Configurable instrumentation components and their use by Scalasca

2010-08-02  Markus Geimer
Jülich Supercomputing Centre
m.geimer@fz-juelich.de
Source-code instrumentation

- Generic source-code analysis frameworks
  - Program Database Toolkit (PDT)
  - ROSE
- Special-purpose source-code instrumenters
  - OPARI (OpenMP)
  - TAU instrumentor
Source-code instrumentation

- Generic source-code analysis frameworks
  - Program Database Toolkit (PDT)
  - ROSE
- Special-purpose source-code instrumenters
  - OPARI (OpenMP)
  - TAU instrumentor

**Conclusion I**

No configurable source-code instrumenter available.
Source-code instrumentation

- Generic source-code analysis frameworks
  - Program Database Toolkit (PDT)
  - ROSE
- Special-purpose source-code instrumenters
  - OPARI (OpenMP)
  - TAU instrumentor

Conclusion I

No configurable source-code instrumenter available.

Conclusion II

Take the initiative and create one!

- Based on the TAU instrumentor
- Developed in collaboration with UOregon
TAU source-code instrumentor

- Based on Program Database Toolkit (PDT)
  - Uses commercial-grade compiler frontends
  - Creates a database of source-code entities
  - Provides a C++ library to access this data
- Pros
  - Robust, well tested
  - Works for C, C++, Fortran
  - Able to instrument routines, methods, and loops
  - Provides extensive filtering capabilities
- Cons
  - Only inserts instrumentation code for the TAU Performance System
TAU instrumentor workflow

- Application or library source code
  - C/C++ parser
  - F77/F90 parser
- IL analyzer
- PDB
TAU instrumentor workflow

Application or library source code

C/C++ parser

F77/F90 parser

IL analyzer

Instrumentor

Instrumented application or library source code

PDB
TAU instrumentor workflow

- Application or library source code
  - C/C++ parser
  - IL analyzer
  - IL
  - F77/F90 parser
- Filter file
  - Instrumentor
  - Instrumented application or library source code
  - PDB
TAU instrumentor workflow

Application or library source code

C/C++

C/C++ parser

IL

F77/F90 parser

IL

IL analyzer

Filter file

Instrumentor

Specification file

Instrumented application or library source code

PDB
“Building blocks” for user-defined instrumentation

- Entering a routine
  entry file="..." routine="..." code="..."

- Leaving a routine
  exit file="..." routine="..." code="..."

- Insert arbitrary code (e.g., to include header files)
  file="..." line=... code="..."
“Building blocks” for user-defined instrumentation

- Entering a routine
  `entry file="..." routine="..." code="..."`

- Leaving a routine
  `exit file="..." routine="..." code="..."`

- Insert arbitrary code (e.g., to include header files)
  `file="..." line=... code="..."

- Declaration of local variables
  `decl file="..." routine="..." code="..."

- Aborting the application
  `abort file="..." routine="..." code="..."

- Initialization
  `init file="..." code="..."`
Wildcards

- Files and routines can be specified using wildcards
  - ‘?’ matches a single character
  - ‘*’ matches multiple characters in file names
  - ‘#’ matches multiple characters in routine names
    - Avoids escaping ‘*’ characters in pointer types of arguments and return values
- If file and/or routine clause is omitted, ‘*’ or ‘#’ is implicitly assumed
Code clauses

- Code clauses support C-style escaping of characters
  - \" Quotation mark
  - \n Newline character
  - \t Horizontal tab
  - ...

- Instrumentor knowledge can be referenced through keyword substitution
## Keyword substitution

