Generic Library Interception for Improved Performance Measurement and Insight

Ronny Brendel[1(✉)], Bert Wesarg[2], Ronny Tschüter[2], Matthias Weber[2], Thomas Ilsche[2], and Sebastian Oeste[2]

[1] Oak Ridge National Laboratory, Oak Ridge, USA
ronnybrendel@gmail.com
[2] Technische Universität Dresden, Dresden, Germany
{bert.wesarg,ronny.tschueter,matthias.weber,
thomas.ilsche,sebastian.oeste}@tu-dresden.de

Abstract. As applications grow in capability, they also grow in complexity. This complexity in turn gets pushed into modules and libraries. In addition, hardware configurations become increasingly elaborate, too. These two trends make understanding, debugging and analyzing the performance of applications more and more difficult.

To enable detailed insight into library usage of applications, we present an approach and implementation in Score-P that supports intuitive and robust creation of wrappers for arbitrary C/C++ libraries. Runtime analysis then uses these wrappers to keep track of how applications interact with libraries, how they interact with each other, and record the exact timing of their functions.

Keywords: Clang · Instrumentation · Library · LLVM ·
Performance analysis · Performance optimization · Software module ·
Tracing · Wrapper

1 Introduction

To push science and businesses further, today's software becomes increasingly powerful but also complex. Software libraries allow offloading this complexity into subunits, so that developers can focus on adding functionality by using
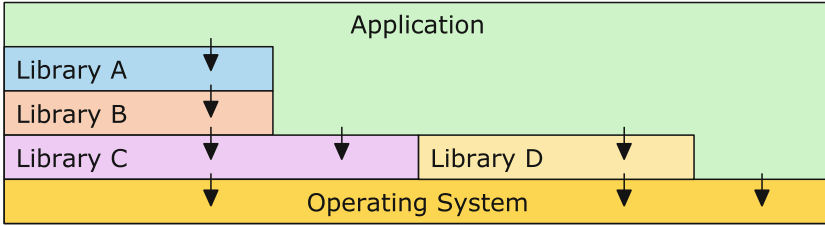
Fig. 1. Typical software stack with an application relying on four libraries.

them, rather than implementing every detail themselves. But the complexity does not disappear—It gets pushed down to lower levels. The gained development convenience is traded for an increased effort of debugging and overall reasoning about the application including its performance characteristics. Figure 1 depicts a typical application and its software dependencies.

A similar development takes place in computer architecture. The adoption of multiple cores per CPU, heterogeneous architectures, complex cache/memory hierarchies, elaborate interconnect networks, as well as deep I/O hierarchies gives rise to a multitude of potential performance problems. This increasing complexity in software and hardware makes performance analysis an integral part of the software life cycle.

Tool chains providing modern performance analysis capabilities include Linux Perf [23], NVIDIA profiling tools [26], Intel VTune Amplifier [30], Score-P [20], Arm MAP [2] and HPCToolkit [1]. These tools combine multiple data collection techniques, like sampling, call stack unwinding, tools interfaces, library wrapping, compiler instrumentation, and manual instrumentation in various ways. The goal is to gather data as detailed as needed while alleviating the disadvantages of individual techniques. For example, it is common to combine sampling and call stack unwinding with library wrapping for important libraries. Sampling gives coarse-grained stochastic timing information of the application's function call sequence, while library wrappers count and measure exact timings of library calls. Aside from counting calls and measuring time, libraries like POSIX Threads [6], and I/O libraries like HDF5 [16] and ADIOS [22] are commonly wrapped to extract semantic information (e.g. written bytes) from arguments passed to library functions.

Which wrappers are available is limited to what each performance tool supports. Even if an application developer is interested in exact function call tracking of certain libraries, there is no well supported way to achieve this in any performance analysis tool today.

To address this, we introduce *user library wrapping*, which is included in the upcoming release of the open-source performance monitor Score-P. The feature empowers application developers to easily generate library wrappers for any C/C++ library. This is significant, because:

– With just link time changes, developers can now get exact performance information on any C/C++ library they want.
– They can analyze closed-source libraries, like the Intel MKL [24].
– They can track function calls from a library to itself and between libraries.

Score-P benefits from user library wrapping for the following reasons. First, regular compiler instrumentation provides no call-backs upon library entry/exit. Second, compiler instrumentation often yields high event rates, which leads to diminished performance and large event recordings. This necessitates a filtering workflow that in turn complicates the whole measurement process. With user library wrapping, developers can forego compiler instrumentation and still capture critical performance data, and have a small low-overhead recording at the same time—no filtering needed. Third, wrappers give exact function call counts and timings as opposed to the statistical information from sampling and call stack unwinding. Additionally, it simplifies creating fixed wrappers that capture library semantics not only in Score-P, but for all tools that rely on library wrappers.

We took great care to make wrapper creation and usage as intuitive and simple as possible. Numerous checks with polished error messages ensure the wrapper works correctly or let the developer know why it might not.

This paper is divided as follows: Sect. 2 enumerates related work. Section 3 first presents basics on library wrapping. It then details the workflow for creating and using wrappers while highlighting some implementation choices, by the example of wrapping the QtWidget and QtGui modules. Section 4 demonstrates how our approach aids investigating the performance characteristics of two real-world scientific applications. The last two sections offer conclusions and indicate points of interest for future development.

