FerbJmon Tools - Visualizing Thread Access on Java Objects using Lightweight Runtime Monitoring

Marvin Ferber(✉)

Institute of Computer Science, Technical University Bergakademie Freiberg, Freiberg, Germany
marvin.ferber@informatik.tu-freiberg.de

Abstract. Learning and teaching parallel programming in Java can sometimes be tedious, because the correct behavior of a parallel program can hardly be debugged. A runtime monitoring can help to gather information on the behavior of a parallel program. As Java Bytecode is executed inside a JVM, a runtime monitoring can be applied with almost no changes to the source code. In this paper, FerbJmon tools for monitoring Java objects at runtime are proposed. FerbJmon tools also include a tool for call graph creation and a tool for visualizing chronological accesses of threads on Java objects in a timeline diagram. A lightweight monitoring is necessary to minimize the influence of the monitoring on the program behavior itself. FerbJmon tools address this issue by selecting only one class for the monitoring at the same time and by a fast logging implementation. A producer/consumer program and a program for cooperative task execution are used to demonstrate the applicability and the performance of the logging. FerbJmon tools can be used to understand and optimize thread synchronization in Java programs.

Keywords: Java · Runtime monitoring · Parallel computing · Distributed computing · Education

1 Introduction

The programming language Java is already used in a variety of educational activities such as lectures or practical sessions [7]. Reasons for its popularity include the consistent object-orientation in Java source code, the platform independence, and its large system library, which includes many classes, e. g., for parallel processing. Java has already built-in threading and synchronization mechanisms such as mutual exclusion and signals. Even higher level parallel constructs such as thread pools and complex thread-safe data structures like hash tables are built-in. This makes parallelism easy to use in Java programs.

However, the proper usage of mechanisms for thread synchronization can be difficult when starting to learn parallel programming [4,9]. The chronological order of activities in multiple threads of execution is hard to debug. As a consequence, race conditions may occur or threads may block due to missing signals. A source code analysis may not always detect the causes for such unintended behavior. Runtime monitoring can be used to investigate the behavior of program *in action*. As Java Bytecode is executed in a Java Virtual Machine (JVM), a monitoring can be applied to a JVM at runtime with only minor changes to the source code. This makes it easy to apply for, e. g., multi-threaded programs. Runtime monitoring requires the execution of additional code. It needs to be applied carefully to minimize the influence on the actual program behavior.

In this paper, FerbJmon tools for runtime monitoring of Java programs are proposed. FerbJmon tools can be used to capture runtime information of Java programs in order to create call graphs and timeline diagrams that visualize the chronological accesses of threads on Java objects. FerbJmon tools use Bytecode instrumentation and focus on class instances to monitor access on them. Only parts of the program code are monitored, which allows a lightweight monitoring.

FerbJmon tools can easily be applied in teaching, because only few arguments of the invocation of the Java interpreter and almost no changes to the source code of the program under monitoring are necessary to automatically monitor and visualize the program behavior. Furthermore, the created timeline diagrams provide a different view on execution traces than the widely used UML sequence diagrams, which is better suited to handle fine-grained parallelism in Java. FerbJmon tools are available for Windows and Linux[1].

The article is structured as follows. First, related work on the analysis of multi-threaded programs is discussed. In Sect. 3 the FerbJmon tools, their implementation, and their usage are presented. Example programs and the application of FerbJmon tools in teaching parallel programming are discussed in Sect. 4. Benchmarks investigating the monitoring performance are presented in Sect. 5. Section 6 concludes the paper.

2 Related Work

First steps for successful parallel programming in Java include the usage of programming patterns for parallelism in general [10] and the proper usage of mechanisms and data structures for parallel programming in Java [6]. However, the necessity for tool support for parallel program validation and verification has already been discussed in current literature on parallel programming education [4]. Questions are raised such as “How can I introduce parallelism into my algorithm?” or “How can I measure the benefits of parallelism?” [11]. Different ways exist to capture and analyze the behavior of parallel programs.

