A Generic and Configurable Source-Code Instrumentation Component

Markus Geimer[1], Sameer S. Shende[2], Allen D. Malony[2], and Felix Wolf[1,3]

[1] Jülich Supercomputing Centre
Forschungszentrum Jülich, Germany
{m.geimer,f.wolf}@fz-juelich.de
[2] Department of Computer and Information Science
University of Oregon, Eugene, OR, USA
{sameer,malony}@cs.uoregon.edu
[3] Department of Computer Science
RWTH Aachen University, Germany

Abstract. A common prerequisite for a number of debugging and performance-analysis techniques is the injection of auxiliary program code into the application under investigation, a process called *instrumentation*. To accomplish this task, source-code preprocessors are often used. Unfortunately, existing preprocessing tools either focus only on a very specific aspect or use hard-coded commands for instrumentation. In this paper, we examine which basic constructs are required to specify a user-defined routine entry/exit instrumentation. This analysis serves as a basis for a generic instrumentation component working on the source-code level where the instructions to be inserted can be flexibly configured. We evaluate the identified constructs with our prototypical implementation and show that these are sufficient to fulfill the needs of a number of todays' performance-analysis tools.

1 Introduction

As a prerequisite for various performance-analysis and debugging techniques, it is often necessary to insert additional code fragments into the application that is currently under investigation, e.g., to validate parameters given to a function call, read hardware counter values such as the number of cache misses, or query the system clock to calculate the time spent in a certain code region. This process of adding extra code to be executed at runtime is called *instrumentation* and can be accomplished in a number of different ways.

A well-accepted technique of instrumenting an application is the so-called *source-code instrumentation* method, which is the subject matter of this paper. With this approach, additional code fragments such as function calls are directly inserted into the application's source code at appropriate places before compilation. Although this can be done manually by the developer—being quite time-consuming and error-prone—it is generally more convenient to perform this step automatically using a source-code preprocessor. Since instrumentation is entirely performed on the source-code level, its granularity can be flexibly controlled and is even not restricted to functions, but can also be done, e.g., for program phases, basic blocks, loops or individual statements. In

addition, correlating analysis results gained from such an instrumentation with locations in the source code is trivial. And finally, this approach is platform-independent as a source-code preprocessor can be implemented in a very portable way.

Unfortunately, to the authors' knowledge, none of the source-code instrumentation tools available today is flexible enough to satisfy the need of the tool developer community for a generic instrumentation component, since the commands to be inserted into the application's source are typically hard-coded for a particular purpose or toolset. To overcome this situation, this paper investigates the general requirements for such a configurable source-code instrumentor. As a starting point, our initial focus is primarily on instrumenting routine entries and exits, a feature which is commonly needed by performance-analysis and debugging tools. Not to start entirely from scratch, our prototypical implementation used to evaluate the identified constructs is based on the instrumentor of the TAU performance-analysis framework [1].

The remainder of this paper is structured as follows: after a review of related work in Section 2, we summarize the architecture of the aforementioned TAU instrumentor in Section 3. Section 4 then discusses the requirements of a configurable instrumentation component and the basic constructs that we identified as "building blocks" for user-defined instrumentation. Next, Section 5 evaluates the presented configuration concepts by mapping the manual instrumentation API of various performance-analysis toolsets onto the generic constructs, before we conclude the paper and outline directions of future work in Section 6.

2   Related Work

A simple way of inserting instrumentation code into an application is specified by the Message Passing Interface (MPI) standard [2]. Here, all library calls also exist with a second entry point name using the PMPI_ prefix, allowing a user or tool developer to provide an interposed wrapper library intercepting MPI_ calls issued by the user code. However, this approach can only capture the behavior of the instrumented MPI routines and has to be used in conjunction with one or more of the techniques described below to also gain insights into the computational core of the application.

Somewhat similar interfaces for instrumenting communication-related events are provided by the PERUSE MPI extension [3] as well as the GASP performance-analysis tool interface [4] targeting partitioned global address space (PGAS) languages. In both cases, the user of these interfaces is given the possibility to register callback functions for events of interest. Although providing very detailed information about the internals of the communication, pure user functions are still ignored.

