*Adaptive SFA*		
	Exec. time [s]	Exec. time [s]	ovh.	Exec. time [s]	ovh.	Speedup
avrora	14.74	698.33	47.38	307.63	20.87	2.27
batik	5.92	74.77	12.63	66.72	11.27	1.12
eclipse	74.26	8124.28	109.40	3180.25	42.83	2.55
fop	3.07	54.68	17.81	41.68	13.58	1.31
h2	40.72	2352.87	57.78	1047.30	25.72	2.25
jython	14.20	542.11	38.18	295.45	20.81	1.83
luindex	1.52	74.49	49.01	29.46	19.38	2.53
lusearch	2.99	281.07	94.00	75.58	25.28	3.72
pmd	6.80	121.94	17.93	76.86	11.30	1.59
sunflow	4.12	1097.21	266.31	796.78	193.39	1.38
xalan	10.18	2497.54	245.34	392.84	38.59	6.36
Total	178.52	15919.29	89.17	6310.55	35.35	2.52

Table 3. Execution times and overhead factors for *Fixed SFA* and *Adaptive SFA*. Overheads calculated with respect to uninstrumented benchmark execution, speedup for *Adaptive SFA* calculated with respect to *Fixed SFA*.

time), and the smallest speedup of a factor of 1.12 with batik. For the three longest-running benchmarks (eclipse, h2, and avrora—accounting for approx. 72% of the total execution time of the whole uninstrumented benchmark suite), the speedup factor was over 2. This is consistent with our expectation that the adaptive instrumentation will have the biggest impact on long-running analyses. Based on the measurements, we conclude that adaptive instrumentation significantly improves SFA performance, while preserving completeness and accuracy to a large extent.

4. Case Study: Method Execution Time Analysis

Compared to the previous case study, which maintains a per-field state machine to classify the field access pattern, the dynamic analysis in our second case study measures execution time of each method invocation. Hence, in the second case study, measurement perturbation due to inserted instrumentation code influences the analysis results.

To evaluate our approach with this type of analysis, we use a FRANC-based recast of JRat[12], a profiler that measures execution time of each method invocation, and produces a call

[12] http://jrat.sourceforge.net/

graph with execution time statistics, which allows attributing the time spent in a particular method to individual callers. To gather the necessary data, JRat instruments each method entry and exit to collect a time stamp and to update the call graph for the current thread. Our implementation reuses parts of JRat responsible for collecting data and producing traces, and replaces the instrumentation part with a FRANC-based variant to enable adaptive instrumentation at runtime.

The instrumentation required by JRat introduces significant overhead. The overhead is caused both directly by the instrumentation code at each method entry and exit, and indirectly by inflating method bodies and thus limiting inlining and other compiler optimizations. Since the measured method execution time includes the instrumentation overhead, the results tend to over-approximate the actual execution times—a "light" method that just calls many other methods may easily accumulate more execution time than a "heavy" computational method that does not invoke any other methods.

Since the original JRat measures execution times with 1-millisecond resolution, methods with execution time less than the resolution are considered unimportant. However, if such a method is invoked many times in a loop, its execution time will accumulate in the caller, whose execution time may become significant.

Based on this observation, we have implemented an adaptive controller for our implementation of JRat that eliminates instrumentation from unimportant methods at runtime. To identify such methods at runtime, the controller measures execution time of all methods for some time, and for methods that have been executed at least a certain number of times, it marks those with execution time below a certain threshold as unimportant. It then periodically triggers retransformation of the corresponding classes to remove instrumentation from those methods, thus improving tool performance.

This approach has a drawback in that there is a risk of eliminating instrumentation from methods with variable execution times depending on their parameters. This may happen if we fail to obtain a representative sample of execution times for a method during its first few tens or hundreds of invocations. This situation could be alleviated by switching first to a different instrumentation that might sample method execution times selectively, but longer into program execution, and possibly completely eliminating the instrumentation later. For this type of analysis, eliminating instrumentation from a method will inevitably result in less complete results. However, unless it affects methods at the top of the resultant profile, the loss of completeness might not be considered an issue. The reduced instrumentation overhead will also influence the accuracy of the results, yet we would expect this influence to be mainly positive.

