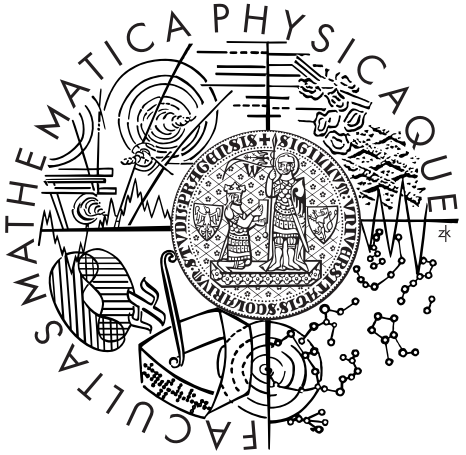
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Peter Júnoš

Flexible Event Processing Subsystem for the Java Performance Monitoring Framework

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	Ing. Lubomír Bulej, Ph.D.
Study programme:	Informatika
Specialization:	Software systems

Prague 2015

I would like to thank to my consultant Lubomir Bulej, who was leading this thesis for his support and for his questions forcing me to think about the thesis in detail.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Flexible Event Processing Subsystem for the Java Performance Monitoring Framework

Autor: Peter Júnoš

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: Ing. Lubomír Bulej, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Java Performance Measurement Framework (JPMF) je framework zaměřen na popis sledovacích bodů, které se používají pro měření výkonnosti. Popis se použije na získávání dat vztahujících se k výkonnosti v těhto bodech. Data jsou vždy získána od operačního systému a zapsána bez dalšího zpracování. Popsaný tok dat přez framework je velký a zvyšuje zátěž úložiště. JPMF neumožňuje uživateli zredukovat množství těhto dat.

Tato práce se zaměřuje na vyřešení popsaných problémů. Používá se při tom filtrování dat a jejich agregace, co vede k zmenšení množství dat na zápis. Kromě toho se tato práce věnuje úzkym hrdlám při zpracování dat v JPMF, řeší jejich důvody a snaží se je odstranit.

Klíčová slova: analýza výkonnosti software, měření výkonnosti, sledování výkonnosti

Title: Flexible Event Processing Subsystem for the Java Performance Monitoring Framework

Author: Peter Júnoš

Department: Department of Distributed and Dependable Systems

Supervisor: Ing. Lubomír Bulej, Ph.D., Department of Distributed and Dependable Systems

Abstract: Java Performance Measurement Framework (JPMF) is a framework dedicated to description of points, where the performance is measured. This description is used to gather performance data in these running points. Data are gathered and written without any processing. The handling increases bandwidth and puts high load on the storage. JPMF does not provide any possibility for user to reduce this data.

This thesis aims to solve the described problem by introduction of filtering and aggregation, that should reduce the bandwidth. Additionally, performance bottlenecks in various parts of JPMF are investigated and removed.

Keywords: application performance analysis, performance measurement, performance monitoring

Contents

1. Introduction

Computers are part of our lives. The complexity of computer systems leads to incorrect predictions of necessary computing power. Incorrect predictions in turn caused multiple well-known incidents resulting in angry users, money loss and was cited to be a cause contributing to the a fall of a government.

Czech vehicle registry has been reprogrammed to introduce connection with other registers. Unexpected load lead to the major slowdown[**car-reg**].

Financial administration of Slovak republic (IRS) experienced malfunctioning system with anticipated losses exceeding ten million eur[1], being one of the causes of fall of Slovak government[2]. According to an employee, one action took nine minutes instead of few seconds[3], which corresponds with performance issues. A completely new system was made, but it was also reported running slowly and thus increasing the time required to process taxes[4].

HealthCare.gov[5] is the health insurance website for US citizens. Despite the estimated costs reaching $1.7 billion [6], it could not fulfill the performance demands. Barack Obama, the president of United States of America, stated: "There's no sugar coating - the website has been too slow, people have been getting stuck during the application process and I think it's fair to say that nobody's more frustrated by that than I am."[7].

Multiple solutions to the aforementioned issues arise. The problem can be shifted to the future by developing of newer and faster algorithms. Performance problems can be revealed during the development using benchmarking and by monitoring after the development is finished.

1.1 Background

There are multiple tools focused on measurement and benchmarking differing in the area of usage and also in the abstraction level, where the tools operate. They range from low-level tools monitoring the processes and finishing with profilers exposing slow instructions and functions.

The most low-level performance monitoring tools are Unix-based `sar`[8] and Windows-based `PerfMon`[9]. They provide the summary of CPU usage by different processes running on one hardware. Followingly, misbehaving processes can be identified.

`Nagios`[10] and `Zabbix`[11] are enterprise solutions spanning across multiple platforms. In addition to monitoring, alerts can be sent and the plugin interface is provided. The measurements are visualized to provide overall image of the system.

Google-wide profiling[12] provides continuous profiling on of whole machine as well as per-process profiling for cloud applications. The per-process measurement can expose the performance differences based on platform including microarchitectural peculiarities to get performance improvements of "up to 15"%[12].

Contention aware execution runtime (CAER) measures differences in cache misses. That leads increased CPU utilization by 58% and thus overall performance is improved[13].

On application level, Application Response Measurement[14] describes common method for application management including diagnosing a performance problems.

Performance measurement and application monitoring are provided by Kieker[15]. Application is instrumented to deliver performance-related data.

Last, but not least, one solution to application monitoring is Java Performance Measurement Framework (JPMF). We will focus on it in this work.

1.2 JPMF Overview

JPMF aims to be high-performance low-overhead instrumentation framework focused on application monitoring. It focuses on live instrumentation of running program performed when the unexpected runtime condition appears.

Despite of the requirements mentioning low overhead, configuration and measurement concerns are strictly separated to provide generic framework. More overhead could be eliminated by placing measurements directly to the code, but the strict separation would be lost.

Overview

Measurement data are sampled in specified points, that are exactly defined in `event sources`. User of framework is able to define new `event source` as well as send new event to the framework.

When event arrives to the framework, performance data are obtained from data sources, which are operating system-dependent. Linux data source gathers data using syscalls, netlink interface and virtual file systems `/proc` and `/sys`[16]. Windows gathers data using Windows Performance Objects[17]. The requested set of data to measure can be configured either during runtime or in a static file.

The performance data are written to a single file for further analysis. It can provide insight into the program and to target a specific outcome.

Examples of usage

System administrator diagnosing high I/O utilization can configure sampling length of I/O queues and I/O bytes read and written in various parts of running program along with the other I/O values provided by framework. These values are compared with the samples of the running program without observable bugs to find the difference and therefore the cause of high I/O utilization.

In case of serious performance issues, JPMF can be used to get the raw samples, that will not be compared to the other samples. When the programmer or administrator thinks the action takes unexpected amount of time to complete, data from operating systems can advise, where to look for the issue. High iowait point to slow I/O, that can be solved either programmatically or by the hardware upgrade.

1.3 Goals

JPMF suffers from multiple performance ineffectivities as well as missing features, that should be solved by this work.

It does not provide any form of aggregation or possibility to skip measured data. Everything sampled must be written to a single output. While JPMF provides programming interface to extend provided storage, which could use multiple outputs, adding another storage does not solve aggregation and filtering of data. More data being sent increases the pressure on storage and results in imprecise measurements.

JPMF also uses platform-dependent sensor naming. With different names on each platform, measurements have to be reconfigured when moving across platform boundaries.

To sum up, our goals are to analyze architecture of JPMF, its event processing and provide a support for configurable pipeline. It should be able to aggregate and filter performance data, provide them to multiple storages, provide cross-platform naming without further performance degradation.

2. Analysis

In this section, we provide an analysis of JPMF architecture and implementation. We point out the existing shortcomings and derive specific goals to address them.

Profilers are well known to programmers. They are used to find "hot code" – the code where the significant portion of time is spent.

There are at least two possibilities of profiling implementation. One is based on sampling and the another is based on direct calls of benchmarking functions. Furthermore, small parts of code can be profiled standalone in microbenchmarks.

2.1 Profiling

Sampling-based profiler instructs the processor to sample running instructions in specified intervals. According to the law of large numbers, by regularly sampling running instruction, the distribution of appearances in each instruction approaches to the real distribution of the time taken during analysis.

The approach is useful, ensuring low overhead while providing ability to measure performance of whole system, including the system kernel.

One example of statistical profiler is OProfile compiled with Java support.

Despite the low overhead, optimizations of compiler and incorrect reading of native code can cause an incorrect interpretation of running functions and instructions[18].

2.2 Benchmarking

2.2.1 Microbenchmarks

One solution to the profiling is using microbenchmarks. They provide a possibility to measure a performance of individual functions as the smallest parts of program.

Microbenchmarking requires its own setup. It can be accomplished by calling one function in loop, while measuring the time taken to complete N runs. Set of tools such as JMH solve the problems of naive loop-based implementation by providing tools, that ensure the code is not optimized away while not taking measurable amount of resources[19].

Microbenchmarks are performance-sensitive, as only short functions are being executed. Their solution to save measured data is simple, but still understandable – they avoid it and use only memory. Using memory does not cause extreme performance losses while allowing sampling for a short time. Solutions suggested for sampling does no work, as microbenchmarks precisely know, which functions are being executed.

Practice of microbenchmarking of all parts of code was discouraged, as it leads to premature optimization.

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified" Donald Knuth[20]

Furthermore, microbenchmarked functions run on their CPU not taking the other parts into consideration. Depending on Java scheduler, parallel programs can show linear scalability while providing only little performance benefit. This can be caused by thread interleaving, when benchmarked thread is run alone on one CPU and the completion takes less time than expected. Therefore, it is not always reliable.

On the other hand, microbenchmarking is not without advantages. It can provide measurement of performance in performance critical parts of code and identify small issues. By utilizing a correct framework, issues with usage are eliminated and the resource consumption of the benchmarking code.

2.3 Monitoring

Performance measurement based on direct calls can measure the performance of application focused on individual functions, without the requirements like custom runner in case of microbenchmarking. It provides ability to measure performance of parts of the application like an electrical circuit. One possible use includes measurement of application, that is already deployed, via live instrumentation.

Calling benchmarking framework from application code is cross-cutting concern. It can be solved either by aspect oriented programming, conditional compilation or instrumentation.

Java performance measurement framework

JPMF is designed to be used in call-based performance measurement. It is up to programmer to implement event source in any way. However, programmer is provided with triggers designed to be in used in manual instrumentation.

It evolved from benchmarking solution. Changes were introduced to provide a framework for harvesting the performance data, that in turn lead to reduced performance. Therefore, it cannot be used for microbenchmarking.

JPMF samples performance data in specified points and outputs them to a file. The performance data can be defined in order to sample the data necessary to provide insight while not slowing the sampling down. The data contain the stats provided by operating system, including, but not limited to CPU cycles, time, cache misses, memory usage, bytes read and written to storage and to the network adapter.

The data can be analyzed to provide insight into the program and to target a specific outcome. System administrator diagnosing high I/O utilization would sample length of I/O queues and I/O bytes read and written in various parts of running program along with the other I/O values provided by framework. These values could be compared with the samples of the running program without observable bugs to find the difference and therefore the cause of high I/O utilization.

Programmer could try the similar sampling while evaluating the performance. The data would be compared between older and newer implementations to provide the speedup factor between different versions.

In case of serious performance issues, JPMF can be used to get the raw samples, that will not be compared to the other samples. When the programmer or administrator thinks the action takes unexpected amount of time to complete,

data from operating systems can advise, where to look for the issue. High iowait point to slow I/O, that can be solved either programmatically or by the hardware upgrade.

Kieker

Besides from JPMF, Kieker[15] uses instrumentation to specify the position to execute measurement code. Kieker does not provide configurable filtering or aggregation, but can still serve as an example.

Kieker utilizes SIGAR[21] and JMX to sample the performance data. Code generation is used to generate class to transfer sampled data to storage, so that the data are not stored in generic structures using generic algorithms, but the simple copying algorithm is generated. Note that despite these microoptimizations, transfer objects are allocated on each transfer. That increases the load put on garbage collector.

Furthermore, it provides the aggregation. As the provided aggregation and filtering is only static, code must be generated to change sampling. Without external compilation, everything measured must be always written and thus extending the waiting queues. However, basic aggregation functions are provided in `AggregationMethod`. It is called on the measured data just after obtaining them. This approach reduces the load while preserving the essence of data by using average, median or an another aggregating function.

Kieker also offers architecture discovery. Monitored program is being run with Kieker agent and Kieker reports, which functions are called from which parts of program. Using SynchroViz [22], user can get the idea about which synchronization is being done. It is visualized in 3D (see 2.1) and it could be useful, but it is outside the scope of performance measurement and PMF.

2.4 Original JPMF implementation for user

As the name says, JPMF is Java-based framework divided into more subsystems depicted in Figure 2.2.

JPMF contains application-driven subsystems to be executed in the context of measured application during the measurement and externally driven subsystems, that provide the API to access JPMF from the application.

2.4.1 Application-driven subsystems

Event source

These are subsystems to be run by instrumentation during the measurement. The main role is to sample and save data related to performance, whenever Event source emits new event. The Event Source can be either called by instrumentation of the application or can be called externally. Furthermore, it allows disabling events, so that they will not be emitted anymore to save the resources.

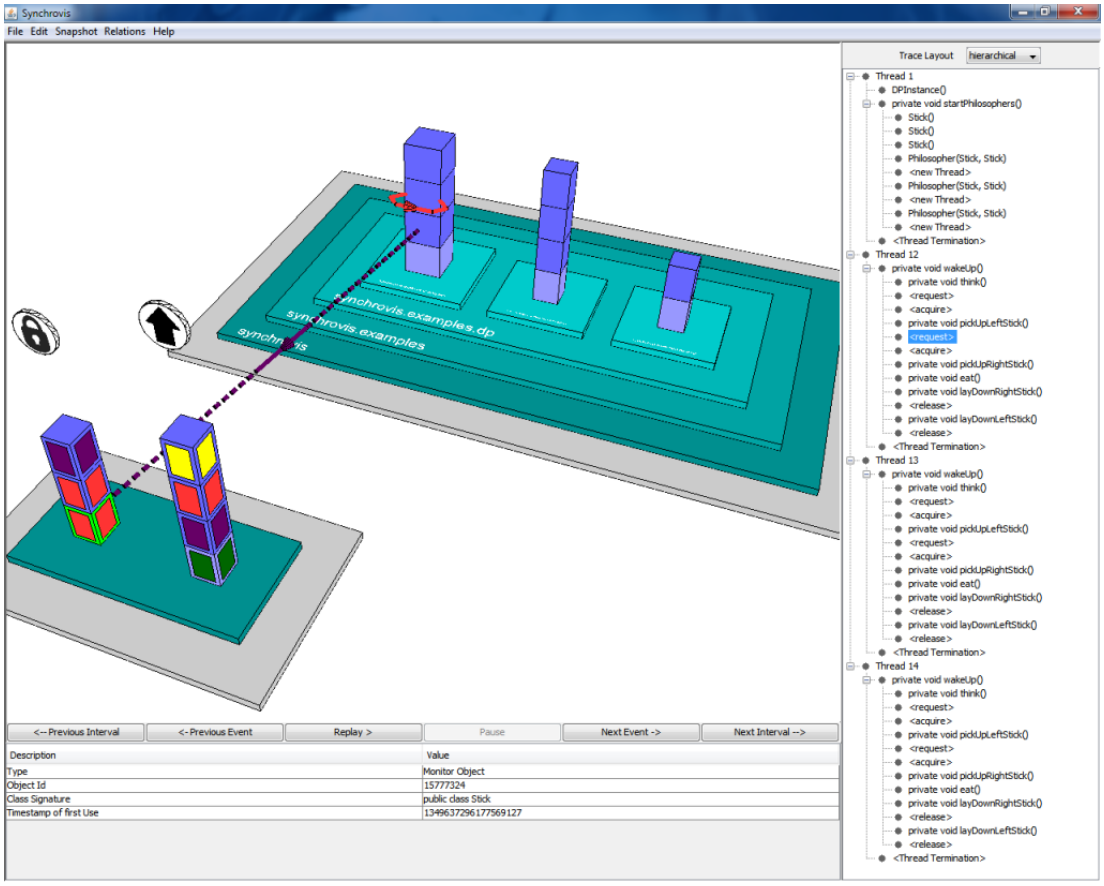


Figure 2.1: Synchroviz example from [22]