A complementary approach applicable to user code is to leverage the capability provided by many of todays' compilers to automatically instrument the entry and exit points of functions. Although this sounds like a convenient way to instrument user code, this approach has several drawbacks. First, this feature sometimes has to rely on undocumented or unsupported compiler functionality (e.g., for the IBM xl compilers). Second, it is absolutely compiler-dependent whether instrumentation is performed before or after code optimization, i.e., the granularity of the results may differ significantly when switching between compilers. And third, the user has only very limited control over

what is instrumented. Enabling or disabling the instrumentation on a per-file level is of course always possible, but control on the function level is only supported by few compilers, typically using relatively inconvenient command-line interfaces [5].

In contrast to compiler-based instrumentation, the binary instrumentation technique [6,7] inserts measurement calls after the program's binary code is generated. In this case, the additional instrumentation code is injected either at runtime by patching the application's binary code in memory, or through rewriting the application executable prior to execution. However, this low-level technique is very architecture- and compiler-dependent, which restricts its applicability to the supported set of platforms/compilers. In addition, it suffers from a non-negligible runtime overhead, since calls to the inserted instrumentation code typically cannot be performed directly but have to go through some sort of indirection (e.g., using so-called trampolines). Nonetheless, this technique is the only choice if the application's source code is not available.

As an example of a source-code preprocessing tool, OPARI [8] specifically focuses on instrumenting OpenMP directives, requiring it to be used in conjunction with some other technique to instrument user functions. This could be done, e.g., using the afore-mentioned TAU source-code instrumentor, which forms the basis of our prototypical implementation and will therefore be covered in more detail in the next section.

An alternative framework that can be used to write source-to-source translation tools is ROSE [9]. Although being very powerful through the ability of regenerating source code after modifying the abstract syntax tree im memory, ROSE is currently only distributed for x86 and x86-64 architectures, limiting the portability of tools written on top of it.

3 TAU Source-Code Instrumentor Overview

Altering the source code of an application by a preprocessor before it is passed to the compiler typically involves parsing the source code to infer the locations of potential instrumentation points. To relieve developers of source-to-source translators from the burden of writing their own parsers and to support the development of such tools, the TAU project has developed the Program Database Toolkit (PDT) [10]. As depicted in Figure 1, PDT consists of several components that are used in different steps of the instrumentation workflow described below.

The first step is to parse the source-code files using commercial-grade compiler front-ends which build an internal representation in form of an abstract syntax tree and write this information to an intermediate language (IL) file. Next, IL analyzers walk the abstract syntax tree stored in the IL file and extract a reasonable subset of the syntactic entities, storing the result in a program database (PDB) ASCII text file. PDB files provide information such as the list of all input files read, a list of all routines including the source-code locations of their declaration and definition, and a list of all statements for each routine, again providing their source-code locations. To simplify tool development, PDT also provides a C++ class library (DUCTAPE) as a convenient interface to access the PDB data.

The TAU source-code instrumentor, built on top of the DUCTAPE library, first reads the generated PDB file, analyzes the contained syntactic information and generates a list
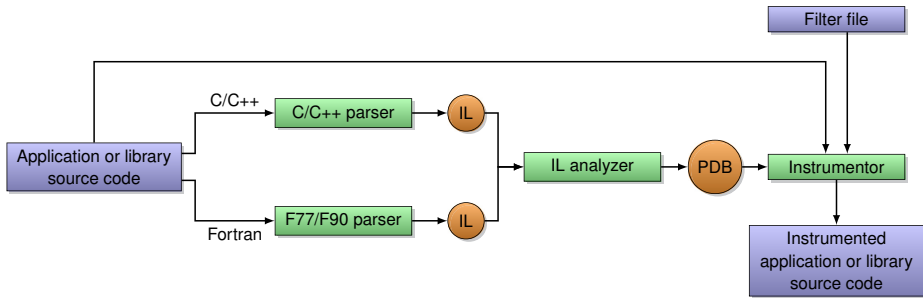
Fig. 1. Overview of the TAU instrumentor workflow. The source code is first processed by a parser front-end for the corresponding programming language, generating an intermediate language (IL) file. This file is then converted by an IL analyzer into an program database (PDB) file. The instrumentor itself then reads the PDB file, the application or library source code as well as a filter file and generates the modified, instrumented source.