To quantify the effects of adaptive instrumentation on the results and performance of a method execution time analysis, we performed an experimental evaluation of our FRANC-based reimplementation of JRat. We now discuss the experimental setup and results.

4.1 Experimental setup

In our experiments, we again evaluate two variants of the tool. One is designed to mimic the original JRat tool as closely as possible (referred to as *Fixed JRat*) and only uses fixed instrumentation. The other uses adaptive instrumentation (referred to as *Adaptive JRat*), with a controller thread that periodically (every 500 milliseconds) triggers removal of instrumentation from unimportant methods. To identify such methods, the controller observes method execution times and looks for methods with execution times below 1 millisecond that have been invoked at least 100 times.

To compare the two variants, we use the benchmarks from the DaCapo suite. The selection of benchmarks and the experimental environment is carried over from the previous case study.

4.2 Evaluation results

Completeness. Unlike in the previous case study, we know that adaptive instrumentation necessarily impairs result completeness. However, as discussed in Sect. 2.1, this need not be detrimental to result accuracy. Our evaluation is therefore aimed at quantifying the impact on completeness, followed by evaluation of accuracy achieved with adaptive instrumentation.

To assess the impact on completeness, we determine the number of code locations considered unimportant by the adaptive instrumentation controller when executing a single iteration of each DaCapo benchmark with *Adaptive JRat*. We contrast this with the total number of code locations instrumented when running the benchmarks with *Fixed JRat*, which serves as the baseline.

The results are summarized in Table 4. We observe that the adaptive instrumentation avoids instrumenting 12-39% of code locations that would normally be instrumented, as shown in the results for *Fixed JRat*. This directly quantifies the decrease in result completeness. However, given the type of analysis, a developer will be mostly interested in several methods at the top of the profile and the loss of completeness will hardly be an issue.

To illustrate the effect of the adaptive instrumentation at runtime, Table 4 also shows the number of analysis invocations that were avoided by eliminating the instrumentation from unimportant methods. We observe that *Adaptive JRat* avoided 35-93% of analysis code invocations.

Accuracy. To evaluate the effect of adaptive instrumentation on accuracy, we compare the profiles produced by our tools to a profile obtained using hprof executing in CPU sampling mode. The sampling profiler introduces minimal overhead, which will cause only minor measurement perturbation. Hence, the hotness profiles produced by the sampling profiler offer good accuracy. Here we measure how closely

	Fixed JRat		*Adaptive JRat*			
	Instr. Locations	Analysis Invocations	Avoided Locations		Avoided Invocations	
avrora	2376	3.97E+09	648	27.27%	2.68E+09	67.47%
batik	5463	5.68E+07	1587	29.05%	3.65E+07	64.21%
eclipse	13440	3.52E+09	5250	39.06%	3.28E+09	93.32%
fop	5162	4.99E+07	1476	28.59%	3.80E+07	76.27%
h2	3492	6.67E+09	1057	30.27%	5.24E+09	78.56%
jython	5453	6.78E+09	1649	30.24%	6.29E+09	92.81%
luindex	2301	1.14E+08	504	21.90%	5.34E+07	46.93%
lusearch	1799	7.57E+08	427	23.74%	2.64E+08	34.94%
pmd	4044	2.52E+08	1233	30.49%	1.68E+08	66.70%
sunflow	2549	4.45E+09	310	12.16%	3.70E+09	83.22%
xalan	3314	4.51E+09	1445	43.60%	3.32E+09	73.58%

Table 4. Number of instrumented code locations and analysis invocations for *Fixed JRat*, followed by the number and percentage of code locations and analysis invocations avoided by *Adaptive JRat*.

