Analysis of Overhead in Dynamic Java Performance Monitoring

Vojtěch Horký, Jaroslav Kotrč, Peter Libič, Petr Tůma

Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics, Charles University
Malostranské náměstí 25, Prague 1, 118 00, Czech Republic

first.last@d3s.mff.cuni.cz

ABSTRACT

In production environments, runtime performance monitoring is often limited to logging of high level events. More detailed measurements, such as method level tracing, tend to be avoided because their overhead can disrupt execution. This limits the information available to developers when solving performance issues at code level.

One approach that reduces the measurement disruptions is dynamic performance monitoring, where the measurement instrumentation is inserted and removed as needed. Such selective monitoring naturally reduces the aggregate overhead, but also introduces transient overhead artefacts related to insertion and removal of instrumentation. We experimentally analyze this overhead in Java, focusing in particular on the measurement accuracy, the character of the transient overhead, and the longevity of the overhead artefacts.

Among other results, we show that dynamic monitoring requires time from seconds to minutes to deliver stable measurements, that the instrumentation can both slow down and speed up the execution, and that the overhead artefacts can persist beyond the monitoring period.

Keywords

performance measurement overhead; dynamic instrumentation; Java

1. INTRODUCTION

Software performance is not only a common term, but also something of a misnomer, because it suggests performance is a property of software. In reality, software performance is a product of executing the software on a particular platform and neither the software nor the platform alone determines performance. This is also one of the reasons why performance monitoring is used – by observing the actual performance, it takes into account the software, the platform and the workload, something that is difficult to do otherwise.

ICPE'16, March 12-18, 2016, Delft, Netherlands

DOI: http://dx.doi.org/10.1145/2851553.2851569

Technically, essential tasks of performance monitoring include data collection and data storage or data processing, or both. These tasks consume resources, giving rise to monitoring overhead. The overhead can easily range from units of percent – for example when monitoring selected methods in an enterprise benchmark application [36] – to orders of magnitude – for example when collecting calling context profile in standard application benchmarks [29]. This is obviously a practically significant factor.

Because the monitoring overhead depends on the amount of data collected, it can be reduced by collecting less data at fewer locations. Particularly interesting is dynamic monitoring, where individual components of the monitoring infrastructure are enabled and disabled, or even inserted and removed, to cater to changing monitoring demands. Dynamic monitoring support exists in many contexts, from operating systems [4, 25, 34] to enterprise application monitoring frameworks [23, 5, 7].

An important influence on dynamic monitoring overhead is exerted by probes – data collection components that are inserted directly into the monitored application. Probes can be inserted either through static instrumentation, which happens before the monitored application is executed, or through dynamic instrumentation, which happens during execution. In the former case, the probe code is always in place and contains support for enabling or disabling data collection. In the latter case, the probe code is simply inserted or removed as needed. Dynamic instrumentation is technically more challenging, because it entails modifying an executing application, but also more attractive, because it carries the implied promise of achieving zero overhead when not collecting data.

In this paper, we focus on dynamic performance monitoring in the context of Java. Starting with version 1.6, Java provides a standard support for changing the code of an executing application through mechanisms called class redefinition and class retransformation. By operating on bytecode, these mechanisms are much more portable than dynamic instrumentation based on machine code manipulation, but also much less transparent where performance overhead is concerned. We address this issue by presenting an extensive overhead study focused particularly on dynamic performance monitoring in Java.

We conduct our overhead study in the broader context of our research on performance awareness. Our general goal is to provide developers with information on software performance that is timely and relevant – that is, presented at a

time and in a manner that makes it useful rather than distracting. Towards that goal, we have implemented a framework capable of both static and dynamic performance monitoring, which we use for example to answer performance related queries in the context of regression testing [3] or to provide performance information in software documentation during development [12]. Here, we therefore analyze the overhead of the framework.

The structure of the paper follows our main contributions. In Section 2, we describe our performance monitoring framework, with focus on dynamic instrumentation as the new feature. Section 3 contributes a detailed analysis of overhead sources specific to dynamic instrumentation. In Section 4, we present the experimental overhead evaluation itself. Related work discussion and concluding remarks close the paper.

2. MEASUREMENT FRAMEWORK

Figure 1 presents a high level architecture of the performance monitoring framework we use throughout this paper. The framework executes in two virtual machines – the data collection components reside in the same JVM as the measured application, the data storage and data processing components use a helper JVM. This helps minimize the framework footprint in the application JVM and provides the possibility of running the helper JVM on a separate host. It also matches the architecture of the underlying instrumentation framework we use, called DiSL [21].

The framework uses the launcher component to perform the necessary initialization and set up the connection between the application JVM and the helper JVM. Once the application executes, the measurement coordination component decides when a measurement should start – depending on circumstances, this can be in response to an interactive developer request, favorable load conditions, or other triggers. The component uses the control connection to deliver the instrumentation request to the application JVM, where the transformation agent fires a class transformation request. The application JVM reacts by asking the DiSL agent to transform the measured class, the DiSL agent in turn uses the DiSL framework in the helper JVM to perform the transformation – which in this case takes the form of inserting probe code. Once the probe code is inserted, it starts feeding measurements to the data transfer component, which uses the data connection to deliver the measurements to the helper JVM for processing. Similar process is used when removing probe code.

Listing 1 provides a compact pseudocode listing of the probe code. The code simply collects the time at the entry to and the exit from the measured method – the somewhat more complicated listing is due to the need to handle recursion. When the probe is called recursively, only the top level iteration is measured. The probe state is thread local, implemented using efficient thread local variables offered by DiSL. This minimizes synchronization.

We omit other elements of the framework, which are not essential for the purpose of this paper. These include the ability to differentiate between invocations of the same method based on the actual argument values, and the applications for regression testing and documentation generation. For more details on the performance regression testing features, refer to [3], for performance documentation generation

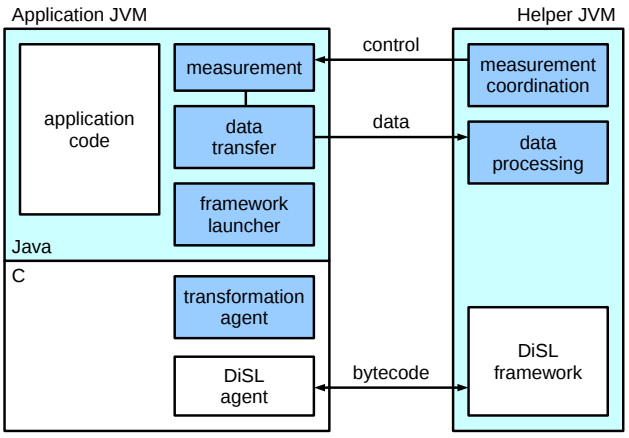

Figure 1: *High level architecture of the dynamic performance monitoring framework.*

features, refer to [12]. The framework is available as open source at http://d3s.mff.cuni.cz/software/spl.

3. OVERHEAD SOURCE ANALYSIS

A characteristic feature of contemporary computing platforms is the potential for complex interactions across multiple levels of the hardware and software stack. Dynamic measurement instrumentation influences these interactions in many ways, with the collective impact on performance forming the observed measurement overhead. Here, we discuss the sources of measurement overhead relevant to Java-like platforms – that is, platforms with applications written in a high level language, garbage collected memory, dynamic class loading and just-in-time (JIT) compilation. The discussion steers mostly clear of technical detail, available in platform-specific sources such as [14].