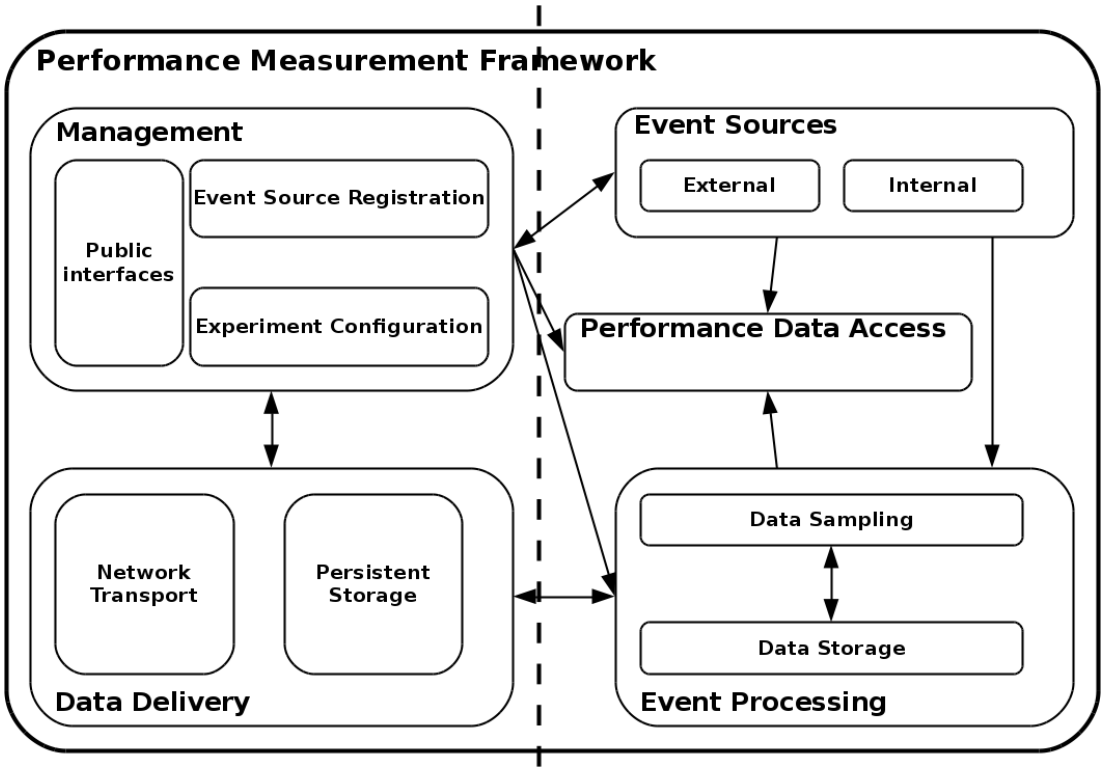


Figure 2.2: JPMF modules from [22]

Performance Data Access

Performance Data Access subsystem measures data, that can be useful when assessing performance of application. The performance data are provided by applications, operating systems and some are provided directly by hardware. It relies on data sources and time sources, that should provide more types of performance and time data, such as I/O stats, CPU usages, network stats and even more data.

Data sources provide data in sensors. These are of unified naming denoted by URI-like syntax:

`[sensor://]<datasource>/<group>/<sensor>[#<instance>]`

Sensors are sorted into groups, that merge together similar areas of sampling. For example, there is a group for network, for block devices, for CPU and similarly.

Name of sensor then provides the name of data, that is sampled – sensors in group CPU could provide sensor load. To avoid more CPUs having different groups, instances were introduced. Instance is the naming of the unique resource. More CPUs in one system would be part of one sensor, but the sensor would provide and instance for each CPU available. There are also singletons, where there can be only one instance – such as in the case of system RAM.

To sum up, to describe sensor denoting length of I/O queue on sda1 provided by ProcFS, one would use URI-like syntax

`sensor://linux.procfs/io/queue_length#sda1`

Data storage

The data sampled by Performance Data Access are sampled only in memory. Data storage is provided to store them persistently. Nowadays, only direct writing of measured data to files is supported. However, it could be extended to save data over network.

Data storage can save data to the one predefined storage in one format, that is either xml or custom indexed data format. All the sampled data are saved in the format they were sampled.

2.4.2 Externally-driven subsystems

Externally-driven subsystems allows access to JPMF from the outside of the measured application.

Data delivery provides the measured data back to the application based on the subscription. Also, it provides metadata about the running storages.

Infrastructure managements is the management API of the whole JPMF. It also serves as a registry to save and provide shared resources.

2.5 Event processing

The event processing part describes, how events are generated and the effects they have on the other parts of the framework. It starts a measurement of the performance data to be sampled and saved. Class diagram of the most important classes used in old implementation is depicted in Figure 2.3.

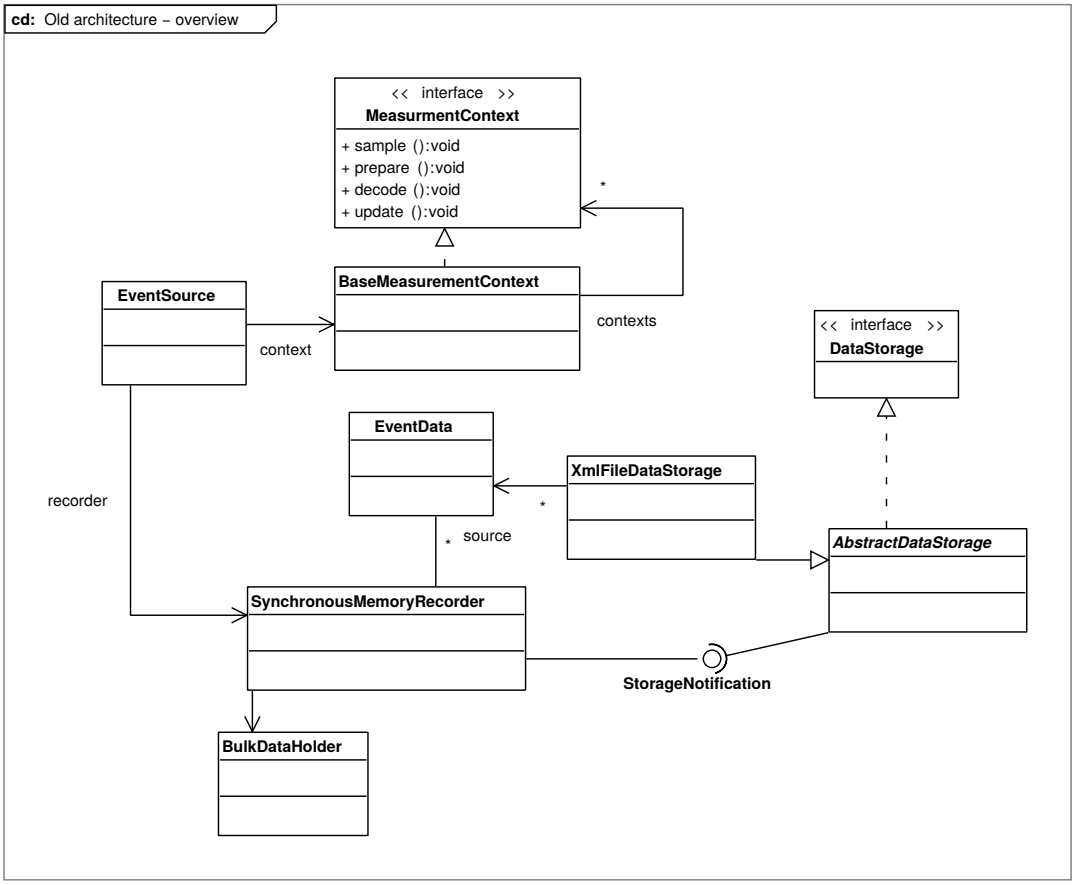


Figure 2.3: Most important classes when processing performance data

2.5.1 Event source

The first step in performance data gathering is the request to get the data. This is called event and is produced by an event source. Although there are some implementations of event sources, programmer is encouraged to implement additional ones.

The implemented helpers contain only the simplest, yet sufficient events. There are events for entering and leaving method, cycles and one-time fired events. Each event contains fields, that bear additional information to allow identification of calls. This identification is bound to the event and cannot be simply moved elsewhere.

The event source is expected to be called, when the measurement should be performed. One way to call it is using the call directly in code. Direct call is done by programmer and is called everytime the part of program is executed. If the programmer does not want to call it in production, he is free to use `ifdef`ed code. Then, the code will be executed only with certain compile-time configuration, when `ifdef` is satisfied. When one compiled program is to be used both in production and during the measurements, the event source can be called by instrumentation.

When the event source is called, it fires an event to be processed. The event is picked up by Java Performance Data Access.

2.5.2 Java performance data access

Java performance data access (JPDA) is the part responsible for obtaining the performance data via available source. As all sources of performance data are platform dependent, there are different implementations for different operating systems. Performance data are obtained via netlink, virtual file systems and syscalls on Linux; via WPO on Windows. The data are obtained and transferred to be saved.

On a lower level, event is dispatched to all currently registered measurement contexts for current sampling. Measurement code requests getting the performance data from `Probes` — the units of code dedicated to accomplish sampling from one physical resource. Probes are designed to accomplish quick sampling of data when told to `sample()` while they can spend more time, when user only wants to `prepare()` them for sampling. Therefore, the sampling is prepared and sampled afterwards. The data are propagated to `SynchronousMemoryRecorder`, where the JPDA section ends.

2.5.3 Memory recorder

`SynchronousMemoryRecorder` is designed to hold data in its temporary buffer. Custom synchronized buffer is provided by `BulkDataHolder` and `SingleContextMemoryStore`. All measured data are stored there till the time they are be requested.

When the filled capacity reached specified percentage, which is 75% by default, `StorageLowNotification` is sent to the storage, so that the storage can request the sampled data. When requested, the data were read from their temporary store and copied to the structure called `EventData`. The structure contained event fields, that are the data directly related to measurement and to the event received when sampling along with measured data.

After the `EventData` were filled, they were sent to the storage.

2.5.4 Writing to disk

Storage is composed of `AbstractDataStorage` and the storage implementation. Division has its sense - while `AbstractDataStorage` is requesting the data to be read, the real storage implementation is called from its abstract ancestor without caring about the source of the data.

The stored data are therefore propagated to writer, that writes either to XML file usable for developer overview or to indexed file which is developed to be faster.

Simplified diagram of the dispatching of performance data can be seen in Figure 2.4.

Original storage was built on an idea, that we cannot avoid writing to disk and it is the slowest part of the system, so we should not request writing to disk too often (amount of data written is still the same). Although the idea is correct, there is not an ideal time, when PMF will write its data to disk. PMF is not a microbenchmarking suite, so that it could hold all the sampled data in memory till the next run. Program runs continuously and the measurements have to be stored continuously.

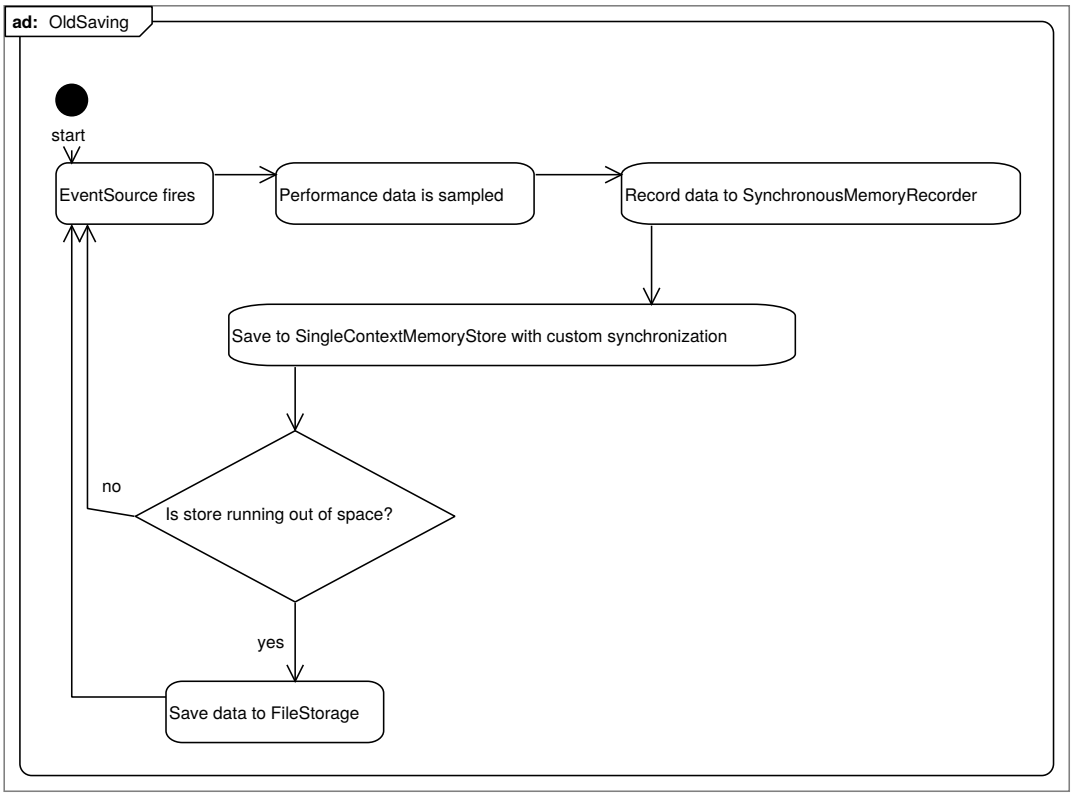


Figure 2.4: Saving data in old versions of PMF

Storing data in bursts avoids regular overhead during traversal of measured code. However, spikes in execution time appear, when there are a lot of records to write and the disk is already utilized. When program continues traversing measurement point and generating big amounts of sampled data, original program dropped these records.

2.5.5 Synchronization

Furthermore, implementation of custom synchronization is error prone. Bugs in synchronization are hard to detect, as they cannot be observed consistently, but only when timing of multiple threads interleaves in a way, that triggers the bug. Finding race conditions is NP hard, therefore less apparent races are relied upon debugging in practice [23].

The possible solution is to utilize formal verification to solve this bug. However, that would need more work on formal verification itself or creation of model and then would require more work from the maintainer.

The easiest solution, albeit not so rigorous and without any guarantees, is using standard algorithms and trusting them, as it is often quoted as "Don't reinvent the wheel" [24]. This reduces the surface, where the bug could appear. This solution also provides possibilities of future speedups, that do not need attention of programmer. New update of JVM can always use the best algorithm available.

2.6 Goals revisited

There are also abilities outside the scope of JPMF and this work, that will not be implemented. Interesting feature too far from JPMF range is architecture mapping. Besides that, only functional parts will be used. Therefore, useful tools from another performance measurement programs will not be reimplemented.

One of the main problems alternative implementations tried to solve was performance of the measurement itself. Therefore, we want to provide:

To get high performance of sampling, we will support omitting of events and allow only one event per specified time. Microbenchmarks show we should concentrate on the lowest overhead possible and we will be able to measure the performance of all slower executing functions. We could also support the aggregation like in Kieker, that could be improved by having aggregation stackable.

2.6.1 Configurable pipeline

Pipeline implementation, requires saving performance data to the pipeline instead to the `SynchronousMemoryRecorder`. `SynchronousMemoryRecorder` also holds important details about fields, so it cannot be simply eliminated. Using `SynchronousMemoryRecorder` in pipeline would cause slowdown, as it is the part, that synchronizes accesses and is necessary on every data reading.

Moreover, the data are still copied between buffers, so that the time is spent on copying and more memory is allocated when not really required. The main reason for not adding this or another overhead is targeting of JPMF. With overhead, sampling is dominated by noise.

Therefore, the first subgoal is to design configurable pipeline between the temporary and persistent storage without adding significant overhead.

2.6.2 Pipeline filtering

More data flowing thru the whole framework slows it down. We could simply place our pipeline after dispatching data to write, but then the majority is framework is already traversed and the filtering cannot offer expected performance benefits.

The second subgoal is to provide filtering, aggregation and transformation of events to avoid writing them to the storage. Filtering should be provided at runtime. This will reduce the overhead even further.

2.6.3 Virtual data source

Virtual data source is designed but not implemented in the thesis of Lubomir Bulej. We see the data can be sampled in rapid way, but sometimes cannot be saved. It is suggested as a solution to unification of sensor naming.

