A configurable binary instrumenter

making use of heuristics to select relevant instrumentation points

12. April 2010 | Jan Mussler | j.mussler@fz-juelich.de

JÜLICH FORSCHUNGSZENTRUM

Presentation outline

Introduction

Instrumentation

Configurable instrumenter

Heuristics to select relevant points

Architecture

Example

Introduction

Student at the RWTH Aachen, Germany

Helmholtz-University Young Investigators Group

“Performance Analysis of Parallel Programs”

Lead by Professor F. Wolf

Located at the “Jülich Supercomputing Center”

JÜLICH
FORSCHUNGSZENTRUM

scalasca

Integrated measurement & analysis toolset

- Runtime summarization (aka profiling)
- Automatic event trace analysis

Objective

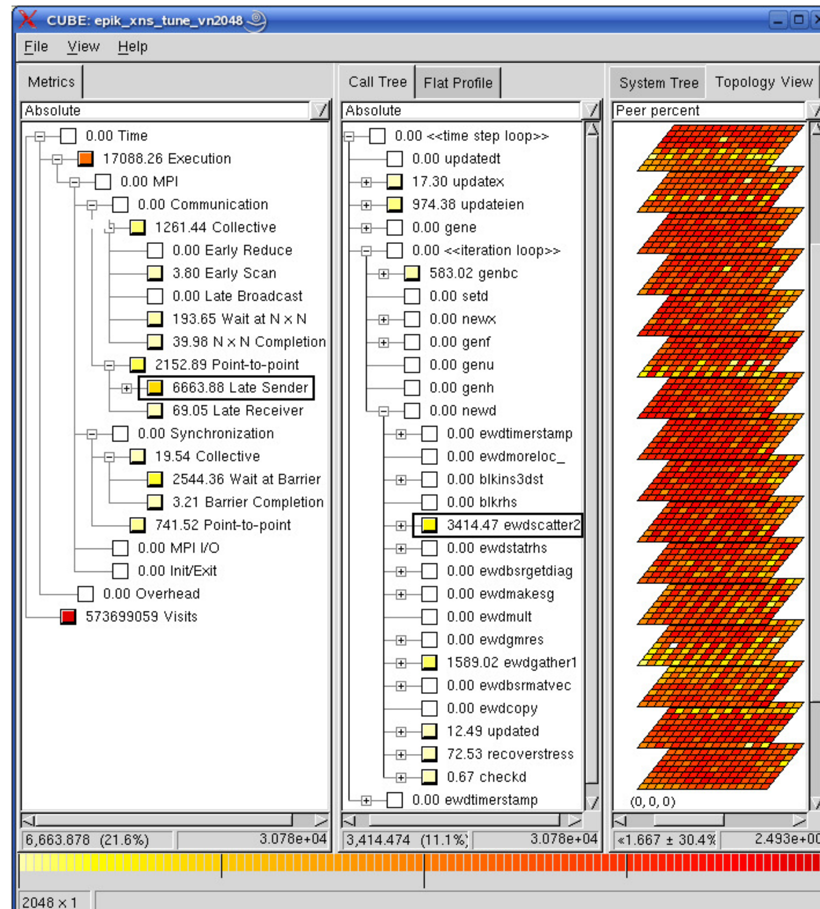
- Development of a **scalable** performance analysis toolset
- Specifically targeting **large-scale** applications

Supports various languages & parallel programming paradigms

- Fortran, C, C++
- MPI, OpenMP & hybrid MPI/OpenMP

More information at:

www.scalasca.org



Presentation outline

Introduction

Instrumentation

Configurable instrumenter

Heuristics to select relevant points

Architecture

Example

Instrumentation

Two ways to gather information

- By direct instrumentation
- By sampling, periodic measurement

Link between program and measurement system

- Trace events during program execution
- Profile to evaluate where time is spent

Possibilities of instrumentation

Source code transformation

- Manually added by user
- Automatically, e.g. TAU, OPARI

Compiler supported

- Wrapper functions
- Adding function calls

Library interposition

- MPI <-> PMPI

Binary instrumentation

- Static, e.g. TAU
- Dynamic, e.g. Paradyn

Static binary instrumentation

Advantages

- Language independent
- Instrumentation of optimized code
- Possible if no source available, e.g. libraries
- Templates are instantiated
- No need to recompile

Disadvantages

- Limited information available
- Not all platforms are supported

Information provided by Dyninst

Method identification

- E.g. Namespace::Class::Method in C++

List of called subroutines in given function

Control flow graph and loop tree

Possibility to access basic blocks

What information is available?