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>All constructs:</strong></td>
<td></td>
</tr>
<tr>
<td>@FILE@</td>
<td>File name</td>
</tr>
<tr>
<td>@LINE@</td>
<td>Source line of insertion</td>
</tr>
<tr>
<td>@COL@</td>
<td>Column of insertion</td>
</tr>
<tr>
<td><strong>decl, init, entry, exit, abort only:</strong></td>
<td></td>
</tr>
<tr>
<td>@ROUTINE@</td>
<td>Routine name</td>
</tr>
<tr>
<td>@BEGIN_LINE@</td>
<td>Begin line of routine body</td>
</tr>
<tr>
<td>@BEGIN_COL@</td>
<td>Begin column of routine body</td>
</tr>
<tr>
<td>@END_LINE@</td>
<td>End line of routine body</td>
</tr>
<tr>
<td>@END_COL@</td>
<td>End column of routine body</td>
</tr>
<tr>
<td><strong>decl, entry, exit, abort only (C++):</strong></td>
<td></td>
</tr>
<tr>
<td>@RTTI@</td>
<td>Dynamic routine name (class/membe function templates)</td>
</tr>
<tr>
<td><strong>init only (C/C++):</strong></td>
<td></td>
</tr>
<tr>
<td>@ARGC@</td>
<td>Name of first parameter to <code>main()</code></td>
</tr>
<tr>
<td>@ARGV@</td>
<td>Name of second parameter to <code>main()</code></td>
</tr>
</tbody>
</table>
Example

- Print a message at each routine entry stating
  - the routine name
  - how often it has been called so far
- Do this only for routines in files with prefix `foo`
Example

- Print a message at each routine entry stating
  - the routine name
  - how often it has been called so far
- Do this only for routines in files with prefix `foo_`

Specification

decl file="foo_*" code="static int count=0;"
entry file="foo_*"
  code="printf("@ROUTINE@ called %d times\n",
     ++count);"
Language issues

- Rules often need to be restricted to a particular language
  - All rules accept an optional `lang="..."` clause
  - Argument: comma-separated list of language names
    ("c", "c++", "fortran")
Language issues

- Rules often need to be restricted to a particular language
  - All rules accept an optional `lang="..."` clause
  - Argument: comma-separated list of language names ("c","c++","fortran")

- Fortran issues
  - Line-length limit
  - Different line continuation syntax for free-/fixed-form
Language issues

- Rules often need to be restricted to a particular language
  - All rules accept an optional `lang="..."` clause
  - Argument: comma-separated list of language names
    (“c”, “c++”, “fortran”)

- Fortran issues
  - Line-length limit
  - Different line continuation syntax for free-/fixed-form

- C++ issues
  - Template support
    - Solvable for member function templates through RTTI
    - Information returned is implementation-dependent
    - For non-members, only generic template prototype available
  - Exception support
    - Needs to be (partially) handled by the user’s code
Evaluation

Usability evaluated using three different performance-analysis toolsets

- Scalasca
  - Documented user API uses macros and `__FILE__/__LINE__`
  - Lower-level API needs to be used
  - Requires `line, decl, entry and exit constructs`
Evaluation

Usability evaluated using three different performance-analysis toolsets

- **Scalasca**
  - Documented user API uses macros and `__FILE__/`__LINE__`
  - Lower-level API needs to be used
  - Requires line, decl, entry and exit constructs

- **VampirTrace**
  - API very similar to Scalasca
  - Only minor modifications required
Evaluation

Usability evaluated using three different performance-analysis toolsets

- **Scalasca**
  - Documented user API uses macros and `__FILE__/__LINE__`
  - Lower-level API needs to be used
  - Requires `line, decl, entry and exit` constructs

- **VampirTrace**
  - API very similar to Scalasca
  - Only minor modifications required

- **TAU**
  - Far more challenging
  - Use of all provided constructs required
  - Two minor differences remaining
    - Default function grouping for C/C++
    - Slightly different semantics for C++ templates
Current status

- Instrumentor available as part of the PDT distribution
- Supported by Scalasca as optional component on most platforms
  - Configure Scalasca using
    
    ```bash
    --with-pdt=<DIR>
    ```
  - Instrument your code using
    
    ```bash
    scalasca -instrument -comp=none -pdt <compile_cmd>
    ```
  - Optionally provide filter using
    
    ```bash
    -optTauSelectFile=<filter_file>
    ```
- Language-specific issues still work in progress
Lessons learned

- Writing a configurable instrumenter is possible!
  - Can leverage existing technologies
  - Keyword substitution provides enough information for existing instrumentation APIs
    - New keywords can be added if needed
- Usage by existing tool compiler wrappers is no big deal either
Lessons learned