3.1 Probe Presence

The instrumentation inserts probes directly into the application, to be executed just before and just after the measured application code. The probe code consumes processor resources just as the application code does, introducing execution overhead.

As illustrated on Listing 1, the probe code samples time. The part of the probe code situated between the sampling points and the application code of interest will be measured together, introducing systematic measurement error. With some simplifications, the error is likely to be additive and can be possibly compensated by calibration. The remaining probe code, which resides outside the sampling points, is not measured but still counts towards the application execution time.

The systematic measurement error can grow when the measured application code is called recursively. When this is the case, the error accumulates with the depth of the recursion and, except for the top level iteration, includes the entire probe code rather than just the part of the probe between the sampling points. A similar situation arises when multiple instrumented methods call each other.

When examined in detail, the execution overhead is further influenced by interactions inside the processor microarchitecture. The probe code may or may not cause or suffer

```
▷ Thread local variable
recursion : map String to integer

▷ Invocation local variables
entryTime : integer
exitTime : integer
name : String

advice at method entry
    ▷ Call is converted to constant by DiSL
    ▷ at class loading (weaving) time
    name ← GetCurrentMethodName
    Increment(recursion[name])
    ▷ Time sampled close to real entry moment
    entryTime ← GetCurrentTime
end advice

advice at method exit
    ▷ Time sampled close to real exit moment
    exitTime ← GetCurrentTime
    name ← GetCurrentMethodName
    Decrement(recursion[name])
    if at top level of recursion then
        SendMeasurement(name, entryTime, exitTime)
    end if
end advice
```

Listing 1: *Probe pseudocode for dynamic instrumentation.*

relatively expensive events such as cache misses or branch prediction failures, whose occurrence depends on the interaction with the surrounding application code. In principle, applications that are particularly tightly tuned to the processor microarchitecture – such as numerical applications that rely on tiling to efficiently utilize caches [26] – may be disrupted significantly, however, such tight tuning is not common on platforms that do not expose memory layout to applications.

With both the application and the probe written in a high level language, control over the execution overhead is somewhat limited. Still, it is possible to minimize the overhead by structuring the probe code so that the sampling points are close to the measured application code and by avoiding potentially expensive constructs such as synchronization or polymorphic invocations. Ultimately, the overhead determines practical measurement granularity – if the overall disruption to application execution is to be reasonable, the measured application code should execute orders of magnitude longer than the probe code.

3.2 Code Manipulation

The code manipulation associated with inserting and removing probes also consumes resources. The instrumentation needs to parse the application class to be measured, insert the probe code, and have the virtual machine load the instrumented application class. In general, these are operations that are about as disruptive as other class loading activity.

As an important consequence, class manipulation during instrumentation may trigger JIT compilation. If some methods of the class were JIT compiled before instrumentation, then these compiled versions are discarded, and may be JIT compiled again after instrumentation. The impact may extend to methods of other classes whose compiled versions depend on the instrumented application class, leading to cascades of JIT compilations that reflect prior inlining decisions.

Depending on circumstances, the virtual machine may initiate JIT compilation immediately after loading the instrumented application class, at some later time, or even never. Until the JIT compilation completes, those methods whose compiled versions were discarded can execute less efficiently or even block, again in effect contributing to overhead. In general, it is not possible to tell whether some future JIT compilation will deliver a more efficient compiled version of a method, it is therefore not possible to minimize the impact on measurement simply by waiting for the compiled version. It is, however, possible to wait for JIT compilations that immediately follow instrumentation to finish – those JIT compilations should cover most hot code, where the impact on measurement is also most likely significant.

3.3 Code Optimization

The JIT compilation involves optimization decisions that may change with instrumentation. This is true even when the interaction between the probe code and the application code is kept to a minimum – most importantly, the very presence of the probe code influences the heuristics that drive method inlining. Although these heuristics may vary, they are likely to include a limit on the size of the inlined method. Inserting probe code increases code size and therefore reduces the chance of the measured methods being inlined.

Method inlining is an important optimization because it impacts the scope of most other optimizations – with JIT compilation working on methods as compilation and optimization units, inlining one method into another means the caller and the callee are optimized together. The impact of inlining on performance experiments was demonstrated in detail with JMH benchmarks [24] that can selectively disable method inlining to prevent interaction between the benchmark harness and the measured method [31]. In very general terms, we can assume that by reducing the chance of inlining, instrumentation reduces the opportunity for optimization. We can therefore expect instrumentation to introduce another systematic measurement error, due to observing possibly less optimized versions of the measured methods. Keeping probe code small, however, should make this error less likely.

The optimization decisions made during JIT compilation also depend on past application execution. Factors such as method invocation count, loop iteration count, or type variability are taken into account – it is therefore not guaranteed that the same method will be compiled in the same way at different moments in application execution. In particular, it is not guaranteed that a method will have the same compiled version after removing probes as it had before inserting probes.

3.4 Other Overhead Sources

Significant sources of measurement overhead are also associated with data storage. Whatever data a probe collects or aggregates needs to be stored in memory and then exported outside the measured application. The memory storage overhead begins with allocation – when using the application heap, additional allocations will either cause the heap

to expand or the garbage collector to run more often [16, 17]. After allocation, storing data in memory consumes additional memory bandwidth, and export similarly incurs additional storage or network bandwidth. The magnitude of these effects grows with the data volume.

Because there is no principal difference between data storage overhead coming from dynamic measurement instrumentation and similar overhead from standard instrumentation or even application I/O, we do not analyze this overhead source further. A thorough analysis of the export overhead and the related measurement framework implementation issues can be found in [37].

4. EXPERIMENTAL EVALUATION

The analysis of potential overhead sources associated with dynamic measurement instrumentation directly translates into questions we want to answer using experimental evaluation:

Q1. Given that some part of the probe code is necessarily situated between the sampling points and the measured application code, what is the typical difference between the measured execution time and the actual execution time ?

Q2. Does the difference between the measured execution time and the actual execution time remain stable ?

Q3. Given that the measured code is potentially interacting with the probe code through code optimization decisions and other channels, is the execution time of the measured code different from the execution time with no measurement ?

Q4. Given that the measured code is potentially compiled differently before and after measurement, does the execution time after measurement differ from the execution time before measurement ?

Q5. What is the typical duration of JIT compilation associated with dynamic measurement instrumentation ?

For completeness, we also want to answer the ever present question associated with instrumentation, even if there is no reason why the result should be significantly different from other instrumentation overhead studies:

Q6. What is the total overhead in terms of application performance that can be attributed to dynamic measurement instrumentation ?

