Waller, Jan and Hasselbring, Wilhelm (2012) A Comparison of the Influence of Different Multi-Core Processors on the Runtime Overhead for Application-Level Monitoring [Paper] In: International Conference on Multicore Software Engineering, Performance, and Tools (MSEPT), May 31 - June 1, 2012, Prague, Czech Republic.

# A Comparison of the Influence of Different Multi-Core Processors on the Runtime Overhead for Application-Level Monitoring

Jan Waller[1] and Wilhelm Hasselbring[1,2]

[1] Software Engineering Group, Christian-Albrechts-University Kiel, Germany
[2] SPEC Research Group, Steering Committee, Gainesville, VA, USA

**Abstract.** Application-level monitoring is required for continuously operating software systems to maintain their performance and availability at runtime. Performance monitoring of software systems requires storing time series data in a monitoring log or stream. Such monitoring may cause a significant runtime overhead to the monitored system.
In this paper, we evaluate the influence of multi-core processors on the overhead of the Kieker application-level monitoring framework. We present a breakdown of the monitoring overhead into three portions and the results of extensive controlled laboratory experiments with micro-benchmarks to quantify these portions of monitoring overhead under controlled and repeatable conditions. Our experiments show that the already low overhead of the Kieker framework may be further reduced on multi-core processors with asynchronous writing of the monitoring log.
Our experiment code and data are available as open source software such that interested researchers may repeat or extend our experiments for comparison on other hardware platforms or with other monitoring frameworks.

## 1 Introduction

Through the advent of multi-core processors in the consumer market, parallel systems became a commodity [12]. The semiconductor industry today is relying on adding cores, introducing hyper-threading, and putting several processors on the motherboard to increase the performance, since physical limitations impede further performance gains based on increasing clock speed. A fundamental question is how to exploit these emerging hardware architectures for software applications. Parallel programming languages intend to offer the programmer features for explicit parallel programming, while parallelizing compilers try to detect implicit concurrency in sequential programs for parallel execution. In this paper, we report on our research for exploiting parallel hardware for monitoring software systems.

In addition to studying the construction and evolution of software systems, the software engineering discipline needs to address the *operation* of continuously running software systems. A requirement for the robust operation of software

systems are means for effective monitoring of the software's runtime behavior. In contrast to profiling for construction activities, monitoring of operational services should impose only a small performance overhead [5].

Various approaches attempt to reduce the overhead of monitoring large software systems. Self-adaptive monitoring approaches start with a comprehensively instrumented system and reduce the monitoring coverage through rule-based (de-)activation of selected probes at runtime [3, 10, 17]. Ehlers et al. [4] demonstrate the feasibility of this approach, given that the remaining overhead of deactivated probes is negligible. Another approach to reducing the overhead of monitoring is a reduction of the amount of data written with each monitoring record. Instead of using human readable file formats (e.g. XML, CSV, or ASCII), binary files provide an efficient storage of monitoring data. Chen et al. [2], for instance, integrate a compression algorithm to reduce the amount of log data. This compression could either be handled by a spare processor core or by using generally available free CPU resources in enterprise systems.

Our complementary idea is to use (potentially) underutilized processing units (processors, cores, hyper-threads) for further reducing the overhead of collecting monitoring data. To evaluate our approach of exploiting parallel hardware for monitoring software systems with the Kieker[3] framework and to determine the positive or negative (e.g., higher overhead of communication) influence of multiple available processing units we use benchmarks. As discussed by Tichy [21], benchmarks are an effective and affordable way of conducting experiments in computer science. As Georges et al. [6] state, benchmarking is at the heart of experimental computer science and research.

As contribution of this paper, we present a breakdown of the Kieker monitoring overhead into three portions and the results of extensive micro-benchmarks on various multi-core processor configurations with application-level monitoring tools on the example of the Kieker framework for application monitoring and dynamic software analysis [9].

A major result is a quantification of the individual portions of monitoring overhead, the identification of the main sources of overhead, and the proof that for asynchronous monitoring writers with the Kieker framework, the availability of (idle) processing units (processors, cores, hyper-threads) significantly reduces the (already very low) overhead of monitoring software applications. Thus, we show that multi-core processors may effectively be exploited to reduce the runtime overhead of monitoring software systems with Kieker. The micro-benchmarks available with our releases of Kieker can be applied to other monitoring frameworks and on other hardware platforms.