We also need aggregation, that could be provided by Virtual data source to avoid saving megabytes of data, when only a part of them will be ever read.

The third subgoal is to design virtual data source capable of data aggregation and unification of sensor naming across platforms.

3. Design

In this chapter, we outline the concepts of the proposed solution. Details of implementation along with implementation difficulties are described in Implementation (Chapter 4)

When there are more events from EventSource than the size of buffer, file writer has to write performance data as fast as possible to avoid slowing down of program execution. Slowdown means the benchmarked program is waiting for benchmarking program, which invalidates the results of benchmarks.

Writing of big amounts of data to the target file causes unnecessary slowdowns. Therefore, we decided to implement filtering to filter out as much measured data as possible.

3.1 Processing pipeline

3.1.1 Simplified data flow

When the performance data are sampled, they have to be written or thrown away to not fill the memory. We describe the investigation of the path from sampling to writing and suggest improvements.

Measured data in old model were immediately stored to a temporary storage called `SynchronousMemoryRecorder`. They were then put to transfer objects to be sent for recording to storage when temporary storage is full. The original implementer predicted aggregation[1] of requests. Less, but bigger chunks of data sent to disk could speed it up.

However, there are multiple problems in this approach. There were multiple buffers on the path from sampling to storage — one on the level of aggregation and one on level of open file in Java; another one could have been provided by operating system. With each buffer requiring deep-copy of data, part of processing power was spent in getting data from one transfer object and copying it to the other transfer object.

Moreover, large buffers to group up writes resulted in periodic floods of the storage. This was observed in case of measured program writing data to the same slow hard disk as the JPMF used. Time spent in writing program data was higher than expected. Consequently, the results contained outliers resulting in incorrect marking, which functions are hot.

Furthermore, multiple buffers always reduce predictability of the produced load. With bigger jitter, less implications of measured program execution can be deduced.

Writing everything individually spreads the time taken by I/O to a wider time window. Continuous monitoring of target application results in small I/O overhead in all samples, in exchange for hard-to-predict bigger overhead in part of samples. Regularity can help an administrator to spot the problems and to avoid choking in case we run out of our buffers.

[1] The aggregation was only performed on the level of Java calls between temporary storage and writer implementation. All original data were retained and written

Slight increase in overhead is expected in this step. We decided stable and therefore more usable results are much more important than the overhead is.

Nevertheless, with the previously noted, we do not aim to eliminate all buffers by design. One buffer to be used by writer on the Java level (`BufferedFileWriter`) can be preserved to optimize access to hardware. Contrary to the previous parts, hardware storage is significantly slowed down by writing chunks in the sizes of bytes and therefore we leave there aggregation.

Considering some simplification, this is shown in Figure 3.1.

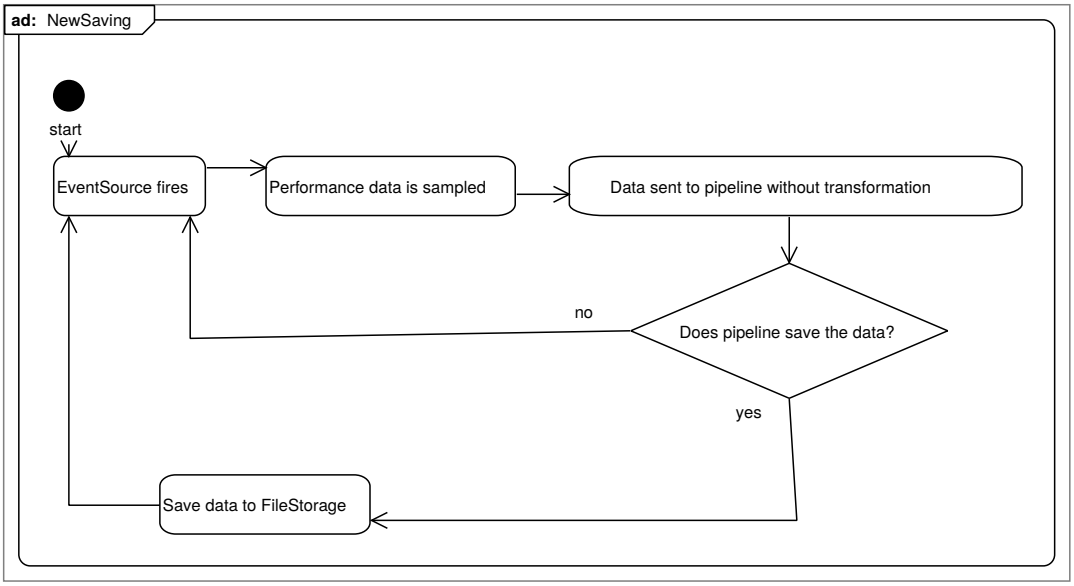


Figure 3.1: Saving data in proposed version of PMF

3.1.2 Reduced amount of synchronization

When running sampling, each copying of data flow mentioned earlier used its custom synchronization. In chapter "Simplified data flow" is the description of omitted steps, that also reduces amount of synchronization, which in turn results in less contention. However, there is still a connection between sampling and storage. We were considering changes there.

We considered direct and synchronous passing of performance data from sampling to storage. That would decrease the contention, improve the cache usage and therefore, would allow more rapid sampling. That would mean writing performance data to disk will always block the measured method, which is undesirable.

To avoid extra buffers, we have decided to use simple queue as the only synchronized part. The queue (`BlockingQueue`) in our usage can be implemented even as a lock-free data structure using Hazard Pointers [25], that allows affordable performance without custom implementation. To avoid allocation during the measurement, we use two queues. One stores records to be delivered to pipeline and possibly follow to disk, the other one returns used records for further reuse. By using this setup, we can avoid any allocation while sampling, that results in more consistent sampling results.

3.1.3 Buffer resizing requirements

Previous implementation provided temporary aggregation storage described in "Simplified data flow". That was extending the buffers as soon as they were full. As they were scheduled to be freed at 75% of their capacity, the resizing was not often executed, but it happened when producing data in more rapid fashion. Therefore it seems we would also need temporary storage, if we wanted to mimic the old behavior.

The main problem of omission of their implementation is only the bounded amount of unwritten data in queues. This results in slower program executions, when the buffer size is underestimated.

We could, of course, extend our buffers. However, extending buffers costs CPU time and only postpones the problem of them being filled. When the buffering memory extends, it should also shrink to avoid memory leak.

If we were to shrink buffers, then it is questionable, when the shrinking occurs. If it shrank whenever the buffer was used below specified percentage of its capacity, shrinking and possibly regrowing array would generate high extra load and therefore would result in unusable performance data. Predictions of future array usage are similar to other predictions of future - they can be wrong and one cannot rely on them.

Last, but not least, allocation changes the measured times as:

- There is an extended working set for CPU. Bigger working set means lesser part of it can fit in CPU caches. This can, but do not have to, affect the performance
- Java has to allocate memory to be freed later. Allocation on newer JVM with garbage collector can be done in as low as 10 machine instructions **??**. On the other hand, allocation puts load on garbage collector traversing memory to be freed.
- In the worst case, it allocates whole available memory and starts to use swapping. It could slow down the whole system by high factor.

Therefore, we decided not to reimplement indefinitely extending buffers. There is a configuration of buffer size, that is static. In case application produces more events than it can be written to disk, the application is paused as in producer-consumer model. User can avoid this pause by configuring filter, that drops the extra data.

The new pipeline was designed to be composed of stages named elements, as shown in Figure 3.2. Each of the elements should be responsible to send the data to the next element in the row. It could also choose to filter the data our simply by not resending the data, that will not proceed. On the other hand, one part of pipeline should be able to resend the data to multiple targets solely at its own discretion to provide filtering.

Expected interactions of pipeline working are depicted in Figure 3.3. Pipeline is passively fed by the data from measurements and provides its outputs to a selectable target or sink of measurements. As the data transferred are produced by different events and the new events can be created, it also provides an interface to register new events to be transferred (not depicted in Figure). These metadata

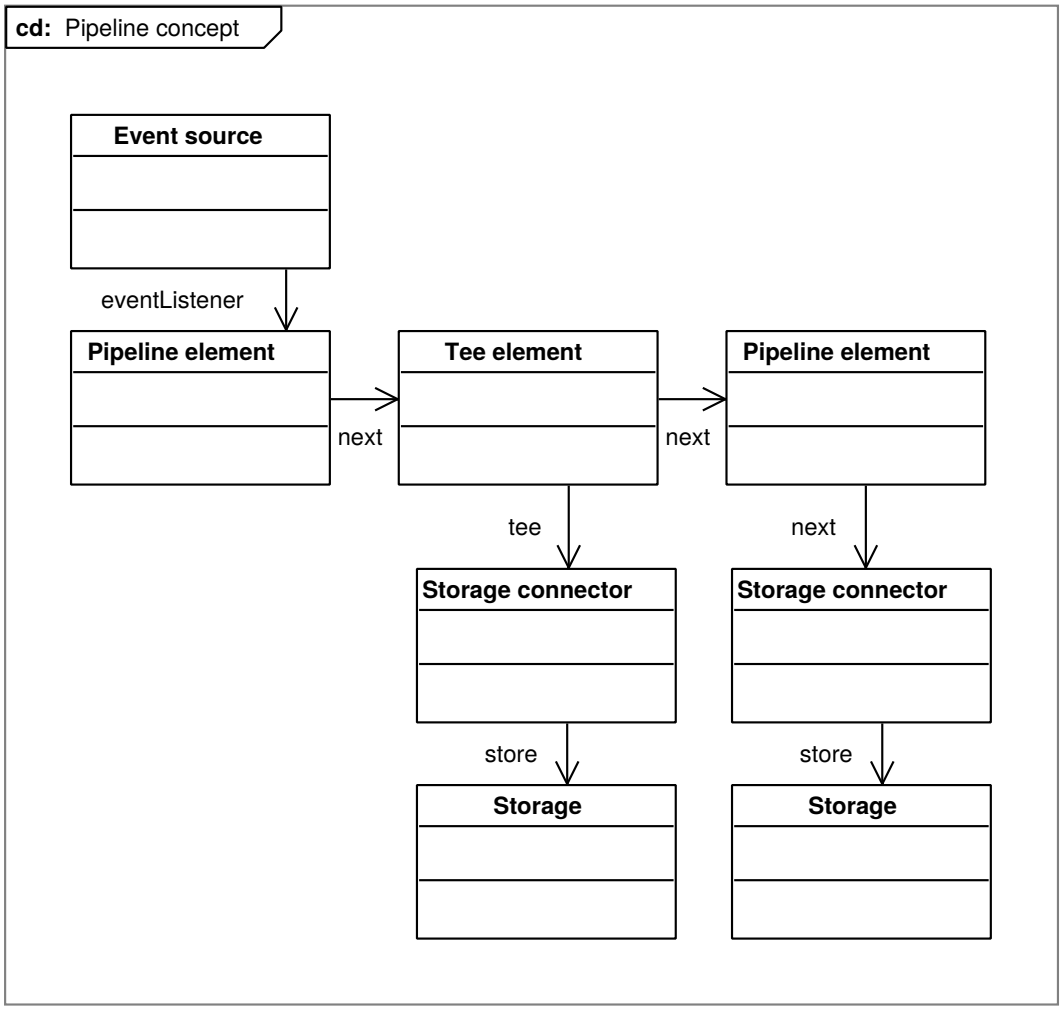


Figure 3.2: Concept of pipeline (simplified)

could be transferred by the out-of-band channel, but it would need extra channel and the metadata could not be used in created pipeline elements.

The main idea behind editing the existing implementation of PMF was to reduce overhead and to simplify overall architecture, that can reward us by avoiding possible bugs in code. Secondly, we wanted to implement new functionality. Programmers do not want to be restricted by only one file, where is everything measured written. One would expect writing everything to multiple targets or even no target, if he does not need data. This can provide additional speed, as hard disk, its queue and whole I/O subsystem are not utilized by measurement, when it is not absolutely necessary. Such utilization causes unexpectable results, which, in turn, renders measurement useless.

Its implementation of this pipeline along with the implementation problems are described more closely in the pipeline part of Implementation chapter (see Chapter 4.1).

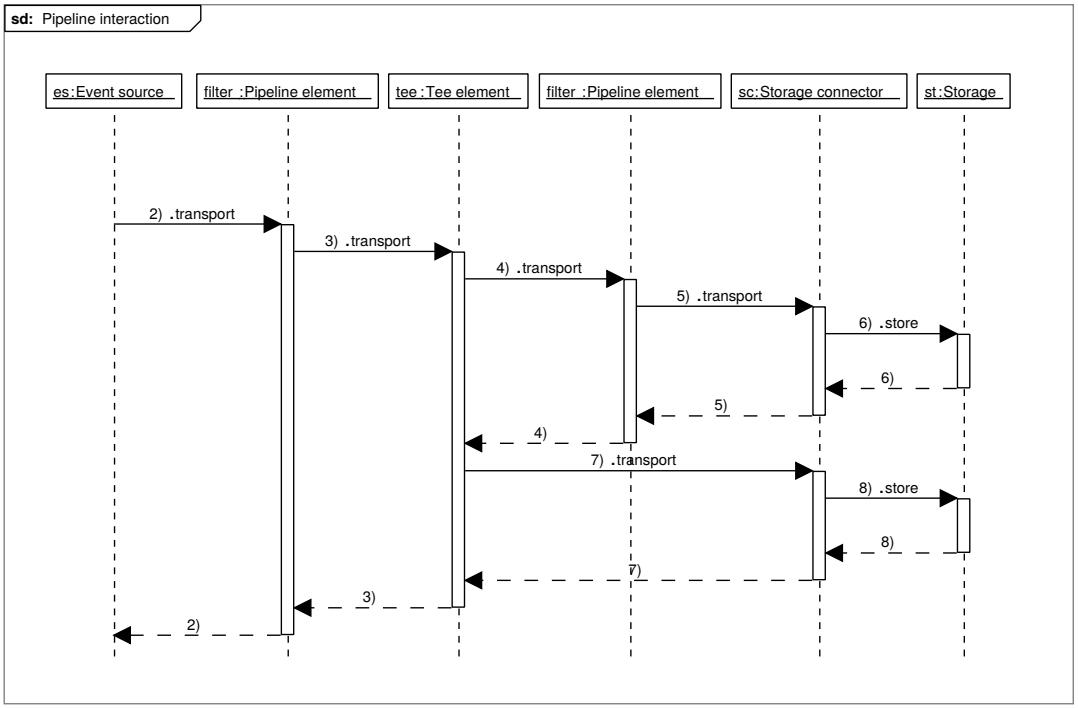


Figure 3.3: Expected interaction of pipeline (simplified)

3.2 Division of virtual data source

The original author of JPMF envisioned building a virtual data source [26], that was designed to serve two basic ideas. The more pronounced one is an implementation of DataSource interface, that serves as an intermediary and unifies naming of different sensors across operating systems. Such data source gets platform independent name of sensor, that is going to be measured. As a return value, it measures and provides data from platform specific sensor. `MapDataSource` was separated to facilitate efficient renaming.

The other one is `VirtualDataSource`, that is the extra `DataSource` used for aggregation of values. It can provide any other transformations including aggregation, but with the significantly higher costs.

Targets and even implementation of these data sources differ significantly. While one does not need to store nor read any data, the other one has to store at least its own sampled value. In cases like flowing averages, more values, with the size proportional to input size, would have to be stored.

3.2.1 MapDataSource

If the data source was implemented in the most generic fashion, it would have to provide renaming and also aggregation together. JPMF does not directly allow rewriting of sampled data, as they are transferred in container to avoid rewriting them mistakenly.

If we designed only the virtual data source, that would transform all data, it would need an extra step even with data left without transformation. `MapDataSource` can be obviously implemented more effectively, as shown in its interaction

with other data sources in Figure 3.4. For comparison, virtual data source will be shown in Figure 3.5. Thanks to PMF architecture, MapDataSource does not have to allocate its own storage and can use sampled state of other sensors. The only prerequisite is, that the `MapDataSource` provides the descriptors in the way JPMF samples sensors underneath. By this implementation, user gets only a renamed sensor without sampling overhead.