	Fixed JRat			*Adaptive JRat*		
	match	sad	mad	match	sad	mad
avrora	13	154	9	13	117	7
batik	11	3983	248	14	3738	233
eclipse	10	10739	671	11	9991	624
fop	15	3638	227	15	3380	211
h2	8	5658	353	8	4764	297
jython	13	3322	207	12	2828	176
luindex	14	1609	100	14	1534	95
lusearch	10	224	14	10	218	13
pmd	11	198	12	11	156	9
sunflow	16	32	2	16	4	0
xalan	8	501	31	8	458	28

Table 5. Distance metrics between the hprof profile and profiles from the *Fixed JRat* and *Adaptive JRat* variants of the JRat tool. Calculated for 16 top-ranking methods in the hprof profile. The metrics include the number of the match methods , the Sum and the Mean of the Absolute Distances.

	Uninstrumented	*Fixed JRat*		*Adaptive JRat*		
	Ex. time [s]	Ex. time [s]	ovh.	Ex. time [s]	ovh.	speedup
avrora	14.74	363.80	24.68	273.10	18.53	1.33
batik	5.92	31.52	5.33	31.15	5.27	1.01
eclipse	74.26	1285.46	17.31	504.10	6.79	2.55
fop	3.07	25.87	8.43	23.68	7.72	1.09
h2	40.72	1098.44	26.98	266.83	6.55	4.12
jython	32.37	1453.87	44.91	255.08	7.88	5.70
luindex	1.52	30.28	19.89	21.25	13.96	1.42
lusearch	2.99	58.76	19.64	47.39	15.84	1.24
pmd	6.80	33.19	4.88	67.58	9.94	0.49
sunflow	4.12	200.15	48.64	47.82	11.62	4.19
xalan	10.18	157.40	15.46	127.56	12.53	1.23
Total	196.69	4738.74	24.09	1665.54	8.47	2.85

Table 6. Execution times and overhead factors for the *Fixed JRat* and *Adaptive JRat* variants of the JRat tool. Overheads calculated with respect to uninstrumented benchmark execution, speedup for *Adaptive JRat* calculated with respect to *Fixed JRat*.

the profiles produced by our versions of JRat approximate the sampling profiles. However, please note that the profiles generated by JRat provide more information than the sampling profiles (otherwise, JRat would be obsolete).

The comparison entails three metrics. We first calculate the cardinality of intersection between sets of 16 methods that have accumulated the most execution time (including time spent in the callees). Then, for the set of 16 top-ranked methods in the hprof profile, we calculate the sum and mean absolute difference between their rankings in the hprof profile and the profiles produced by our tools. We report both metrics, because while the sum of absolute differences better captures the distance between two profiles, the mean absolute difference is the more intuitive of the two.

The results of the comparison are shown in Table 5. The "match" column corresponds to the cardinality of intersection between the sets of 16 top-ranked methods, while the "sad" and "mad" columns correspond to the sum and mean absolute difference in rankings, respectively. We observe that due to its high overhead, the *Fixed JRat* analysis produces results that show considerable distance from the hprof profile. The *Adaptive JRat* analysis produces results that are closer to the hprof baseline, thereby improving result accuracy with respect to the metrics used. Based on the results, we therefore conclude that the adaptive instrumentation slightly improves result accuracy.

Performance. Like in the previous case study, we expect the adaptive instrumentation to improve performance of the

dynamic analysis represented by JRat. Even though this analysis is not as "heavy" as SFA, the reasons for the expected performance improvement remain the same.

To evaluate the performance of our FRANC-based variants of JRat, we execute them on the benchmarks from the DaCapo suite, and calculate their overhead compared to uninstrumented execution of the benchmarks. We then calculate the speedup of *Adaptive JRat* with respect to *Fixed JRat*, which serves as the baseline. Again, to mimic the typical usage of a tool like JRat, we target startup (instead of steady-state) performance. Consequently, we collect data from a single iteration of each benchmark.

The results are summarized in Table 6. As expected, the results for *Adaptive JRat* indicate a significant performance increase over the *Fixed JRat* variant of the analysis. We observe the highest speedup of a factor of 5.70 with jython, average speedup of a factor of 2.85 (calculated from total execution time), and the lowest speedup of a factor of 0.49 (or a slow-down of a factor of 2.04) in case of pmd. The slowdown with pmd was caused by the retransformation overhead associated with the adaptive instrumentation. The overhead grows with the number of cores available to the JVM, and in the case of pmd was not amortized by the relatively short execution.