The remainder of this paper is organized as follows. Section 2 provides a brief overview of the Kieker framework. Section 3 introduces a partition of monitoring overhead into three portions. Section 4 presents the results of three different micro-benchmarks with the Kieker framework to quantify these portions of overhead in different scenarios. Section 5 discusses related work. Finally, Section 6 draws our conclusions and indicates areas for future research.

[3] `http://kieker-monitoring.net`

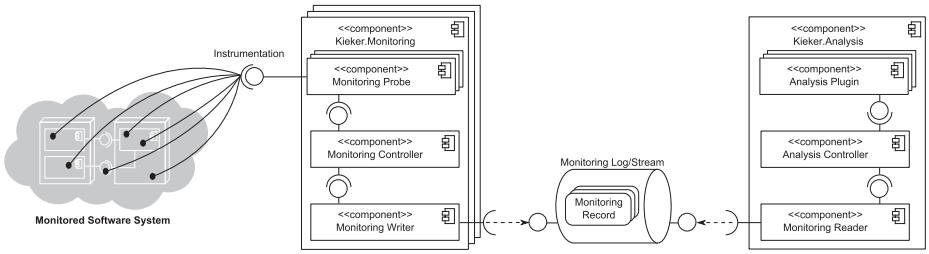



**Fig. 1.** Top-level view on the Kieker framework architecture

## 2 Overview on the Kieker framework

The Kieker framework [8, 9] is an extensible framework for monitoring and analyzing the runtime behavior of concurrent or distributed software systems. It provides components for software instrumentation, collection of information, logging of collected data, and analysis/visualization of monitoring data. Each Kieker component is extensible and replaceable to support specific project contexts. A top-level view on the Kieker framework architecture and its components is presented in Figure 1.

In this paper, we focus on the Kieker.Monitoring component to monitor software systems. This configuration allows for the insertion of Monitoring Probes into the Monitored Software System, e.g., instrumenting methods with probe code. With any execution of the monitored (instrumented) methods, these probes collect data and store it in Monitoring Records. The Monitoring Controller coordinates the activation and deactivation of Monitoring Probes and connects the probes with the single Monitoring Writer. This writer receives the records and forwards them to the Monitoring Log/Stream, e.g., a file system, a database, or a remote message queue connected to a Monitoring Reader. A more detailed description of how method executions are monitored is presented in Section 3.

The Monitoring Log/Stream acts as an interface between the Kieker.Monitoring and the Kieker.Analysis component, facilitating both online and offline analyses.

The Kieker.Analysis component consists of an Analysis Controller component that instructs the Monitoring Reader to retrieve Monitoring Records from the Monitoring Log/Stream enabling a series of Analysis Plugins to analyze and visualize the recorded data. Some Analysis Plugins available with the Kieker framework support the reconstruction of traces, the automated generation of UML sequence diagrams, dependency graphs, and call graphs. Refer to van Hoorn et al. [8, 9] for more information on analysis and visualization with Kieker.

Note that the execution of all the components of Kieker.Monitoring up to the point of storing/transferring the Monitoring Record into the Monitoring Log/Stream is in the same execution context as the Monitored Software System, i.e., its execution time and access to other resources have to be shared with the Monitored Software System. Thus, a low overhead is essential. In the following section, we take a close look at the portions that contribute to this overhead.