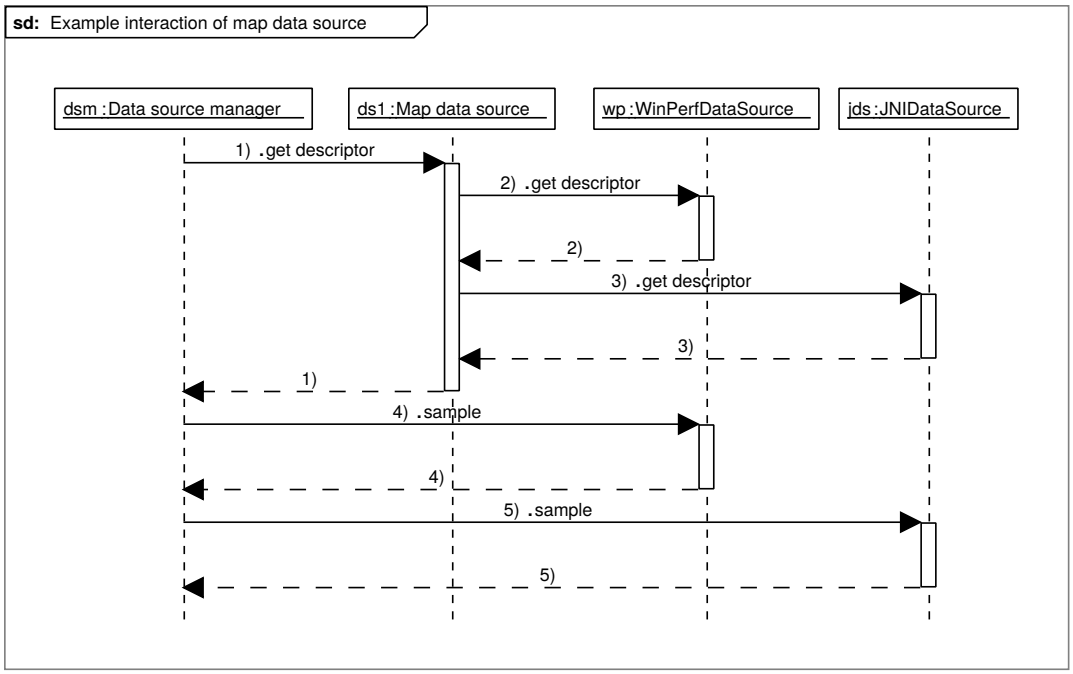


Figure 3.4: Proposed map data source (high-level view) - compare to Figure 3.5

Designing it seems to be trivial, but it has also some drawbacks. JPMF provides its interfaces outside itself, but it does not expect, that the caller could be inside. This newly introduced loop could, for example, yield to infinite recursion. If `MapDataSource` wanted to update state of all sensors, it could call `update()` on its PMF Controller. That would call `update()` on all `DataSource`s again and it yields to an infinite recursion.

We know it and have to be careful about the calls. Avoiding the calls to `update()` solves the issue for any observer outside the API. It has some drawbacks, as the user inside the API will not see updated data, when it calls the `update()` only in this data source, that can lead to confusion. However, it should never be used as stated and therefore no confusion is possible. It will be supported by a comment in the mentioned method.

3.2.2 VirtualDataSource

The second implementation is `VirtualDataSource`, that is able to change the values. It gets sampled values from the other data sources and uses their outputs for its custom sampling.

Changing values in regular pipeline comes with hazard of unpredictable results. When user decides to send the measured values to more consumers, where one contains transformations of source data, also the second consumer can get

the transformed data. In case of sending values to two tee branches in parallel, the results could be vastly worse, as they would depend on a data race condition. Details are in the implementation chapter of `MassTransformer` (4.1.5).

This could be trivially solved in implementation by defensively cloning each element containing performance data to be sent to multiple targets. However, we do not want to put extra load on garbage collector and have to solve it on design level. To avoid this issue, we decided altering contents of already measured performance data should not be provided to users.

When changes are forbidden, changing can still be implemented by sampling the data again after transforming them, as shown in Figure 3.5. Therefore there will never be unpredictable results, as the new data are sent in the new pipeline and it seems to be only coincidence they are coming from the same source.

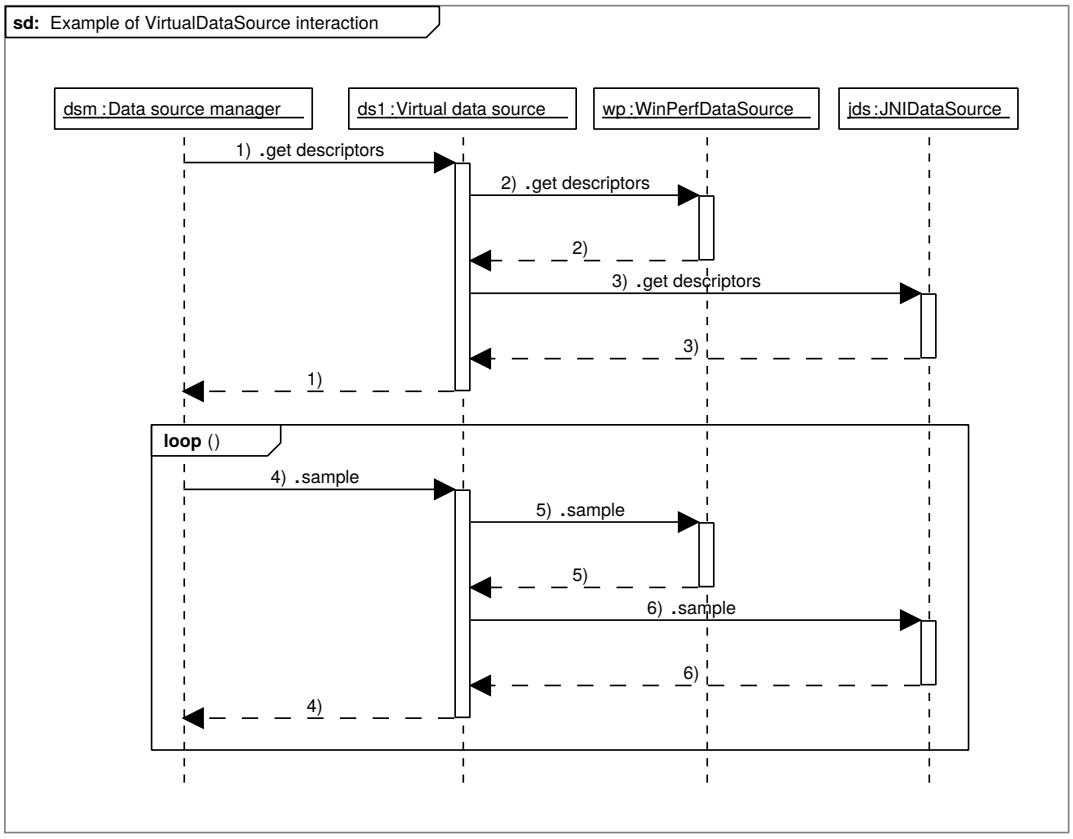


Figure 3.5: Proposed virtual data source (high-level view) - compare to Figure 3.4

It has also some disadvantages to be considered. This data source is not an observable part of any pipeline and so it leads to harder debugging. It is still not as bad as unpredictability, so we can go for it.

More details about its implementation is mentioned in the "Implementation" chapter.

4. Implementation

Implementation should contain configurable and understandable delivery of sampled data. Still, it is the target impossible to attain if it was a coding monolith. With one monolithic implementation, user cannot decompose his problems.

As the decomposition is useful and welcomed, implementation results in pipeline. It is furtherly divided to multiple stages, where each stage is independent.

4.1 Pipeline

When the idea of pipeline is discussed, there are at least two approaches to the meaning of word "pipeline". One is a generalized meaning and the other one is the one, that stands for software design pattern.

The pipeline pattern sees pipelining as a design pattern designed to improve the concurrency. Each of the stages is designed to be ran with separate resources, that provides the speedup factor up to k in case of k-stage pipeline.

The generalized one is the name of concept. It expects there are data flowing between the elements, without any mention of how are the data passed. Data are supplied to pipeline and consecutively transformed by multiple stages, where each stage produces output for the next stage in row. It can be run in one process without leading to any speedup. Pipeline pattern is also the generalized pipeline, though the inverse is not always true.

When we referenced pipeline in chapter Design 3.1.3, it was always the pipeline in the generalized sense, as we were describing concepts. However, we should think about the idea, whether our pipeline can be implemented using pipeline pattern.

If we used pipeline pattern, we should put each stage to separate thread, as it is usual when the throughput has to be maximized. Maximizing output by putting each stage to another thread could work, when each stage does a lot of computation. Then, the synchronization among the stages takes less time than the computation itself. Otherwise, the overhead outweighs the performance gains.

Synchronization puts extra overhead on pipeline and sampled program. In this matter, we decided to implement pipeline, that consists only of two stages, also called as producer-consumer pattern. Consumer gets the data and it is a responsibility of this consumer to save the data. There are two buffers between them – one for delivering sampled data to be saved and the other one for records to be reused.

Note that on the architectural side and from logical viewpoint, the multilevel pipeline is still being used. Implementation as producer-consumer means, that all levels except one happen on the level of internal implementation, as it can be seen in Figure 4.1.

4.1.1 Overview of pipeline elements

Pipeline elements can be divided by their responsibilities. There are elements that treat all measured data as immutable called filters and another elements allowing code mutation originally called transformers. Filters can either serve as

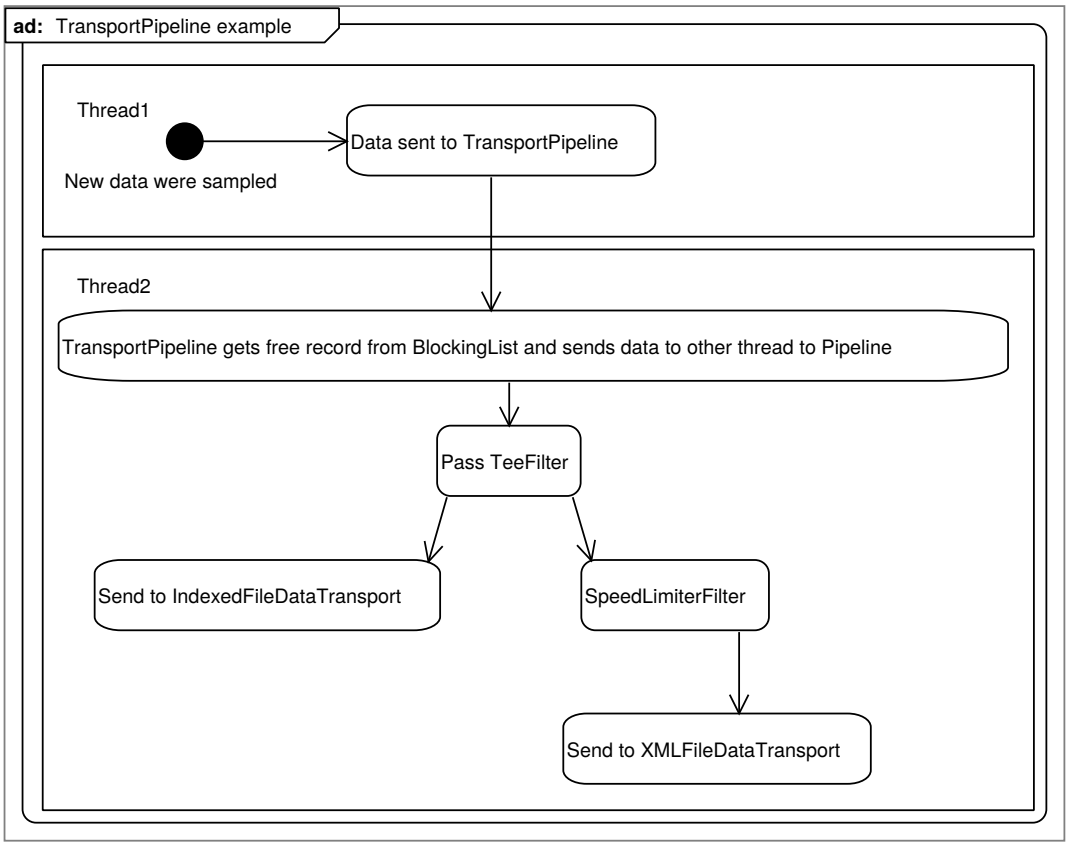


Figure 4.1: Showing `TransportPipeline` – the base part of Pipelines

leafs of pipelines to stop selected measured data from further propagation, or can serve as branching points by sending measured data to more recipients.

Each stage (element) of pipeline is independent, as the whole pipeline is. On the implementation level, this is exhibited by the way each element is called. There is not a single class or object, that knows about all elements and dispatches calls. Instead, the `PipelineElement` itself is responsible for resending all the data, in case it wants to resend it further. Failure to resend data for any reason means the data will not be dispatched to output without any warning. This design decision was backed by the idea of independent elements.

No element is permitted to store the reference to measured data directly after returning from `transport()` function designed to deliver them. In case of necessity, this can be bypassed by copying the data so that the structure can be reused. Failure to comply with this rule may result in unexpected changes of measured data.

One argument to constructor is the next `PipelineElement` in row. This is because it is how can be the pipeline imagined - more elements are connected and the output of one is sent to the input of another `PipelineElement`.

Another constructor parameter passed to `PipelineElement` is `PipelineManager`, that can be used to get any pipeline by name. Together, it allows us to build `TeeFilter`, that behaves like a "tee" command from the Unix world [27]. This `TeePipeline` is sequential for now - one branch happens after another. That is not the strict requirement and in case of independent writers. It can be also

ran in parallel, if changed slightly.

User can see pipelines, when he specifies the output. Output `default` is used every time; afterwards, user can specify more pipelines and forward the output of default pipeline to them using `TeeFilter`.

Implementation of all `PipelineElement`s and even one unimplemented can be seen in Figure 4.2.

Pipeline itself is implemented by synchronous calls from each stage.

4.1.2 SpeedLimiterFilter

By this responsibility division, there is only one synchronization on the whole Pipeline. As the data sampling should not wait for data to be transformed and/or written to disk, there are more threads as consumers of sampled data. The consumer thread count is configurable. Additionally, it is up to consumer, what it does with the sampled data. There is an implementation of multiple solutions, where consumers can decide, what should be done with the data.

One is to omit the data completely and not to save it. This can be useful, when the system is predicted to produce more data, than can be consumed. Such situation can arise, when the possible storages are limited to a consumer-grade hard disk or slow network and the user has instrumented functions, that are often called. User can limit, how often should be the measured data written. This is used in `SpeedLimiterFilter`.

4.1.3 InitialDropFilter

Another idea is to omit some first measurements. The main cause for this decision is, that the startup behaves differently, when compared with running the program for the longer periods of time.

Caches are waiting to be filled and are not fully utilized and some parts of program can still reside on disk being only mapped to memory and waiting to be read. Moreover, PMF runs under JVM and targets programs running under JVM. These are translated to native code, that is not heavily optimized. After getting some statistics about code execution and after considering the methods, that are expected to be ran more often, JVM compiles the optimized version **??**.

To avoid this, one can use `InitialDropFilter`, that can filter out the specified number of samples. However, even this number of samples do not have to suffice, as the storage itself needs to be compiled by JIT compiler. Therefore, if one requires strict omission of warm-up, he should sample it and remove the samples from the provided results.

4.1.4 TeeFilter

When the data are sampled and traversing pipeline, it does not allow multiple filters connected with OR clause. Each consecutive filtering results in logical AND clause. Furthermore, user sometimes require multiple outputs to divide the measurement to burst data to be written to high speed storage and slower samples to be written to slower, but bigger storage. The described items were only about branching, that is provided by `TeeFilter`.