A static source code analysis can be applied to detect possible deadlock situations prior to execution. A tool that detects deadlocks in Java libraries using a lock-order graph is proposed in [12]. However, a static analysis may not detect all dependencies in a Java code because of reflection. Such dependencies appear

[1] https://github.com/marvinferber/ferbjmon.

at runtime. A runtime analysis can additionally detect the number of concurring threads and the occupancy of data structures. Therefore, tools have been developed to monitor the behavior of concurrent Java programs using program traces. Many early attempts to generate program traces such as JACOT [8] were based on the Java Virtual Machine Debug Interface (JVMDI), which was removed in Java 6. Current attempts such as Javashot [1] and JThreadSpy [9] are therefore based on Bytecode instrumentation to gather runtime information.

Javashot is a tool that generates call graphs from runtime traces of Java programs. JThreadSpy can generate augmented UML sequence diagrams from runtime traces highlighting thread interactions and concurrent access to critical sections in concurrent Java programs. Because UML sequence diagrams are not well prepared for parallel programs, actions of different threads are marked in different color for visualization. JThreadSpy provides a custom visualization software for this purpose. Performance benchmarks on the overhead for logging are not given. FerbJmon tools create a different visualization based on timeline diagrams that visualize the accesses of different threads on each object instance separately, which is better suited for fine-grained access monitoring.

Other tools for runtime monitoring of CPU usage or memory occupation such as the Java Monitoring and Management Console (JConsole) and VisualVM [3] are already shipped with an installation of the Java Development Kit (JDK). All tools named provide different information that can be combined in order to get a comprehensive knowledge on a parallel Java program.

3 Java Runtime Monitoring Using FerbJmon Tools

FerbJmon tools generate traces from program executions using a Bytecode transformation that is executed at runtime in order to inject monitoring code into user-defined classes. FerbJmon tools are applied in three steps:

1. Preparation → mark class of interest and transform Bytecode,
2. Monitoring → run program with transformed Bytecode and generate logs,
3. Visualization → process logs and generate diagrams.

All steps are performed automatically, except the annotation of the class of interest. Only instances of one specific class are monitored at the same time in order to only execute a minimal set of additional code. The class under monitoring must not have public fields because only method invocations can be monitored. Access to public fields is not captured. Getter and setter methods need to be implemented in order to realize a complete monitoring. The monitoring is applied to a class by annotating it using the FerbJmon-specific `@Monitored` annotation, see Fig. 3 for an illustration. FerbJmon tools require the monitored parts to be encapsulated into one Java class. This sometimes requires to adapt the program code in order to enable the monitoring for the intended behavior. Many data structures such as thread pools or work queues are well encapsulated and can be monitored easily. In contrast, a matrix multiplication may be difficult to monitor unless each matrix update can be encapsulated into a corresponding method.

After a class has been annotated, the monitoring can easily be applied to any Java program by substituting the `java` command-line tool by a FerbJmon-specific wrapper script.

- `java_threadorder` monitors the chronological access of threads on a Java object with the intention to create a timeline diagram.
- `java_callgraph` captures runtime dependencies with the intention to create a call graph.

Both wrapper scripts take exactly the same command-line parameters like the `java` command-line tool. The wrapper scripts additionally apply the Bytecode transformations necessary for monitoring, implement the generation of trace logs while the program is executed, and perform the generation of a default visualization after the execution has finished. Furthermore, the trace logs are also saved to generate user-specific visualizations. FerbJmon tools take optional parameters to influence the logging behavior, which are specified as environmental variables of a Java program. The tools are described in detail in the following subsections.

3.1 Bytecode Instrumentation

Bytecode instrumentation is a mechanism to modify Java Bytecode at runtime. It is applied while a specific Java class is loaded by the classloader. The instrumentation is transparent to the actual program code and can be used to inject additional output messages into the Bytecode. It is helpful to use a Bytecode manipulation library for Bytecode processing. FerbJmon tools use the Javassist Bytecode manipulation library [2]. A filter transformation is used to modify the relevant parts inside a class. Thereby, the syntax tree of the class is traversed and transformations are implemented as callback functions on relevant nodes such as method calls. FerbJmon tools use this technique to inject additional code each time a method is invoked and each time a method returns.