4.1 Overall Design

To answer the overhead related questions, we need to observe an application both with and without dynamic measurement instrumentation in place – in other words, we need independent observation capabilities that exist alongside the dynamic instrumentation. We employ static instrumentation deployed throughout the measured application to perform continuous measurement. From the perspective of the dynamic instrumentation, the static instrumentation is just a part of the measured application that provides baseline measurements, as outlined in Figure 2. Compared to Figure 1, the application is now augmented with the static probe code, which relies on the static measurement agent

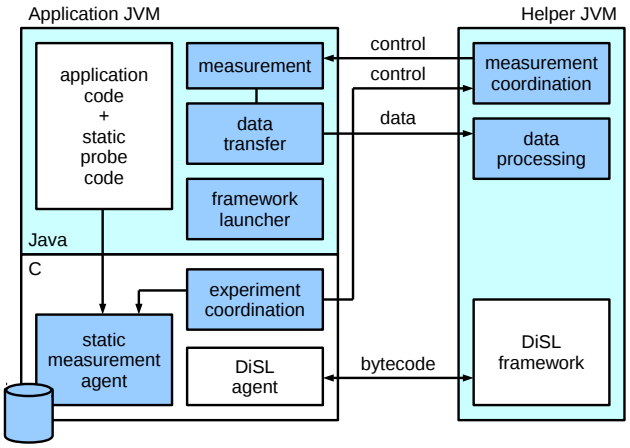

Figure 2: *High level architecture of the experiment.*

to record the baseline measurements in local storage. An experiment coordination component is introduced to direct when a measurement should start and stop, but the dynamic instrumentation remains otherwise unchanged.

With both static and dynamic instrumentation available, we pretend that we perform dynamic measurements on an application that also produces baseline measurements for comparison. We structure the experiment to model a situation where a developer chooses to observe the execution time of an arbitrary application method using dynamic instrumentation, and use the static instrumentation to measure the overhead associated with the dynamic instrumentation. To collect a representative sample, we repeatedly choose the observed methods at random. In individual steps, outlined in Figure 3, the experiment proceeds as follows:

S1. Before launch, we use static instrumentation to augment the application. The statically instrumented application continuously reports the execution times of all methods considered in the experiment and the processor utilization.

S2. We launch the application and wait for the warmup period to pass before commencing measurement.

S3. We measure and record the execution time of all methods considered in the experiment using the static instrumentation. This data describes the performance before all dynamic measurements.

S4. We choose one of the methods considered in the experiment at random to be the observed method. We use dynamic instrumentation to insert the probe code at the start and the end of the observed method and measure the time it takes the JIT compilation associated with the code manipulation to complete.

S5. At all times between inserting and removing the probe code, we measure and record the execution time of the observed method using the dynamic instrumentation. This data is the dynamic measurement sample.

S6. We measure and record the execution time of all methods considered in the experiment using the static instrumentation. This data describes the performance during dynamic measurement.

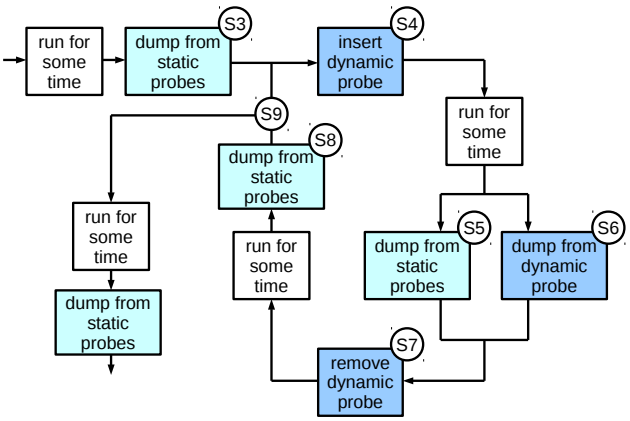

Figure 3: *Control flow of the experiment.*

S7. We use dynamic instrumentation to remove the probe code at the start and the end of the observed method and measure the time it takes the JIT compilation associated with the code manipulation to complete.

S8. We measure and record the execution time of all methods considered in the experiment using the static instrumentation. This data describes the performance after dynamic measurement.

S9. We continue with step S4 until enough methods are observed.

S10. Finally, we again measure and record the execution time of all methods considered in the experiment using the static instrumentation. This data describes the performance after all dynamic measurements.

The individual steps provide data to answer the overhead related questions – by comparing the measurements from steps S5 and S6, we evaluate the dynamic measurement accuracy ; relating the measurements from steps S6 and S8 reveals the dynamic measurement overhead ; comparing steps S3 and S10 identifies any permanent performance changes due to inserting and removing probe code, and so on.

4.2 Technical Specifics

The complete experiment implementation and configuration is available as open source, as is the performance monitoring framework. Here, we provide selected technical details necessary for interpreting the experiment results.

4.2.1 Static Probes

The static instrumentation is implemented independently of the dynamic measurement framework. AspectJ™ [1] is used to insert the probe code in the form of a `before` advice and an `after` advice. Both pieces of advice consist of a single JNI call to the actual probe code implemented natively, with statically assigned integer method identifier as the only argument. Listing 2 provides a compact pseudocode listing.

Implementing most of the static instrumentation natively provides more technical advantages including dynamic memory allocation independent of the application heap. We also obtain access to JVM state information, such as notifications about JIT compilation.

```
▷ Thread local variable holding execution times of
▷ individual methods
data : array of record
  ▷ Individual samples
  samples : array of record
    generation : integer
    entryTime : integer
    exitTime : integer
  end record
  ▷ Index into samples
  nextFree : integer
  ▷ Pending call data
  recursion : integer
  entryTime : integer
end record

procedure StartMeasurementJNI(id : integer)
  ▷ Time sampled close to what dynamic probe perceives
  ▷ as real entry
  now ← GetCurrentTime
  Increment(data[id].recursion)
  if at top level of recursion then
    data[id].entryTime ← now
  end if
end procedure

procedure EndMeasurementJNI(id : integer)
  Decrement(data[id].recursion)
  if not at top level of recursion then
    return
  end if
  i ← data[id].nextFree
  ▷ Generation indicates data validity
  Increment(data[id].samples[i].generation)
  data[id].samples[i].entryTime ← data[id].entryTime
  ▷ Time sampled close to what dynamic probe perceives
  ▷ as real exit
  data[id].samples[i].exitTime ← GetCurrentTime
  Increment(data[id].samples[index].generation)
  ▷ Overwrite oldest data if necessary
  ModuloIncrement(data[id].nextFree, 256)
end procedure
```

Listing 2: *Probe pseudocode for static instrumentation.*

To avoid excessive synchronization between the probe code and the static measurement agent, which records the measurements in thread local storage, we use versioning as in sequential locks [2]. Each measurement updates the generation counter twice, the agent records only measurements whose generation counter was odd and the same both before and after access. We also take care to use the same clock source in both the static and the dynamic instrumentation (`clock_gettime` with `CLOCK_MONOTONIC`). This makes it possible to pair the static and the dynamic measurement of the same method invocation, which is used in some parts of the evaluation.

Keeping the Java part of the static probe code as simple as possible is essential to preserve a realistic interaction between the application code and the dynamic probe code that the experiment examines. We note that AspectJ™ does not simply inline the JNI call at the method entry and method

exit points, but uses a somewhat more complex invocation sequence that first locates the (singleton) aspect and then invokes the aspect method which contains the JNI call. The code involves monomorphic invocation and predictable conditional branching, which should optimize reasonably well. The use of JNI carries some overhead as well [10].

4.2.2 Measured Application

Because the potential overhead sources depend on interaction between the application code, the probe code, and the execution platform, we need to conduct the experiment in a reasonably realistic context. We have chosen the SPECjbb2015™ benchmark [33], a Java server business benchmark that approximates a business information system of a supermarket company.

In the experiment, we consider methods that reside in the main JAR file of the benchmark as methods that the developer of the application would be likely to observe. As a technical necessity, we omit methods of anonymous classes, which cannot be selected by static instrumentation pointcuts. This leaves us with 5628 statically instrumented methods in 957 classes. For the dynamic instrumentation, we select methods that are invoked frequently enough to provide some data in 60 s of measurement. To do this, we run the benchmark with static instrumentation for 40 min and select methods called at least 100 times in the last 10 min. This yields 1286 methods.