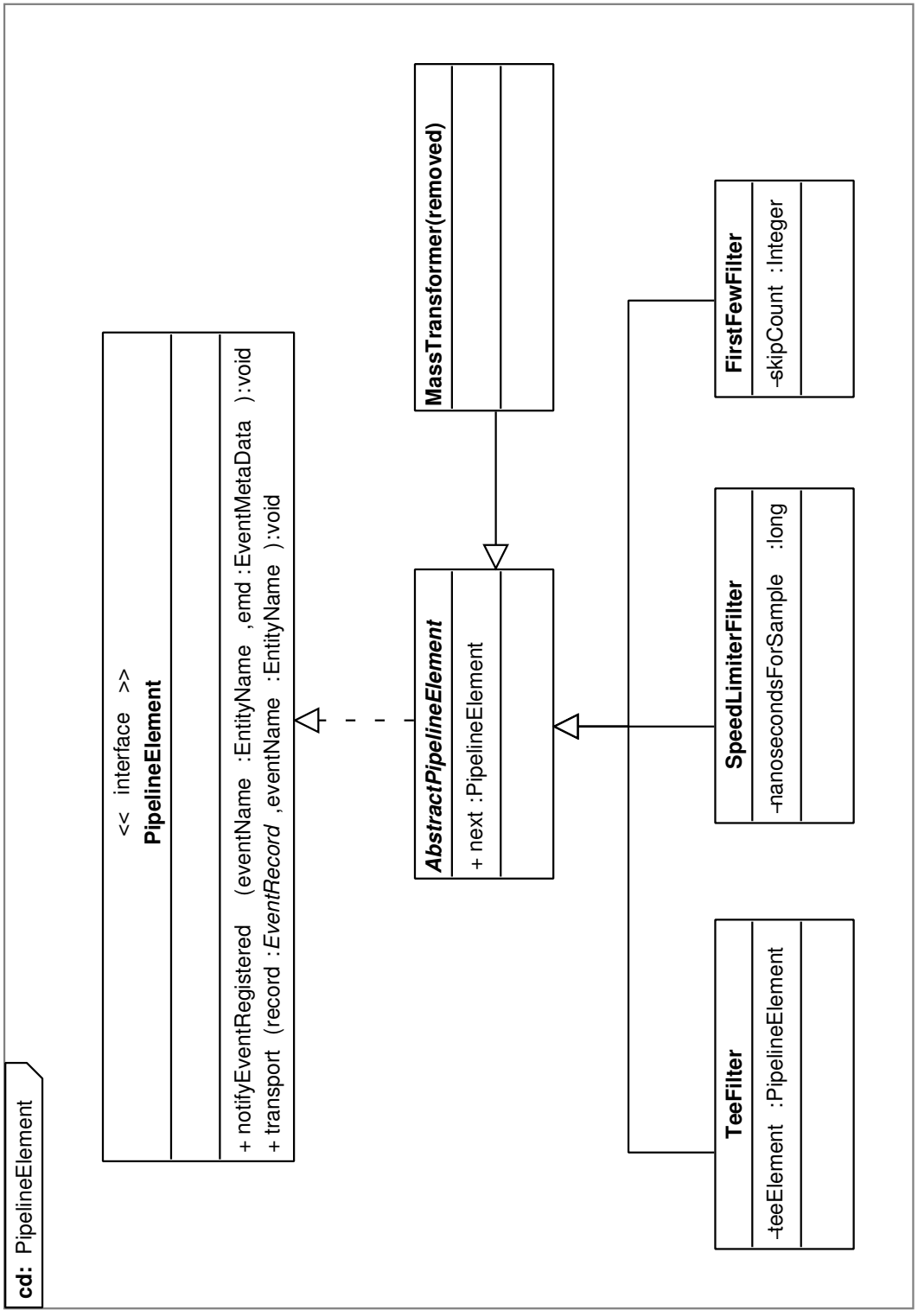


Figure 4.2: PipelineElement as in original work

With another implementation, it would be also possible to turn the storage on and off. Such implementation could use big slow storage, such as HDD to save all samples during the measurement and user would turn on the detailed sampling, when he would decide "something is going on". Earlier versions of PMF did not allow this without editing of storages. It could be avoided by not

turning some parts of instrumentation "off", but that will not allow oversight from administrator.

The implementation of `TeeFilter` expects immutability of the data, as we do not want to defensively deep-copy each element sent. The reason of our decision was purely based on performance. Without copying and by data-sharing, one branch of execution could see the modified data although there were no element modifying them in the path from root to that branch. A care must be taken to avoid data modification.

4.1.5 MassTransformer

The concept of "tee" does not rigorously define the next element in the row and therefore, no one can be sure about the ordering of these actions. Possibly parallel implementation only highlights the possibility of problems.

When one path after `tee` contains the part transforming data and the ordering of these actions is not well-defined, the other part of `tee` can also see the transformed data, but does not have to. With the parallel execution of both branches, results can be not only incorrectly transformed, but also completely incorrect. Parallel execution enables bots branches to be ran with one rewriting the source data of another. In fact, long value rewrite is not guaranteed to be atomic[**jls**], hence new unseen values, that were never sampled, can be observed.

`MassTransformer` was implemented as a `PipelineElement`, that was designed to change the data based on user requirements. More details can be seen in Figure 4.2. As it was written in previous paragraph and in Chapter 4.1.4, it is hard to implement consistent and predictable behavior without special status of `TeeFilter` or `MassTransformer`. As a solution, one can duplicate every record send thru `TeeFilter`, but it is useless when nothing is changed, it puts more load on garbage collector and it also brings in extra overhead.

The alternate solution based on providing custom `ValueHandles` means allocation of something of arbitrary length, where the data should reside during the calls. As sampled data size can change, this is not the best idea, as this should generate some garbage. Furthermore, tracking of the data by programmer cannot be done in a timely fashion. Where more records are transported, then traversing them, cloning them and replacing part of the data does not seem to be an efficient move.

Consequently, `MassTransformer` was removed from the implementation to bring something predictable and efficient, which can be used even by the user not knowing the implementation.

4.1.6 Configuration

Configuration resides in default `jpmf.conf`, which is by default in jpmfconf.xml.

There is a new pipeline configuration. Each pipeline should end with sink. Enforcement of this rule restricts users, but helps to ensure there are be no pipelines only consuming computing power without any valuable output.

Example of the configuration is displayed in Listing 4.1.

By default, default pipeline is called `default`, like here. It can be set as user wants, only with configuration of `jpmf.pipeline`. This enables multiple output

```
<?xml version="1.0" encoding="utf-8"?>
<a:configuration
        xmlns:a="http://jpda.dsrg.ow2.org/jpmfconfig.xsd">
  <!-- source and event configuration are
        like they were, nothing changed here... -->
  <!-- ... -->

  <pipeline name="direct_out">
    <sink type="jpmf.indexed">
      <property name="file">eventdata</property>
    </sink>
  </pipeline>

  <pipeline name="default">
    <filter type="speedlimit">
      <property name="nspersample">10</property>
    </filter>
    <tee>
      <property name="output">direct_out</property>
    </tee>

    <filter type="firstfew">
      <property name="count">1000</property>
    </filter>
    <sink type="jpmf.xml">
      <property name="file">eventdata.xml</property>
    </sink>
  </pipeline>

</a:configuration>
```

Listing 4.1: Example JPMF configuration

```
...
  <filter type="tee">
      <property name="output">direct_out</property>
   </filter>
...
```

Listing 4.2: Alternative specification of tee

configuration per one file.

This configuration configures pipeline `direct_out`, that saves its data directly to `jpmf.indexed`. Properties are passed to classes. In the example, by default, first filter to be applied is a speedlimit. It limits, how many nanoseconds is required between samples. Sampling too often is often unproductive, because almost nothing can change in a short time and it just increases I/O load. This filter solves that problem.

Next filter is tee. It was described earlier, so this is the special form of TeeFilter in configuration file. There is an another way to accomplish the same tee shown in Listing 4.2.

These two ways — with the tag and then itself — are interchangeable. Tag tee is interpreted exactly like a filter with type `tee`. This is true only for tee; there are no other exceptions for other types.

4.1.7 Further extensions

Custom filter implementation can be done by extending `VirtualDataTransformerProvider` and adding the class to service loader META-INF. That means editing the file of its service.

4.2 Virtual data source

As described in TeeFilter (Chapter 4.1.4), the pipeline expects immutable elements. It cannot be accomplished as long as there are elements in pipeline, that are allowed to change sampled data.

The `VirtualDataSource` was implemented as a replacement for `MassTransformer`. No transformer can transform the data, if the whole pipeline is designed to behave predictable and some of its parts, like `TeeFilter` and `MassTransformer` are not be taken specially. Note that it is not only the matter of changing them, we would also like to aggregate them, which means removal of some data from the user viewpoint. More about this problem is described in Chapter 4.1.5.

The problems with `MassTransformer` can be leveraged by a new implementation of `DataSource`. The `DataSource` should provide changing and aggregating, based on already existent `DataSource`s.

Therefore, one provides the data, for example raw performance counters of multiple CPUs. User should get one performance counter, that could provide the sum of these performance counters, their average or an another function of the statistics. Consequently, this should be done on the `DataSource` level.

`VirtualDataProbe` owned by `VirtualDataSource` creates its own `MeasurementContext` (Figure 4.3) and it samples the data required. Then, the data are transformed, even multiple times and eventually, they are sent to output as one number under given virtual sensor. This number can be sampled like any other numbers.

A bit more detailed view showing synchronous calls can be found in Figure 4.4. Synchronous calls were the only way to maintain independence of pipeline stages while preserving call speed. Synchronization itself can be slow when associated with cache ping-pong and MESI algorithm; our implementation should therefore avoid it, if possible. Intel CPUs since Core2Duo correctly predict returns in depth up to 16 [28] and so, the idea of implementation with call stack depth up to 16 will not suffer from branch misprediction. Our pipeline is not expected to be longer than few entries, so it this return stack buffer will preserve correct branch prediction even with our implementation.

Note that in case someone opts to make time-consuming aggregation, the version with one thread per `PipelineElement` could be more effective. The interface was designed to allow changes in the internal workings of `PipelineElement`. Changing the internal workings of method by using asynchronous `transport()` call will not affect that interface nor its callers.

As said, `VirtualDataSource` samples the data itself and outputs their aggregation. This aims to reduce bandwidth between two sides and it also helps to reduce I/O bandwidth. Reducing I/O bandwidth by editing `ValueHandle`s was mentioned in assignment. They hold the data, while transporting them further. Yet, they allow multiple data types. Therefore, the aggregation interface should be provided for all of them.

As far as comparable data is concerned, nothing uses `StringValueHandle`. This `ValueHandle` is hard to aggregate. Despite this, the aggregation mechanism was implemented for each data type. Another still unused data type is `DoubleValueHandle` and it will not need to be aggregated.

Remaining data types can be all handled as long value with the special care for unsigned long values. This is because IntValueHandle holds integers, that can be losslessly converted to longs and LongValueHandle are long themselves. This can simplify implementation, together with our `ParserUtil.setHandleInteger`.

The interface is made of transform calls, where the user of Performance measurement framework supplies the `ValueHandle` he wants to transform. Transformed results are not returned, but the correct output has to be set beforehand.

We are aware any independent settings outside constructor are discouraged, because programmers can forget calling it, but it simplifies the overall implementation with the possibility of reduced overhead. In this matter, we are getting information, how many outputs are provided for given input and then set outputs and next inputs accordingly, so that there is no more configuration required and every `Transformer` gets its temporary storage based on input. Thus, thanks to the exact fitting, there will not be unused space allocated just for case the Performance measurement framework could need it.

4.2.1 Configuration

Configuration of `VirtualDataSource` is similar to the configuration of other Linux probes. Programmer has to define and describe, what he is going to do and which value types should be used. The configuration should be placed under `etc/probeconfig.d/` and the name should end with `.xml`.

Here is an example of configuration. There is a sensor group called `netststs` and sensors `bytes` and `packets`. Value-kind, name and long descriptions are provided only for user getting metadata. The name should be unique, but now, it does not affect framework internal workings.

There are two used properties used to set `VirtualDataSource`. One is used as a specification, where should be the data obtained and the other specifies, what should be done with them.

More precisely on configuration level, the first one is called `sources`. It is a space separated list of all sources programmer wants to aggregate to the given instance. URI-like sensor naming in form of `datasource/probe/sensor#instance` is used. The instance part can be omitted, but it should be avoided, if one does not want to get the unstable sampling. There are computers with USB network cards, which fail in this example in the case `sum` would be ready for only exact number of inputs. When the card is disconnected, the network interface disappears.

Specification of the instance is therefore generally recommended. Note that it is not the ultimate solution in case hardware can vary.

The second property is `aggregators`, that contains space-separated aggregators in the order they should be ran. Numbers of inputs, outputs and their types should match. The number of `ValueHandle`s accepted by the first aggregator should match the number of sources specified in properties. When they do not match and the problem cannot be solved by omitting some of the records, an exception will be thrown. We considered throwing an exception every time the expected input does not match the real input, but that makes this sampling unstable. More about this instability is described further in the text.

The configuration shown in Listing 4.3 should be placed somewhere under `etc/probeconfig.d` with the extension `.xml`.

This is the first part of the configuration. The other part is much easier. Programmer needs to add a text to `etc/probeconfig.d/probeconfig.xml` while preserving its XML structure. Fortunately, it is not hard. Moreover, any other record can be copied and modified according to required configuration. Resulting configuration is shown in the Listing 4.4.

The group here should be the same as the group in sensor configuration. Otherwise, the `DataSource` will not match the proper data together and the group will not be offered.

4.3 Mapping data source

The last and the most omitted of the changes is `MapDataSource`. It is not a real `DataSource`, that would get the data from the operating system and provide it to the other parts.

```
<?xml version="1.0" ?>
<a:configRoot
        xmlns:a="http://jpda.dsrg.ow2.org/config.xsd">

<sensorCfg group="netststs">
    <instanceCfg for="bytes"
            value-type="ulong" value-kind="gauge">

      <name>All bytes</name>
      <longdesc>Sum of all bytes</longdesc>
      <property name="aggregators">sum avg</property>
      <property name="sources">
                linux.jni/netstats/rx_bytes#eth0
                linux.jni/netstats/tx_bytes
      </property>
    </instanceCfg>
    <instanceCfg for="packets"
            value-type="ulong" value-kind="gauge">

      <name>All packets</name>
      <longdesc>Avg of all packets</longdesc>
      <property name="aggregators">avg</property>
      <property name="sources">
                linux.jni/netstats/tx_packets
                linux.jni/netstats/rx_packets
      </property>
    </instanceCfg>

</sensorCfg>

</a:configRoot>
```

Listing 4.3: Sensor configuration in the new VirtualDataSource

```
...
<virtual group="netststs">
    <identifier>netstats</identifier>
    <name>Network stats</name>
</virtual>
...
```

Listing 4.4: Data source config in the new VirtualDataSource

```
<?xml version="1.0" ?>
<a:configRoot
xmlns:a="http://jpda.dsrg.ow2.org/config.xsd">

<sensorCfg group="netststs">
    <instanceCfg for="rxbytes" value-type="ulong"
                                    value-kind="gauge">
      <name>All bytes</name>
      <longdesc>Sum of all bytes</longdesc>
      <sourceInstances for-instance="default">
            linux.jni/netstats/rx_bytes#eth0
            linux.jni/netstats/tx_bytes
      </sourceInstances>
    </instanceCfg>
    <instanceCfg for="txbytes" value-type="ulong"
                                    value-kind="gauge">
      <name>All packets</name>
      <longdesc>Avg of all packets</longdesc>

      <sourceInstances for-instance="default">
            linux.jni/netstats/tx_packets
            linux.jni/netstats/rx_packets
      </sourceInstances>
    </instanceCfg>
  </sensorCfg>

</a:configRoot>
```

Listing 4.5: Configuration for MapDataSource

It is dedicated to mapping names, when multiple sets of names are provided. Although `VirtualDataSource` provides aggregation, that could replace this data source, the configuration cannot be so usable, as it is much more generic. Furthermore, when glued together, it cannot be so fast.

On the one hand, `VirtualDataSource` own the storage and can save whatever there. It can transform everything into different value, but the transformation costs CPU time. With `MapDataSource`, everything can be mapped and there will be zero overhead after the initial cost.

On the other hand, `MapDataSource` reads the descriptions of other `DataSource`s and mimics them. It provides new names for the contents, that are still to be sampled. However, after providing sampling context, all requests are sent directly to the real `DataSource`s, without any redirection.

Configuration is shown in Listing 4.5. Note the `sourceInstances` tag, that is used for quick grouping of values. It could not be used in `VirtualDataSource`, as there was more configuration to save.

```
lock.lock ();
{
  // The buffer's empty...
  if (__firstFilledItem < 0) {
    lock.unlock ();
    return false;
  }

  line = __firstFilledItem;
}
lock.unlock ();
// ....
// something with buffer
// ...
lock.lock ();
{
   __occupiedCount--;
   if (__occupiedCount == 0) {
     __firstFilledItem = -1;
   } else {
     __firstFilledItem =
       (__firstFilledItem + 1) \% __capacity;
   }
}
lock.unlock ();
```

Listing 4.6: Triggering race condition in the original implementation

4.4 Bugs in the old implementation

4.4.1 Race condition

Old implementation was tried and benchmarked before we started working on it. After the first benchmarks, random bug causing cycling appeared.