of *instrumentation requests*. At this point, filter rules specified in a configuration file given to the instrumentor are applied to selectively enable or disable instrumentation for certain code regions (e.g., functions or loops). Finally, the original application or library source code is read line by line and augmented with calls to the TAU measurement library according to the instrumentation requests that remain after filtering.

The TAU source-code instrumentor currently works with C, C++, and Fortran. It is robust, can process very large source files, and is able to instrument routines, methods, and loops. However, the instrumentor only generates TAU measurement code. A general approach would allow any measurement library to be used and would be applicable to other languages such as Java or the emerging HPCS languages X10, Chapel and Fortress, provided that suitable parser front-ends are available.

4   Requirements for a Configurable Instrumentation Component

To generalize the instrumentor to be used with performance-analysis or debugging tools other than TAU, the hard-coded insertion of calls to the TAU measurement API has to be replaced by the injection of arbitrary code fragments that can be specified by the user (i.e., typically the developer of the corresponding tool). In this context, the most important questions to be considered are:

– What are the basic constructs needed to specify a user-defined instrumentation?
– Which additional information available at instrumentation time might be useful?
– How can this information be referenced in the users' code fragments?

The following subsections examine these questions in more detail and present our current solution. They are structured based on the basic constructs that we have identified as the "building blocks" for user-defined instrumentation. All examples are given in the syntax used by our current prototype implementation based on the TAU instrumentor.

4.1   Entering a Routine

One important point where tool developers typically want to insert extra instrumentation code is at the begin of functions, to be notified when the routine is entered. For this type of instrumentation request, the `entry` construct is provided:

```
entry file="*" routine="#" code="printf(\"Entering\\n\");"
```

To make this construct generic and flexible, both file and routine names can be specified using wildcards. Note that we are using the hash character (#) as a wildcard for routine names, since the asterisk (*) can be part of the function signature in C/C++ and we wanted to avoid introducing an escape character for routine names. In addition, since such "catch all" rules as shown in the example are commonly used, we decided to make them the default behavior, allowing the user to omit the `file` and `routine` parts of the specification line. For the specified code fragment, we adopted the standard C syntax to quote special characters within strings using the backslash character, also supporting line breaks (\n) and tabulators (\t). This allows for inserting multiple code lines with a single specification rule. Alternatively, several `entry` clauses for the same file/line combination can be given since their code fragments will be concatenated in the order of appearance in the specification file, separated by a line break. Of course, all filter rules defined in a filter file given to the instrumentor still apply, allowing generic specification rules which are then not used for certain files or routines.

To leverage "instrumentor knowledge" in the code snippets to be inserted, a number of textual substitutions are performed just before they are written to the output file. For example, the keyword `@FILE@` will be replaced by the name of the input file and `@LINE@` by the line number in the original source file at which the code is inserted. Although this information is in principle also available through the C preprocessor macros `__FILE__` and `__LINE__`, using these macros will usually insert the wrong values since the file that is actually being compiled will be the instrumented source file with a temporary name as well as displaced code lines due to the instrumentation. Theoretically, this can be corrected using `#line` directives, however, adding them correctly is non-trivial. Moreover, the `__FILE__` and `__LINE__` macros cannot be used inside of strings.

Besides file and line number information, the instrumentor can also provide the name of the routine (i.e., the full signature) as well as the line and column of both begin and end of the function body. Table 1 provides a full list of all keyword substitutions that we deemed useful and that are currently supported.