4.2.3 Workload Generation

The SPECjbb2015™ benchmark uses an elaborate workload generation mechanism that first identifies the request rate bounds and then generates requests with gradually increasing rate to identify the benchmark score. For our experiment, the changing workload is not practical because individual measurements would be collected at different request rates – we therefore execute the benchmark with a fixed request rate. We choose the rate to be close enough to maximum rate to maintain high utilization, because that is where the instrumentation overhead is easily visible, but low enough to make overload situations rare. On the experiment platform, this is 4000 req/s.

The workload generation mechanism of SPECjbb2015™ implements an open workload model, where individual requests arrive at the configured rate regardless of the request processing speed (except for overload situations, which are detected and reported). Hence, the instrumentation overhead does not necessarily translate to changes in request rate – instead, the processor utilization rises so that the configured request rate can be maintained. Similarly, request queueing and thread scheduling effects may mask changes in response time [22]. We therefore monitor changes in processor utilization as an indication of instrumentation overhead.

Technically, we monitor processor utilization using the processor accounting subsystem of the process control group associated with the application JVM running the benchmark. This provides accurate information at nanosecond granularity, which we express as percentage of full utilization – 0 % means no processor was executing the application JVM threads in the measurement period, 100 % means all processors were exclusively executing the application JVM threads in the measurement period.

4.3 Experiment Platform

We perform the measurements on an Intel Xeon machine with 32 logical processors (E5-2660, two packages, 8 cores per package, 2 hardware threads per core). The processors are running at 2.2 GHz, the frequency is fixed for all measurements because frequency scaling and turbo boost would otherwise distort the processor utilization measurements that we use as an indication of the instrumentation overhead. The operating system is Fedora 20, 64 bit kernel 3.19.8, OpenJDK 1.7.0-79, AspectJ 1.8.6, DiSL 1.0.

The machine has 48 GB RAM in 2 NUMA nodes. We use the default configuration for heap size and force a garbage collection cycle before each processor utilization measurement to avoid including garbage collection in data intended to characterize instrumentation overhead. As a consequence, JVM arrives at a stable heap size of less than 5 GB that reflects the allocation rate between the utilization measurements.

We use a 5 min warmup period before collecting measurements, taking care to also exercise probe code during warmup, and restart the experiment every 2 h to randomize the initial conditions [13]. Figure 4 shows the initial processor utilization, indicating that by the end of the 5 min period, the benchmark execution is stable, the same is indicated by the JIT log.

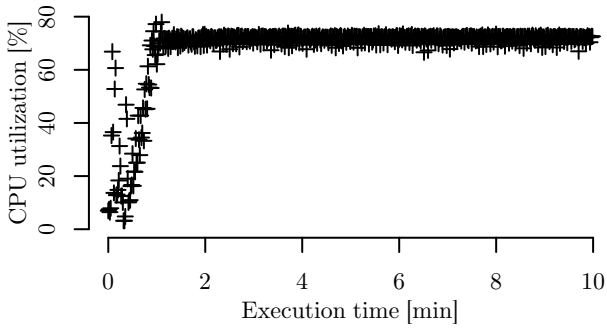

Figure 4: *Processor utilization during warmup.*

In the experiment steps that collect measurements using the static instrumentation – S3, S6, S8 and S10 – we collect the processor utilization for 30 s and the execution time of all methods for 60 s with cyclic buffers of 256 elements per thread. In the steps that wait for the JIT compilation to complete – S4 and S7 – we consider the JIT compilation complete when no new compiled method appears for 20 s, with a timeout of 60 s. We also insert a random delay of 30 s to 90 s in step S9 to prevent inadvertent synchronization between the experiment and the application.

4.4 Measurement Results

We examine the measurement results in the same order as the overhead questions. Question Q1 deals with the measurement accuracy, that is, the difference between the measured and the actual time. Figure 5 answers with a distribution of the average difference between the time reported using the static and the dynamic instrumentation in steps S5 and S6. In numbers, the minimum average difference was observed to be 76 ns, the median was 1.34 µs, the maximum was 166.09 ms.

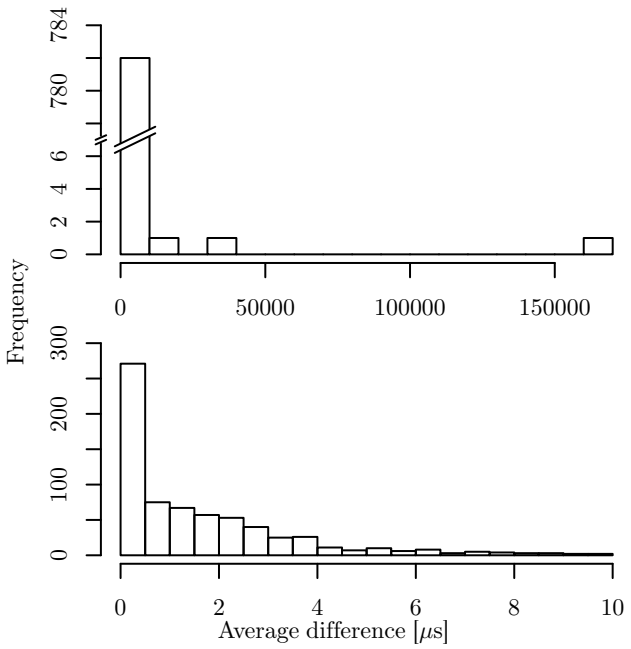

Figure 5: *Difference between measurements reported by static and dynamic instrumentation. We (1) pair static and dynamic measurements of the same invocation, (2) compute paired difference, (3) compute average difference per method, (4) plot distribution of the averages.*

Note broken scale in the top plot, the bottom plot provides a zoom in view. Results closer to zero indicate more accurate dynamic measurement, but not necessarily zero disruption due to dynamic measurement, which is examined later.

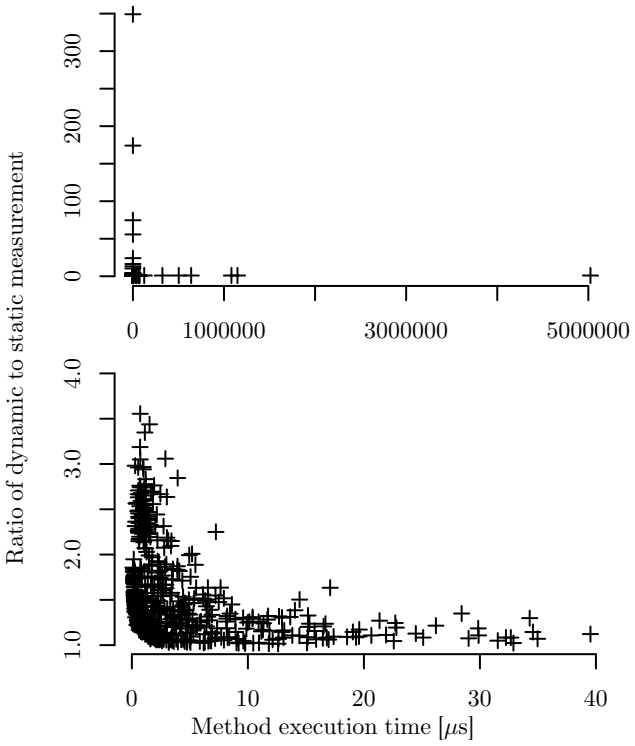

Figure 6: *Ratio between measurements reported by static and dynamic instrumentation. We (1) pair static and dynamic measurements of the same invocation, (2) compute ratio of dynamic to static measurement, (3) compute geometric average ratio per method, (4) compute average static measurement per method, (5) plot the averages of ratios relative to the averages of static measurements.*

The bottom plot provides a zoom in view. Results closer to one indicate relatively more accurate dynamic measurement, helping to identify the minimum method execution time where the relative accuracy is acceptable.