2   Related Work

Wrapping C/C++ libraries is not new. SWIG [4], first released in 1996, generates wrappers for C/C++ libraries so they can be called from other languages like Python, Go and Lua. SWIG does not provide library interception for extracting, e.g., performance data. Furthermore, it is not possible to create C++ or C wrappers for C/C++ libraries. SWIG uses its own C/C++ preprocessor and parser.

Recently, Google released the C++ Language Interface Foundation (CLIF) [8] which provides similar functionality to SWIG. It uses Clang [7] to analyze the library headers, and for now only generates wrappers for Python.

Some libraries, like OpenMP 5 [13] and CUDA [10], offer a so-called *tools interface* for analysis tools to hook into. In that case, library wrapping is not needed. But for most libraries, wrappers are required to gain insight into their usage. MPI [25] provides a special *profiling interface*, which helps create wrappers by providing all functions as weak symbols so that they can be overridden. Wrapper functions call the actual MPI functions through a P-prefixed symbol, e.g. `PMPI_Send`.

One possible application for tools interfaces and library wrappers is to check for correct API usage. For example MUST [19] uses MPI's profiling interface to ensure correct use and to detect possible deadlocks. The wrapping code is generated manually with a simple proprietary wrapper generator.

Software performance analysis tools commonly use fixed wrappers to gain insight into the use of specific libraries. For example Arm MAP [2] is a commercial profiler specialized in analyzing multi-paradigm applications. It wraps MPI and OpenMP functions and uses the tools interface of CUDA. Various open-source performance analysis tools exist. Some of them are Extrae [14], HPCToolkit [1] and Score-P [20]. All three support a variety of parallelization schemes and hardware platforms. They differ in techniques, focus and user interface, but are similar in terms of utilizing library wrapping.

VampirTrace includes a simple implementation of user library wrapping [11]. It is based on CTool [9] (abandoned in 2004), supports only C, has several technical limitations and needs manual intervention in most cases. Score-P is VampirTrace's successor.

TAU offers user library wrapping via the `tau_wrap` and `tau_gen_wrapper` commands [28]. It uses the Edison Group's commercial C/C++ parser [12]. TAU's implementation has multiple limitations. For example, it does not support C++, cannot wrap functions with function pointers or ellipsis arguments, and compile and link flags are not customizable.

3   Methodology

Our goal is to provide a simple and robust way to record performance data on library function calls. For this, we need an opportunity to intercept them. That means whenever a library function is called, the measurement system has to be invoked.

3.1   Library Call Interception

We distinguish two wrapping methods based on when interception is set up: *link time* and *runtime*. These two methods also differ in the kind of functions that can be intercepted.

Link Time: The first approach is based on the `--wrap` option of the GNU linker[1]. For example, to wrap the function `foo`, we have to implement the corresponding wrapper function `__wrap_foo`. The original function is available via the `__real_foo` symbol. Then specifying `--wrap foo` in the link command enables wrapping `foo`, and the GNU linker resolves these symbols appropriately. This approach is limited to instances where the link step of the application can be modified, as the symbols of interest need to be specified at link time. Wrapping symbols called from shared libraries does not work, because the linker resolves these symbols at runtime.

---

[1] https://sourceware.org/binutils/docs-2.28/ld/Options.html.

Runtime: At the start of executing an application, the dynamic linker loads and links all dependent shared libraries. The second approach modifies the order in which the dynamic linker loads them. To wrap a function, we provide a replacement function with the same symbol name as the wrapped function. The linker, then, needs to link the wrapper before the target library. One way to achieve this is modifying the link step to put the wrapper library before the original one. Alternatively, let the environment variable `LD_PRELOAD`[2] point to the wrapper library before executing the application. The latter method has the advantage that it does not need to modify the link step. Once called, the wrapper function loads the target library via `dlopen`, searches for the address of the original symbol using `dlsym`, and then forwards the original call. With link step modification, this approach can intercept all calls that link time wrapping can, plus those that originate from shared libraries. The `LD_PRELOAD`-based version can only intercept calls to shared libraries, not to statically linked ones.

Both mechanisms require wrappers that pose as the original functions. For each call, a wrapper function notifies the performance monitor before and after forwarding the call to the original function. In the next section we present the workflow with which users create their own library wrapper.

3.2   Workflow

In this work we extend Score-P—a state-of-the-art software performance monitor. Figure 2 shows its high-level architecture.

The goal is to make calls to library functions available for performance analysis. For this, we add functionality to record timestamped *enter*- and *exit*-events for these calls.

The process of generating a library wrapper is intricate and error-prone. Thus, the highest priority in the design of user library wrapping is to make it reliable. To guide the user through these potential problems we introduce a workflow, which the following paragraphs explain. We motivate some of the implementation choices by highlighting the intricacies that necessitate them. Figure 3 depicts the steps involved.

Initialize the Working Directory. The tool `scorep-libwrap-init` initiates the bootstrapping process. For this it creates a working directory where all subsequent steps take place in. The command takes a number of arguments that concern compilation setup, linking setup, and the name of the user library wrapper. Essentially, the user specifies how to compile and link an application using the target library.

In this step, Score-P tries to locate potential shared versions of the target libraries. It lets the user know if it cannot find any to avoid confusion due to failing `dlopen`-calls later on.