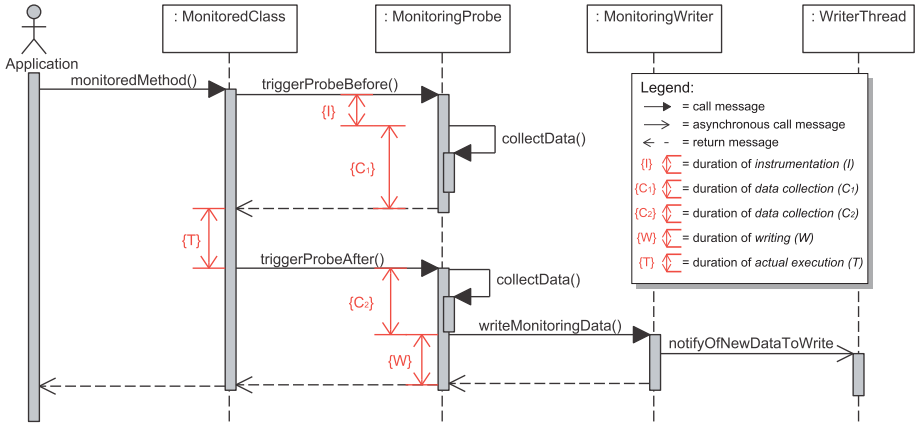



**Fig. 2.** UML sequence diagram for method monitoring with the Kieker framework

## 3 Portions of the Monitoring Overhead

A monitored software system has to share some of its resources (e.g., CPU time or memory) with the monitoring framework. In Figure 2, we present a UML sequence diagram representation of the control flow for monitoring a method execution. With a typical monitoring framework, such as Kieker, there are three possible causes of overhead while monitoring an application:

*I* Before the code of the monitoredMethod() in the MonitoredClass is executed, the triggerProbeBefore() part of the MonitoringProbe is executed. Within the probe, it is determined whether monitoring is activated or deactivated for the monitoredMethod(). If monitoring is deactivated, no further probe code will be executed and the control flow immediately returns to the monitoredMethod().

*C* The probe will collect some initial data during $C_1$ (in main memory), such as the current time and the operation signature before returning the control flow to the monitoredMethod(). When the execution of the actual code of the monitoredMethod() is finished with activated monitoring, the triggerProbeAfter() part of the MonitoringProbe is executed. Again, some additional data is collected during $C_2$ (in main memory), such as the response time or the return values of the monitored method. ($C = C_1 + C_2$)

*W* Finally, writeMonitoringData() forwards the collected data to the MonitoringWriter. The MonitoringWriter either stores the collected data in an internal buffer, that is processed asynchronously by a WriterThread into a Monitoring Log/Stream, or it synchronously writes the collected data by itself into the Monitoring Log/Stream. Here, only asynchronous writing is illustrated.

To sum up, in addition to the normal execution time of the monitoredMethod() $T$, there are three possible portions of overhead: (1) the instrumentation of the method and the check for activation of the probe ($I$), (2) the collection of data ($C$), and (3) the writing of collected data ($W$).

## 4 Micro-Benchmarks with Kieker

In order to determine these three portions of monitoring overhead, we perform a series of micro-benchmarks designed to determine the overhead of each individual portion. These experiments are performed with the Kieker framework, but any typical application-level monitoring framework should produce similar results. A detailed description of the used benchmarks will be published later.

First, we document the configuration of the experiment (Section 4.1). Then, we describe benchmarks to measure the influence of available cores (Section 4.2), to determine the linear scalability of the Kieker framework (Section 4.3), and to compare the influence of different multi-core platforms (Section 4.4).

### 4.1 Micro-Benchmark Configuration

The micro-benchmark used in our experiments is designed to measure the three individual portions of monitoring overhead. In order to allow an easy repeatability, all necessary parts of the benchmark are included in releases of Kieker.

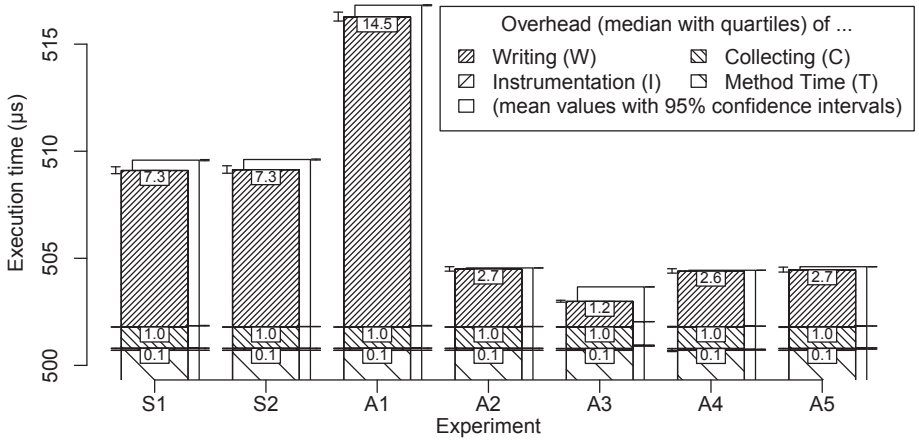
Each experiment consists of four independent runs. Each individual portion of the execution time is measured by one run (see $T$, $I$, $C$, and $W$ in Figure 2). In the first run, only the execution time of the monitoredMethod() is determined ($T$). In the second run, the monitoredMethod() is instrumented with a Monitoring Probe, that is deactivated for the monitoredMethod(). Thus, the duration $T + I$ is measured. The third run adds the data collection with an activated Monitoring Probe without writing any collected data ($T + I + C$). The fourth run finally represents the measurement of full monitoring with the addition of an active Monitoring Writer and possibly an active Writer Thread ($T + I + C + W$). This way, we can incrementally measure the different portions of monitoring overhead.