Figure 6 offers an alternative view, plotting the average ratio between the time reported using the static and the dynamic instrumentation, relative to the method execution time. The figure suggests that the relative measurement accuracy sharply declines for methods shorter than about 10 µs to 20 µs.

Figures 5 and 6 also relate to question Q2. The interquartile range of average differences is 3.33 µs, more than two times the median difference. This suggests the overhead is far from stable and therefore not easy to compensate by subtracting the average difference. We have also used one-way ANOVA to decide whether the choice of the measured method is an important factor. When ignoring the few methods with average difference over 10 µs, ANOVA returns p close to 1, suggesting that the variability does not depend on which method is measured.

Questions Q1 and Q2 concern different observations of the same invocations. In contrast, the remaining questions concern observations of different invocations, we can therefore only talk about effects on average behavior. As a consequence, outliers and fluctuations have more influence over the results. To compensate for outliers, we compute averages after discarding 2.5 % of the smallest and 2.5 % of the largest measurements for each method.

Figure 7 is related to question Q3, examining the effect of repeated JIT compilation on dynamic measurement. The figure shows how the average method execution time, as measured by the static instrumentation, changes during dynamic measurement. Although the median rate of 1.014 indicates an intuitively reasonable small slow-down, the variability is again large, with 25 % of methods exhibiting a slow-down of more than 1.23, and, more surprisingly, 25 % of methods exhibiting a speed-up of more than 0.84. To distinguish the effects of instrumentation from normal execution time variability, we employ statistical testing with t-test – the slow-down is statistically significant at $\alpha = 0.05$ for 13.2 % of methods, and the speed-up for 18.0 % of them.

To provide more detail, Figure 8 shows a typical behaviour during dynamic measurement, from initiating the measurement in step S4 to concluding the measurement in step S8. Upon inserting the probe code, the method execution time jumps up because the compiled version of the instrumented class, and possibly other related methods, is discarded. Soon after that, the executed code is compiled again and the performance returns to normal levels. Similar behavior appears upon removing the probe code. The few other outliers that are visible throughout the measurements appear at random

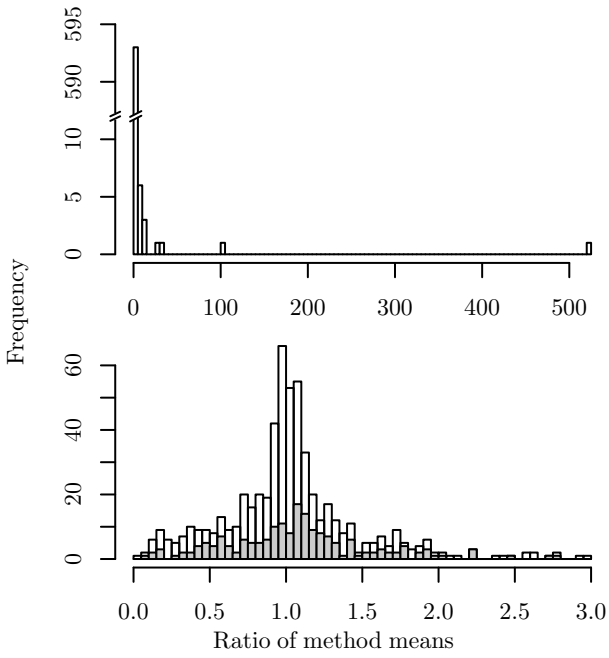

Figure 7: *Ratio between measurements reported by static instrumentation on methods during and after dynamic instrumentation. We (1) compute average static measurement per method from step S6, (2) compute average static measurement per method from step S8, (3) compute ratio of the dynamic average to the static average, (4) plot distribution of the ratios.*

Note broken scale in the top plot, the bottom plot provides a zoom in view. The gray bars denote statistically significant differences at $\alpha = 0.05$. Results smaller than one indicate methods that run faster when instrumented and vice versa.

and are probably not due to instrumentation. We have observed this behavior with most methods.

Figure 9 is related to question Q4, examining the effect of repeated JIT compilation on application outside measurement. Here, the figure shows how the average method execution time changes from near the start to near the end of the benchmark, with dynamic measurement performed in between. The median rate of 0.994 indicates a reasonably stable performance, however, at the end of the benchmark 25 % of methods are slower by a factor of over 1.12, and 25 % of methods are faster by a factor of over 0.87. The slow-down is statistically significant at $\alpha = 0.05$ for 8.9 % of methods, and the speed-up for 15.0 % of methods.

To determine whether the changes of method execution time in Figure 9 are due to dynamic measurement, Figure 10 shows how the average method execution time changes from near the start to near the end of the benchmark when no dynamic measurement is done. The median rate of 1.003, as well as the lower and upper quartiles of 0.76 and 1.16, are similar, however, the extreme values are further apart in Figure 9 than in Figure 10. We conclude that although the benchmark exhibits long term changes in the average method execution time all by itself, dynamic measurement increases the magnitude of the most extreme changes. When no dynamic measurement is done, the slow-down is statistically significant at $\alpha = 0.05$ for 12.3 % of methods, and the speed-up for 16.2 % of methods.

By observing JIT compilation in steps S4 and S7, we also obtain statistics on the temporary disruptions due to code manipulation. As shown on Figure 11, JIT compilation takes more than 6.4 s to complete in 50 % of the probe insertion operations, and more than 3.7 s to complete in 50 % of the probe removal operations. Between 1 % and 2 % of code manipulation operations kept JIT compilation active for more than 60 s.

We conclude with Figure 12, which provides an answer to question Q6 about the total overhead associated with dynamic measurement instrumentation. The figure plots the distribution of processor utilization without dynamic instrumentation, observed in step S6, and the distribution of processor utilization with dynamic instrumentation, observed in step S8. Both cases are very similar, confirming earlier findings that small scale instrumentation does not incur significant overhead – in fact, the average utilization is 73.03 % without dynamic instrumentation and 72.17 % with dynamic instrumentation, with the difference statistically significant at $\alpha = 0.05$.

4.5 Threats To Validity

We close our results with discussing threats to validity. We focus on the threats to statistical validity, internal validity and external validity as the most relevant validity categories.

4.5.1 Statistical Validity

To guard against threats to statistical validity, we report detailed statistical properties alongside summary results. We also provide complete data at http://d3s.mff.cuni.cz/resources/icpe2016.

The statistical analysis is complicated by the fact that, for reasons inherent to the SPECjbb2015™ benchmark implementation, the individual observations of the method execution times are not necessarily independent. As a particular consequence, if too many methods exhibit sufficiently large phases in behavior, then the conclusions on the statistical significance of the results may be distorted due to observing method behavior in different phases.