- Depends in part on available symbol table
- Improves when debug information are present
  - Sourcefile and sourceline become available

Presentation outline

Introduction

Instrumentation

Configurable instrumenter

Heuristics to select relevant points

Architecture

Example

Configurable binary instrumenter

Configurable by both the tool provider and user

Tool provider focuses on adapter specification

- Define code for initialization
- Define code for instrumentation
- Includes filter for the measurement system

User starts with provided filter

- Refines the filter to his or her needs

Possible instrumentation points

Functions

- Function enter and exit

Loops

- Before and after the loop
- Loop body enter and exit

Callsites

- Before the function call
- After the return

Filter requirements

Selective binary instrumentation

- Provide a usable default filter
- Allow the user to refine which parts to instrument

Configurable set of instrumentation points

- Filter by function, class and module names
- Filter by properties
- Ability to combine filters

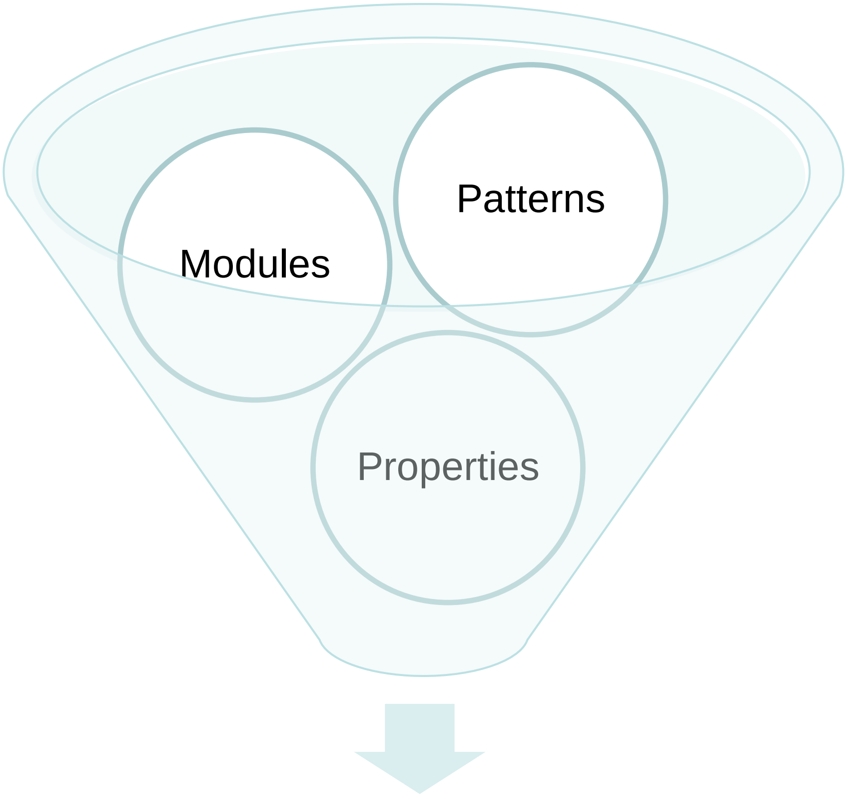
Filter

Start with set of functions

- All
- None

Filter set further using

- String patterns for
  - *Filename (module)*
  - *Namespace, classname*
- Properties
  - *E.g. callgraph, depth*



What to instrument?

Filter specifcation

A single XML document

- Patternlists as plain text for elements taking lists

Filter

- Include or exclude elements containing
  - *„and", „or", „not" and „true" or „false"*
  - *Functions, classes, namespaces, modules*
  - *Property*
- Callsite filter for restricting instrumentation

Filter specification example

```
<filter name="pathtest" instrument="functions=handletest" start="all">
<exclude>
        <or>
                <not>
                        <property name="path">
                                <functionnames match="simple">
                                        MPI*
                                </functionnames>
                        </property>
                </not>
                <functionnames>main</functionnames>
        </or>
</exclude>
</filter>
```

Inserted code

The instrumenter has to support

Additional dependencies (measurement system)

Variable declarations (e.g. region handles)

Code for initialization (run once at startup)