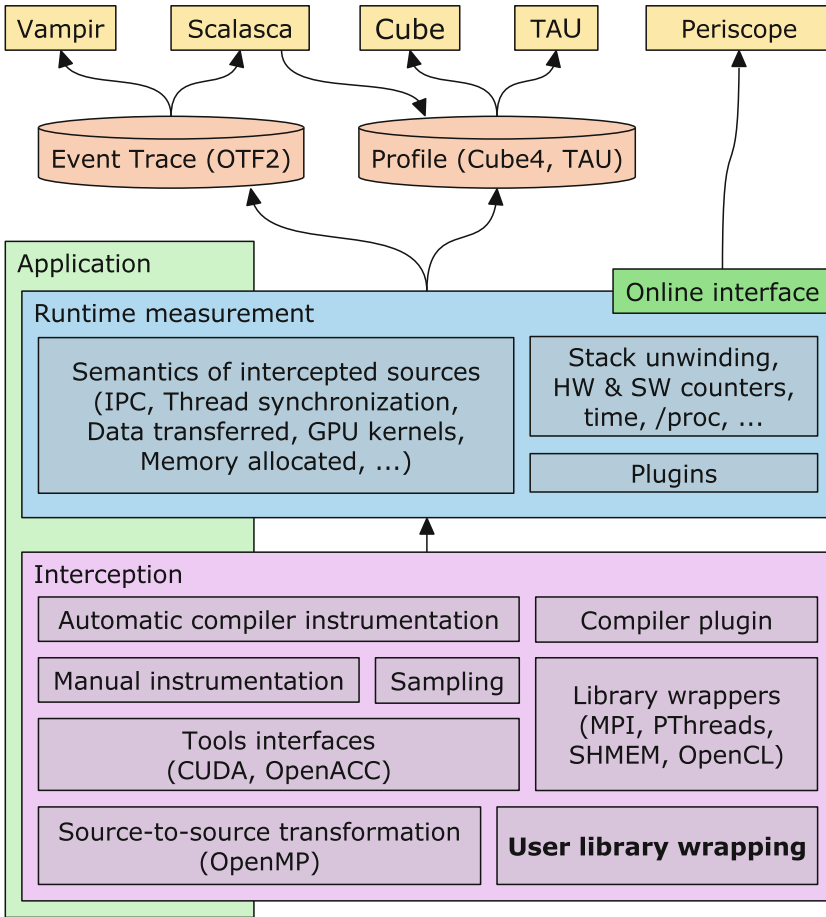[2] http://man7.org/linux/man-pages/man8/ld.so.8.html.

Fig. 2. Overview of the Score-P measurement system architecture. User library wrapping provides an additional interception mechanism.

`scorep-libwrap-init` creates a number of files in the new working directory:

– A detailed documentation with explanations of possible warnings and errors
– A `Makefile` that guides the user through the next steps
– Stub source-, header-, and filter-files, which subsequent paragraphs explain

At the end, the command prints out what the next steps are.

For example, the following command initializes the working directory for a library wrapper of the QtGui and QtWidget modules [27][3]:

---

[3] Full example: https://github.com/score-p/scorep_libwrap_examples/tree/1564c2723 11d04575f9886cd982fc611e07eb295/qt5/qtgui-and-qtwidgets.
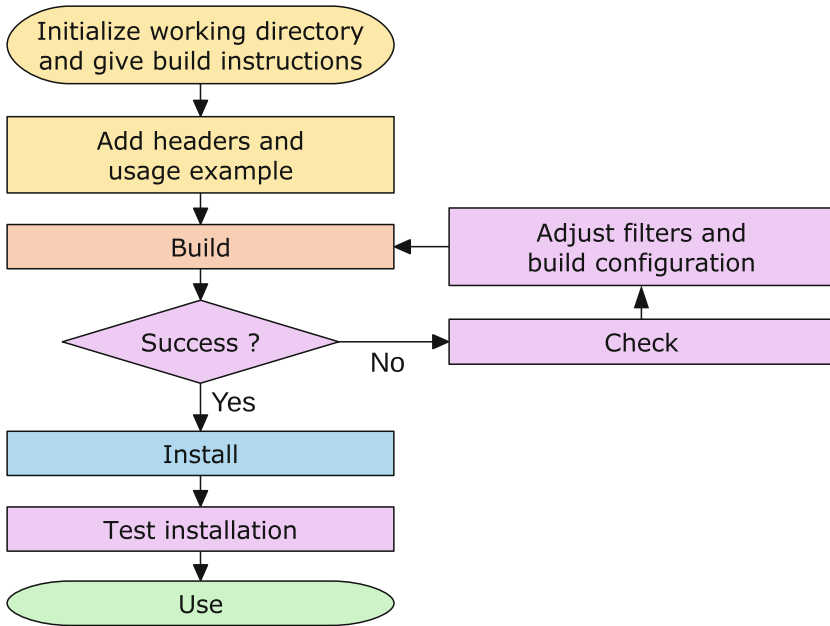
Fig. 3. High-level workflow for creating a user library wrapper.

```
$ scorep-libwrap-init -x c++                    \
    --name qtgui_and_qtwidgets                  \
    --display-name "Qt Gui & Widgets"           \
    --cppflags "-fPIC -I${QT_INCLUDE}"          \
    --ldflags "-fPIC"                           \
    --libs "-lQt5Widgets -lQt5Gui -lQt5Core"
```