We utilize a typical enterprise server machine for our experiments, in this case a X6270 Blade Server with two Intel Xeon 2.53 GHz E5540 Quadcore processors and 24 GB RAM running Solaris 10 and an Oracle Java 64-bit Server VM in version 1.6.0_26 with 1 GB of heap space. We use Kieker release 1.4 as the Monitoring component. AspectJ release 1.6.12 with load-time weaving is used to insert the particular Monitoring Probes into the Java bytecode. As mentioned before, the Kieker framework can easily be replaced by another monitoring framework to compare our benchmark results with results of similar frameworks.

We repeat the experiments on ten identically configured JVM instances, calling the monitoredMethod() 2,000,000 times on each run with an execution time of 500 µs per method call. We discard the first 1,000,000 measured executions as the warm-up period and use the second 1,000,000 steady state executions to determine our results.

In this configuration, each experiment consists of four independent runs and each run takes a total time of 20 minutes. Each run with an active Monitoring Writer produces at least 362 MB of Kieker monitoring log files.

We perform our benchmarks under controlled conditions in our software performance engineering lab that is exclusively used for the experiments. Aside from the experiment, the server machine is held idle and is not utilized.




| Exp. | Writer | Cores | Notes |
|---|---|---|---|
| S1 | SyncFS | 1 | single physical core |
| S2 | SyncFS | 2 | two logical cores on the same physical core |
| A1 | AsyncFS | 1 | single physical core |
| A2 | AsyncFS | 2 | two logical cores on the same physical core |
| A3 | AsyncFS | 2 | two physical cores on the same processor |
| A4 | AsyncFS | 2 | two physical cores on different processors |
| A5 | AsyncFS | 16 | whole system is available |

**Fig. 3.** Single-Threaded Monitoring Overhead

### 4.2 The Influence of Available Cores on the Monitoring Overhead

The focus of this series of experiments is to quantify the three portions of monitoring overhead and to measure the influence of different assignments of multiple cores or processors to the application (and to the Monitoring component) on the monitoring overhead. In order to achieve this goal, we are using operating system commands to assign only a subset of the available cores to the monitored application and to the monitoring framework.[4] Our X6270 Blade Server contains two processors, each processor consists of four cores, and each core is split into two logical cores via hyper-threading. The assignment of cores is documented in Figure 3.

The configuration of all experiments in this section is as specified in Section 4.1. The Monitoring Writer that is used by the Monitoring component during the measurement of the portion $W$ of the overhead is either the Kieker asynchronous file system writer (AsyncFS) or the Kieker synchronous file system writer (SyncFS).

The results of the experiments are presented in Figure 3 and described below.

[4] On our Solaris 10 server we use the psrset command. Similar commands are available on other operating systems.

S1 We start our series of experiments with a synchronous file system MonitoringWriter, thus disabling the internal buffer and the asynchronous WriterThread, yielding a single-threaded benchmark system. First, we assign a single physical core to the application and disable its second logical core, thus simulating a single core system. The main portion of overhead in S1 is generated by the writer $W$ (7.3 µs), that has to share its execution time with the monitored application. The overhead of the instrumentation $I$ is negligible (0.1 µs), the overhead of the data collection $C$ is low (1.0 µs).

S2 In Experiment S2 we activate two logical cores (hyper-threading) in a single physical core and repeat the experiment with the synchronous writer. There is no significant difference between one or two assigned cores. For this reason we omit further synchronous experiments. Only with asynchronous writing, multiple processing units may reduce the monitoring overhead.

