

# Instrumenting C and Fortran Software With Kieker

Reiner Jung<sup>1</sup>, Sven Gundlach<sup>1</sup> and Wilhelm Hasselbring<sup>1</sup>

<sup>1</sup>Kiel University, Christian-Albrechts-Platz 4, 24103 Kiel, Germany

## Abstract

Kieker is a versatile monitoring framework well established for performance analysis and dynamic architecture recovery for JVM languages. As applications are not only written in Java, in the past Kieker has been supplemented with probes for Perl, Pascal, and C++ in context of embedded systems.

In the context of our project OceanDSL, which aims to provide DSLs for ocean system models, we need to comprehend existing climate models written in Fortran and C. Thus, probes for C and Fortran were required. In this paper, we report on our efforts in realizing minimal invasive monitoring utilizing compiler features, probes, and tools tailored to instrument application based on these languages.

## Keywords

Application Level Monitoring, Kieker, Instrumentation, C, Fortran

## 1. Introduction

Performance analysis and architecture recovery are key tasks software engineers perform, especially for long-living systems [1]. Kieker provides the facilities to observe applications at runtime and analyze performance and architectures based on runtime and design time data [2, 3].

Kieker was first developed for JVM-based languages, but provides monitoring probes for Perl, Visual Basic, and DotNet. In the context of our project OceanDSL [4], which aims to provide DSLs for ocean system models, we need to comprehend existing climate models written in Fortran and C. Thus, probes for C and Fortran were required.

In this paper, we report on the Kieker language pack for C, utilizing the GNU Compiler Collection (GCC) [5] and compatible compilers. As GCC supports additional languages beside C and Fortran, our probes can also be used with these languages. We illustrate the probes, the adjunct tooling, and the process of the language pack using two Earth System Climate Models (ESCMs).

The Kieker language pack for C is introduced in Section 2 and its application to the two examples is illustrated in Section 3. Finally, we provide a summary and outlook in Section 4.

---

SSP'21: Symposium on Software Performance, November 09–10, 2021, Leipzig, Germany


✉ reiner.jung@email.uni-kiel.de (R. Jung); sven.gundlach@email.uni-kiel.de (S. Gundlach);

hasselbring@email.uni-kiel.de (W. Hasselbring)

ORCID 0000-0002-5464-8561 (R. Jung); 0000-0003-4060-2754 (S. Gundlach); 0000-0001-6625-4335 (W. Hasselbring)



© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

## 2. Instrumentation

The overall Kieker architecture comprises of a monitoring component which is embedded with a weaving technique into an application, an analysis component located outside the application, and a set of shared data structures for monitoring events. The communication between both is realized via files, TCP and a variety of other means of transport and storage.

To support C and other object file languages, we only extend the monitoring side while we reuse the analyses provided by Kieker and Kieker-based projects. Furthermore the Kieker language pack for C [6] supports only binary logging via TCP which allows offsite logging and has the least impact on the analyzed application. In the following, we introduce all the necessary building blocks to monitor such programs.

**Configuration** Similar to its Java cousin, the C variant of Kieker monitoring uses a configuration file to set parameters. The configuration file can be placed anywhere in the file system. To inform Kieker where the file can be found, a `KIEKER_CONFIG` environment variable must be set. In case the variable is not set, Kieker will work with built-in defaults.

**Event Types** Kieker uses a language independent notation for its event types called Instrumentation Record Language (IRL) [7]. With the IRL generator, we can generate event types for various languages like C or Fortran. The generator produces `structs` and `typedefs` for all Kieker events alongside serializer functions that produce binary serializations using big-endian encoding also known as network byte order.

Kieker's default binary protocol uses string tables or string registries to avoid redundantly transferred string values. Each unique string is given an ID and sent once following the Kieker format specification for binary data. In the serialized event, the string is then represented by its ID and the string is sent ahead of the event by the monitoring controller.

**Monitoring** The monitoring component handles strings, timestamps, trace metadata, and writing the data via TCP. While in Java the number of event types can be resolved at runtime, in C this is done at compile time. Thus, event types have to be registered upfront. Therefore, three default flow events are currently supported, i.e., `TraceMetadata`, `BeforeOperationEvent` and `AfterOperationEvent`. All events can be overwritten with a event type mapping file.

In the current implementation, logging is supported via TCP. While other options can be added, in all our use cases it is beneficial to be able to collect and process the monitoring data in a separate process, preferably on a separate machine to reduce interference. The TCP writer can be configured with two parameters, for the destination host and port where a collector or analysis tool is listening. The defaults for both values are `localhost` and port `5678`.

**Probes** The Kieker language pack for C provides probes for manual instrumentation, i.e., modifying the code, and probes to be woven in by compilers utilizing the GCC instrumentation facility. The latter declares two functions for the entry and exit points of a function, respectively.