4.2   Leaving a Routine

Similar to the point of entering a routine, the location where the routine is left is another important point to insert instrumentation code. This applies to the end of the function body as well as to every intermittent `return` statement. For this purpose, the `exit` construct is provided:

```
exit file="*" routine="#" code="printf(\"Leaving\\n\");"
```

Again, the same wildcard, quoting and keyword substitution rules as described in the context of the `entry` construct (Sec. 4.1) apply. Note that it is possible to distinguish

Table 1. Keyword substitutions performed while inserting user-defined code fragments

Keyword	Substitution
All constructs:	
`@FILE@`	File name
`@LINE@`	Source line of insertion
`@COL@`	Column of insertion
`decl, init, entry, exit, abort` only:	
`@ROUTINE@`	Routine name
`@BEGIN_LINE@`	Begin line of routine body
`@BEGIN_COL@`	Begin column of routine body
`@END_LINE@`	End line of routine body
`@END_COL@`	End column of routine body
`decl, entry, exit, abort` only (C++):	
`@RTTI@`	Dynamic routine name (class/member function templates)
`init` only (C/C++):	
`@ARGC@`	Name of first paramater to `main()`
`@ARGV@`	Name of second parameter to `main()`

different `return` statements of a routine using the `@LINE@` keyword substitution, which might be handy for debugging purposes.

For C and C++, the expression after the `return` keyword defining the return value can be arbitrarily complex. To insert the `exit` code fragment as late as possible (e.g., for accurate time measurements), the source-code needs to be slightly rewritten. First, the result of the return expression is assigned to a local variable. Next, the given `exit` code snippet is inserted and finally, the `return` statement returning the value of the aforementioned local variable is generated. Note that replacing a single-line expression with multiple lines of code might require the creation of a new {...} block in C/C++ or modifying the surrounding `if` statement in Fortran.

4.3 Variable Declarations

The code fragments specified by a user to instrument routine entries and exits might require the declaration of local variables. For C and C++, this does not seem to be an issue since new variables can either be declared at any position in the code (C99/C++) or at the beginning of a new block (C89), which could be opened as part of an `entry` construct's code fragment. However, this approach would require to close the block at the end of the function body, which cannot be accomplished using a simple `exit` construct as this is also applied to intermittent `return` statements. In addition, Fortran requires the declarations of local variables to precede the first executable statement. It therefore seems reasonable to provide a separate `decl` construct to specify local variable declarations:

```
decl file="*" routine="#" code="static int count = 0;"
```

Depending on the purpose of the instrumentation, initializing such a variable with the result of a function call should be avoided, since this would be executed *before* any code fragment specified via an `entry` construct.

4.4 Inclusion of Header Files

Inserting calls to a performance-measurement or debugging API into the source code
of an application typically also requires including one or more header files defining the
corresponding function prototypes. Fortunately, the TAU instrumentor already provides
a mechanism which can be exploited to accomplish this task: using a special file rule,
some arbitrary code fragment can be inserted at a particular line in the specified source
file. For example, the specification line

```
file="*" line=1 code="#include <stdio.h>"
```

can be used to include the header file "stdio.h" at the top of every processed source file.

4.5 Aborting the Application

Other interesting locations where the insertion of, e.g., clean-up code might be useful
are calls to the exit() or abort() functions in C/C++ or the occurrences of the stop
keyword in Fortran. For this purpose, the abort construct is provided:

```
abort file="*" routine="#" code="printf(\"Abort\\n\");"
```

As already described in Sec. 4.2 in the context of the exit construct, the keyword
substitutions can be used to distinguish different abort locations from each other.

4.6 Initialization

Finally, a tool library might need to be initialized before any other API call is executed.
For C and C++ this could be accomplished by providing an entry rule restricted to
the function main(), however, for Fortran the name of a program can be arbitrary.
Therefore, a separate init construct is necessary:

```
init file="*" code="init_api();"
```

This construct does not need a routine part, as it implicitly applies to main() in C/C++
or the main program routine in Fortran, respectively.

   As a tool library might want to parse the command line arguments given to the in-
strumented application, e.g., to configure a measurement run, two special keyword sub-
stitutions have been implemented for the init construct, although for C and C++ only.
In this case, the names of the first and second parameter of main() are substituted for
the @ARGC@ and @ARGV@ keywords, respectively. If main() has been defined without
arguments, the names of two artificially created local variables are inserted, providing
the values "1" and "unknown".