4.5.2 Internal Validity

When examining internal validity, we are concerned with the possibility that the observed overhead is not due to dynamic instrumentation, and the possibility that the dynamic instrumentation introduces overhead that is not observed. Here, most dangerous are effects that can synchronize with dynamic measurement, because such effects can introduce a systematic error when measuring the overhead. We believe such systematic synchronization is unlikely, because we randomize both the choice of the measured method and the delay between measurements. Effects due to events inherent to dynamic measurement, such as dynamic code manipulation, are obviously part of the overhead by definition.

Measuring the total overhead as a change in processor utilization similarly ensures we observe all processor overhead. The benchmark is configured to perform a constant amount of work per unit of time, anything that changes the processor demand per unit of work is bound to change the processor utilization. This deserves some attention – while

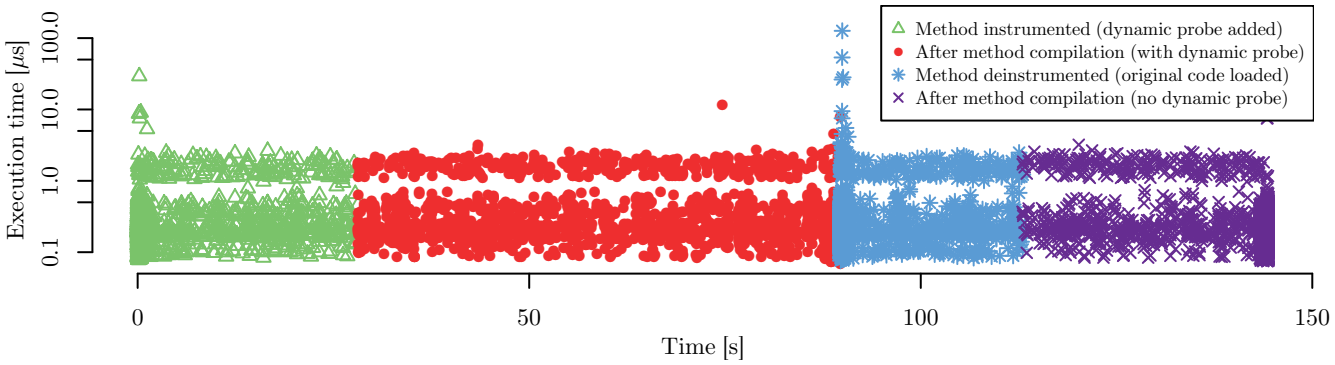

Figure 8: *Detailed measurement for the* `getArray` *method of the* `Data` *class in* `transport` *package. Method picked to demonstrate typical behavior. Note logarithmic scale.*

the benchmark does maintain a stable request rate, brief periods of increased overhead are likely to be compensated by queueing inside the benchmark. Because we perform every dynamic measurement for more than a minute, we believe we are likely to exhaust any queues that might mask the measurement overhead entirely.

As noted, we force a garbage collection cycle before each processor utilization measurement, and therefore influence the garbage collection overhead. Because utilization measurements happen much less frequently than young garbage collection cycles, we are not likely to influence the young collection overhead directly. We do make the full collection cycles more frequent, with multiple consequences – the total time spent in full collections is likely to be longer and the young collections may become more efficient because the references between generations are more likely to be live [17]. We believe this influence to be minor because no dynamic measurement instrumentation is likely to keep significant amounts of live data on the application heap for long, and the young collection overhead – which we are less likely to influence – should therefore dominate.

4.5.3 External Validity

External validity is concerned with how much the observed overhead generalizes to other dynamic instrumentation frameworks, other applications and other platforms. Much of the dynamic instrumentation framework revolves around the ability to redefine and retransform classes, frameworks that use the same mechanism are therefore likely to induce the same overhead due to code manipulation and code optimization. We note that this is pretty much the only reasonably portable dynamic instrumentation method currently available for Java, differences therefore should not be big.

Other dynamic instrumentation frameworks can also differ in their data storage and data processing implementation. There are many ways how this implementation can be optimized [37], we believe our implementation is reasonably straightforward to keep the results comparable with other probes written in Java.

To generalize to other applications, we must ask how much our measured application resembles other applications in those features that are relevant to dynamic instrumentation. Assuming the SPECjbb2015™ benchmark is reasonably representative, we have to account for the differences introduced by static instrumentation:

- The instrumentation slows the benchmark down roughly by a factor of four. The effect is somewhat similar to using a slower platform, but the overhead is not distributed evenly – by adding similar overhead to each method, we slow down shorter methods more than longer ones in relative terms. With the measured application becoming faster, the dynamic instrumentation overhead will become relatively smaller.
- The instrumentation increases the size of all methods by a small constant amount, making compilation and inlining somewhat less likely. Examining the JIT log, we see a total of 520 kB in 24 k inlined methods and 7 k failed inline attempts for the original benchmark, and a total of 750 kB in 36 k inlined methods and 21 k failed inline attempts for the benchmark with static instrumentation.
- The instrumentation inserts JNI calls, whose impact on compiler behavior may depend on subtle memory model implementation details [15]. Hypothetically, JNI calls may require optimization barriers, leading to more conservative optimization of the measured application. We have not included a specific evaluation of this possibility into our experiment.

Given the platform specific character of our experiment, we do not make any specific claims outside our platform. We believe the platform is representative enough to account for a large percentage of existing systems, however, different platforms – especially different JVM implementations – may behave in an arbitrary manner, yielding entirely different dynamic instrumentation overhead.

5. RELATED WORK

Instrumentation overhead is an obvious concern for any measurement framework. Instrumentation can interact with the measured system, making the measured performance different from the performance exhibited otherwise. This problem is carefully explained by Malony in [18] – in this sense our work is an experimental study of performance intrusion and performance perturbation due to dynamic instrumentation in Java.

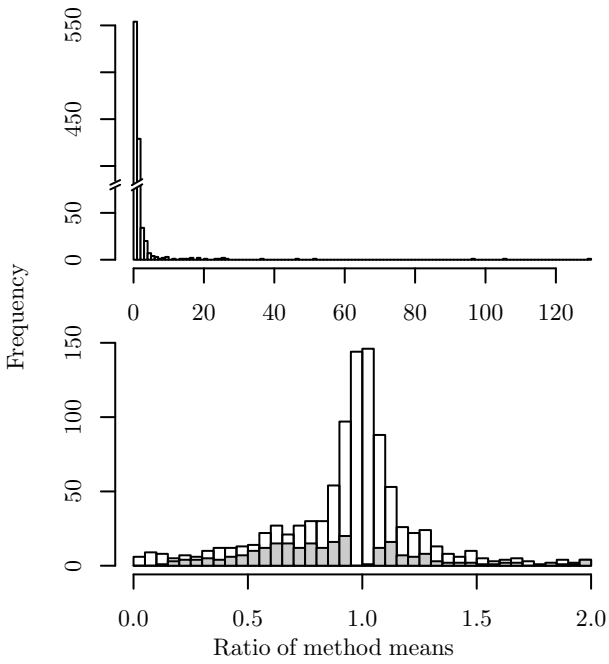

Figure 9: *Ratio between measurements reported by static instrumentation on methods before and after dynamic instrumentation. We (1) compute average static measurement per method from step S3, (2) compute average static measurement per method from step S10, (3) compute ratio of the initial average to the final average, (4) plot distribution of the ratios.*

Note broken scale in the top plot, the bottom plot provides a zoom in view. The gray bars denote statistically significant differences at $\alpha = 0.05$. Results smaller than one indicate methods that run faster at experiment startup than teardown and vice versa.

