Software Performance Anti-Patterns Observed and Resolved in Kieker

Christian Wulf
Software Engineering Group
Kiel University
24098 Kiel, Germany
chw@informatik.uni-kiel.de

Wilhelm Hasselbring
Software Engineering Group
Kiel University
24098 Kiel, Germany
wha@informatik.uni-kiel.de

ABSTRACT

Software performance anti-patterns describe bad-practice solutions for particular problems. They help in sensitizing software engineering to such situations. In this paper, we focus on anti-patterns of monitoring and dynamic analysis frameworks, such as Kieker. These frameworks typically have high requirements on a low monitoring overhead and a high-throughput analysis performance.

We describe three observed anti-patterns which influenced previous versions of Kieker with a high impact on the performance. Moreover, we present our solution for each of the anti-patterns.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Patterns; E.1 [**Data Structures**]: Records; E.2 [**Data Storage Representations**]: Object representation

General Terms

Design, Performance

Keywords

Application Performance Monitoring, Dynamic Analysis, Kieker Framework, Performance Anti-Patterns

1. INTRODUCTION

In software engineering, patterns are a well-known means to capture knowledge at least since Gamma et al. [1] present their collection of design patterns. Such patterns represent good-practice solutions to problems which occur frequently. Apart from that, anti-patterns [2, 6] have emerged. Anti-patterns represent bad-practice solutions to common problems and should thus sensitize software engineers to such solutions. A subset of anti-patterns are performance anti-patterns [3]. They focus on solutions to problems which have a negative impact on the performance of software system.

In this paper, we present three performance anti-patterns (PAA) which we observed in the monitoring and dynamic analysis framework Kieker [4]. All address problems with the serialization and deserialization of monitoring records. For each of them, we describe its context, its problem, and a solution which is faster and in some case even less complex.

The paper is structured as follows. In Section 2, we describe a performance anti-pattern which comprises the parallelization of sequential dependencies. Afterwards, in Section 3, we describe a performance anti-pattern which covers the reflection-based record reconstruction. In Section 4, we describe a performance anti-pattern which addresses the problem of an exception-based buffer underflow detection. Finally, we conclude this paper in Section 5.

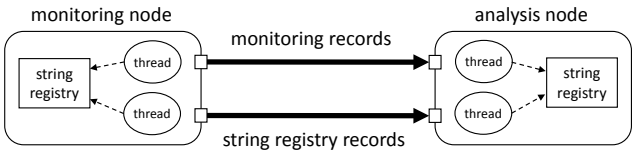

Figure 1: Parallel transfer of monitoring records and string registry records in Kieker

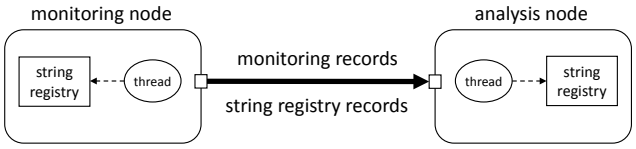

Figure 2: Our proposed sequential transfer of monitoring records and string registry records

2. PAA #1: PARALLELIZING SEQUENTIAL DEPENDENCIES

Context. In Kieker 1.10, we find the following implementation for transferring monitoring records from the monitoring node to the analysis node via TCP. Each node runs two threads as illustrated by Figure 1. Below, we describe the serialization tasks of the two threads of the monitoring node. The threads of the analysis node perform analogous deserialization tasks.

The first thread of the monitoring node continuously receives the monitoring records from the monitoring controller and serializes them as a byte sequence to a buffer. If the buffer is full, the thread sends its content to the analysis node via TCP. Afterwards, the thread clears its buffer and continues with receiving monitoring records..

To reduce the monitoring overhead and to increase the throughput, Kieker compresses string attributes of monitoring records. For this purpose, the monitoring controller maintains a string registry which holds a map of string/identifier entries. Once a new string attribute is recognized while serializing a monitoring record, a new unique 4-byte identifier is generated. The string and the identifier are then

stored in the string registry. Hereupon, the string registry creates a new string registry record, copies the string and the identifier into it, and finally sends it to the second thread. The second thread performs the same procedure as the first thread, however, for string registry records.

Problem Description. When the first thread of the analysis node deserializes a monitoring record, it needs to replace each string identifier by its corresponding string. For this purpose, it looks up the string identifier in a string registry which is similar to the one used on the monitoring node. If the second thread has not yet received or registered the corresponding string registry record, the first thread blocks.