The manual probe functions have two parameters: `class_signature` and `operation_signature`. As C does not have classes, the class signature contains the name of the file where the function

is defined in. Further we use the name class signature to conform to the Java induced naming scheme. Therefore the operation signature contains the function signature or a pointer.

The GCC based probes reuse the manual probes. However, the GCC instrumentation interface does not provide names for files and operations. While this data can be linked to an executable, they are not available at runtime. Instead, the caller and callee are represented by a memory address. Thus, we use the address as operation signature and use a placeholder as class signature.

**Weaving** The probes are injected with the instrumentation feature of the GCC [5] and the Intel Fortran compiler [8]. This is activated via the compiler (option `-finstrument-functions`) and weaves two calls into each function that are invoked when a function is entered or left, respectively (cf. Listing 1). We had used AspectC++ in the past, but it is only able to process C and C++ code [9].

Listing 1: GCC instrumentation function names [10]

```
void __cyg_profile_func_enter (void *this_fn ,  
    void *call_site) __attribute__((no_instrument_function));  
void __cyg_profile_func_exit (void *this_fn ,  
    void *call_site) __attribute__((no_instrument_function));
```

The weaving should work with any compiler optimization settings. However, we suggest to avoid inlining functions. Also to resolve function names after data collection, debugging information must be added by the compiler (option `-g`). To collect monitoring data and store it in log files, Kieker provides a collector that listens on a TCP port (default 5678) for monitoring data and stores the data in Kieker log files supporting all logging features, including compression.

Further, in case the collector is too slow and blocking the monitored application, monitoring data can also be collected with `netcat` and then replayed to the collector afterwards.

**Post-Processing** Post processing is required when the GCC instrumentation function has been used, as the log only contains function pointers. Thus, we developed a log rewrite tool that resolves function and file names in the Kieker log utilizing the `addr2line` tool and the binary program file containing debugging symbols. The tool can be found in OceanDSL tool project.<sup>1</sup>

### 3. Application to Climate Models

We applied the language support to two existing earth system climate models, i.e., MITgcm [11] and UVic [12]. To instrument them with Kieker, we used an instrumentation feature respectively a compiler option `-finstrument-functions`. The illustration of the required setup with a short introduction is as follows.

**MITgcm** is a fairly modular general circulation model from the Massachusetts Institute of Technology. It is used to simulate the atmosphere and the ocean. It is configured and

---

<sup>1</sup>OceanDSL tool <https://git.se.informatik.uni-kiel.de/oceandsl/oceandsl-java-tools>

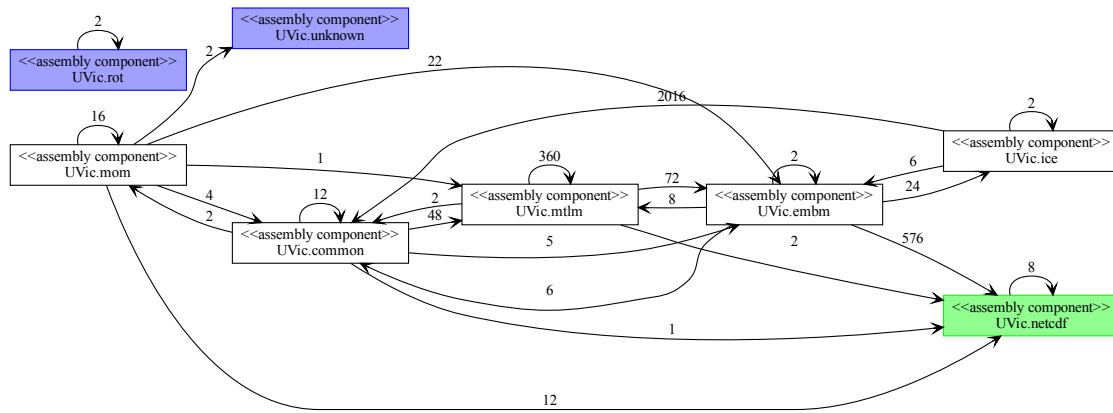
parameterized via a large set of small configuration and parameterization files which allow to setup different experiments. These can be seen as different variants of the model.

MITgcm compiles with GCC and provides a set of configuration files for different architectures and compilers. The setup required only to modify the existing setup files by extending `linux_amd64_gfortran` and deactivating the optimization by setting the compiler flags for C and Fortran and lastly to extend the library path (cf. Listing 2).