Overall, the increased performance of *Adaptive JRat* corresponds to the number of analysis invocations avoided by using adaptive instrumentation (see Table 4). Based on the results, we conclude that adaptive instrumentation significantly improves performance of instrumentation-based dynamic method execution-time analysis as represented by JRat. While completeness is reduced, the accuracy is slightly improved compared to the baseline results produced by the *Fixed JRat* variant.

5. Related Work

Our framework is technically an application of the Java code hotswapping technique which allows redefinition of previously loaded classes and thus enables dynamic adaptation of bytecode instrumentation. An early technique, JFluid [8], implements Java code hotswapping by modifying the JVM. Although our framework uses standard JVMTI interfaces in later versions of the JVM, it shares common restrictions of Java code hotswapping: Only method bodies can be modified; fields and methods cannot be added, removed, or renamed; method signatures and the class hierarchy cannot be changed. Nevertheless, the current implementation of our framework is an embodiment of our characterization of dynamic analyses, providing a general view and an operational way of adaptive instrumentation. In contrast to the presented general framework, our previous work on dynamic aspect-oriented programming [1, 16] supported only a limited set of dynamic analyses.

Program profiling is indispensable in code optimizations and performance tuning, but its runtime overhead may be prohibitively high or distort the resulting profiles. Arnold and Ryder [3] propose to reduce instrumentation using counter-based sampling combined with code duplication. Their approach maintains a global counter and starts collecting data when the counter reaches a certain threshold. The data collection is performed on a duplicated version of the original program; code instrumentation concentrates in the duplicated version. Similar to this approach, our adaptation of code instrumentation is triggered by certain runtime events, while we use state machines instead of a global counter. However, since our framework supports runtime undeployment and redeployment, no code duplication is required. Path profiles are particularly useful to path-based optimizations, but path profiling is generally much more expensive than basic block profiling or edge profiling due to the sheer number of paths in non-trivial CFGs. For efficient path profiling, instead of placing instrumentation on both edges of each branch statement, Ball and Larus [4] use a spanning tree of the CFG to select edges for instrumentation. In addition, the edges are assigned certain integer values so that each path computes a unique value by going through its corresponding edges. Sumner et al. [14] extend Ball and Larus' approach to precisely encode calling contexts, by handling recursive contexts and using stack offsets to disambiguate contexts. In these approaches the overhead reduction comes from properties of specific structures (i.e., control flow graphs, call graphs, and stacks) and specially designed algorithms. The places for instrumentation are determined by static analysis prior to program execution. Our framework is orthogonal to such kind of approaches in that it can take effect at runtime to achieve further performance improvement, especially when completeness (or even accuracy) is no longer strictly demanded. More importantly, our framework is generally applicable to different profiling techniques.

In solving software engineering problems, a lot of techniques use sophisticated program instrumentation to control runtime overhead. Liblit et al. [11] design a remote sampling framework to collect execution information from large user communities, in order to help localize and diagnose bugs. Their sampling framework is similar to that proposed by Arnold and Ryder [3] in that both frameworks are based on sampling and code duplication. However, Liblit et al. propose to use numbers from geometric distributions together with weights of acyclic regions to decide the execution of instrumented code. Our framework can be incorporated into this remote sampling framework to remove the necessity of duplicate code and, more importantly, to enable adaptive instrumentation driven by fault-related information only available at runtime.

There is a number of frameworks for runtime monitoring, among which JavaMOP [10] is a full-fledged framework which enables parametric properties and supports multiple logical formalisms to express properties. JavaMOP incorporates knowledge of specific properties and uses static analysis

to avoid creating or retaining unnecessary monitors [7, 9]. Differing from JavaMOP, our framework does not focus on checking runtime properties. Meanwhile, the optimizations in JavaMOP focus on creating fewer monitors or removing useless ones via garbage collection. Our framework is based on hotswapping techniques, thus it can also support optimizations by lazily creating monitors, that is, monitors can be created on-demand and enabled at runtime. It is also worth noting that our framework is built on the top of FRANC so that it is more flexible than AspectJ, which JavaMOP relies on, in terms of choosing locations for program instrumentation.