Java has the ability to execute Bytecode instrumentation through a Java agent. The Java agent code is executed before the actual user program starts. Therefore, the method `premain()` needs to be implemented. The agents library can be passed as command-line argument to the `java` command-line tool (e. g., `java -javaagent:PATH/agent.jar -cp CLASSPATH MAINCLASS`). Figure 1 shows the point of execution in a Java program run. The Java agent is executed after the JVM has been initialized and before the `main()` method of the actual program is invoked. Its main purpose is to substitute the default classloader by a custom classloader (modifying loader) that performs the Bytecode modification. Furthermore, the Java agent also registers a global shutdown hook, which is necessary for log file creation after the program has finished.

At runtime, Bytecode instrumentation can only be performed for user classes. System classes are locked and prevented from being substituted once they are loaded. However, it is possible to instrument system classes offline and to load

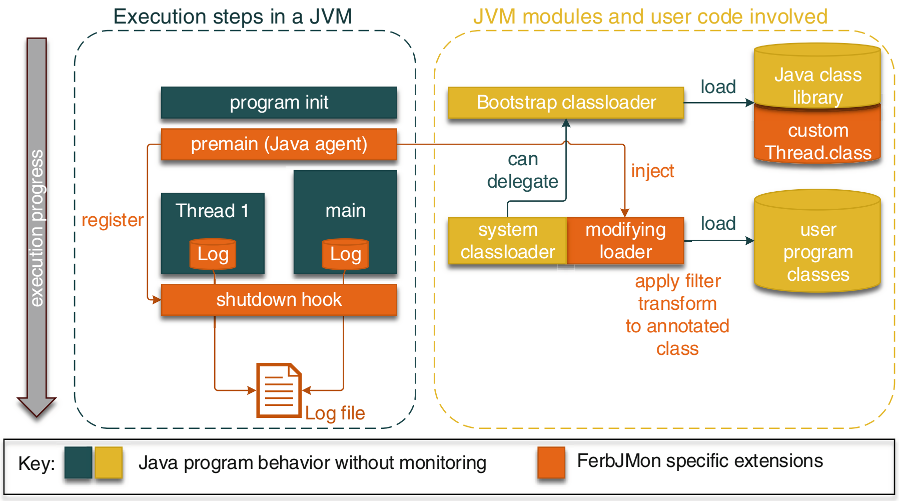


Fig. 1. Overview of the implementation of the runtime monitoring using a JVM. User classes are instrumented based on annotation. Logging is performed by each thread itself using a custom implementation of `java.lang.Thread`. A shutdown hook pulls all log data and writes a common log file (Color figure online).

them during JVM startup. System class loading is performed by the bootstrap classloader even before loading the Java agent. A specific library can be passed to the `java` command-line tool in order to be loaded by the bootstrap classloader, see Fig. 1 for an illustration. So, it is possible to even substitute system classes like `java.lang.Thread`. An exemplary command line would be `java -Xbootclasspath/p:PATH/sys.jar -cp CLASSPATH MAINCLASS`.

FERBJMON tools use a modified version of the system class `java.lang.Thread`, because a custom logging is directly integrated into each thread. So, lock free logging is performed by each thread. This ensures an equally distributed influence of the logging among all threads that access Java objects under monitoring. There are no additional dependencies between threads or threads and common data structures. At the end of a program run, a shutdown hook is executed that actively pulls the logs from each thread and merges them into a single log file.

3.2 FERBJMON Call Graph

A call graph that is captured at runtime can visualize dependencies between those classes that were instantiated in a specific program run. This set of instantiated classes and method invocations is often smaller than the set of all classes of the program, because not all classes may be used in a specific program run. A runtime call graph can be less complex than a call graph captured from a static code analysis. Prior to the actual thread monitoring a call graph analysis can be useful to identify classes and threads that access the object under monitoring.