Code to be executed at points

- Enter / exit
- Before / after

Provide access to context information

- @linenumber@, @functionname@,…

Instrumentation specification

Independent XML document

- Include adapter filter

Dependencies

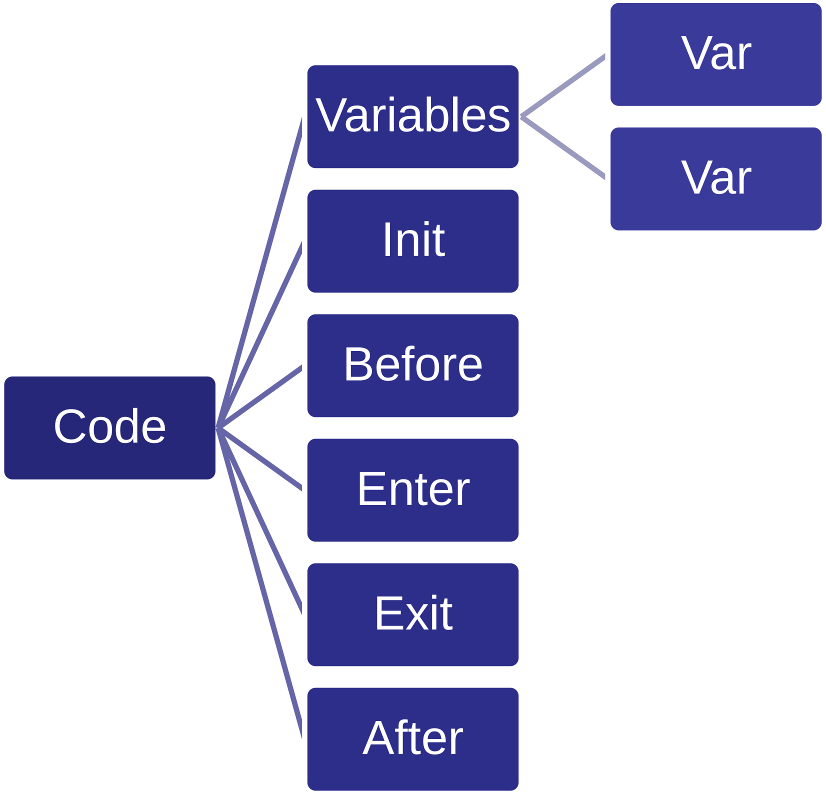
- Add dynamic libraries

Variable element

- Type information
- Memory to allocate

Code in plain text

- C-like syntax



Example specification

```
<code name="handletest">
        <variables>
                <var name="handle" type="void*" size="4" />
        </variables>
        <init>
                initNotify(@functionname@,@linenumber@,@filename@);
                handle = createHandle(@functionname@);
        </init>
        <enter>enterHandle(handle);</enter>
        <exit>exitHandle(handle); </exit>
</code>
```

Presentation outline

Introduction

Instrumentation

Configurable instrumenter

Heuristics to select relevant points

Architecture

Example

Goal

Automatic selection of relevant instrumentation points

How to select instrumentation points

What makes a point relevant?

- Granularity of trace to locate possible problems
- Ability to profile where time is spent

- Communication
- I/O

Is decision possible with available information?

Heuristics using binary code

Aim here: do not instrument short functions

- Instrumentation costs exceed function costs

Complexity of function

- Contains „if" and „loop" statements
- Amount of instructions
- Subroutine calls

Cyclomatic Complexity

- Complexity M = E(dges) – N(odes) + 1

Heuristics using debug information

Lines of code

- May be obscured by comments and code style

Method name hints

- Exclude e.g., helper functions „get*“, „set*“
- Include „do*“ ,“process*“, „calculate*“, or „solve*“

Classname and namespace

Heuristics using callpath

Callpath of functions

- Leads to I/O functions?
- Leads to MPI functions?
- Leads to functions using OpenMP?

Depth of function in call graph

- Instrument only to specified depth

Problem for static callpath construction

- Virtual functions, function pointers

Unevaluated results

CP2K Fortran code with Intel 10 compiler

- 12652 functions (50MB binary)
- Using MPI path reduced to 5194
- Using adapter filter and mpi path 767 remain

GENE Fortran code

- 7095 functions (13MB binary)
- Using adapter filter and MPI path reduced to 3144
- Remove nodes on direct path, leaves 2510 function

BT (NAS Parallel Benchmark)