A1 We continue the rest of our series of experiments with the asynchronous file system Monitoring Writer. Similar to experiment S1, we assign a single physical core to the application and disable its second logical core. The portion $W$ of the overhead caused by the writer (14.5 µs) is almost doubled compared to the synchronous writer. This can be explained by the writer thread sharing its execution time with the monitored application. Compared to the experiment S1, context switches and synchronization between the two active threads degrade the performance of the system.

A2 Next, we activate two logical cores in a single physical core. The additional core has no measurable influence on the overhead of instrumentation $I$ (0.1 µs) and collecting data $C$ (1.0 µs). Due to the additional available core, which is exclusively used by the writer thread, the overhead of writing the data $W$ (2.7 µs) is significantly reduced. Even though both logical cores have to share the resources of a single physical core, the second logical core proofed to be an enormous improvement. The overhead could be reduced by 55% of the overhead of the synchronous writer (S1) and by 76% of the overhead of the single core system with the asynchronous writer (A1).

A3 In this experiment we assign two different physical cores on the same processor to the benchmark system. This setup provides the best results of the series of experiments with again greatly improved writer performance $W$ (1.2 µs). The improvement can be explained by no longer sharing the processing resources of a single physical core by two logical cores (via hyper-threading). Thus, the overhead of monitoring could be reduced by 73% of the overhead of the synchronous writer (S1) and by 85% of the overhead of the single core system with the asynchronous writer (A1).

A4 Next, we assign two physical cores of two different processors on the motherboard. The increased synchronization overhead between two different processors causes results similar to A2.

A5 Finally, we activate all physical and logical cores in the system. Since the monitored software system uses a single thread and the monitoring framework uses an additional writer thread, no additional benefit of more than two available cores is measurable: the two threads (one for the application and one for monitoring) cannot exploit more than two cores.

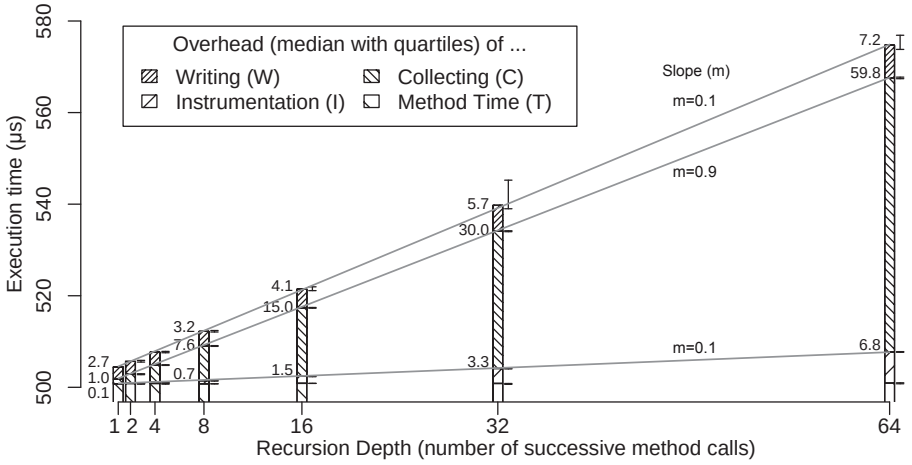



**Fig. 4.** Linear Increase of Monitoring Overhead

### 4.3 The Scalability of Monitoring Overhead

Only a linear increase of monitoring overhead is acceptable for good scalability. In order to determine whether the increase of the amount of monitoring overhead with each additional monitored method call is linear, we perform a series of experiments with increasing recursion depths. Thus, in each experiment run, each call of the monitoredMethod() results in additional recursive (monitored) calls of this method, enabling us to measure the overhead of monitoring multiple successive method calls. The benchmarks are performed with the Kieker asynchronous file system writer (AsyncFS) in a configuration similar to the one described in experiment A5 in the previous section. Apart from the increasing recursion depths, the configuration of the experiment is as described previously.

The results of this experiment are presented in Figure 4 and described below.