Add Library Headers. Next, the user adds an include-statement for each header an application usually includes from the target library to `libwrap.h`. This approach allows the user to specify a sequence of includes and preprocessor macros, which the wrapper generator can then process. Continuing the example, we add:

```
#include <QtGui/QtGui>
#include <QtWidgets/QtWidgets>
```

Create an Example Application. To be able to verify the results, the process needs a test case. For this, the user adds a small usage example to `main.c/cc`. It will be compiled, linked and executed later to test whether the target library and wrapper work. Continuing the example, we write a simple Qt application that opens a window and creates an unused image:

```
int main(int argc, char** args) {
    QApplication app(argc, args);
    QWidget w;
    QImage i;
    w.show();
    return app.exec();
}
```

Create the Wrapper. Before building the wrapper, the user can always adjust the compile- and link-setup by either directly changing the top lines in the makefile, or by invoking `scorep-libwrap-init` again with the `--update` argument.

The user can now attempt to build the wrapper via the `make`-command. First, this links the example application to the target library. If that fails, the provided example is wrong.

Next, it preprocesses `libwrap.h` to create `libwrap.i` with the same compiler and flags that are used to create the provided example. Our *libclang*-based analyzer then processes this file to generate the complete list of library functions (plus name spaces, classes and types). During this step, the analyzer consults a *filter file* for functions to ignore.

The generated list of functions is then used to create an example application, which contains a call to each of these functions. If linking this application to the target library fails, there are wrapper functions that do not have an original function in the target library. For example this happens for some class constructors and inline functions.

If `make` fails, the next step is `make check`, which the next paragraph explains. If `make` passes, it creates the wrapper. The wrapper consists of up to four different wrapper libraries. One dimension is whether the wrapper is a shared or static library. The other dimension is whether the wrapper contains the code for link time or runtime wrapping. All four versions are useful depending on the application/library/system setup. If this succeeds, the user can move on and install the wrapper.

While processing the header files, there are a number of warnings and errors that can occur. For example, the wrapper warns about functions that contain ellipsis arguments, because they cannot be forwarded in C. In case a v-version (like `vprintf` is to `printf`) exists, the `LIBWRAP_ELLIPSIS_MAPPING_SYMBOLS`-variable in the makefile lets the user create a mapping so the wrapper can forward the call to the v-version via the `va_arg` argument.

In C, having an empty argument list in a function declaration means the argument list is unknown, i.e. *variadic*. In C++, on the other hand, the same syntax means it is really empty. In C, you need to use `(void)` as argument list for this. Calling a variadic function without parameters is valid C. This means valid C can trip up library wrapping, if the library developer did not use `(void)` for an empty argument list. To work around this, the makefile provides the `LIBWRAP_VARIADIC_IS_VOID_SYMBOLS`-variable, which names functions that are to be treated as having an empty argument list.

In our example, the `make`-step warns about a number of ellipsis functions, for example in the `QMessageLogger` and `QString` classes. It exits with an error message, because there is a mismatch between functions found in the headers and the symbols in the library. To find out which functions these are, we need to run `make check` and then adjust the filter.

Verify the Wrapper. Because the function list generated by the library header analysis rarely matches the symbol table in the library, for each wrapper function, `make check` generates a source file, and tries to compile and link it with and without the target library. The result is a complete list of symbols that are missing from the target library, and a list of symbols where linking works even without the target library. The latter tries to weed out functions that are not intended to be wrapped, because they are in system libraries.[4]

Using the two generated lists, the user has to adjust the filter to remove unwrappable, and perhaps some unwanted functions. This not only ensures the soundness of the wrapper, but also makes sure the user chooses the functions deliberately. Accidentally wrapping more than intended should be avoided.

After this, the user has to repeat `make` and `make check` until `make` succeeds.

Executing this step in our example first informs us that it is doing this check for over 13,000 functions, and this may take some time. Looking at the list of these functions (in the `.wrap`-file), we notice that it wraps more than just QtGui and QtWidget's components. This is because the header analysis cannot read the users intention perfectly. It initially only includes functions that it finds in files in directories specified via the `-I`-compilation-flag.[5] Thus we refine the filter from

```
INCLUDE /usr/include/x86_64-linux-gnu/qt5/*
```

to

```
INCLUDE /usr/include/x86_64-linux-gnu/qt5/QtGui/*
INCLUDE /usr/include/x86_64-linux-gnu/qt5/QtWidgets/*
```

and repeat `make check`. This yields a list of 818 missing functions, which we add to the filter. No symbols were found that exist when not linking to Qt.

Repeating `make` still fails due to a restriction in libclang with C++. If a function uses a type that is created via `typedef` or `using` in a class, our header analysis cannot always determine the fully qualified type. This case requires user intervention. In our example we can fix this by looking up the types in Qt's documentation and adding the class scopes via text replacement to the wrapper code.[6]

---

[4] Creating a source file for each function and try compiling and linking it is a common technique among configure tools. Doing this in one compile-link-step would require parsing the output of each supported compiler and version, which is not portable across compilers and linkers.

[5] Not doing this would initially always wrap everything including functions from system headers.

[6] https://github.com/score-p/scorep_libwrap_examples/blob/1564c272311d04575f988 6cd982fc611e07eb295/qt5/qtgui-and-qtwidgets/fix-type-scopes.sh.