Compared with a single-threaded solution, this implementation has the following three disadvantages: First, two TCP connections are established: one for transferring monitoring records and one for transferring string registry records. Thus, an administrator needs to maintain and to observe two open ports which can lead to a higher maintenance effort and a higher security risk. Second, the threads on each node must synchronize with each other via the string registry. Thus, a higher communication effort can be perceived. Third, the first thread waits for the second thread if a monitoring record is being deserialized before its string attributes have been registered in the string registry.

In summary, the deserialization of monitoring records depends on their string attributes. Nevertheless, Kieker 1.10 tries to parallelize this intrinsically sequential dependency.

Our Solution. We propose an approach that handles the transfer of both the string registry records and the monitoring records in one single thread per node. We call these threads the *writer thread* and the *reader thread*, respectively. This approach, as illustrated by Figure 2, requires the following two changes in the source code: First, we added to each record type a new method called `registerStrings` which registers all string attributes of the record with the passed string registry. Second, we adapted the string registry so that it does not send a new string registry record to another thread. Instead, it sends the string registry record to the writer thread. The writer thread then writes the record to the buffer.

In this way, we ensure that all string registry records are transferred before its corresponding monitoring record is serialized. Moreover, this version requires one open port only and avoids an unnecessary synchronization overhead. Finally, the reader thread does not block anymore on the analysis node except for an empty buffer.

This anti-pattern is a special case of the anti-pattern *Unnecessary Processing* [3] with a focus on parallel processing.

3. PAA #2: REFLECTION-BASED RECORD RECONSTRUCTION

Context. In Kieker 1.10, we find the following implementation for reconstructing a monitoring record from a TCP input stream. First, a software buffer is filled with a fixed number of bytes from this stream. This buffer serves as a cache to reduce the amount of accesses to the application programming interface of the operating system. Then, the following algorithm is executed as long as the buffer has enough bytes left. Figure 3 shows the corresponding

```
1 int classId = buffer.getInt();
2 recordClassName = stringRegistry.get(classId);
3 record = AbstractMonitoringRecord.
      createFromByteBuffer(recordClassName,
      buffer, stringRegistry);
```

Figure 3: Reflection-based record reconstruction in Kieker 1.10

```
1 int classId = buffer.getInt();
2 recordClassName = stringRegistry.get(classId);
3 recordFactory = cachedRecordFactoryCatalog.get(
      recordClassName);
4 record = recordFactory.create(buffer,
      stringRegistry);
```

Figure 4: Factory-based record reconstruction in Kieker 1.11

source code. The TCP reader component first reads a 4-byte string identifier from the buffer (Line 1). The corresponding string represents the fully qualified class name (FQCN) of the record type which is about to be reconstructed (Line 2). The TCP reader then passes the FQCN, the buffer, and the string registry to a generic record factory which constructs a new instance of the correct record based on the passed information (Line 3).

Problem Description. The generic record factory utilizes Java's Reflection API to search for a record type based on its FQCN in the class path. If the record type was found, the factory stores the FQCN and the reflective constructor of the record type in a map which serves as a cache. In this way, the factory only needs to search for each record type once. Subsequently, the factory again uses the Reflection API to invoke the reflective constructor with the buffer and the string registry as parameters. Finally, the record's constructor reads its attributes from the buffer and thus returns a correctly reconstructed instance. Hence, although the constructor of each record type is cached, it must still be invoked using the Reflection API.

This implementation has the following major disadvantage: Java's Reflection API in general and the reflective invocation of a constructor in particular are slow. The invocation of a constructor via Java's keyword `new` is always faster than its reflective counterpart.[1]

In summary, Java's Reflection API should not be used in frequently occurring situations to avoid performance problems. Nevertheless, Kieker 1.10 makes use of it to reconstruct monitoring records.

Our Solution. We propose an approach that uses the abstract factory pattern [1] and a naming convention. Instead of using a generic record factory, each record type is assigned a dedicated record factory. This record factory implements a factory method that statically creates and returns a new instance of the corresponding record type via Java's keyword `new`. The name of the factory consists of the name of the record type and the suffix `Factory`.

[1] `http://docs.oracle.com/javase/tutorial/reflect/index.html`

Figure 4 shows the adapted source code. Now, the TCP reader searches for and caches the record type's factory instead of the record type (Line 3). Afterwards, the resolved factory is used to deserialize and instantiate a new record (Line 4).