The measured overhead of instrumentation $I$ increases with a constant value of approximately 0.1 µs per call. The overhead of collecting data $C$ increases with a constant value of approximately 0.9 µs per call. The overhead of writing $W$ consists of two parts: a constant overhead of approximately 2.5 µs during the period of 500 µs and an increasing value of approximately 0.1 µs per additional call.

Our experiments include recursion depths up to 64 method calls per 500 µs. With higher values of the recursion depth, the monitoring system records method calls faster than it is able to store monitoring records in the file system.

In each experiment run, the Monitoring Writer has to process 362 MB of monitoring log data per step of recursion depth. In the case of a recursion depth of 64, 23 GB Kieker monitoring log data were processed and written to disk within the 20 minutes execution time (at 19.3 MB/s).

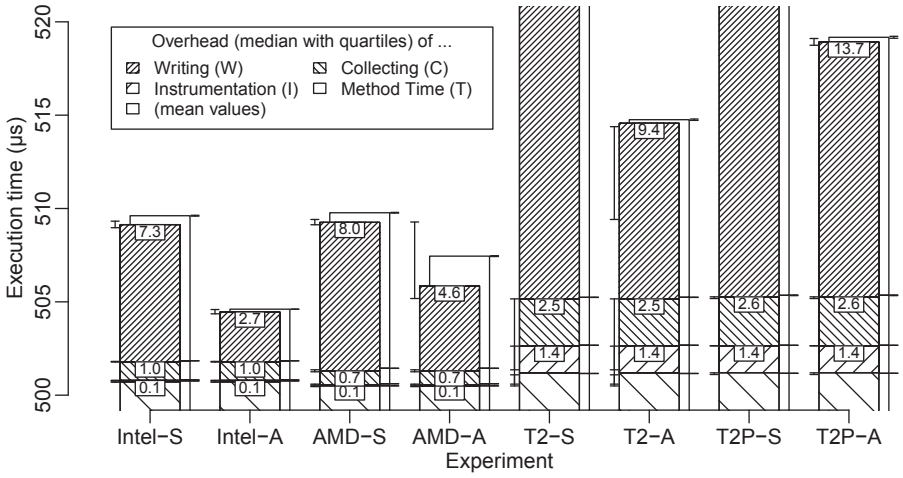



**Fig. 5.** A comparison of different multi-core architectures

### 4.4 The Influence of Different Multi-core Architectures

In this final experiment, we compare the results of our benchmarks on several different multi-core architectures with each other. The goal of this experiment is a generalization of our results in the previous sections.

Besides the X6270 Blade server with two Intel Xeon E5540 (Intel), we use a X6240 Blade with two AMD Opteron 2384 2.7 GHz processors (AMD), a T6330 Blade with two Sun UltraSparc 1.4 GHz T2 processors (T2), and a T6340 Blade with two Sun UltraSparc 1.4 GHz T2+ processors (T2P).

On each server, we compare two different benchmark runs. The first run is performed with a synchronous writer (S) and is similar to S2 in Section 4.2. The second run is performed with an asynchronous writer (A) and corresponds to experiment A5. The results of these experiments are presented in Figure 5 and are described below.

Compared to our Intel experiments, the AMD architecture provides slightly improved performance in the collecting portion $C$ with a similar performance of the synchronous writer while the performance gain of the asynchronous writer is slightly worse. The Sun UltraSparc architectures provide lots of slower logical cores (64 on the T2, 128 on the T2+) compared to the Intel or AMD architectures. The result is a significant increase of the monitoring overhead. Yet, an asynchronous writer provides an even greater benefit compared to a synchronous writer. In the case of the T2 processor, the overhead of writing $W$ is reduced from 69.4 µs to 9.4 µs. In the case of the T2+ processor, the overhead is reduced from 64.9 µs to 13.7 µs.

In all experiments, the writing portion $W$ of the overhead can be greatly reduced with the usage of an asynchronous monitoring writer and available cores.

## 5 Related Work

In this section, we compare related approaches to exploiting multiple cores to reduce the runtime overhead for monitoring software systems.

One approach is the usage of additional specialized hardware mechanisms to reduce the overhead of profiling, monitoring, or analysis of software systems [2, 16]. Contrary to this, our goal with the Kieker framework is to build a software-based, portable framework for application performance monitoring that imposes only a very small overhead to the monitored application, particularly via exploiting multi-core processors. Furthermore, additional cores reserved or used for monitoring are comparable to dedicated profiling hardware.