4.7 Restricting Rules to a Language

Although it is possible to create separate instrumentation specification files for each
supported programming language, we believe that it is more convenient to keep every-
thing together in a single file. All of the aforementioned constructs therefore support an
optional lang="..." part taking a comma-separated list of language names, restricting
the corresponding specification clause to only those languages.

4.8   Language Peculiarities: Line-Length Limit, Templates and Exceptions

Due to the keyword substitutions performed at instrumentation time, the actual lengths of the code fragments to be inserted are not known in advance. This poses a problem in the context of instrumenting Fortran codes, since the maximum length of an individual source line is restricted by the language standard. It is therefore necessary to preprocess the code snippets before inserting them into the output file and eventually introduce additional line breaks and continuation marks, taking into account whether fixed-format or free-format style is used.

Another challenging language feature are C++ templates. At instrumentation time, the `@ROUTINE@` keyword substitution can only provide the generic template prototype, but not the concrete instantiation. If it is a class or member-function template, however, this information can be queried using the run-time type information system (RTTI). As this constraint can be verified at instrumentation time, an additional keyword substitution for `@RTTI@` can be performed, which either expands to `typeid(*this).name()` in case of a class or member-function template or the generic template prototype (i.e., the same value as for `@ROUTINE@`) otherwise. However, the value of the `typeid` expression is compiler-dependent and might be a linker decorated name, which has to be taken into account when using this feature.

Finally, a source-code instrumentor can handle C++ exceptions only to a certain extend, since this is a highly dynamic language feature. Although `throw` statements could be instrumented similar to `return` statements, they do not necessarily leave only the current routine, but all routines up to the next matching `catch` block. However, tools can leverage destructors of local objects [11] to get a correct sequence of exit events.

5   Evaluation

To evaluate whether the proposed specification clauses presented in the previous section are already sufficient to satisfy the needs of current tools to perform a simple per-routine entry/exit instrumentation, we have implemented them in our prototype based on the TAU instrumentor, except for the language-specific features described in Section 4.8. Afterwards, we have developed a set of specification files for a number of performance-analysis toolsets using their manual instrumentation API and verified their correct mode of operation by applying the instrumentor to various test codes.

Our first target was the Scalasca toolset [12]. As the documented user instrumentation API is basically a set of convenience C preprocessor macros heavily using the predefined names `__FILE__` and `__LINE__`, we had to use the lower-level routines these macros are build upon. For all three supported languages, a header file defining the API had to be included. In addition, instrumenting C++ code only required a single `entry` construct due to the availability of a measurement class employing the aforementioned "local object" technique. By contrast, instrumenting C code required the `entry` and `exit` constructs, as shown in the following self-contained example:

```
file="*" line=1 code="#include <epik_user.h>"
entry code="EPIK_User_start(\"@ROUTINE@\", \"@FILE@\", @BEGIN_LINE@);"
exit code="EPIK_User_end(\"@ROUTINE@\", \"@FILE@\", @END_LINE@);"
```

For Fortran, an additional local variable needed to be declared, i.e., the `decl` construct had to be used as well. In all three cases, the @ROUTINE@, @FILE@ and @LINE@ keyword substitutions were sufficient to fully exploit the current functionality of the provided instrumentation API.

As a second example, we investigated the VampirTrace performance measurement system [13]. Providing an instrumentation API very similar to Scalasca, it was straightforward to come up with a specification file using the same constructs. For both toolsets, the instrumentor could in fact provide more details about source-code locations than necessary, indicating potential for extending the tool APIs to collect even more expressive information.

A far more challenging problem was to clone the TAU instrumentation originally performed by the instrumentor using the generic specifications. It turned out that all of the constructs described in Sec. 4 are needed to fulfill this task. However, two minor issues still remained were the original TAU instrumentor behaved differently.