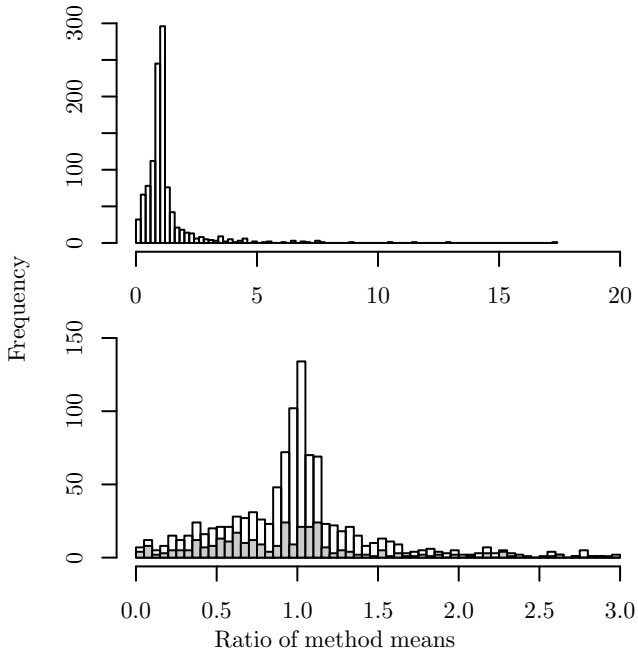

Figure 10: *Ratio between measurements reported by static instrumentation on methods at benchmark startup and before benchmark teardown without dynamic instrumentation. We (1) compute average static measurement per method from step S3, (2) compute average static measurement per method from step S10, (3) compute ratio of the initial average to the final average, (4) plot distribution of the ratios.*

Note broken scale in the top plot, the bottom plot provides a zoom in view. The gray bars denote statistically significant differences at $\alpha = 0.05$. Results smaller than one indicate methods that run faster at experiment startup than teardown and vice versa.

Malony and Shende have investigated the measurement overhead issues especially in the context of the Tau Performance System [30]. In [19], they describe a method for compensating the measurement overhead by subtracting the execution time added by the instrumentation from the individual measurements. The method assumes the computation is calibrated for particular application and platform. Our experiment is a case of such calibration that highlights the limits of accuracy in a system where the overhead of the same probe code can vary depending on the measured method, the call site, or even ephemeral compilation decisions. Our experiment extends the overhead investigation towards dynamic instrumentation, Tau focuses on more heterogeneous platforms and more distributed applications [20].

Technologically, our work is related to Java performance monitoring frameworks that collect data through instrumentation. A prominent representative is the Kieker Framework [36], which can use multiple aspect oriented instrumentation frameworks. Detailed experiments with AspectJ™ instrumentation are in [35], where a microbenchmark consisting of a single method with known execution time is used to measure the overhead of the individual instrumentation components, and two real life monitoring tasks are reported to have no observable overhead.

We extend the results reported in [35] in multiple directions. Some are related to the differences between static and dynamic instrumentation – in particular, we measure and examine dynamic instrumentation effects, which the static instrumentation constrains to the warmup period where the measurements are discarded. On the overall design level, we consider multiple threads, and we preserve realistic conditions for interaction between the probe code and the application code. In contrast, the microbenchmark in [35] enforces method timing by observing virtual thread time and waiting for a computed deadline [37]. This solution masks possible application timing changes due to instrumentation.

Kieker overhead experiments in [35] and [37] also very much complement our results – we do not deal in detail with overhead sources that are not unique to dynamic measurement instrumentation, in particular data storage and data processing. These are examined in detail especially in [37].

Another performance monitoring framework where our results are likely to apply is SPASS-meter [32]. SPASS-meter supports dynamic instrumentation, which can be used together with configurable monitoring scopes to restrict the instrumentation to relevant locations and therefore reduce overhead. Experiments that measure the instrumentation

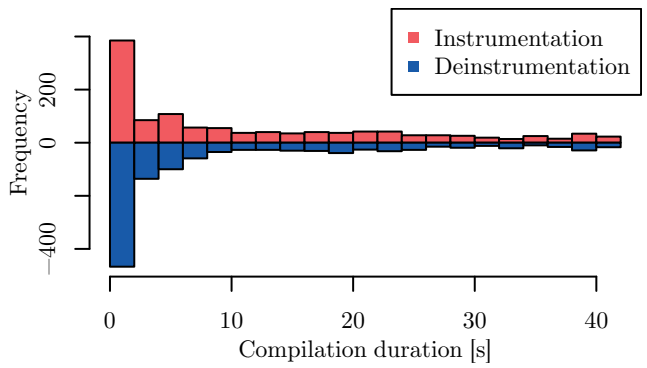

Figure 11: *Distribution of time from start of code manipulation to end of immediately following compilations.*

overhead are presented in [8], where SPECjvm2008™ is used as the benchmark application and processing overhead is defined as the change in the combined benchmark score. Again, we complement these experiments by providing a much more detailed look at the dynamic instrumentation effects, and we consider our results complemented by these experiments where the more general overhead issues are concerned.

Some monitoring framework experiments [27, 9] analyze overhead in terms of average changes to application throughput or response time, which is certainly reasonable with static instrumentation and enterprise application context. Our results are generally compatible as far as the overhead magnitude is concerned.

Instrumentation overhead is analogous to overhead introduced through aspect weaving, which is examined and attributed to particular code constructs in [6]. The need for overhead analysis in dynamic aspect weaving is advocated in [11], however, the authors performed only a limited set of experiments for dynamic aspect features supported at that time. A study examining the use of aspects for profiling of heap usage, object lifetime and execution time on the SPECjvm2008™ benchmark is available in [28], again with static instrumentation – in this context, we contribute experimental results relevant to dynamic aspect weaving.

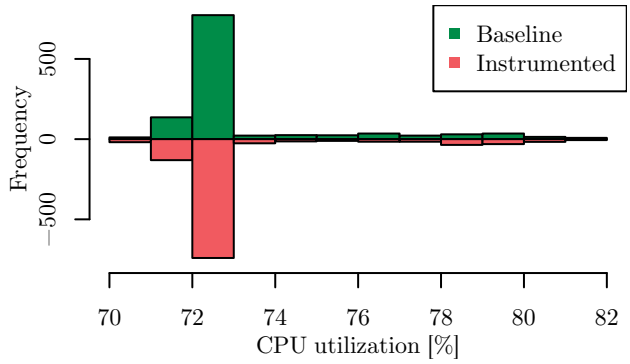

Figure 12: *Distribution of processor utilization with and without dynamic instrumentation.*

6. CONCLUSION

Dynamic performance monitoring is a promising method of reducing monitoring overhead. Coupled with dynamic instrumentation, it carries the promise of achieving zero overhead when not monitoring, because the probes that collect the monitoring data can be inserted and removed at will. On the other hand, dynamic instrumentation can interact with the execution platform in complex ways that give rise to new sources of overhead. We investigate these sources in the context of a dynamic measurement framework for Java.

Using experiments on a modified version of the SPECjbb-2015™ benchmark, we first show that the loss of measurement accuracy due to instrumentation overhead is not constant. On our platform, this limits practical measurements to methods whose execution time exceeds tens of microseconds, and also impacts the overhead compensation methods described in [19].