Install the Wrapper. Once the wrapper builds, `make install` installs it. If not specified otherwise, this installs the wrapper into Score-P's installation directory.

Verify the Installed Wrapper. Invoking `make installcheck` links the example application to the link time and runtime wrapper library in the same way the user would. This step creates two executables, and prints out how to run and check the resulting Score-P measurement.

Running the example yields a profile with over 5000 calls to 251 unique Qt functions. Figure 4 shows an excerpt. Without the wrapper, compiler instrumentation would only recognize and record the `main`-function. Sampling with stack unwinding yields a more detailed call graph (e.g., it includes system and desktop system functions), but misses many function calls due to the nature of sampling, and it also cannot capture exact timing and call counts.
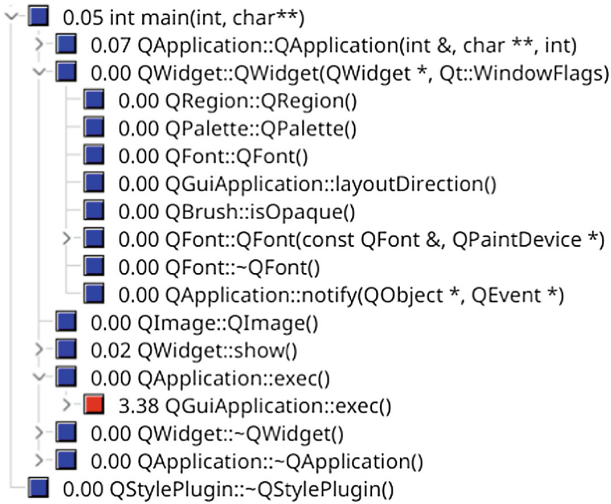
```
 0.05 int main(int, char**)
   0.07 QApplication::QApplication(int &, char **, int)
   0.00 QWidget::QWidget(QWidget *, Qt::WindowFlags)
       0.00 QRegion::QRegion()
       0.00 QPalette::QPalette()
       0.00 QFont::QFont()
       0.00 QGuiApplication::layoutDirection()
       0.00 QBrush::isOpaque()
       0.00 QFont::QFont(const QFont &, QPaintDevice *)
       0.00 QFont::~QFont()
       0.00 QApplication::notify(QObject *, QEvent *)
     0.00 QImage::QImage()
     0.02 QWidget::show()
     0.00 QApplication::exec()
         3.38 QGuiApplication::exec()
     0.00 QWidget::~QWidget()
     0.00 QApplication::~QApplication()
     0.00 QStylePlugin::~QStylePlugin()
```

Fig. 4. Partially collapsed Cube profile of the Qt example application. It accurately resembles the source code. Numbers are seconds, inclusive execution time for collapsed entries and exclusive for expanded ones.

Use the Wrapper. If the wrapper has not been installed into Score-P's installation directory (the default), the environment variable `SCOREP_LIBWRAP_PATH` (`PATH`-like) needs to point to the wrapper's path before using it.

Score-P's new `--libwrap=<wrappername>`-flag then modifies the link step to activate one or more wrappers.

To use our example Qt wrapper, we link the to-be-analyzed application according to the instructions we initially gave `scorep-libwrap-init` and simply prefix it with `scorep --libwrap=qtgui_and_qtwidgets`. I.e.:

```
$ scorep --libwrap=qtgui_and_qtwidgets g++ \
    -fPIC -I${QT_INCLUDE}                   \
    application.cc                          \
    -fPIC -lQt5Widgets -lQt5Gui -lQt5Core   \
    -o application
```

Optionally, the user can specify the wrap method by prefixing the wrapper name with either `linktime:` or `runtime:`.

Auxiliary Commands. In HPC centers, we expect support staff, not only users themselves, to install wrappers of analysis-worthy libraries alongside a Score-P installation. Users can still install wrappers into their own directories. One advantage is that the staff can update the wrappers at the same time as they update Score-P or the target libraries.

The command `scorep --help`, among other information, gives a list of installed wrappers. Users can invoke `scorep-info libwrap-summary`, with an optional wrapper name, to view wrapper configurations in greater detail.

3.3   Implementation Details

Because compile-time commands, e.g., `#ifdef`, can influence the list of declared functions, we decided to employ the user's compiler to preprocess the library's headers. To generate this list of functions, we read the header using libclang. This mismatch between preprocessor and reader can sometimes lead to problems because they might not agree on the language standard to use. Specifying the standard explicitly solves this.

During development we realized that Score-P's configured compiler cannot always link libclang to the wrapper generator. The configure step would need to know the compiler with which libclang has been created. To circumvent this, contrary to other parts of Score-P, it builds the wrapper generator using Clang, if available.

The presented approach relies on wrapping facilities offered by the linker and dynamic linker. Many C++ libraries heavily rely on inlining and templates. Wrapping libraries based on symbols being present in the target library means that this technique is unable to intercept inlined function calls.

4   Case Study

The previous section proves that our approach is robust by wrapping two Qt modules. This section demonstrates how user library wrapping benefits performance analysis for two real-world scientific applications. We repeat all measurements five times, and pick the median.