First, TAU supports so-called *profile groups* as a mechanism to further classify sets of functions. The default behavior of the instrumentor for C and C++ codes is to add the program's main function to the group TAU_DEFAULT and all the other functions to the group TAU_USER. This behavior could be partially emulated by specifying a separate `entry` rule restricted to the `main()` function, however, there is currently no way of restricting a clause to every function except `main()`. This issue could potentially be solved by supporting full regular expressions in the `routine` part of the specification rules.

The second issue will show up once we have fully implemented template support in our prototype as proposed in Sec. 4.8 because the @RTTI@ keyword substitution has slightly different semantics than what the TAU measurement system currently assumes. Here, a minor change to the measurement system API would be required, however, this could be implemented as an extension not to break backward compatibility.

6   Conclusion

In this paper, we have investigated which basic constructs are required to specify a user-defined function entry/exit instrumentation in a generic way. We identified six different constructs as the "building blocks" that can be applied to individual files, routines or programming languages, as well as a set of keyword substitutions to take advantage of instrumentor knowledge at instrumentation time. We then evaluated the applicability of the proposed constructs by defining appropriate specification files for three different performance-analysis toolsets and showed that this small set of constructs can already fulfill almost all the needs of a number of todays' tools with respect to routine enter/exit instrumentation. Our prototypical implementation supporting all described constructs except for the language-specific features described in Section 4.8 is available as part of the PDT distribution.

As part of our future work, we plan to first address the open language-specific issues mentioned in Section 4.8. In addition, we will investigate how the configurability can be extended beyond the current enter/exit instrumentation, e.g., to support instrumenting `throw` statements as well as `try` and `catch` blocks, loops, or specially marked program

phases such as OpenMP regions. As a result, the configurable source-code instrumentor component described in this paper should ultimately be able to replace the existing special-purpose instrumentators currently used by various toolsets.

References

1. Shende, S.S., Malony, A.D.: The TAU parallel performance system. International Journal of High Performance Computing Applications 20(2), 287–331 (Summer 2006)
2. MPI Forum: MPI – A Message-Passing Interface Standard, Version 2.1. ch.14 (June 2008)
3. MPI PERUSE: An MPI extension for revealing unexposed implementation information (May 2006), http://www.mpi-peruse.org
4. Leko, A., Bonachea, D., Su, H.H., George, A.D.: GASP: A performance analysis tool interface for global address space programming models, specification version 1.5. Technical Report LBNL-61606, Lawrence Berkeley National Lab (September 2006)
5. Free Software Foundation: GCC 4.3.2 manual – options for code generation conventions (2008), http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/Code-Gen-Options.html
6. Buck, B., Hollingsworth, J.K.: An API for runtime code patching. Journal of High Performance Computing Applications 14(4), 317–329 (2000)
7. De Rose, L., Hoover Jr., T., Hollingsworth, J.K.: The dynamic probe class library – an infrastructure for developing instrumentation for performance tools. In: Proc. 15th International Parallel & Distributed Processing Symposium (IPDPS 2001), Washington, DC. IEEE Computer Society, Los Alamitos (2001)
8. Mohr, B., Malony, A.D., Shende, S., Wolf, F.: Design and prototype of a performance tool interface for OpenMP. The Journal of Supercomputing 23, 105–128 (2002)
9. Schordan, M., Quinlan, D.: A source-to-source architecture for user-defined optimizations. In: Böszörményi, L., Schojer, P. (eds.) JMLC 2003. LNCS, vol. 2789, pp. 214–223. Springer, Heidelberg (2003)
10. Lindlan, K.A., Cuny, J., Malony, A.D., Shende, S., Mohr, B., Rivenburgh, R.: A tool framework for static and dynamic analysis of object-oriented software with templates. In: Proc. SC 2000: High Performance Networking and Computing Conference (November 2000)
11. Meyers, S.: More Effective C++. Addison-Wesley, Reading (1996) (Item 9)
12. Geimer, M., Wolf, F., Wylie, B.J.N., Mohr, B.: Scalable paralle trace-based performance analysis. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 303–312. Springer, Heidelberg (2006)
13. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir performance analysis tool set. In: Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A. (eds.) Tools for High Performance Computing, pp. 139–155. Springer, Heidelberg (2008)