- Writing a configurable instrumenter is possible!
  - Can leverage existing technologies
  - Keyword substitution provides enough information for existing instrumentation APIs
    - New keywords can be added if needed

- Usage by existing tool compiler wrappers is no big deal either

- However...
  - Combining code specification and definition of what to instrument does not always work
  - Example: loops
    - User: “Instrument loop 2 in routine ‘foo’”
    - Tool developer: “Use code snippet ‘...’ to instrument loops”
Binary instrumentation

- Dynamic instrumentation frameworks
  - PIN
  - Dyninst
    - Better portability
    - Also allows static binary rewriting (though x86/x86_64 only)
- Special-purpose binary instrumenters
  - P^nMPI
  - tau_pin / tau_run
Binary instrumentation

- Dynamic instrumentation frameworks
  - PIN
  - Dyninst
    - Better portability
    - Also allows static binary rewriting (though x86/x86_64 only)
- Special-purpose binary instrumenters
  - P\textsuperscript{n}MPI
  - tau\_pin / tau\_run

**Conclusion I**

No configurable binary instrumenter available.
Binary instrumentation

- Dynamic instrumentation frameworks
  - PIN
  - Dyninst
    - Better portability
    - Also allows static binary rewriting (though x86/x86_64 only)
- Special-purpose binary instrumenters
  - P^nMPI
  - tau_pin / tau_run

Conclusion I

No configurable binary instrumenter available.

Conclusion II

Take the initiative and create one!
- Based on Dyninst with support from UW Madison
Design decisions

- Focus on static binary rewriting
- Prototype new specification language
  - XML-based
  - Fully separate code and filter specifications
- Experiment with property-based filters
  - Number of instructions
  - Lines of Code
  - Cyclomatic complexity
  - Callpaths to MPI/OpenMP only
  - ...
Binary instrumenter workflow

- Application binary
- Filter file
- Specification file
- Instrumenter
- Instrumented application binary
- Measurement library
Filter file

- Specifies what to instrument
  - Functions
  - Callsites
  - Loops (as a whole / loop body)

- Allows filtering by
  - Function names
  - Class names
  - Namespaces / Fortran modules
  - Properties

- Supports black- and whitelisting
- Supports boolean operations
Example

- Instrument all functions in files with prefix `foo_`
- Use code snippet “func_inst” provided by specification file
Example

- Instrument all functions in files with prefix `foo_`
- Use code snippet “func_inst” provided by specification file

```
<?xml version="1.0" encoding="UTF-8"?>
<filter name="foo_funcs"

    instrument="functions=func_inst"
    start="none">

    <include>
    <modulenames match="prefix">foo_</modulenames>
    </include>

</filter>
```
Specification file (adapter)

- Provides named code snippets referenced from filter file
  - This is the tool specific part!
  - Uses a C-like syntax
- Allows specification of additional library dependencies
- Can contain special adapter filter to exclude, e.g., functions of a measurement library
- Supports keyword substitution
Example

- Print a message at each routine entry stating
  - the routine name and how often it has been called so far
Example

- Print a message at each routine entry stating
  - the routine name and how often it has been called so far

Specification

```xml
<?xml version="1.0" encoding="UTF-8"?>
<instrumentation>
  <dependencies>
    <library name="libc.so" />
  </dependencies>
</instrumentation>

<!-- continued on next slide -->
Example (cont.)

Specification

<!-- continued from previous slide -->

<code name="func_inst">
  <variables>
    <var name="count" type="int" size="4" />
  </variables>
  <init>
    count = 0;
  </init>
  <enter>
    count = count + 1;
    printf(@functionname@);
    printf("called %d times\n", count);
  </enter>
</code>
Current status

- Work in progress
  - Any feedback is welcome!
- Evaluation mostly using Scalasca
  - DROPS (C++)
  - Cactus benchmarks PUGH / Carpet (C++)
  - Gadget (C)
- Small proof-of-concept experiments using TAU
- Full integration into Scalasca pending
- Release as a component is planned
Acknowledgments

- Jan Mußler (JSC)
- Bernd Mohr (JSC)
- Sameer Shende (UOregon)
- Madhavi Krishnan (UW Madison)
- Drew Bernat (UW Madison)