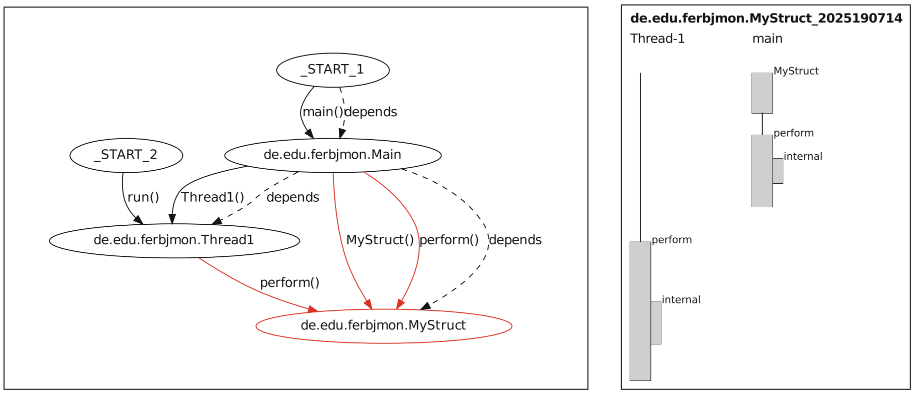


Fig. 2. *Left:* Call graph visualization showing dependencies captured by the runtime monitoring. The monitored class `MyStruct` is marked red. *Right:* Timeline diagram showing the chronological order of accesses from different threads (`Thread-1` and `main`) on a Java object instance of the class `MyStruct`.

The tool `java_callgraph` directly creates a text file as output that contains all captured dependencies in the Graphviz dot language [5]. Graph layout is done by the `dot` command-line tools. A exemplary graph is shown in Fig. 2 (left). Classes are shown as ellipses containing their fully qualified class name. Call dependencies are visualized as solid directed lines that show the direction of method invocation. Dependencies that arise from non-method invocations, e. g., through parameters in method calls, are visualized as dashed directed lines. The class under monitoring and all ingoing call dependencies of method invocations are marked red. Each thread start (including the main thread) is also visualized by ellipses containing the word `_START_`. Additionally, there is a dependency from this start ellipse to the corresponding thread class via its startup method. This can be seen in Fig. 2 (left) at the methods `main()` and `run()`.

The call graph visualization does not contain invocations of methods within a class itself. This would lead to looping dependencies that have the same origin and target. These are not relevant for inter-class dependencies. Invocations on system classes are not visualized, because they can hardly be monitored. Furthermore, the monitoring of system classes can lead to large dependency graphs, because even small programs can involve many system classes in the background.

3.3 FerbJmon Timeline Diagram of Thread Accesses

The chronological order of accesses to a Java class instance can be relevant to validate or optimize the proper usage and implementation of synchronization mechanisms in a parallel Java program. The monitoring is performed by each thread itself, because each thread is extended by a logger implementation for this purpose. The log messages are generated by the instrumented Bytecode of the class under monitoring. Each log entry contains the current system time by means of `System.nanoTime()`, the hash value of the current class object, the method that is considered, the thread that performed an action, and the action itself (enter method or exit method). Using this information, a timeline diagram of thread accesses can be created for each object instance.

As a requirement, logging should not influence the program behavior significantly. If many log entries appear in a short period of time, writing these entries to a file directly can lead to delays in thread execution. This can influence the program behavior. Logging into main memory is limited by the capacity of the RAM but normally much faster. Java provides the class `java.lang.StringBuilder` to collect an unknown amount of strings in main memory. However, writing strings to a preallocated array of characters may provide the best performance. As a drawback, preallocation further limits the available log space, because the size of the array needs to be specified in advance. To suit different logging demands the FerbJmon tool `java_threadorder` implements the following logger:

- `File Logger` a `java.io.BufferedWriter` that writes log entries directly into a file (this is the default logger),
- `StringBuilder Logger` uses `java.lang.StringBuilder` to capture all logs in main memory, and
- `MemArray Logger` creates a fixed size array of characters in main memory and captures all logs in this array.

Data from memory loggers is written to disk during the shutdown hook execution. The selected FerbJmon logger must be specified as global parameter to the JVM invocation (`-Dmyjavamon.logger=<Logger>`). The `MemArray Logger` needs an additional parameter (`-Dmyjavamon.logger.size=<size>`) that specifies the size of the array. Benchmarks on the performance of the different logging implementations in an example application are given in Sect. 5.