Since programming a record factory for each record type by hand is cumbersome, we extended the generator of Kieker's Instrumentation Record Language.[2] Now, the generator does not only generate the record types, but it also generates the record type's factory.

This anti-pattern is not covered by any of the anti-patterns described by Smith et al. [3].

4. PAA #3: EXCEPTION-BASED BUFFER UNDERFLOW DETECTION

Context. In Kieker 1.11, we find the following implementation for detecting and reacting on a buffer underflow. Figure 5 shows the corresponding source code. First, the current position of the buffer is saved for later use (Line 2). Then, the record reconstruction process starts by invoking the method `reconstruct` (Line 3). The record identifier is read from the buffer to resolve the corresponding record factory. Afterwards, the factory invokes the record's constructor which in turn deserializes each of the record's attributes one by one from the buffer. Whenever the buffer does not contain enough bytes for the next data element, it throws a `BufferUnderflowException` signaling an abort of the record reconstruction. Hereupon, the TCP reader catches the exception (Line 4) and triggers a refill of the buffer (Line 5). Finally, it continues with the record reconstruction from the previously saved buffer position (Line 6).

Problem Description. The most CPU and memory consuming part of this implementation strategy lies in resolving and storing the current stacktrace in the newly created exception. For example, if we assume a buffer size of 8192 bytes and a mean record size of 40 bytes, then a buffer underflow exception is thrown each 204th record reconstruction. If we consider that Kieker achieves a throughput of more than one hundred thousand records per second [cf. 5], 500 exceptions are thrown each second. In general, the smaller the buffer size, the more frequently exceptions are thrown, and the slower is the overall performance.

Our Solution. We propose an approach that completely avoids creating and throwing buffer underflow exceptions. For this purpose, we use the buffer's capability to check how many bytes are remaining within the buffer. The record reconstruction process works as follows. Besides saving the buffer's current position as previously (Line 1), the TCP reader now also checks within the method `reconstruct` (Line 2) whether the buffer has four bytes remaining for the record identifier. If remaining, the TCP reader resolves the record factory and invokes its new method `getRecordSizeInBytes`. The method's return value represents the size of the record which is about to be reconstructed by the factory. If this size is smaller than the bytes remaining in the buffer, then the record is successfully reconstructed (Line 3). Otherwise,

[2] `https://github.com/research-iobserve/instrumentation-language/wiki`

```
1 try {
2   // save buffer's current position
3   reconstruct(buffer);
4 } catch (BufferUnderflowException e) {
5   // refill buffer
6   // reset buffer's position
7 }
```

Figure 5: Exception-based buffer underflow detection in Kieker 1.11

```
1 // save buffer's current position
2 boolean success = reconstruct(buffer);
3 if (!success) {
4   // refill buffer
5   // reset buffer's position
6 }
```

Figure 6: Boolean-based buffer underflow detection in Kieker 1.12

`reconstruct` returns false indicating that the buffer needs to be refilled (Lines 4–5).

Our solution replaces the exception-based buffer underflow detection by checking the buffer's remaining bytes before actually reading from the buffer. We avoid to trigger the exception and return a boolean value instead. In this way, our approach is faster, especially for small buffer sizes.

This anti-pattern is a special case of the anti-pattern *Excessive Dynamic Allocation* [3] with a focus on exceptions.

5. CONCLUSION

In this paper, we presented the following three performance anti-patterns which we observed in previous versions of Kieker: (1) Parallelization of sequential dependencies, (2) reflection-based record reconstruction, and (3) exception-based buffer underflow detection. For each of the performance anti-patterns, we explained its problems and gave a better solution.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Prentice Hall, 1995.

[2] A. Koenig. Patterns and antipatterns. In *The Patterns Handbooks*. Cambridge University Press, 1998.

[3] C. U. Smith and L. G. Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *Proc. of the Int. CMG Conference*, 2003.

[4] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proc. of the ICPE*, 2012.

[5] J. Waller, F. Fittkau, and W. Hasselbring. Application performance monitoring: Trade-off between overhead reduction and maintainability. In *Proc. of the Symposium on Software Performance*, 2014.

[6] M. Wooldridge and N. R. Jennings. Pitfalls of Agent-oriented Development. In *Proc. of the AGENTS*, 1998.