Next, we show that dynamic instrumentation can change the execution time of the instrumented method even when the overhead due to probes is not considered. This change can both slow down and speed up the method, sometimes significantly. We also show that the duration of the changes can vary, with short periods corresponding to compilation bursts on one end of the spectrum and periods spanning entire application execution on the other.

Looking at the compilation bursts, we show that although the code manipulation operations due to dynamic instrumentation are fast, the associated JIT compilation can last from several seconds to more than a minute. Any dynamic measurement framework looking to avoid disruptions due to JIT compilation should expect these effects to delay data collection.

Finally, we confirm that the total overhead in terms of application performance remains negligible when small scale instrumentation is deployed.

Our work is provided together with complete data and tools, available at http://d3s.mff.cuni.cz/resources/icpe2016.

Acknowledgments

This work was partially supported by Charles University Institutional Funding (SVV) and by the Research Group of the Standard Performance Evaluation Corporation (SPEC).

7. REFERENCES

[1] AspectJ, 2015. http://eclipse.org/aspectj.

[2] H. Boehm. Can seqlocks get along with programming language memory models?, 2012. http://www.hpl.hp.com/techreports/2012/HPL-2012-68.html.

[3] L. Bulej, T. Bureš, J. Keznikl, A. Koubková, A. Podzimek, and P. Tůma. Capturing performance assumptions using stochastic performance logic. In *Proc. ICPE 2012*, pages 311–322, New York, NY, USA, 2012. ACM.

[4] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proc. USENIX 2004*, 2004.

[5] A. Diaconescu, A. Mos, and J. Murphy. Automatic performance management in component based software systems. In *Proc. ICAC 2004*, 2004.

[6] B. Dufour, C. Goard, L. Hendren, O. de Moor, G. Sittampalam, and C. Verbrugge. Measuring the dynamic behaviour of AspectJ programs. In *Proc. OOPSLA 2004*, pages 150–169, New York, NY, USA, 2004. ACM.

[7] J. Ehlers, A. van Hoorn, J. Waller, and W. Hasselbring. Self-adaptive software system monitoring for performance anomaly localization. In *Proc. ICAC 2011*, pages 197–200, Karlsruhe, Germany, June 2011. ACM.

[8] H. Eichelberger and K. Schmid. Flexible resource monitoring of Java programs. *Journal of Systems and Software*, 93:163–186, July 2014.

[9] K. Govindraj, S. Narayanan, B. Thomas, P. Nair, and S. Peeru. On using AOP for application performance management. In *Proc. AOSD 2006*, pages 18–30, 2006.

[10] M. Grimmer, M. Rigger, L. Stadler, R. Schatz, and H. Mössenböck. An efficient native function interface for Java. In *Proc. PPPJ 2013*, pages 35–44, New York, NY, USA, 2013. ACM.

[11] M. Haupt and M. Mezini. Micro-measurements for dynamic aspect-oriented systems. In *Object-Oriented and Internet-Based Technologies*, number 3263 in LNCS, pages 81–96. Springer Berlin Heidelberg, 2004.

[12] V. Horký, P. Libič, L. Marek, A. Steinhauser, and P. Tůma. Utilizing performance unit tests to increase performance awareness. In *Proc. ICPE 2015*, pages 289–300, New York, NY, USA, 2015. ACM.

[13] T. Kalibera, L. Bulej, and P. Tuma. Benchmark precision and random initial state. In *Proc. SPECTS 2005*, pages 853–862. SCS, 2005.

[14] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the java hotspot client compiler for java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1):7:1–7:32, May 2008.

[15] D. Lea. The JSR-133 cookbook for compiler writers, 2011. http://gee.cs.oswego.edu/dl/jmm/cookbook.html.

[16] P. Libič, L. Bulej, V. Horky, and P. Tůma. On the limits of modeling generational garbage collector performance. In *Proc. ICPE 2014*, pages 15–26, New York, NY, USA, 2014. ACM.

[17] P. Libič, L. Bulej, V. Horký, and P. Tůma. Estimating the impact of code additions on garbage collection overhead. In *Proc. EPEW 2015*, number 9272 in LNCS, pages 130–145. Springer International Publishing, Aug. 2015.

[18] A. D. Malony. *Performance Observability*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1990. AAI9114332.

[19] A. D. Malony and S. S. Shende. Overhead compensation in performance profiling. In *Proc. Euro-Par 2004*, number 3149 in LNCS, pages 119–132. Springer Berlin Heidelberg, Aug. 2004.

[20] A. D. Malony and S. S. Shende. Models for on-the-fly compensation of measurement overhead in parallel performance profiling. In *Proc. Euro-Par 2005*, number 3648 in LNCS, pages 72–82. Springer Berlin Heidelberg, Aug. 2005.

[21] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. DiSL: A domain-specific language for bytecode instrumentation. In *Proc. AOSD 2012*, pages 239–250, New York, NY, USA, 2012. ACM.

[22] T. Martinec, L. Marek, A. Steinhauser, P. Tůma, Q. Noorshams, A. Rentschler, and R. Reussner. Constructing performance model of JMS middleware platform. In *Proc. ICPE 2014*, pages 123–134, New York, NY, USA, 2014. ACM.

[23] A. Mos and J. Murphy. COMPAS: Adaptive performance monitoring of component-based systems. In *Proc. ICSE 2004 RAMSS*, 2004.

[24] Oracle. Java microbenchmark harness, 2013-2015. http://openjdk.java.net/projects/code-tools/jmh.

[25] P. Panchamukhi. Kernel debugging with kprobes, 2004. http://www.ibm.com/developerworks/library/l-kprobes/index.html.

[26] N. Park, B. Hong, and V. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):640–654, July 2003.

[27] T. Parsons, A. Mos, and J. Murphy. Non-intrusive end-to-end runtime path tracing for J2EE systems. *Software, IEEE Proceedings*, 153(4):149–161, Aug. 2006.

[28] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly. Profiling with AspectJ. *Software: Practice and Experience*, 37(7):747–777, June 2007.

[29] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, and M. Mezini. JP2: Call-site aware calling context profiling for the Java Virtual Machine. *Science of Computer Programming*, 79:146–157, Jan. 2014.

[30] S. S. Shende and A. D. Malony. The Tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[31] A. Shipilëv. Java Microbenchmark Harness (The Lesser of Two Evils). Presentation at Devoxx, 2013. http://shipilev.net/talks/devoxx-Nov2013-benchmarking.pdf.

[32] SPASS-meter monitoring framework, 2015. http://www.sse.uni-hildesheim.de/spass-meter.

[33] SPEC Java server business benchmark, 2015. http://www.spec.org/jbb2015.

[34] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proc. OSDI 1999*, pages 117–130, Berkeley, CA, USA, 1999. USENIX Association.

[35] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous monitoring of software services: Design and application of the Kieker framework. Report, Department of Computer Science, Kiel University, Germany, Nov. 2009.

[36] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proc. ICPE 2012*, pages 247–248, Boston, Massachusetts, USA, Apr. 2012. ACM.

[37] J. Waller, F. Fittkau, and W. Hasselbring. Application performance monitoring: Trade-off between overhead reduction and maintainability. In *Proceedings of the Symposium on Software Performance 2014*, pages 1–24, Stuttgart, Germany, Nov. 2014. University of Stuttgart.