A timeline diagram is the default visualization of the log data captured during runtime. It is also created by `java_threadorder`, see Fig. 2 for an illustration. One diagram is created for each object instance that is identified by the fully qualified class name followed by the hash value of the object, see top of Fig. 2 (right). The timeline for each thread and the corresponding thread names are drawn below. Figure 2 (right) shows the `main` thread and `Thread-1`. First, the `main` thread instantiates an object of the class `de.edu.ferbjmon.MyStruct` and calls the constructor. Afterwards, both threads invoke the method `perform()` on this object instance. A nested method call to the public function `internal()` can also be seen. Method calls to private functions are not shown here.

4 Examples

Examples of common problems in parallel programming and their visualization using FerbJmon tools are given in the following.

4.1 Producer and Consumer

The problem of producer and consumer is a common problem in a variety of multi-threaded scenarios. A set of producers offers tasks to a set of threads that execute the work. An application of this problem can be found, e. g., in a Web server. A fixed set of dispatch threads (producer) accept client requests and enqueue them into a queue data structure. A fixed set of worker threads then executes these request in their order of appearance and the results of the requests (a Web document) are returned to the dispatcher thread. This example demonstrates how FERBJMON tools can be used to visualize the usage of a queue.

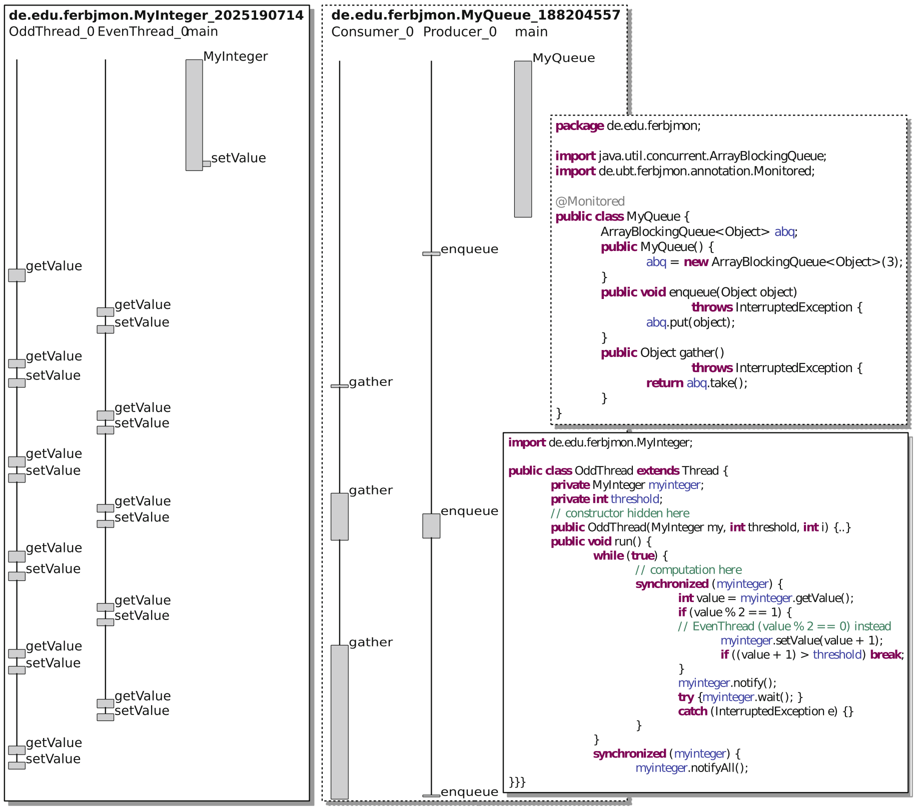
Java offers different implementations of the interface `java.util. concurrent. BlockingQueue`. This interface requires the methods `put(...)` and `take()` that block the executing thread if the queue is empty or if no space is available (queue full). This data structure can be used to synchronize a producer and consumer scenario. As an example, the data structure `MyQueue` is implemented as a wrapper class for an `java.util.concurrent.ArrayBlockingQueue`. `MyQueue` sets the queue size and realizes the methods `enqueue(...)` and `gather()` to insert and remove elements from this queue. The `Producer` and `Consumer` thread are derived from the class `java.lang.Thread`.