- Reduced to 27 functions with MPI callpath filter
- More in the example later

Presentation outline

Introduction

Instrumentation

Configurable instrumenter

Heuristics to select relevant points

Architecture

Example

Architecture

Mutatee

- Layer between Dyninst and filter component

Filter

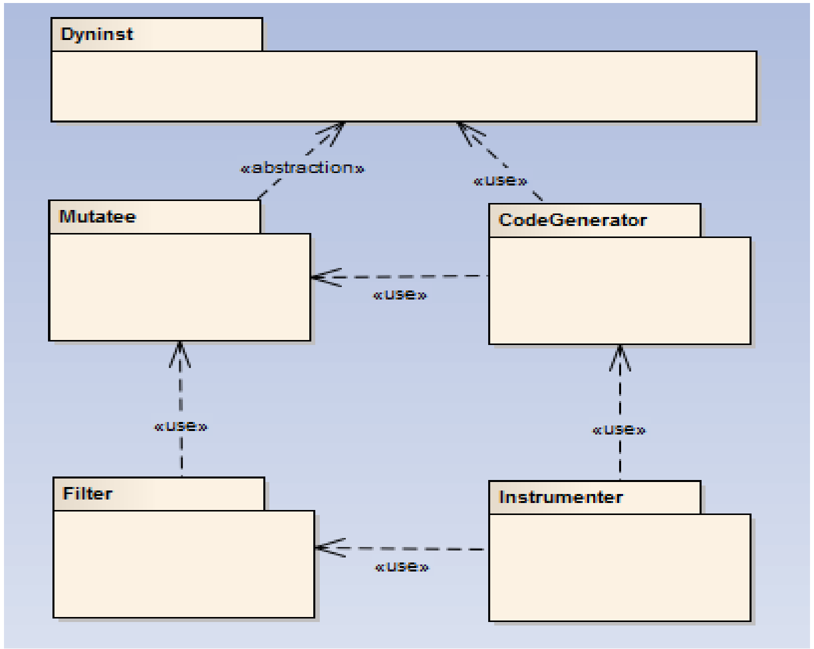
- Responsible for reading filter
- Evaluate filter

CodeGenerator

- Parses code specification
- Generates Dyninst snippets

Instrumenter

- Instruments the filtered set with generated code



Dependencies

Dyninst

Boost

- Spirit – Parser for adapter code
- Regex – Regular expressions in filter
- Tokenizer

Apache Xerces

- XML DOM parser for the adapter and filter files

Open issues

Binaries contain a lot of functions

Compiler-specific functions added

Scalasca does not provide dynamic library

- Need to preinstrument with “skin –comp=none –user”

Future work

Adding more properties

- sourceLines, hasControlStructure, calledInLoop

Evaluate reduction in instrumented functions

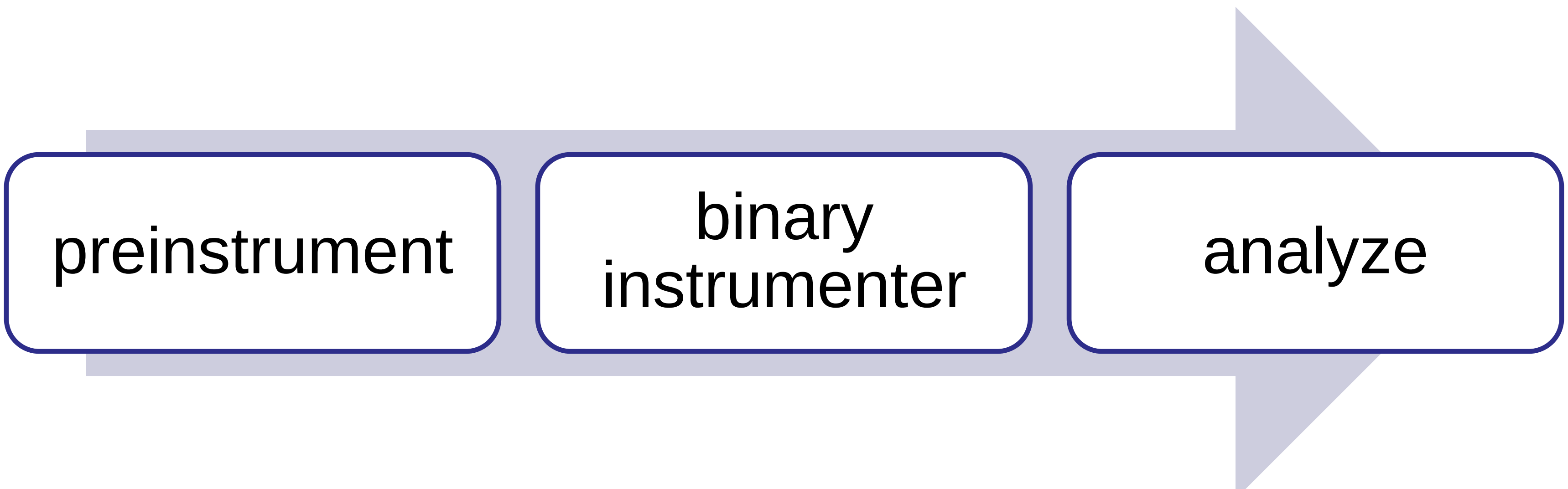
- Instrument benchmarks
- Instrument sample application

Evaluate advantage over filtering at runtime

Evaluate advantage of instrumenting optimized code

Example

Instrumenting NAS Parallel Benchmark BT



skin = scalasca –instrument –comp=none –user
scan = scalasca –analyze mpirun –n 4 ./mutated