6. Conclusion

The value of a dynamic analysis largely depends on the quality and utility of the results it provides and the cost associated with obtaining them. While the cost is typically associated with performance, the quality of the results is largely determined by their completeness and accuracy. Dynamic analyses providing the most insight often require denser, more fine-grained instrumentation, which in turn comes at a greater cost in terms of performance overhead. Completeness, accuracy, and performance are therefore the main concerns for both authors and users of dynamic program analyses.

In this paper, we presented a framework that enables runtime adaptation of program instrumentation. This allows reducing the cost of certain dynamic program analyses by removing instrumentation and avoiding execution of analysis code in situations that do not contribute to the analysis results. We evaluated our framework on two case studies with different sensitivity to performance overhead and adaptation requirements, studying the impact of adaptive instrumentation on completeness and accuracy of the results.

In both cases, using adaptive instrumentation resulted in a significant performance improvement. For the analysis insensitive to performance overhead, both completeness and accuracy of the results were preserved. For the analysis where performance overhead directly influenced the results, the performance improvement was traded for reduced completeness, while slightly improving accuracy of the results. While adaptive instrumentation does not come entirely for free, the associated overhead is easily amortized in heavy, long-running dynamic analyses, where it provides the most benefit.

Acknowledgments

This work was supported by the Swiss National Science Foundation (project CRSII2_136225), by a Sino-Swiss Science and Technology Cooperation (SSSTC) Institutional Partnership (project IP04–092010), and by the European Commission (Seventh Framework Programme grant 287746).

References

[1] D. Ansaloni, W. Binder, P. Moret, and A. Villazón. Dynamic aspect-oriented programming in Java: the HotWave experience. In G. T. Leavens, S. Chiba, M. Haupt, K. Ostermann, and E. Wohlstadter, editors, *Transactions on Aspect-Oriented Software Development IX*, pages 92–122. 2012.

[2] D. Ansaloni, S. Kell, Y. Zheng, L. Bulej, W. Binder, and P. Tůma. Enabling modularity and re-use in dynamic program analysis tools for the Java virtual machine. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP '13, pages 352–377, 2013.

[3] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 168–179, 2001.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pages 46–57, 1996.

[5] E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 5–14, 2010.

[6] E. Bodden and K. Havelund. Aspect-oriented race detection in Java. *IEEE Trans. Softw. Eng.*, 36(4):509–527, 2010.

[7] F. Chen, P. O. Meredith, D. Jin, and G. Rosu. Efficient formalism-independent monitoring of parametric properties. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 383–394, 2009.

[8] M. Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *Proceedings of the 4th international workshop on Software and performance*, WOSP '04, pages 139–150, 2004.

[9] D. Jin, P. O. Meredith, D. Griffith, and G. Rosu. Garbage collection for monitoring parametric properties. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 415–424, 2011.

[10] D. Jin, P. O. Meredith, C. Lee, and G. Roşu. JavaMOP: efficient parametric runtime monitoring framework. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1427–1430, 2012.

[11] 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 '03, pages 141–154, 2003.

[12] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. DiSL: a domain-specific language for bytecode instrumentation. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, pages 239–250, 2012.

[13] S. Nelson, D. J. Pearce, and J. Noble. Profiling object initialization for Java. In *Proceedings of the Conference on Runtime Verification*, RV '12, pages 292–307, 2012.

[14] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 525–534, 2010.

[15] C. Unkel and M. S. Lam. Automatic inference of stationary fields: a generalization of Java's final fields. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 183–195, 2008.

[16] A. Villazón, W. Binder, D. Ansaloni, and P. Moret. Advanced runtime adaptation for Java. In *Proceedings of the eighth international conference on Generative programming and component engineering*, GPCE '09, pages 85–94, 2009.