That seemed to be a race condition, that was already mentioned in this chapter. The bug was found in `SingleContextMemoryStore`, more precisely in its `read()` method. The locking was used to synchronize the access, but its use was incorrect, as it did not cover whole critical section (bug in similar area was predicted in Section 2.5.5).

The snippet in Listing 4.6 shows, how to trigger bug with synchronization.

This part of code contains "Time of check to time of use" (TOCTOU) bug. Variable `__occupiedCount` is subtracted every time, but the program never checks, whether it is negative. From here, if there is only one free item to take, two threads can proceed to the line called "something with buffer". Afterward, the crash is inevitable, if there are not be more threads to save this. The first exiting thread sets `__occupiedCount` to 0 and sets `__firstFilledItem`, so that no one can pass the first check. Yet, the second thread have already passed first check and it is there to subtract one from occupiedCount, so it will be negative. The

condition will result in illegal state of application, because `__firstFilledItem` is set to something positive and therefore other parts think there is something new to consume. When summed up, improper use of synchronization results in repetitive writing of some events, that ends, when the hard disk is filled.

Something similar was found in the function `record()`. We decided to not reimplement it, but only to fix the synchronization to get proper results from following benchmarks.

4.4.2 Measurement of time taken

Benchmarking PMF and Kieker revealed the significant differences in sampling speed. The cause was found to be extra 2 calls of `System.nanoTime()`, that were not used in current code base. It was designed to serve with sampling time, but as measurement of sampling time increased the time itself by factor of 3, it was removed.

It can be explained by the way Java gets system time. In Linux JDK, syscall is issued, which results in reduced performance.

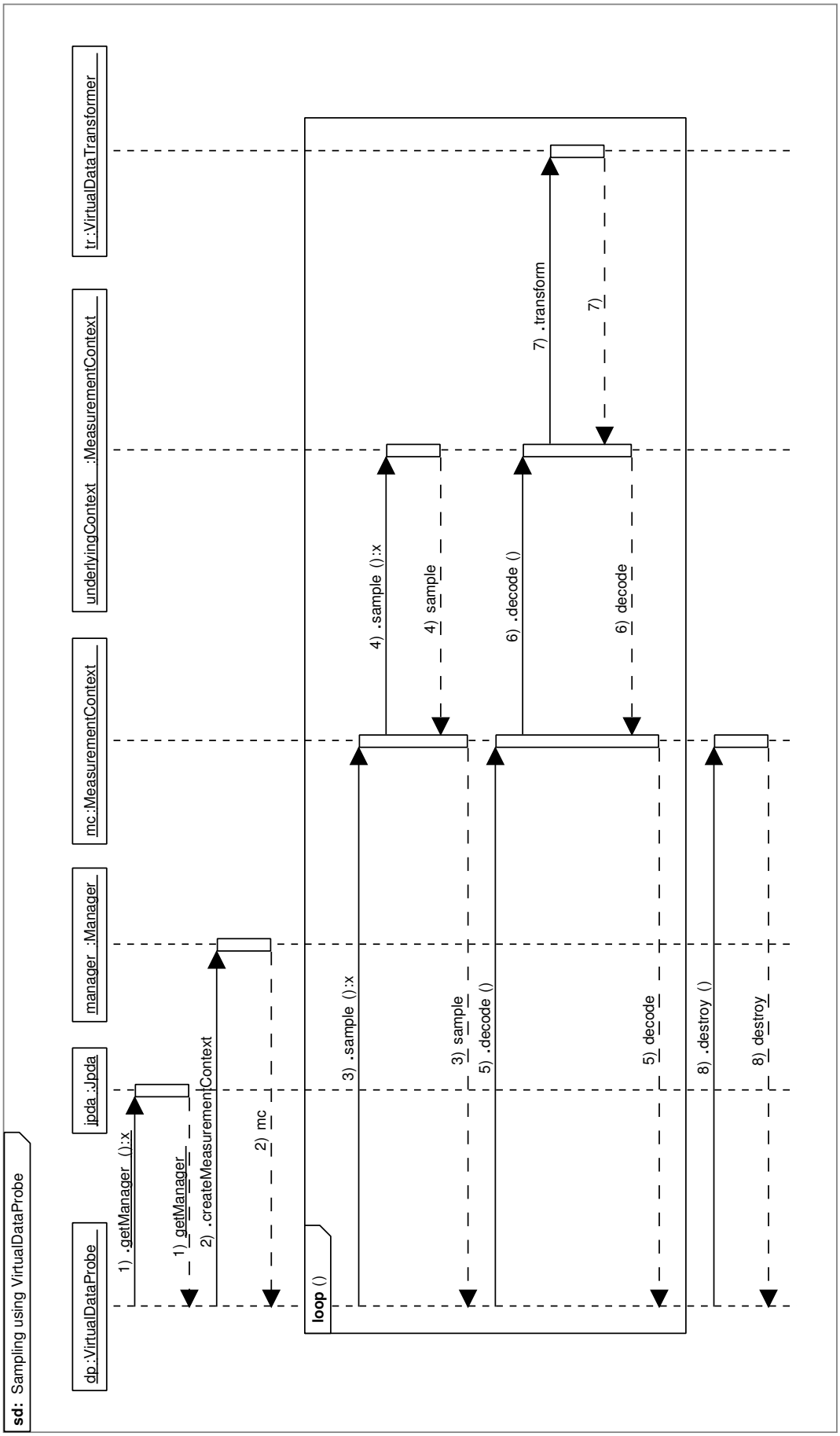


Figure 4.3: Example of sampling as VirtualDataProbe

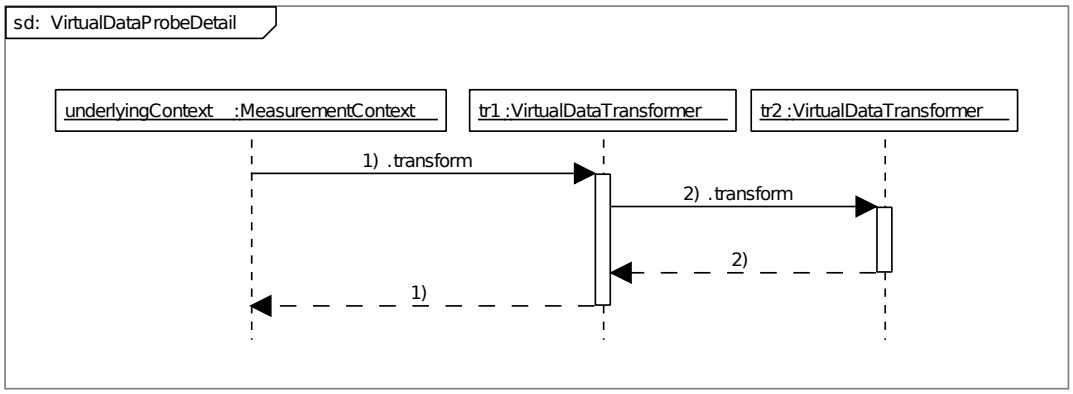


Figure 4.4: Example of sampling as VirtualDataProbe

5. Evaluation

When a new program is programmed or the valuable parts are changed, not only its output is interesting, but also the performance. That means the time required to process input and provide output. The performance and overhead of the implementations have to be compared and evaluated to reveal performance regressions and to recognise the faster implementation.

There are multiple performance-related measurements parts to be evaluated. We would like to compare the performance difference between the old a the new implementation. Also, the results of JPMF should be compared with the results of different programs providing similar services.

5.1 Experiment design

Measurement of overhead is not an easily and objectively solvable problem. First, there needs to be a benchmark, that will run a benchmarked portion of code. An ideal benchmark should perform operations similar to real-life programs, so that similarity in benchmark results correlates with the similarity of real programs.

Furthermore, measurement of overhead only does not provide usable example. Overhead in certain situation could be 1% or 100%, but we cannot say, whether the improvement was significant. Therefore, we decided to compare the performance with another program, that was built with similar ideas in the mind. That should give us usable results, which program is better.

To compare competiveness, it is a great idea to compare it with another tool described here - Kieker. Kieker itself offers benchmark MooBench. The benchmark is not full-fledged. It consists of simple cycling and busy waiting till benchmark time ends. However, it provides the results in CSV format. As it is the part of Kieker, we do not have to edit Kieker to work with it.

Judging the performance should be done on the same level. There is no point in comparing PMF utilizing all features with Kieker using only basics. As optimized and built for MooBench, Kieker offers only sampling, that consists of method name, method entry time and method leave time. PMF configuration was therefore edited to provide only the same amount of information.

Even after edit, PMF provides more robust output formats, that allows turning sensors on and off. This could bring in some slowdown, that we cannot measure precisely without changing the whole architecture.

In spite of that, Kieker provides ability to be ran without any output. With the output omitted, all other parts would send all the data as if output was used. This helps, when comparing overhead of whole framework with the overhead of I/O. We have also implemented the output called EmptyTransportProvider just to measure, how many time is spent on formatting and writing the data.

5.1.1 Possible errors

As we are measuring something, we should not forget to mention, where are the possible errors, even systematic ones. One problem are disk writes. If one cannot reduce them, sampling results will be biased in favor of the one, who writes less.

That seems to be fair, but when writing less affects possible extensions, it should be avoided. It is known, that Kieker uses more effective data format - it writes everything in simple CSV.

One more systematic error arises from use of different instrumentation. PMF uses ASM to instrument the bytecode; Kieker uses AspectJ and does not instrument repeated function entries. When recursive method calls are used, Kieker will instrument and measure only the first invocation of the recursive function. On the other hand, PMF will instrument every invocation, even when it is nested. This is not the case of comparing old PMF with new PMF.

5.1.2 Independence

One unexpected result was unprovable independence. Data looked like self-correlated when seen by eye, but mathematical methods used when finding self-correlation did not lead to any usable results. Self-correlation was observed on the range bigger that is allowed for lag.plot and was unstable, i.e. period was oscillating around one value.

Without independence, we cannot claim anything about a real distribution hidden behind the data, if we are missing the real dependencies. We should say, that the independence was not explicitly denied by data nor we could prove we are missing it. Therefore, we will try to expect it.

We can compare means, medians or standard deviations and if the data are dependent in the same way, we can get some results.

Comparison of means, medians and possibly also extremes can help us when comparing two unknown distributions, but it cannot lead us to difference in an underlying distribution. With standard deviation and Chebyshev's inequality, we can put limit on probability the value will be outside certain bounds. More precisely, we can say,

$$P(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$$

5.2 Experiment setup

We measured the performance using Intel(R) Xeon(R) CPU E5320 @ 1.86GHz, Linux 4.0.4-303.fc22.x86_64, 8175472 kB MemTotal. The benchmark ran from tmpfs to eliminate waiting for I/O. First 10 000 values were dropped, because state during the warmup does not make the overall image of the system.

There are some problems with measurement evaluation. All the benchmarks used in evaluation are built to return one number stating, how long did the benchmarking take to complete. This is a good predictor, as it sums up to everything from benchmark – how long will it take for the program to complete.

There are also some downsides. We cannot assume anything about the data and so, we should not mathematical theorems.

5.2.1 Kieker

We have downloaded Kieker, that provides dynamic software analysis and application monitoring, just like our Performance Measurement Framework.

Kieker contains a benchmarks `MooBench` and is configured to work with it, so we decided to use it.

Kieker, as is, does not provide sampling sensors in its microbenchmark. When we oversimplify it, it just writes method entry time, leave time and its signature to log file. The disadvantage is, that sensors without sampling are not used and we won't get the bottleneck, if it is in non-measured part. However, this is also an advantage, as we can localize bottlenecks more easily. This solution of not providing any additional details was also used by JPMF to measure really minimal overhead, without arguable additional options, that can be turned on and off.

5.2.2 MooBench

`MooBench` is a benchmark provided by Kieker. It is calling measured function in a rapid succession, hence the measured overhead is exaggerated. It is also not real-world and balanced test, but as we are comparing overhead of our implementation and we cannot move other limits (like I/O), it could serve as a help in comparison of loss of overall performance.

We measured Kieker and PMF using the same MooBench benchmark. Full Kieker benchmark can be found in
`kieker-examples/OverheadEvaluationMicrobenchmark/MooBench`
in complete Kieker sources. Full benchmark including our changes to sources can be found inside the distribution package. It is not our work, we have only done some changes.

We changed the sampling in `benchmark.sh`. Now, it should benchmark also PMF in the similar way like Kieker did it.

5.2.3 DaCapo

Another benchmark suite is called `DaCapo` [29]. It should solve the problems with only very synthetic load. DaCapo tries to emulate real-world problems and therefore is closer to the real-world programs. It contains more subbenchmarks, that will be mentioned later.

Slow sampling

`DaCapo` uses custom startup to get stable results. It tries to eliminate overhead by the special preparations like multiple warm-up rounds. However, when they are used, thousands of samples cannot be collected, as it would took too long. To avoid the issues and to get more samples in reasonable time, we do not strictly follow the ideal way to benchmark the differences.

When benchmarking in the preferred way, we should measure the biggest data possible to eliminate overhead. DaCapo usually provides small, default, large and huge datasets. When preparing the benchmark to be run, we tried to run it and single run with default dataset took 51 second, small run took 21 seconds. If we required multiple measurements to avoid distortions by JIT, even 21 seconds of benchmarking is too long, so we have chosen the small dataset.

Furthermore, in benchmarking, warmup rounds should be left to warm benchmark up and we should measure only final results. This solution would take plenty

of time, as with 10 warmup rounds and single measurement rount, it would take minutes to get a single result. Therefoe we use also warm-up rounds with the first few warm-up rounds cropped.

We were deciding, which part of rounds should be cropped. Java JIT in Hotspot runs after 10 000 iterations[30], but we do not have capacity to handle at least ten thousands of iterations. If we took the result from preparation, it is 10 000 iterations, 21 seconds each, that makes almost two and half days of computation of useless results. Then, useful results would still be awaited.

In that matter, we have lowered our requirements. We are making only 10 000 iterations and we will throw away the first 10%. We are aware this computation can bring in systematic error. With 14 benchmarks to perform, choice is harder.

Randomly incorrect results

`DaCapo` uses sampling and verification of results based on checksums to ensure the correct code was run. However, it revealed the problems in 6 of 14 benchmarks, that were not returning the correct results seldomly both with Kieker and PMF. The completion time of benchmark were then incorrectly reported to be few milliseconds instead of thousands of milliseconds.

Thus, we remove the results of the following subbenchmarks from comparison:

- Avrora – random missing files, some successful attempts
- Batik – random missing files, failed digest verification of result, some successful attempts
- Eclipse – some classes not found, failed digest verification of result
- FOP – missing files, failed digest verification of result, no successful attempts
- H2 – failed digest verification of result
- Jython – missing files, some successful attempts
- LUindex – DacapoException: cannot write to index directory

Drawbacks

One of the major drawbacks is the basic property of all benchmarks. When programmer gets the benchmark, he tends to optimize benchmarked program, but the optimization is not based on real performance, but only on the benchmark. It is hard to assess, how this affects the overall performance of Kieker, but we should bear it in mind as a source of systematic error.

On the other hand, during benchmarking, we noticed we are slower than Kieker and investigated it. We found out, that we are doing extra two calls of `System.nanoTime()`, that are not necessary. It was two times on entry and two times on exit, that means four extra calls to the function, which result is never used. These four times were used to measure time spent during the whole measurement. Kieker does nothing like that and no one used it in our system, so we have removed it. Further investigation did not reveal the exact cause

except something well-known - syscalls are slower than the other calls. Our JVM initializes reading for once by assigning value to its own pointer to function and therefore is doing virtual call. Under that, Linux `gettimeofday()` is called, that should not cause the slowdown. However, we should bear in mind, that slowdown could be almost unnoticeable — the time difference caused by measurement was just bigger, than other slowdowns.

5.2.4 JPMF

Comparison of two JPMF implementations was done using MooBench. It provided everything required and was an independent benchmark, which was JPMF not tuned. Therefore, we believe its results can be trusted.