– The `Producer` starts to enqueue three objects immediately with a delay of 5 ms between each.
– The `Consumer` starts to gather three objects with a delay of 2 ms after an initial delay of 2 ms.

The source code of the classes `MyQueue` and the timeline diagram of a monitored program run are shown in Fig. 3. First, `MyQueue` is instantiated by the `main` thread. The `Producer` thread enqueues the first item without delay. The second `enqueue()` action is slightly delayed due to synchronization mechanisms that need to be initialized in this step because a concurring `gather()` is already waiting for an element to be inserted. The third `gather()` is largely delayed until the last element is inserted by the `Producer` thread. A large delay at consumer site may be intended, whereas a large delay on producer site may be caused by an overload. FERBJMON tools can be used to optimize the queue size or the number of `Consumer` threads.

4.2 Cooperative Task Execution

Some applications require a special communication pattern between threads. As an example, the cooperative incrementation of an integer value using two threads is presented. The integer is encapsulated in the class `MyInteger`. Two types of threads manipulate the object. The `EvenThread` is only permitted to increment the value if it is even, the `OddThread` if it is odd. As a result, a mutual incrementation by both types of threads is realized in the manner of a ping-pong game. The access pattern described is implemented using a `synchronized` access to the object by both types of threads. Additionally, the object methods `wait()` and `notify()` are used to implement the signals that hand over control between the thread instances. The source code of the classes `OddThread` and the timeline diagram of a monitored program run are shown in Fig. 3. The program ends



```
package de.edu.ferbjmon;

import java.util.concurrent.ArrayBlockingQueue;
import de.ubt.ferbjmon.annotation.Monitored;

@Monitored
public class MyQueue {
        ArrayBlockingQueue<Object> abq;
        public MyQueue() {
                abq = new ArrayBlockingQueue<Object>(3);
        }
        public void enqueue(Object object)
                        throws InterruptedException {
                abq.put(object);
        }
        public Object gather()
                        throws InterruptedException {
                return abq.take();
        }
}
```

```
import de.edu.ferbjmon.MyInteger;

public class OddThread extends Thread {
        private MyInteger myinteger;
        private int threshold;
        // constructor hidden here
        public OddThread(MyInteger my, int threshold, int i) {..}
        public void run() {
                while (true) {
                        // computation here
                        synchronized (myinteger) {
                                int value = myinteger.getValue();
                                if (value % 2 == 1) {
                                // EvenThread (value % 2 == 0) instead
                                        myinteger.setValue(value + 1);
                                        if ((value + 1) > threshold) break;
                                }
                                myinteger.notify();
                                try {myinteger.wait(); }
                                catch (InterruptedException e) {}
                        }
                }
                synchronized (myinteger) {
                        myinteger.notifyAll();
}}}
```

Fig. 3. Timeline diagrams for the monitored Java classes `MyInteger` (Cooperative Task Execution) and `MyQueue` (Producer and Consumer). The Java codes of the classes `MyQueue` and `OddThread` are also given.

after 10 increments. The timeline shows the proper handover between the two threads. By chance, the `OddThread` starts execution first, but it only determines the value and hands over control to the `EvenThread`, which is allowed to increment the value "0". This scenario of cooperative task execution can naively also be solved without an explicit handover between the threads. But using only a `synchronized` access to the object can lead to many unsuccessful incrementation attempts by both threads, which increases the execution time significantly.

5 Performance of FERBJMON Runtime Monitoring

As FERBJMON tools aim at lightweight runtime monitoring, this section presents benchmarks to better understand the capabilities of FERBJMON tools. The runtime monitoring for the construction of the call graph has no influence on the timely relevant behavior of the program, because only dependencies are visualized. Also, the post-processing step of creating the timeline diagram is not relevant for the monitoring. Thus, the runtime monitoring using `java_threadorder` is benchmarked only. For testing, two different machines are used.