Listing 2: MITgcm configuration

```
FOPTIM=" "
F90OPTIM=" "
FFLAGS="$FFLAGS -finstrument-functions -g"
CFLAGS="$CFLAGS -finstrument-functions -g"
LIBS="$LIBS -L/usr/lib/x86_64-linux-gnu \
-L/kiiker-lang-pack-c/libkiiker/.libs -lkiiker -ldl"
```

**UVic** The second earth system climate model UVic is from the University of Victoria. In contrast to MITgcm it is a monolithic applications which is configured and parameterized with two files able to toggle features and other settings. These feature toggles also allow to create different variants. However, each toggle affects various locations within the code base. Figure 1 shows an architecture recreation of UVic based on dynamic and static analysis with Kieker. The compilation and linking configuration is mingled with many other settings in the `mk.in` file. For the setup we had to extend the library path and set the correct parameters for the compiler as shown in Listing 3.



**Figure 1:** The module structure of the UVic setup. White boxes indicate the modules based on static and dynamic data. Green and blue boxes are derived solely from dynamic or static data, respectively.

### Listing 3: UVic configuration

```
Libraries = -lnetcdf -lnetcdf -L/usr/lib/x86_64-linux-gnu \
-L/kiiker-lang-pack-c/libkiiker/.libs -lkiiker
Compiler_F = ifort -r8 -g -finstrument-functions -O0 \
-warn nuncalled -c
Compiler_f = ifort -r8 -g -finstrument-functions -O0 \
-warn nuncalled -c
Linker = ifort -r8 -g -finstrument-functions -O0 \
-warn nuncalled -o
```

As both examples show, the integration is fairly simple and can easily be introduced into build environments.

## 4. Conclusions

We present our Kieker language pack for C which can also be used with other programming languages that are supported by GCC and compatible compilers. The language pack provides an implementation of all Kieker event data types and probes for manual and compile time instrumentation using the compiler instrumentation facilities. We demonstrated the application of these probes with the GNU Compiler Collection and the Intel Fortran Compiler, applied to two existing earth system climate models. Furthermore, we discussed necessary tooling to collect and post process the monitoring data.

For future work, we will extend the set of probes to be able to log instances and add support for AspectC++ as an alternative instrumentation technique. The current implementation does not support adaptive monitoring which we will add in the future utilizing the Kieker Probe Controller. Furthermore, we will contribute the essential tooling from OceanDSL to Kieker.

The Kieker language pack for C shows that it is simple to provide monitoring for other programming languages and technologies with Kieker. The C implementation shows that large software systems in C and Fortran can be dynamically analyzed with little effort for the developer, as the weaving can be controlled with a few compiler options.

## Acknowledgments

Funded by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation), grant no. HA 2038/8-1 – 425916241.

## References

- [1] U. Goltz, R. Reussner, M. Goedicke, W. Hasselbring, L. Märtin, B. Vogel-Heuser, Design for future: managed software evolution, *Computer Science – Research and Development* 30 (2015) 321–331. doi:10.1007/s00450-014-0273-9.
- [2] W. Hasselbring, A. van Hoorn, Kieker: A monitoring framework for software engineering research, *Software Impacts* 5 (2020). doi:10.1016/j.simpa.2020.100019.

- [3] A. van Hoorn, J. Waller, W. Hasselbring, Kieker: A framework for application performance monitoring and dynamic software analysis, in: Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012), ACM, 2012, pp. 247–248. doi:10.1145/2188286.2188326.
- [4] R. Jung, S. Gundlach, S. Simonov, W. Hasselbring, Developing domain-specific languages for ocean modeling, in: Proceedings of the 8th Collaborative Workshop on Evolution and Maintenance of Long-Living Software Systems (EMLS 2021), volume 2814 of *CEUR*, 2021. URL: <http://ceur-ws.org/Vol-2814/>.
- [5] GCC-GNU, The GNU Compiler Collection, 2021. URL: <https://gcc.gnu.org>.
- [6] Kieker Project, Kieker Language Pack for C, 2021. URL: <https://github.com/kieker-monitoring/kieker-lang-pack-c.git>.
- [7] R. Jung, C. Wulf, Advanced typing for the Kieker instrumentation languages, in: Symposium on Software Performance 2016, 2016. URL: <http://oceanrep.geomar.de/34626/>.
- [8] Intel Corporation, Intel Fortran Compiler, 2021. URL: <https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/fortran-compiler.html>.
- [9] O. Spinczyk, et al., AspectC++ an aspect-oriented extension to the C++ programming language, in: Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications, 2002, pp. 53–60.
- [10] J. Racine, The Cygwin tools: a GNU toolkit for Windows, 2000.
- [11] V. Artale, S. Calmanti, et al., An atmosphere–ocean regional climate model for the mediterranean area: assessment of a present climate simulation, *Climate Dynamics* 35 (2010) 721–740. doi:10.1007/s00382-009-0691-8.
- [12] A. J. Weaver, M. Eby, et al., The UVic earth system climate model: Model description, climatology, and applications to past, present and future climates, *Atmosphere-Ocean* 39 (2001) 361–428. doi:10.1080/07055900.2001.9649686.