4.1   GROMACS

GROMACS [29] is a popular molecular dynamics package specialized in simulating proteins, lipids and nucleic acids. To leverage the compute power of HPC systems, GROMACS relies on MPI, OpenMP, CUDA and either FFTW 3 [17] or the Intel Math Kernel Library for discrete Fourier transforms.

For our demonstration we use GROMACS' current version 2016.3, and simulate a lysozyme in water [21] using one tenth of the default number of time steps. We run the simulation on Oak Ridge National Laboratory's Titan, a Cray XK7 supercomputer. Each node has one AMD Opteron 6274 CPU with eight Bulldozer modules and one NVIDIA Tesla K20X graphics card. We choose to run on two nodes, with four processes each. Every process spawns one additional thread—a total of 16 threads. On the software side, we load the default GNU-based environment, which uses GCC 4.9.3 and FFTW 3.3.4.11.

Executing GROMACS normally takes 330 s, of which it spends 193 in the main part, the actual simulation of the protein (*Production MD*).

To instrument GROMACS with Score-P, we replace the compilers `cc` and `CC` in the CMake-command with Score-P's compiler wrappers `scorep-cc` and `scorep-CC` and prefix the command with `SCOREP_WRAPPER=off`. Building works the same as before. Score-P then, by default, enables automatic compiler instrumentation and injects the performance monitor by modifying compile and link commands. We only use this instrumented GROMACS build on the expensive Production MD part, and execute all other parts with the normal build. Executing this increases Production MD's execution time to 375 s (+94.3%). Score-P registers 3.04 billion function calls, 2.96 billion of which are user functions. The other 80 million are OpenMP loops/calls and MPI calls. `scorep-score` estimates that a trace of this execution is 76 gigabytes (GB) large.

For technical reasons Score-P requires instrumenting MPI and OpenMP events. Therefore, a reduced recording without any user functions takes 214 s (+10.9%), contains 79 million calls, and a trace of this configuration is 3.0 GB large.

Score-P's default (automatic compiler instrumentation) adds significant overhead, and should not be used in tracing mode as is. By following Score-P's filtering workflow we can reduce the overhead and trace recording size. Alternatively we can switch off compiler instrumentation to record a very small amount of information. But none of these three options record anything about FFTW.

To track calls into FFTW, we need to create a wrapper library for it following the workflow described in Sect. 3.2. One thing that confuses our process is that Cray's compiler wrapper `cc` pulls in modules, like FFTW, automatically, if the module is loaded. Thus, to compile a program using FFTW we don't need to add compile and link flags. This is not a problem, but disarms one of our checks and makes wrapper creation slightly confusing. To circumvent this we change the environment variable `PE_PKGCONFIG_PRODUCTS` to not include `PE_FFTW`. The full instructions for building the wrapper are available online[7].

---

[7] https://github.com/score-p/scorep_libwrap_examples/tree/1564c272311d04575f988
6cd982fc611e07eb295/fftw3.

To configure GROMACS with the FFTW wrapper, run CMake with Score-P's compiler wrappers as in the previous case. Then build it using

```
SCOREP_WRAPPER_INSTRUMENTER_FLAGS="--libwrap=fftw3" make
```

instead of just `make` to enable the wrapper. With this, Production MD takes 214 s (the same as the minimum instrumentation) and counts additional 5.9 million function calls. The corresponding trace is 3.1 GB large.

By analyzing this recording, we discover that GROMACS spends the majority of time in OpenMP loops (Fig. 5(a)). FFTW occupies only about 2.4% of the execution time. Nevertheless, for the low amount of time spent in it, there are
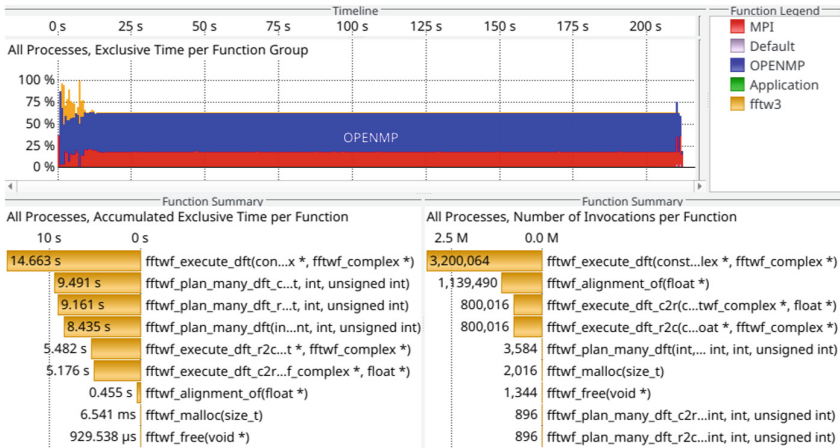
(a) Top: Overview of the whole execution. Bottom left: Accumulated (over all threads and processes) exclusive time for all FFTW functions. Bottom right: Accumulated invocation counts.
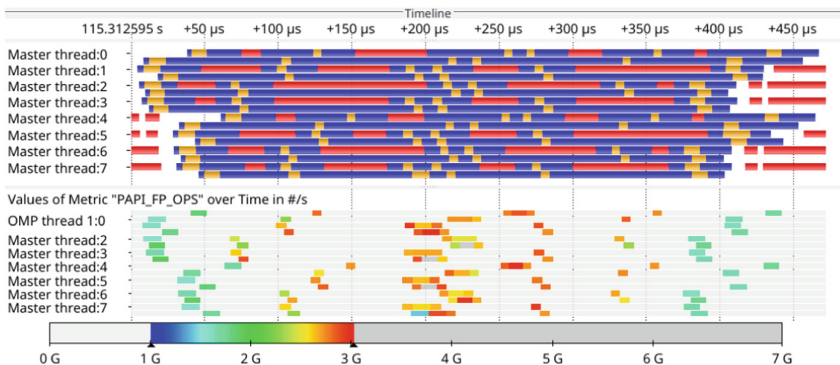
(b) Top: Currently active function active per process and thread. Orange is FFTW. Bottom: FLOPS encoded in color displayed in the same layout.