- **DELL 6530 Laptop** Intel Core i7-3720QM (2.60 GHz), 8 GB RAM, SSHD Seagate ST500LX003 (500 GB), Windows 7 and Linux Mint 13 (Kernel 3.2).
- **Megware Workstation** Intel Xeon E5-1680v3 (3.20 GHz), 64 GB RAM, SSD Sandisk SD6SB1M2 (256 GB), Linux Mint 17 (Kernel 3.13).

For all tests, the Oracle Java JDK Version 7 was used. The experiments were conducted using the example program of cooperative task execution, which was presented in Sect. 4.2. Durations of program executions that are presented in the following reflect the average value of 30 test runs with the same configuration. FerbJmon tools provide different logging mechanisms which are benchmarked in this section. Because the tests were performed on Windows and Linux, time measurement was integrated into the example program itself to obtain comparable results. Therefore, a timestamp at the beginning and at the end of the `main()` function were obtained using the Java method `System.currentTimeMillis()`.

In a first test, the example program for cooperative task execution was executed using a threshold of 10 ($t = 10$) for the incrementation. The execution time measured without monitoring was below 1 ms in most cases. Including the monitoring, the execution times were still below 3 ms. The overall program execution time including Bytecode instrumentation and output creation was measured to approximately 150 ms using the command line tool `time` in Linux (50 ms for the unmodified program). Such low values can hardly be used for comparison. Therefore, the threshold for cooperative task execution was increased to 100000, which resulted in more meaningful time measurements of approximately 1 s. Although it may not be meaningful to monitor this modified program in real life, it was used to generate comparable results for benchmarking.

The different logging mechanisms were investigated in a second experiment. The results are summarized in Table 1. It can be recognized that the results for Linux and Windows differ significantly especially if the CPU frequency scaling is enabled (SpeedStep, Turbo, and extended C-States enabled). To better understand this effect and to get comparable results, all experiments have also been conducted with CPU frequency scaling disabled (SpeedStep, Turbo, and extended C-States disabled). In this test, 800022 log entries have been captured (about 76 MB log file). The experiment only covers benchmarks of the monitoring and logging. Creating a visualization from this amount of data may take a long time. However, a visualization of the example shown in Fig. 3 ($t = 10$) was created in approximately 500 ms.

The benchmarks revealed that the execution times under monitoring are approximately 3 times higher than in an unmonitored program run. The `MemArray Logger` showed the best logging performance in 5 of 6 cases. In the experiment using Linux and a fixed CPU frequency on the Dell 6530 the `File Logger` showed a slightly better performance. In all other tests the `File Logger` showed the worst performance. This observation may be caused by a side effect

Table 1. Benchmarks of logging implementations and logging overhead (in ms)

	Dell 6530 Laptop				**Megware Workst.**	
	CPU Freq. variable		CPU Freq. fixed		CPU Freq. variable	CPU Freq. fixed
	Windows	Linux	Windows	Linux	Linux	Linux
Benchmarks of different logging implementations						
No Monitor.	349	997	462	509	706	460
File Logger	1578	3089	1792	1551	1769	1252
StringBuilder	1246	2884	1601	1711	1548	1202
MemArray	1204	2767	1527	1640	1339	1162
Benchmarks of different ratios between synchronization and computation						
No Monitor.[1]	1070	1129	1048	1059	1091	1076
MemArray[1]	1264 18,1%	1491 32,1%	1220 16,4%	1240 17,1%	1273 16,7%	1189 10,5%
No Monitor.[2]	1038	1070	1008	1016	1048	1033
MemArray[2]	1084 4,43%	1218 13,8%	1070 6,15%	1087 6,99%	1130 7,82%	1076 4,16%

1: t=10000 d=0.1 ms   *2*: t=1000 d=1 ms (*t* and *d* are explained in Section 5)

due to the Linux file system cache. However, such effects can thwart reproducibility and stability of the observation and can lead to a distorted monitoring result. The `MemArray Logger` should be preferred, because it provides a good performance and steady behavior. It has also been investigated that the largely extended threshold log files may fit into main memory of current computers easily. The `StringBuilder Logger` is slightly slower than the `MemArray Logger`. If the program behavior is completely unknown the `File Logger` may be used as first try.

