Improving the flexibility of the TAU instrumentor

2008-08-08 |

Markus Geimer
Jülich Supercomputing Centre

m.geimer@fz-juelich.de

Introduction

Performance analysis typically depends on *instrumentation*

- Insertion of additional code into the application
  - *Query timers*
  - *Collect hardware counter values*
  - *Record other events*
- Usually the first step

Can be done in a number of different ways

Instrumentation techniques

Binary instrumentation

- ☺ Works directly on binaries – no source/recompilation required
- ☹ Hard to set up
- ☹ Limited number of supported platforms/compilers

Compiler-based instrumentation

- ☺ Easy to use
- ☹ Recompilation required
- ☹ “All-or-nothing” approach – filtering at runtime
- ☹ Sometimes disables (certain) optimizations

Instrumentation techniques (cont.)

Source-code instrumentation

- ☺ Portable
- ☺ Flexible filtering possible – at no runtime cost
- ☺ Only way to capture “high-level” user abstractions
- ☹ Recompilation required
- ☹ Source-to-source translation tool hard to implement

TAU instrumentor

Source-to-source translation tool

- Supports C, C++, Fortran
- Provides instrumentation of higher-level abstractions
  - *Loops*
  - *Phases*
  - *...*
- Provides flexible filtering mechanisms
  - *File/routine based*
  - *Include/exclude lists*

However, it only generates instrumentation for the TAU API...

Motivation

There is a common need for a good source-code instrumentor

- Performance tools
  - *Scalasca, VampirTrace, PerfLib, ...*
- Debugging
  - *To insert* `printf()`*'s or additional checks*

Long-term vision

- Make it a generic, configurable, stand-alone component!
  - *Remove all the hard-coded TAU stuff*
  - *Use instrumentation code from a configuration file instead*
- Distribute it separately from TAU (e.g., as part of PDT)

Previous work

TAU instrumentor already provided two features towards more flexibility

- PerfLib instrumentation (“-p” option)
  - *Disables all TAU instrumentation*
  - *Inserts PerfLib API calls when entering/leaving a routine*
- Insertion of user-specified code
  (through selective instrumentation file, “-f” option)
  - *At a specific source-code location*
  - *At routine entry/exit*

Well... at least that's what they were intended to do... ☺

Extended “file” construct

```
file=”<file>”  line=<line>  code=”<code>”  [lang=”<lang>”]
```

- Inserts the given <code> at <line> in <file>
  - *Files can be specified using wildcards*
- ☑ New, optional “lang” token restricts construct to the given language(s)
  - *Comma-separated list of languages*
  - *Recognized: c, c++, fortran*

Extended “entry” construct

```
entry  [file=”<file>”] [routine=”<r>”] code=”<code>”
       [lang=”<lang>”]
```

- Inserts the given <code> at the beginning of the code section of routine <r> in <file>
  - *Files can be specified using wildcards*
  - *Routines can be specified using wildcards*

☑ Specifying a routine name is now optional (default: “#”)

☑ Construct is subject to file/routine filter rules

☑ New, optional “lang” token restricts construct to the given language(s)

  - *Comma-separated list of languages*
  - *Recognized: c, c++, fortran*

Extended “exit” construct

```
exit  [file=”<file>”] [routine=”<r>”] code=”<code>”
      [lang=”<lang>”]
```

- Inserts the given <code> at each exit of routine <r> in <file>
  - *Files can be specified using wildcards*
  - *Routines can be specified using wildcards*

☑ Specifying a routine is now optional (default: “#”)

☑ Construct is subject to file/routine filter rules

☑ New, optional “lang” token restricts construct to the given language(s)

  - *Comma-separated list of languages*
  - *Recognized: c, c++, fortran*

New “init” construct

```
init  code=”<code>”  [lang=”<lang>”]
```

- ☑ Inserts the given <code> at the beginning of `main()` for C/C++ or a program in Fortran
- ☑ Construct is subject to file/routine filter rules
- ☑ Optional “lang” token restricts construct to the given language(s)

New “decl” construct

```
decl  [file=”<file>”] [routine=”<r>”] code=”<code>”
      [lang=”<lang>”]
```

- ☑ Adds the given <code> to the declaration section of routine <r> in <file>
  - *Files can be specified using wildcards*
  - *Routines can be specified using wildcards*
- ☑ Construct is subject to file/routine filter rules
- ☑ Optional “lang” token restricts construct to the given language(s)

Code substitutions

To use “instrumentor knowledge” in the inserted code snippets, some substitutions are performed:

- ☑ file, decl, init, entry, exit

```
@FILE@          →   Filename
@LINE@          →   Insertion line
@COL@           →   Insertion column
```

- ☑ decl, init, entry, exit

```
@ROUTINE@       →   Routine name
@BEGIN_LINE@    →   Line   of routine begin
@BEGIN_COL@     →   Column of routine begin
@END_LINE@      →   Line   of routine end
@END_COL@       →   Column of routine end
```

- ☑ init

```
@ARGC@          →   First  argument to main()
@ARGV@          →   Second argument to main()
```

Modified TAU instrumentor

The existing TAU instrumentor had to be modified as well

- ☑ New “-spec <specfile>” command line option
  - *Reads configuration from <specfile>*
  - *Format identical to selective instrumentation file*
  - *Suppresses generation of TAU API instrumentation*
- ☑ New/enhanced constructs should also work in “TAU mode” (almost...)

Putting it all together: Example

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

    init  code="printf(\"Argument count: %d\\n\", @ARGC@);"

    decl  code="static int ex_count=0;"
    entry code="printf(\"Entered @ROUTINE@ [%d]\\n\", ex_count++);"
    exit  code="printf(\"Leaving @ROUTINE@ at line @LINE@\\n\");"
END_INSTRUMENT_SECTION
```

Example (cont.)

```
int foo(int value) {
    if (value > 1)
        return 1;
    else
        return 0;
}

int main(int count, char** args) {
    int i;

    for (i=0; i<4; ++i)
        foo(i);

    return 0;
}
```

```
#include <stdio.h>
int foo(int value) {
    int tau_ret_val;
    static int ex_count=0;

    printf("Entered int foo(int) C"
           " [%d]\n", ex_count++);
{
    if (value > 1) {
      tau_ret_val=1;
      printf("Leaving int foo(int) C"
             " at line 3\n";
      return (tau_ret_val);
    } else {
      tau_ret_val=0;
      printf("Leaving int foo(int) C"
             " at line 5\n");
      return (tau_ret_val);
    }
}
    printf("Leaving int foo(int) C"
           " at line 6\n");
}
...
```

Example (cont.)

```
$ ./example 2 3 4
Argument count: 4
Entered int main(int, char **) C [0]
Entered int foo(int) C [0]
Leaving int foo(int) C at line 5
Entered int foo(int) C [1]
Leaving int foo(int) C at line 5
Entered int foo(int) C [2]
Leaving int foo(int) C at line 3
Entered int foo(int) C [3]
Leaving int foo(int) C at line 3
Leaving int main(int char **) C at line 15
$
```

Limitations

New/enhanced constructs do not work for C++ in “TAU mode”

- It's a bug (and not a feature)...

Can “-spec mode” replace the hard-coded TAU instrumentation?

- Not yet...
  - `stop` *(Fortran) and* `exit`/`abort` *(C/C++) are not instrumented*
  - *Inserted code can exceed line length limit in Fortran*

    No automatic adjustment/workaround at this point
  - *No support for loops & phases*
  - *No support for templates*
  - *Profile groups can only be hard-coded in spec file*
- But it already works to a certain degree... ☺

Example: TAU instrumentation using C API

```
BEGIN_INSTRUMENT_SECTION
    file="*" line=1 code="#include <Profile/Profiler.h>"

    decl code="TAU_PROFILE_DECLARE_TIMER(tautimer);"

    entry code="TAU_PROFILE_CREATE_TIMER(tautimer,
                \"@ROUTINE@ [{@FILE@} {@BEGIN_LINE@,@BEGIN_COL@}-
                 {@END_LINE@,@END_COL@}]\", \" \", TAU_DEFAULT);"
    entry code="TAU_PROFILE_START(tautimer);"

    init code="TAU_INIT(&@ARGC@, &@ARGV@);"
    init code="#ifndef TAU_MPI"
    init code="#ifndef TAU_SHMEM"
    init code="  TAU_PROFILE_SET_NODE(0);"
    init code="#endif /* TAU_SHMEM */"
    init code="#endif /* TAU_MPI */"

    exit code="TAU_PROFILE_STOP(tautimer);"
END_INSTRUMENT_SECTION
```

Outlook

Address critical issues

- Fix “TAU mode” C++ bug
- Address line-length limit issue in Fortran
- Add template support
- Add support for instrumentation of header files

Implement extensions for missing features

- `stop/abort/exit`
- Loops, phases & profile groups

Wish list

- Refactor the code, i.e., make it more object-oriented

JÜLICH
FORSCHUNGSZENTRUM

Thank you!