Fig. 5. Visual analysis of the trace run of GROMACS with Vampir [5] (Color figure online)

a lot of calls to FFTW. This suggests that, if possible, putting more work into one iteration should be considered. Because the vast majority of calls to FFTW take below three milliseconds, a sampling-based analysis would show a distorted picture.

With this exact instrumentation of FFTW, we can now, for example, investigate how efficiently it exploits the underlying hardware by recording performance counters. Figure 5(b) shows how FFTW's use of the floating-point unit varies between calls, and is generally subpar.

4.2   PERMON

The software package PERMON [18] solves quadratic programming problems with the help of FETI methods [15] for domain decomposition. PERMON extends PETSc [3] and is used mainly for simulating mechanical structure, for example linear elasticity, elasto-plasticity and shape optimization.

Score-P offers multiple ways to analyze the interplay of PERMON and PETSc. One is to instrument both. A second way is to analyze only PERMON and use sampling and call stack unwinding to peek into PETSc. A third approach is to intercept all function calls to PETSc by creating a wrapper for it. Score-P also supports combinations of instrumentation, sampling with call stack unwinding and library wrapping.
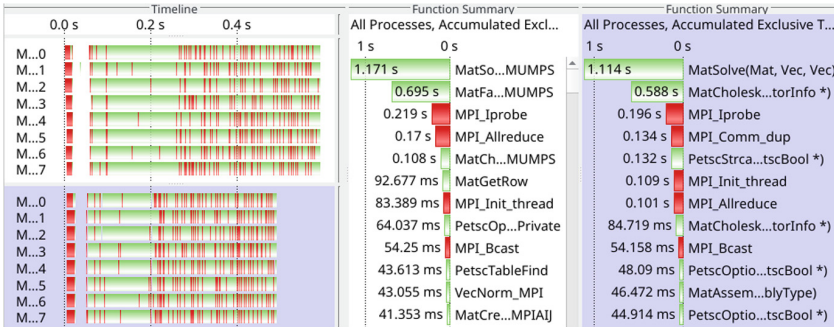

Fig. 6. Vampir Master Timeline and profile excerpt of a PERMON run using eight MPI processes. White background: instrumenting both PERMON and PETSc, blue background: user library wrapping (Color figure online)

Additionally instrumenting PETSc is cumbersome and means creating a custom installation just for measurement with Score-P. The second way is good from an ease-of-use perspective, but has drawbacks. It does not record all PETSc calls, cannot count the number of calls, and cannot record the exact timing of calls. Employing user library wrapping yields a good level of detail while alleviating the drawbacks of the other two methods.

Figure 6 shows the resulting traces from the first and third approach side-by-side. The full instrumentation creates a 112 megabyte trace, whereas the run without automatic compiler instrumentation and with library wrapping results in a 84 megabyte recording. Both recordings are similar in detail and characteristics.

5   Conclusions

In this work we present *user library wrapping*, an extension to Score-P that allows exact tracking of function calls to any C/C++ library. It enables in-depth performance analysis of applications in conjunction with their underlying libraries. Furthermore it tracks calls between libraries and offers insight into closed-source libraries like the Intel Math Kernel Library.

We offer a simple, well-crafted workflow to create and use library wrappers. This workflow guides the user through an otherwise difficult procedure, and minimizes mistakes.

Our approach differs from previous incarnations in that it supports C++, is mature, robust and well documented. It requires minimal manual work and uses modern Clang/LLVM facilities to analyze library headers.

We demonstrate its robustness for non-trivial use-cases by wrapping the QtWidgets and QtGui modules. Furthermore we show how user library wrapping enables better performance analysis for two real-world scientific applications.

6   Future Work

There are multiple interesting areas to pursue. By using compilation databases provided by CMake and GNU Autotools, we might be able to drop the requirement to specify how to build an example application in the first workflow step.

Because HPC systems install multiple versions of the same library, it would be beneficial to explicitly support versioning.

The presented approach forwards parameters from the wrapper to the target function, but does nothing with it. Extending our approach to record parameter values, for example like a performance counter, can be useful.

In order to ensure the soundness of each wrapper, the presented workflow involves a number manual checks. Technically, each wrapper needs to updated if its target library or Score-P is updated. Repeating this procedure for every update is unnecessarily burdensome. Therefore, the workflow should be extended to include automatic updating of generated wrappers.

Due to close consideration of the circumstances of header preprocessing and the symbol tables of library files, each wrapper is tied to the machine it has been created on. It should be investigated how to enable reusing the wrappers across different machines. Ultimately, a public archive of wrappers is desirable.