In the test program used (t = 100000) most of the code is involved in synchronization. To better investigate the real performance of FerbJmon tools the ratio between computation and synchronization was increased using synthetic computations in each incrementation step, see comment in `OddThread` in Fig. 3. Also, the threshold was reduced. Table 1 shows the execution times using this adapted program for cooperative task execution. The delay *d* that is introduced by the computation in each iteration is limited to 0.1 ms and 1 ms using a time measurement. Only the `MemArray Logger` was used. Table 1 also includes the relative overhead of the execution time compared to the unmonitored program run. The overhead decreases significantly when the computation time per iteration increases. Using 1 ms of computation in each iteration the overhead is only 4–14 % in this experiment. In this test, the overall execution times were less influenced by the operating system and CPU frequency scaling behavior chosen. This can be caused by the increased amount of computations, which may cause all operating system to adjust the CPU frequency to the maximum value.

6 Conclusion

FerbJmon tools can generate call graphs and timeline diagrams of thread accesses on Java objects from runtime traces. The runtime traces are captured using a lightweight monitoring based on Java Bytecode instrumentation. FerbJmon tools are helpful to understand the chronological order of synchronizations between threads. Examples of a producer/consumer and a cooperative task execution were presented. Benchmarks of the logging mechanisms revealed that the execution time is not increased too much in the examples.

FerbJmon tools have already been used successfully in a course on parallel programming for undergraduate students at the University of Bayreuth, Germany. Students tasks were the detection of race conditions in a multithreaded program for scalar product computation and the analysis of the lock order on different implementations of read/write locks. Students found it much easier to generate visualizations from their implementations than using *print* statements within the code. Thanks go to Prof. Thomas Rauber for making this possible.

Acknowledgments. The development of FerbJmon tools was improved by many contributors, in particular Sascha Hunold, Thomas Reichel, Björn Krellner, Matthias Korch, and Thomas Rauber are named. Thanks for the discussions and notes.

References

1. Javashot Java Dynamic Call Graph. https://code.google.com/p/javashot/. Accessed 31 May 2015
2. Javassist Java Bytecode Engineering Toolkit. http://jboss-javassist.github.io/javassist/. Accessed 31 May 2015
3. VisualVM. https://visualvm.java.net/. Accessed 31 May 2015
4. Brown, R., Shoop, E., Adams, J., Clifton, C., Gardner, M., Haupt, M., Hinsbeeck, P.: Strategies for preparing computer science students for the multicore world. In: Proceedings of the 2010 ITiCSE Working Group Reports, pp. 97–115. ACM (2010)
5. Ellson, J., Gansner, E.R., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz - open source graph drawing tools. In: Mutzel, P., Jünger, M., Leipert, S. (eds.) GD 2001. LNCS, vol. 2265, p. 483. Springer, Heidelberg (2002)
6. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., Lea, D.: Java Concurrency in Practice. Addison-Wesley Professional, Reading (2006)
7. Grossman, D., Anderson, R.E.: introducing parallelism and concurrency in the data structures course. In: Proceedings of the Technical Symposium on Computer Science Education, pp. 505–510. ACM (2012)
8. Leroux, H., Réquilé-Romanczuk, A., Mingins, C.: JACOT: A Tool to dynamically visualise the execution of concurrent Java programs. In: Proceedings of the International Conference on Principles and Practice of Programming in Java, pp. 201–206. Computer Science Press, Inc. (2003)
9. Malnati, G., Cuva, C.M., Barberis, C.: JThreadSpy: teaching multithreading programming by analyzing execution traces. In: Proceedings of the Workshop on Parallel and Distributed Systems: Testing and Debugging, pp. 3–13. ACM (2007)

10. Mattson, T.G., Sanders, B.A., Massingill, B.L.: Patterns for Parallel Programming. Addison Wesley, Reading (2004)
11. Torbert, S., Vishkin, U., Tzur, R., Ellison, D.J.: Is Teaching parallel algorithmic thinking to high school students possible?: one teacher's experience. In: Proceedings of the Technical Symposium on Computer Science Education, pp. 290–294. ACM (2010)
12. Williams, A., Thies, W., Awasthi, P.: Static deadlock detection for Java libraries. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 602–629. Springer, Heidelberg (2005)