5.3 Measurement

5.3.1 Kieker vs PMF – MooBench

This measurement will be orientational and will also provide data from non-instrumented run to be compared with the other data.

Hypothesis

We tried to use bootstrapping, but is not the silver bullet and therefore, we decided to use non-parametric tests on hypothesis:
H_0: Kieker runs in less time than PMF.
H_1: Kieker runs in equal or greater time than PMF.

Measurement

As described in Chapter 5.2.1, we have disabled all sensors and used only fields in this benchmark.

PMF and Kieker were compared, while their default configuration was being used. That means Kieker is instrumented using `AspectJ` and PMF is instrumented using `ASM`.

We have multiple measurements - dependence between the overhead and the time taken in our test method. We sampled the time taken with respect to in-method time. Sampling was performed in steps of 5000 nanoseconds starting from 0 nanoseconds to 0.5 milliseconds in steps of 5000 nanoseconds.

Evaluation

The distribution is heavily right skewed by outliers. That means we are without possibility to approximate something, as we are not working with the normal distribution. We can find Var or σ, that are big and so unusable. We decided to use subsampling and bootstrapping techniques.

Note that results and improvements differ with different settings. Default configuration uses longer in-method time, that decreases proportional part of the overhead. This can be approximately seen in the plot comparing run without instrumentation (average) with runs with instrumentation (again average).

When the results are evaluated, it can be seen the bigger execution time is, the less overhead will the instrumentation have. This is predictable — extra instrumentation code should execute in the almost constant time, that does not directly losses from the execution duration. There can be a correlation, as a short run does not get enough time to get its own working set to all caches.

There were troubles with recognition between PMF and Kieker results, when we used different types of plots. The plot with 95% confidence intervals is displayed in Figure 5.1. Confidence interval of Kieker is the red and dark red, PMF is the green and dark green. We can clearly see the distinction, although it is not on the the whole range.

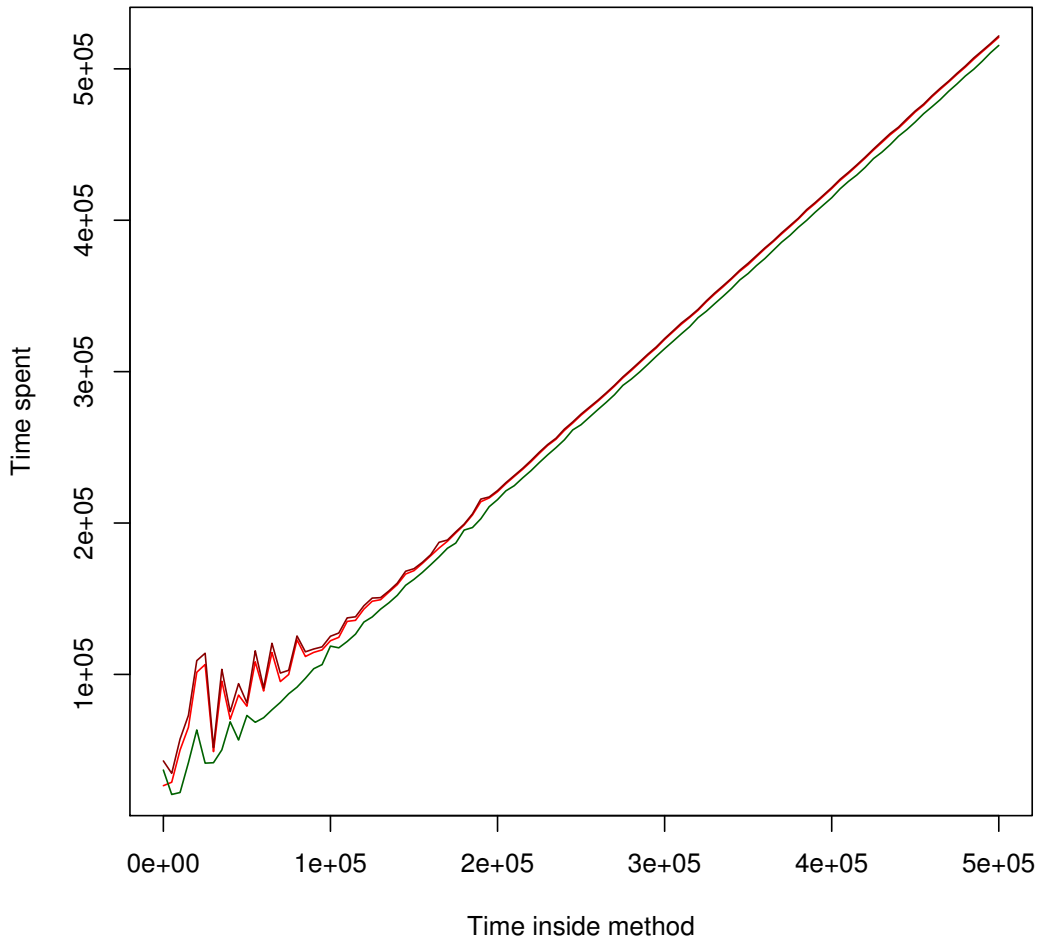


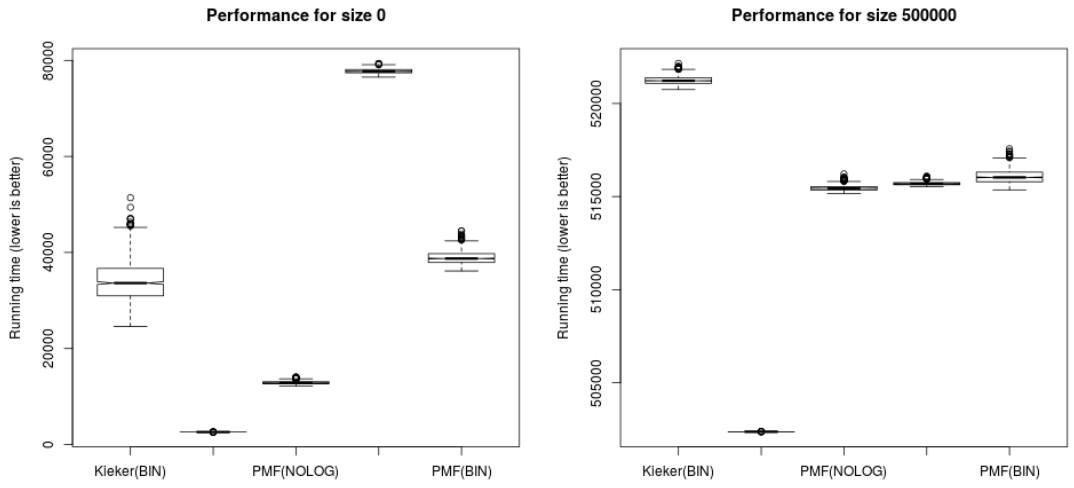
Figure 5.1: Showing almost constant overhead of Kieker and PMF after bootstrapping

These confidence intervals were calculated from resampling to 1000 samples using mean.

These hypotheses were tested using two sample Kolmogorov-Smirnov tests (ks.test) and also double sided Wilcoxon Rank Sum test. Wilcoxon Rank sum test could not get reliable evidence against H_0 with 0 wait time, resulting in p-value of 0.6462.

For all other times and also with the wait time of 0, Kolmogorov-Smirnov test showed very strong evidence against H_0 and refused hypothesis on $\alpha = 0.01$ using Wilcoxon Rank Sum tests and also Kolmogorov-Smirnov test. R data analysis suite showed p-values $\leq$ 2.2e-16 for all of the tests except one noted.

The performance of different outputs in PMF and Kieker was also compared to assess overall slowdown. This disproved the prediction the overhead is almost constant. Some plots can be found in benchmarks. The first there is Kieker, that is noticeable slower from the others. The second one is a measurement of sampling time, when there was no instrumentation. Therefore, the overhead can be seen, even without any comparisons with Kieker. The next one is PMF doing everything, but without output to disk. The another one is our `XmlFileDataStorage` and the final one is `IndexedFileDataStorage`. Complete plots are in `MooBench_byMethodTime/tmp/graf*.png`, some of them for the extreme values follow.



(a) Function execution 0ms

(b) Function execution 0.5ms

Figure 5.2: Comparison of Kieker, nothing, PMF without output, PMF/ASCII and PMF/BIN performance

We also provide "speedup plot" to compare speedups of the two implementations with no instrumentation whatsoever. As you can see in Figure 5.3, using instrumentation and performance measurement causes negative speedup (slowdown). The slowdown vanishes with in-method times increasing. By comparing the performance data of PMF and Kieker, using PMF with binary logging causes average slowdown of 13μ seconds, while using Kieker causes average slowdown of 17μ seconds. When we compare speedup of Kieker compared to PMF, we get the Figure 5.4.

5.3.2 Kieker vs PMF – DaCapo

DaCapo contains multiple benchmarks: LUsearch, PMD, Sunflow, Tomcat, Tradebeans, Tradesoap and Xalan. Other were discarded in part Experiment setup (5.2). To simplify naming and to avoid repetitive writing, we will use X as a placeholder for one of these benchmarks.

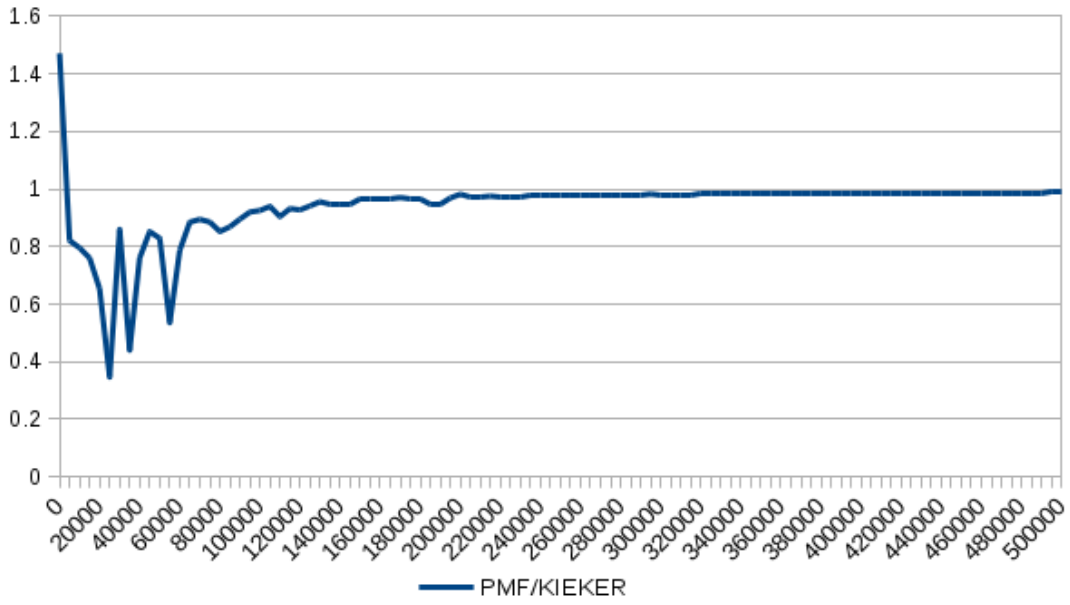


Figure 5.3: Speedup of Kieker and PMF per method time

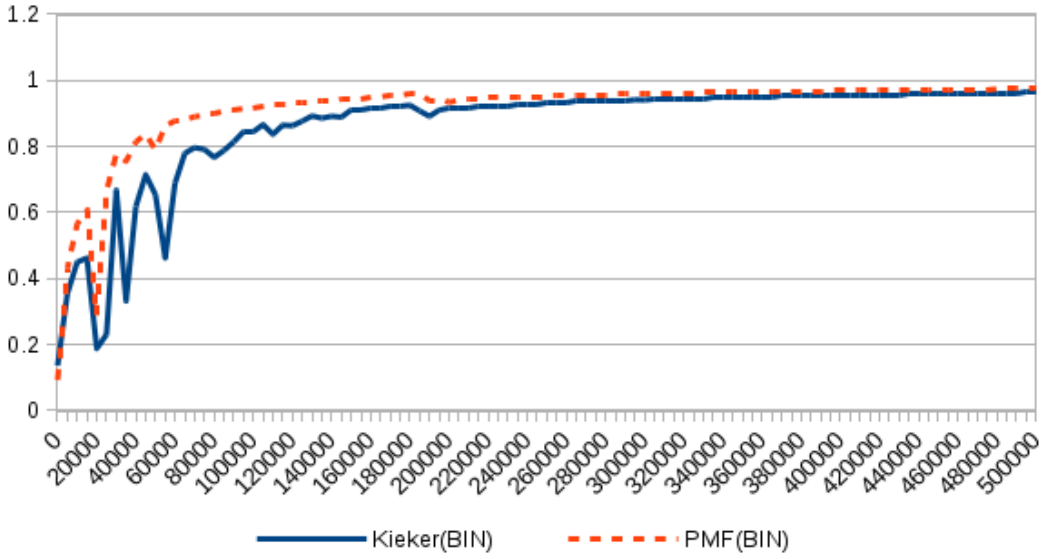


Figure 5.4: Speedup of Kieker with PMF as base per method time

Hypothesis

The hypothesis:
H_0: Kieker runs in less time than PMF in benchmark X.
H_1: Kieker runs in equal or greater time than PMF in benchmark X.

Measurement

Measurement took more than one week to complete, so our approximation of long-running benchmark was true. Benchmark provided values, that were later used. Medians of results are in Table 5.1.

Table 5.1: Overview comparing running times in ms.

	Median (Kieker)	Median (PMF)
LUsearch	246	247
PMD	12	12
Sunflow	701	687
Tomcat	1303	1293
Xalan	122	137
Tradebeans	672	678
Tradesoap	3125	3144

Table 5.2: Results of tests as p-values

	KS test	Wilcox
LUsearch	0.991	1
PMD	0.99	0.9522
Sunflow	**<2.2e-16**	**<2.2e-16**
Tomcat	**<2.2e-16**	**<2.2e-16**
Xalan	1	1
Tradebeans	0.9999	1
Tradesoap	1	1

Evaluation

Kolmogorov-Smirnov test (KS test) and Wilcox test (Wilcox) were used to evaluate the results. The results are in Table 5.2, depicted values are p-valus using the given tests. Highlighted values disprove the hypothesis H_0 at $\alpha = 0.01$.

The measurements show two statistially significant results, but due to the issues mentioned in setup (5.2) and small differences between the values reported, it is hard to draw a conclusion from it. Bottleneck of DaCapo was mentioned to be in kernel and therefore, we cannot get significant differences between the two datasets.

5.3.3 Kieker vs PMF – with DiSL

Hypothesis

Hypotheses used:
H_0: CDF of PMF with DiSL does not lie below the CDF of Kieker with DiSL.
H_1: CDF of PMF with DiSL lies below the CDF of Kieker with DiSL.

Measurement

It was found out, that the instrumentation of Kieker is slower when compared to PMF. One possible reason would be wrong instrumentation framework, as Kieker uses AspectJ and PMF uses ASM. To avoid the issue, we decided to compare performances of Kieker and PMF by a new implementation of DiSL [31] to the both tools.

Kieker already implemented DiSL supporting class, that was designed to work out of box. However, it did not run with new DiSL, because private class variables as constants were used and DiSL failed when running it. After fixing the bug by inlining the constants by hand and the another bug with `NullPointerException` bug caused by unexpected bootstrap `ClassLoader` (`getClassLoader()` was returning `null`), the benchmark could be finally ran.

PMF did not provide the support out of box, hence it had to be implemented. The implementation had some drawbacks, that were caused by state, that had to be preserved across the calls. DiSL implements `@ThreadLocal` annotation with the fast local variable access to facilitate state preservation, but in PMF, the state of sampling can be accessed from multiple threads. With the use of custom holders, `org.ow2.dsrg.jpmf.agent.disl.DiSLClass` and `DiSLEventSource` from the same package were produced.

This fix in Kieker suffered from some problems, as the severe errors were printed to console:

```
SEVERE: Failed to add new monitoring record to queue.  Queue is full.
Either increase 'QueueSize' or change 'QueueFullBehavior' for the con-
figured writer
```

We could reconfigure Kieker, but we think reconfiguration would bring in more parameters and additional bias, that cannot be fully explained. Therefore, everything was measured using the old implementation.

Evaluation

Results are shown in Figure 5.5 and Figure 5.6. As one can easily see without complete evaluation, the performance is comparable when comparing medians, but the difference is shown with means. That shows we achieved one of our requirements – getting stable results.