Several authors [15, 19, 23] propose shadow processes, i.e., instrumented clones of actual parts of the monitored system, running on spare cores, thus minimizing influence on the execution of the main system. The goal of application-level monitoring is recording the actual events in the system, including any side effects, while keeping the overhead and the influence of monitoring minimal. Furthermore, cloning is usually not viable for interactive enterprise systems.

Another possibility is the separation of monitoring and analysis [2, 7, 24] in order to either execute the analysis concurrently on another core or to delegate it to specialized hardware. This is the usual approach in application-level monitoring and also employed by Kieker.

Most of these related works apply to profilers and fine-grained monitoring solutions. In the field of application-level performance monitoring, most performance evaluations are far less sophisticated. Despite the fact that reports on monitoring frameworks often include an overhead evaluation, a detailed description of the experimental design and a detailed analysis of the results, including confidence intervals, is often missing (see for instance [1, 11, 13, 14, 18, 22]).

## 6 Conclusion & Future Work

Multi-core processors may effectively be exploited to reduce the runtime overhead of monitoring software systems on the application level. To evaluate whether monitoring frameworks are really able to use available processing units (processors, cores, hyper-threads) for reducing the overhead of collecting monitoring data, we proposed a splitting of monitoring overhead in three portions and designed a micro-benchmark with a series of experiments to quantify these various portions of monitoring overhead under controlled and repeatable conditions.

Extensive micro-benchmarks were performed with the Kieker framework and the results are presented in this paper. For asynchronous monitoring writers, the availability of (idle) processing units may significantly reduce the (already very low) overhead of monitoring software applications with the Kieker framework.

The benchmarks may be applied to other monitoring frameworks and on other hardware platforms. So far, we performed our experiments on multiple hardware platforms with a specific operating system and a specific Java virtual machine. Other combinations of hardware, operating system and virtual machines may yield other results. Thus, we intend to validate the presented results

on other platforms that are available in our software performance engineering lab. Further experiments can be performed to determine the exact assignment of processing units to active threads within the monitoring framework. Thus, a more detailed analysis of possible contention as a new source of overhead is possible. Additionally, the benchmarks might be adapted to other monitoring concepts, such as event based monitoring, that support a wider range of possible applications. Since, our experiment code and data are available as open source, interested researchers may repeat or extend our experiments for comparison on other hardware platforms or with other application-level monitoring frameworks.

According to Sim et al. [20], our benchmarks are so-called proto-benchmarks since benchmarks require a community that defines and uses its benchmarks. Such a community does not, as yet, exist. However, our intention is that interested researchers may repeat or extend our experiments for comparison. The Software Engineering Group of the University of Kiel is a member of the SPEC Research Group (`http://research.spec.org/`). We share Kieker and our monitoring benchmarks with this group and with other research groups that use Kieker [17, 25] as a start for community building.

## Bibliography


[1] Chawla, A., Orso, A.: A generic instrumentation framework for collecting dynamic information. ACM SIGSOFT Softw. Eng. Notes 29, 1–4 (2004)

[2] Chen, S., Gibbons, P.B., Kozuch, M., Mowry, T.C.: Log-based architectures: Using multicore to help software behave correctly. ACM SIGOPS Operating Systems Review 45, 84–91 (2011)

[3] Ehlers, J., Hasselbring, W.: A self-adaptive monitoring framework for component-based software systems. In: Proc. of the 5th Europ. Conf. on Software Architecture (ECSA 2011). pp. 278–286. Lecture Notes in Computer Science, Springer (2011)

[4] Ehlers, J., van Hoorn, A., Waller, J., Hasselbring, W.: Self-adaptive software system monitoring for performance anomaly localization. In: Proc. of the 8th IEEE/ACM Int. Conf. on Autonomic Computing (ICAC 2011). pp. 197–200. ACM (2011)

[5] Gao, J., Zhu, E., Shim, S., Chang, L.: Monitoring software components and component-based software. In: The 24th Annual International Computer Software and Applications Conference (COMPSAC 2000). pp. 403–412. IEEE Computer Society (2000)