Score-P hinges on modifying compile and link commands to instrument an application. Some features, user library wrapping included, can be used with just link time changes. But that is not strictly necessary. By loading all of Score-P

at runtime using `LD_PRELOAD`, we could skip link command line changes, and attach the performance monitor to an unmodified binary.

Because there are many runtime analysis tools relying on library wrappers, we would like to offer our wrapper creation facility to these projects. Up until now, there was no well-supported, generic way to wrap C/C++ libraries for analysis. Developers need to create wrappers, regularly update them, and keep track of new versions of the target library.

References

1. Adhianto, L., et al.: HPCToolkit: tools for performance analysis of optimized parallel programs. Concurrency Comput.: Pract. Exp. **22**(6), 685–701 (2010)
2. Arm MAP—Arm, August 2017. https://www.arm.com/products/development-tools/hpc-tools/cross-platform/forge/map
3. Balay, S., et al.: PETSc users manual revision 3.7. Technical report, Argonne National Lab. (ANL), Argonne, IL (United States) (2016)
4. Beazley, D.M., et al.: SWIG: an easy to use tool for integrating scripting languages with C and C++. In: TCL/TK Workshop (1996)
5. Brunst, H., Weber, M.: Custom hot spot analysis of HPC software with the vampir performance tool suite. In: Cheptsov, A., Brinkmann, S., Gracia, J., Resch, M., Nagel, W. (eds.) Tools for High Performance Computing, pp. 95–114. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37349-7_7
6. Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley Professional, Boston (1997)
7. Clang: a C language family frontend for LLVM, August 2017. http://clang.llvm.org
8. Google/clif: Wrapper generator foundation to wrap C++ for Python and other languages using LLVM, August 2017. https://github.com/google/clif
9. CTool library, August 2017. http://ctool.sourceforge.net
10. CUDA zone—NVIDIA developer, August 2017. https://developer.nvidia.com/cuda-zone
11. Dietrich, R., Ilsche, T., Juckeland, G.: Non-intrusive performance analysis of parallel hardware accelerated applications on hybrid architectures. In: 2010 39th International Conference on Parallel Processing Workshops (ICPPW), pp. 135–143. IEEE (2010)
12. Edison design group, August 2017. http://edg.com
13. Eichenberger, A.E., et al.: OMPT: an OpenMP tools application programming interface for performance analysis. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 171–185. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40698-0_13
14. Extrae—BSC tools, August 2017. https://tools.bsc.es/extrae

15. Farhat, C., Roux, F.X.: A method of finite element tearing and interconnecting and its parallel solution algorithm. Int. J. Numer. Methods Eng. **32**(6), 1205–1227 (1991)
16. Folk, M., Heber, G., Koziol, Q., Pourmal, E., Robinson, D.: An overview of the HDF5 technology suite and its applications. In: Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases, pp. 36–47. ACM (2011)
17. Frigo, M., Johnson, S.G.: FFTW: an adaptive software architecture for the FFT. In: Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, vol. 3, pp. 1381–1384. IEEE (1998)
18. Hapla, V., Horak, D., Pospisil, L., Cermak, M., Vasatova, A., Sojka, R.: Solving contact mechanics problems with PERMON. In: Kozubek, T., Blaheta, R., Šístek, J., Rozložník, M., Čermák, M. (eds.) HPCSE 2015. LNCS, vol. 9611, pp. 101–115. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40361-8_7
19. Hilbrich, T., Schulz, M., de Supinski, B.R., Müller, M.S.: MUST: a scalable approach to runtime error detection in MPI programs. In: Müller, M., Resch, M., Schulz, A., Nagel, W. (eds.) Tools for High Performance Computing, pp. 53–66. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-11261-4_5
20. Knüpfer, A., et al.: Score-P: a joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In: Brunst, H., Müller, M., Nagel, W., Resch, M. (eds.) Tools for High Performance Computing, pp. 79–91. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31476-6_7
21. Lemkul, J.A.: Gromacs tutorial: Lysozyme in water, September 2017. http://www.bevanlab.biochem.vt.edu/Pages/Personal/justin/gmx-tutorials/lysozyme/index.html
22. Lofstead, J.F., Klasky, S., Schwan, K., Podhorszki, N., Jin, C.: Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In: Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments, pp. 15–24. ACM (2008)
23. de Melo, A.C.: Performance counters on Linux. In: Linux Plumbers Conference (2009)
24. Intel® math kernel library (intel® mkl)—intel® software, August 2017. https://software.intel.com/en-us/mkl
25. Message Passing Interface (MPI) forum, August 2017. http://mpi-forum.org
26. Profiler: CUDA toolkit documentation, August 2017. http://docs.nvidia.com/cuda/profiler-users-guide/index.html
27. Qt—cross-platform software development for embedded & desktop, August 2017. https://www.qt.io/
28. Shende, S., Malony, A.D., Spear, W., Schuchardt, K.: Characterizing I/O performance using the TAU performance system. In: PARCO, pp. 647–655 (2011)
29. Van Der Spoel, D., Lindahl, E., Hess, B., Groenhof, G., Mark, A.E., Berendsen, H.J.: GROMACS: fast, flexible, and free. J. Comput. Chem. **26**(16), 1701–1718 (2005)
30. Intel® VTune^{TM} Amplifier, August 2017. https://software.intel.com/en-us/intel-vtune-amplifier-xe