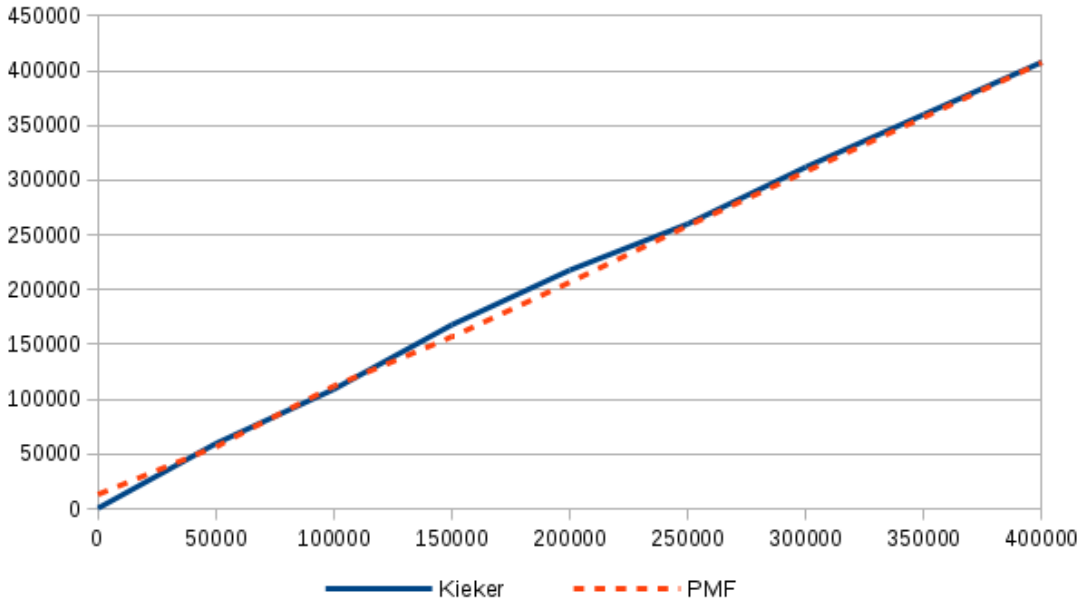


Figure 5.5: Comparing medians of PMF and Kieker with DiSL deployed

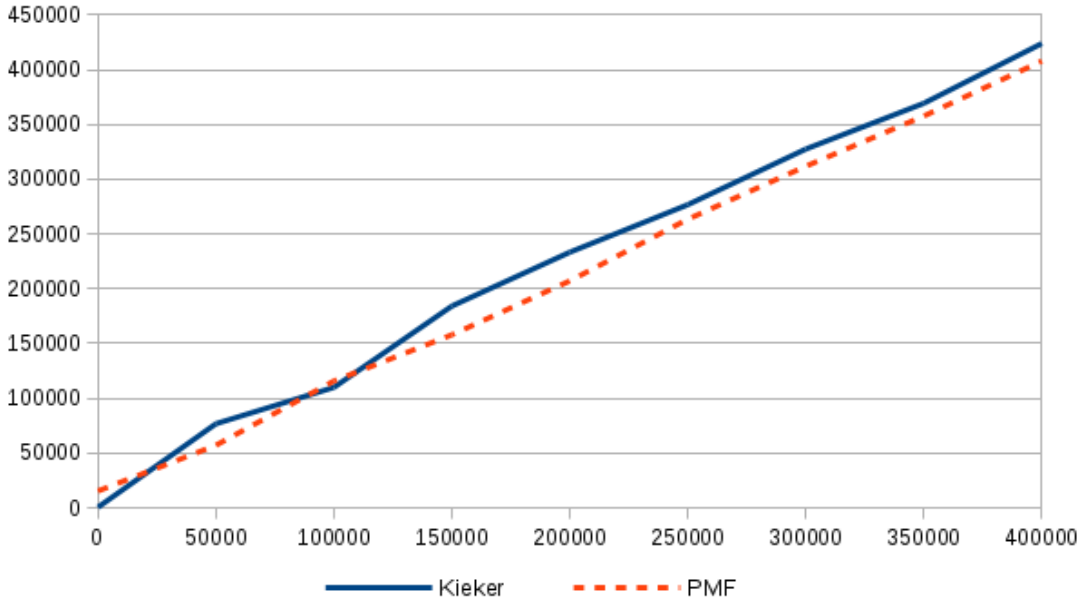


Figure 5.6: Comparing means of PMF and Kieker with DiSL deployed

Table 5.3: Comparing p-values when testing results of benchmarking of PMF and Kieker with DiSL

Running time [μs]	KS	Wilcox
0	1	1
50000	**<2.2e-16**	**<2.2e-16**
100000	**<2.2e-16**	1
150000	**<2.2e-16**	**<2.2e-16**
200000	**<2.2e-16**	**<2.2e-16**
250000	**<2.2e-16**	**<2.2e-16**
300000	**<2.2e-16**	**<2.2e-16**
350000	**<2.2e-16**	**<2.2e-16**
400000	**<2.2e-16**	**<2.2e-16**
450000	**<2.2e-16**	**<2.2e-16**

Although K-S test is used when comparing CDF, we made Table 5.3 comparing also p-values of Wilcox test to compare populations instead of only their CDFs. Bold values are significant and therefore disprove H_0 on $\alpha = 0.01$.

From the table and also plots, PMF is faster in all times except when the function finishes immediately. It is because of the implementation of DiSL in PMF, that uses lazy loading of instrumented functions. Therefore, the first run is slower, but in the consecutive runs, PMF wins (Wilcox show one more exception).

The results and benchmarks were in favour of this setting instead of the original one. We should note that DiSL uses

Sampling time was reduced even when compared with original ASM. Another interesting result to note is the stable returned output and outliers.

5.3.4 The new PMF vs the old PMF – same configuration

As PMF was edited, the performance impact of the edits should be measured. For simplicity, we will call PMF before our edits "old PMF" and performance after our edits "new PMF".

Hypothesis

First, the tests were used as noted before. We took:
H_0: Old PMF completes the benchmarking in less time than new PMF.
H_1: Old PMF completes the benchmarking in equal or greater time than new PMF.

Measurement

First, the technical preparations. We provided XML with changed configuration, so we have to compare it with something similar. We should not use features like `TeeFilter`. Everything can be simply reverted to the time, when our changes were only in our local repository.

Some changes, that could affect performance, like newer ASM, were also performed. After all, we would like to compare the real-life performance. Newer architecture is designed in order to withstand higher load, but it cannot be guaranteed. Therefore, measurements serve as an insurance, that no performance regressions were introduced.

This benchmark should be useful, as the execution time of sampled program is measured, while the benchmark is not slowing the measurement down by performing notoriously known slow operations, such as I/O load.

After fixing the bugs in original implementation, the benchmark returned the expected values.

Evaluation

To compare them, non-parametric tests (wilcox, KS) outlined in the previous subchapter were used.

Both non-parametric tests finished with p-value $\leq$ 2.2e-16, that is the rigorous signal old PMF is not faster.

To assess the difference, mean-based bootstrapping was used again. The 95% confidence interval provided by bootstrapping was, that for running time zero, old measurement took 185.708 to 185.848 ns while the new implementation completed the benchmark in 135.096 to 135.181 ns. The difference can be seen as $\tilde{2}7\%$ speedup in the new PMF.

With the biggest wait settings, old PMF could perform its task in 637515 to 637846ns and new PMF in 587010 to 587142 ns. Everything is still based on bootstrap and 95% CI. That shows a stable trend, that the newer PMF is about $50\mu s$ faster. The difference is small, but it means noticeably lower overhead when sampling functions, that runs only for a short time.

6. Conclusion

There were multiple changes, that lead us to the conclusion about completed goals.

We have implemented `PipelineElement`, that is used during data processing. Except that, `VirtualDataSource` and `MapDataSource` were implemented to serve the goals, that could be accomplished by `PipelineElement`, but with the various implementation and performance issues.

Pipeline had to provide an aggregation, that could be hard to access without dedicated data source. The aggregating datasource was implemented as `VirtualDataSource`.

Renaming of the values from multiple datasources predicted in the JPMF thesis [26] was accomplished by `MapDataSource`. The main reason to design it was improving the performance.

We have compared performance of data harvesting in JPMF with an another framework providing the similar possibilities, Kieker. Comparison yielded a result of advantage of using JPMF. The validity of comparison also led to the idea of DiSL instrumentation. That instrumentation was also implemented and sucessfully tested against the other framework with the similar load. Our implementation was faster in the majority of input values.

Furthermore, we have compared the old implementation before the changes were introduced with the new implementation containing pipeline. Evaluation shows the statistically significant difference in favour of the new implementation.

Therefore, we have implemented new configurable and extensible system for performance data sampling and filtering, that provides aggregation and sensor renaming. Despite extending the framework, we have not sacrificed the performance. Moreover, we have helped framework to get more consistent results, which everyone will profit from.

Bibliography

1 WWW.PLUSKA.SK. Vlanajší kolaps danového systému stál Slovensko aspon 10 miliónov eur. 2013. Available also from WWW: ⟨http://www.pluska.sk/spravy/z-domova/vlanajsi-kolaps-danoveho-systemu-stal-slovensko-aspon-10-milionov-eur.html⟩. ISSN 1336-9776.

2 TVARDZÍK, Jozef. Daniari vymenili systém za desiatky miliónov. Predošlý pokus stál za pádom vlády. *www.etrend.sk*. 2015. Available also from WWW: ⟨http://www.etrend.sk/ekonomika/daniari-vymenili-system-za-desiatky-milionov-predosly-pokus-stal-za-padom-vlady.html⟩. ISSN 1336-2674.

3 ŠPINNER, Vladimír. Kolaps na danovom úrade? — košice:dnes. 2015. Available also from WWW: ⟨http://www.kosicednes.sk/kolaps-na-danovom-urade/⟩. ISSN 1339-7605.

4 TOMA, Branislav. Daniarom viazne nový systém - Pravda.sk. *Pravda.sk*. 2015. Available also from WWW: ⟨http://spravy.pravda.sk/ekonomika/clanok/352491-daniarom-viazne-novy-system/⟩. ISSN 1335-4051.

5 HEALTHCARE.GOV. *Health Insurance Marketplace*. 2015. Available also from WWW: ⟨http://healthcare.gov⟩.

6 LEVINSON, Daniel R. An Overview of 60 Contracts That Contributed to the Development and Operation of the Federal Marketplace Report (OEI-03-14-00231) 08-26-2014. *Oig.hhs.gov*. 2015. Available also from WWW: ⟨http://oig.hhs.gov/oei/reports/oei-03-14-00231.asp⟩.

7 OBAMA, Barack. *Obama addresses healthcare website glitches - BBC News*. 2015. Available also from WWW: ⟨http://www.bbc.co.uk/news/world-us-canada-24613022⟩.

8 ORACLE. *sar - system activity reporter*. 2015. Available also from WWW: ⟨http://docs.oracle.com/cd/E26505_01/html/816-5165/sar-1.html⟩.

9 MICROSOFT. *Perfmon*. 2015. Available also from WWW: ⟨https://technet.microsoft.com/en-us/library/bb490957.aspx⟩.

10 BARTH, Wolfgang. *Nagios: System and Network Monitoring*. San Francisco, CA, USA: No Starch Press, 2006. ISBN 1593270704.

11 OLUPS, Rihards. *Zabbix 1.8 Network Monitoring*. 2010. ISBN 184719768X.

12 REN, Gang; TUNE, Eric; MOSELEY, Tipp, et al. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro*. 2010, vol. 30, no. 4, pp. 65–79. Available also from WWW: ⟨http://dx.doi.org/10.1109/MM.2010.68⟩. ISSN 0272-1732.

13 MARS, Jason; VACHHARAJANI, Neil; HUNDT, Robert; SOFFA, Mary Lou. Contention Aware Execution: Online Contention Detection and Response. In. *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. Toronto, Ontario, Canada: ACM, 2010, pp. 257–265. CGO '10. Available also from WWW: ⟨http://doi.acm.org/10.1145/1772954.1772991⟩. ISBN 978-1-60558-635-9.

14 OPENGROUP. *ARM*. 2015. Available also from WWW: ⟨https://collaboration.opengroup.org/tech/management/arm/⟩.

15 HOORN, André van; WALLER, Jan; HASSELBRING, Wilhelm. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In. *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. Boston, Massachusetts, USA: ACM, 2012, pp. 247–248. ICPE '12. Available also from WWW: ⟨http://doi.acm.org/10.1145/2188286.2188326⟩. ISBN 978-1-4503-1202-8.

16 JÚNOŠ, Peter. *Extending Java Performance Monitoring Framework with Support for Linux Performance Data Sources*. 2012.

17 DRÁB, Martin. *Extending Java Performance Monitoring Framework with Support for Windows Performance Counters*. 2012.

18 MYTKOWICZ, Todd; DIWAN, Amer; HAUSWIRTH, Matthias; SWEENEY, Peter F. Evaluating the Accuracy of Java Profilers. In. *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. Toronto, Ontario, Canada: ACM, 2010, pp. 187–197. PLDI '10. Available also from WWW: ⟨http://doi.acm.org/10.1145/1806596.1806618⟩. ISBN 978-1-4503-0019-3.

19 ORACLE. *OpenJDK: jmh*. 2015. Available also from WWW: ⟨http://openjdk.java.net/projects/code-tools/jmh/⟩.

20 KNUTH, Donald E. Structured programming with go to statements. *Computing Surveys*. 1974, vol. 6, pp. 261–301.

21 MACEACHERN, Doug. *SIGAR - System Information Gatherer And Reporter*. 2015. Available also from WWW: ⟨https://support.hyperic.com/display/SIGAR/Home⟩.

22 DÖHRING, P. *Visualisierung von Synchronisationspunkten in Kombination mit der Statik und Dynamik eines Softwaresystems*. 2012. Available also from WWW: ⟨http://kieker-monitoring.net/download/synchrovis/⟩.

23 NETZER, Robert H. B.; MILLER, Barton P. What are race conditions? some issues and formalizations. *LOPLAS*. 1992.

24 HUNT, John. *The Unified Process for Practitioners: Object-Oriented Design, Uml and Java*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2000. ISBN 1852332751.

25 MICHAEL, Maged M. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* 2004, vol. 15, no. 6, pp. 491–504. Available also from WWW: ⟨http://dx.doi.org/10.1109/TPDS.2004.8⟩. ISSN 1045-9219.

26 BULEJ, Lubomír. *Connector-based Performance Data Collection for Component Applications*. 2007.

27 IEEE, The; GROUP, The Open. *Single Unix Specification version 4 - tee*. 2013. Available also from WWW: ⟨http://pubs.opengroup.org/onlinepubs/9699919799/utilities/tee.html⟩.

28 FOG, Agner. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. [online]. [Visited on 2014-08-07]. Available from WWW: ⟨http://www.agner.org/optimize/microarchitecture.pdf⟩.

29 BLACKBURN, S. M.; GARNER, R.; HOFFMAN, C., et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In. *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. Portland, OR, USA: ACM Press, 2006, pp. 169–190. Available also from WWW: ⟨`http://dx.doi.org/http://doi.acm.org/10.1145/1167473.1167488`⟩.

30 CLICK, Cliff. *The Art of (Java) Benchmarking*. 2009. Available also from WWW: ⟨`http://www.azulsystems.com/events/javaone_2009/session/2009_J1_Benchmark.pdf`⟩.

31 MAREK, Lukáš; VILLAZÓN, Alex; ZHENG, Yudi, et al. DiSL: A Domain-specific Language for Bytecode Instrumentation. In. *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*. Potsdam, Germany: ACM, 2012, pp. 239–250. AOSD '12. Available also from WWW: ⟨`http://doi.acm.org/10.1145/2162049.2162077`⟩. ISBN 978-1-4503-1092-5.

*

List of Tables

Listings

List of Figures

7. Attachments

Attachments for this thesis are provided on an enclosed CD. Folder structure is described in Table 7.1.

Table 7.1: Overview of contents on attached CD

`thesis.pdf`	Text of this thesis with clickable references
`jpmf`	Complete JPMF sources.
`jpmf.diff`	Diff of sources between the original sources and the sources after this thesis was completed.
`benchmarks`	Benchmarks used. Benchmarks are not completely written in this thesis