[6] Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous Java performance evaluation. ACM SIGPLAN Notices 42, 57–76 (2007)

[7] Ha, J., Arnold, M., Blackburn, S., McKinley, K.: A concurrent dynamic analysis framework for multicore hardware. ACM SIGP. 44, 155–174 (2009)

[8] van Hoorn, A., Rohr, M., Hasselbring, W., Waller, J., Ehlers, J., Frey, S., Kieselhorst, D.: Continuous monitoring of software services: Design and application of the Kieker framework. Tech. Rep. TR-0921, Department of Computer Science, University of Kiel, Germany (2009)

[9] van Hoorn, A., Waller, J., Hasselbring, W.: Kieker: A framework for application performance monitoring and dynamic software analysis. In: Proc. of 3rd ACM/SPEC Int. Conf. on Performance Eng. (ICPE 2012). ACM (2012)
[10] Huang, X., Seyster, J., Callanan, S., Dixit, K., Grosu, R., Smolka, S., Stoller, S., Zadok, E.: Software monitoring with controllable overhead. Int. Journal on Software Tools for Technology Transfer (STTT) pp. 1–21 (2010)
[11] Maebe, J., Buytaert, D., Eeckhout, L., De Bosschere, K.: Javana: A system for building customized Java program analysis tools. ACM SIGPLAN Notices 41, 153–168 (2006)
[12] Marowka, A.: Parallel computing on any desktop. Communications of the ACM 50, 74–78 (2007)
[13] Moon, S., Chang, B.M.: A thread monitoring system for multithreaded Java programs. ACM SIGPLAN Notices 41, 21–29 (2006)
[14] Mos, A.: A Framework for Adaptive Monitoring and Performance Management of Component-Based Enterprise Applications. Ph.D. thesis, Dublin City University (2004)
[15] Moseley, T., Shye, A., Reddi, V.J., Grunwald, D., Peri, R.: Shadow profiling: Hiding instrumentation costs with parallelism. In: Int. Symp. on Code Generation and Optimization (CGO '07). pp. 198–208 (2007)
[16] Nair, A., Shankar, K., Lysecky, R.: Efficient hardware-based nonintrusive dynamic application profiling. ACM Transactions on Embedded Computing Systems 10, 32:1–32:22 (2011)
[17] Okanović, D., van Hoorn, A., Konjović, Z., Vidaković, M.: Towards adaptive monitoring of Java EE applications. In: Proc. of the 5th Int. Conf. on Information Technology (ICIT 2011). IEEE Computer Society (2011)
[18] Parsons, T., Mos, A., Murphy, J.: Non-intrusive end-to-end runtime path tracing for J2EE systems. IEEE Software 153, 149–161 (2006)
[19] Patil, H., Fischer, C.N.: Efficient run-time monitoring using shadow processing. In: AADEBUG. pp. 119–132 (1995)
[20] Sim, S.E., Easterbrook, S., Holt, R.C.: Using benchmarking to advance research: A challenge to software engineering. In: Proc. of the 25th Int. Conf. on Software Engineering (ICSE 2003). pp. 74–83. IEEE (2003)
[21] Tichy, W.: Should computer scientists experiment more? IEEE Computer 31, 32–40 (1998)
[22] Vlachos, E., Goodstein, M.L., Kozuch, M., Chen, S., Falsafi, B., Gibbons, P., Mowry, T.: ParaLog: Enabling and accelerating online parallel monitoring of multithreaded applications. ACM SIGPLAN Notices 45, 271–284 (2010)
[23] Wallace, S., Hazelwood, K.: SuperPin: Parallelizing dynamic instrumentation for real-time performance. In: Int. Symp. on Code Generation and Optimization (CGO '07). pp. 209–220 (2007)
[24] Zhao, Q., Cutcutache, I., Wong, W.F.: Pipa: Pipelined profiling and analysis on multi-core systems. In: Proc. of the 6th annual IEEE/ACM int. symp. on Code generation and optimization (CGO '08). pp. 185–194. ACM (2008)
[25] Zheng, Q., Ou, Z., Liu, L., Liu, T.: A novel method on software structure evaluation. In: Proc. of the 2nd IEEE Int. Conf. on Software Engineering and Service (ICSESS 2011). IEEE Computer Society (2011)