Reducing the overhead of direct application instrumentation using prior static analysis

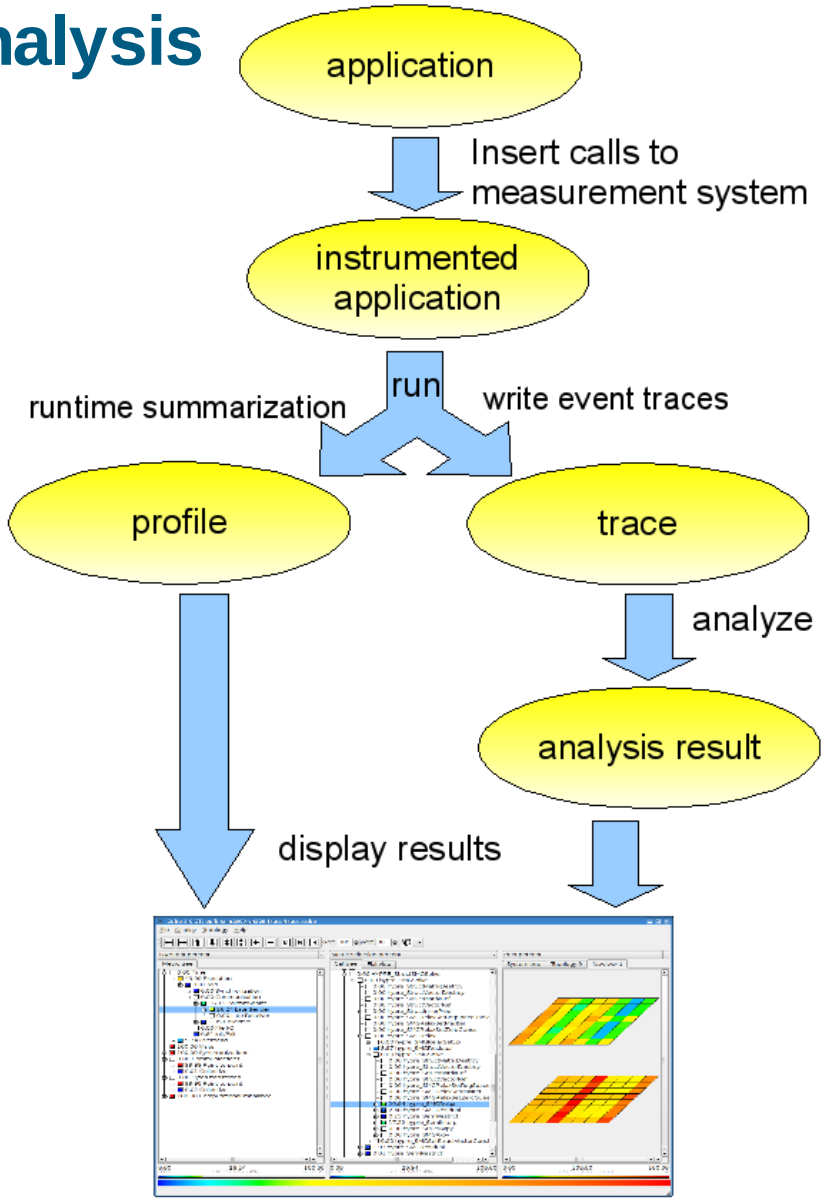
3rd May 2011 | Jan Mußler, Daniel Lorenz, Felix Wolf

Overview

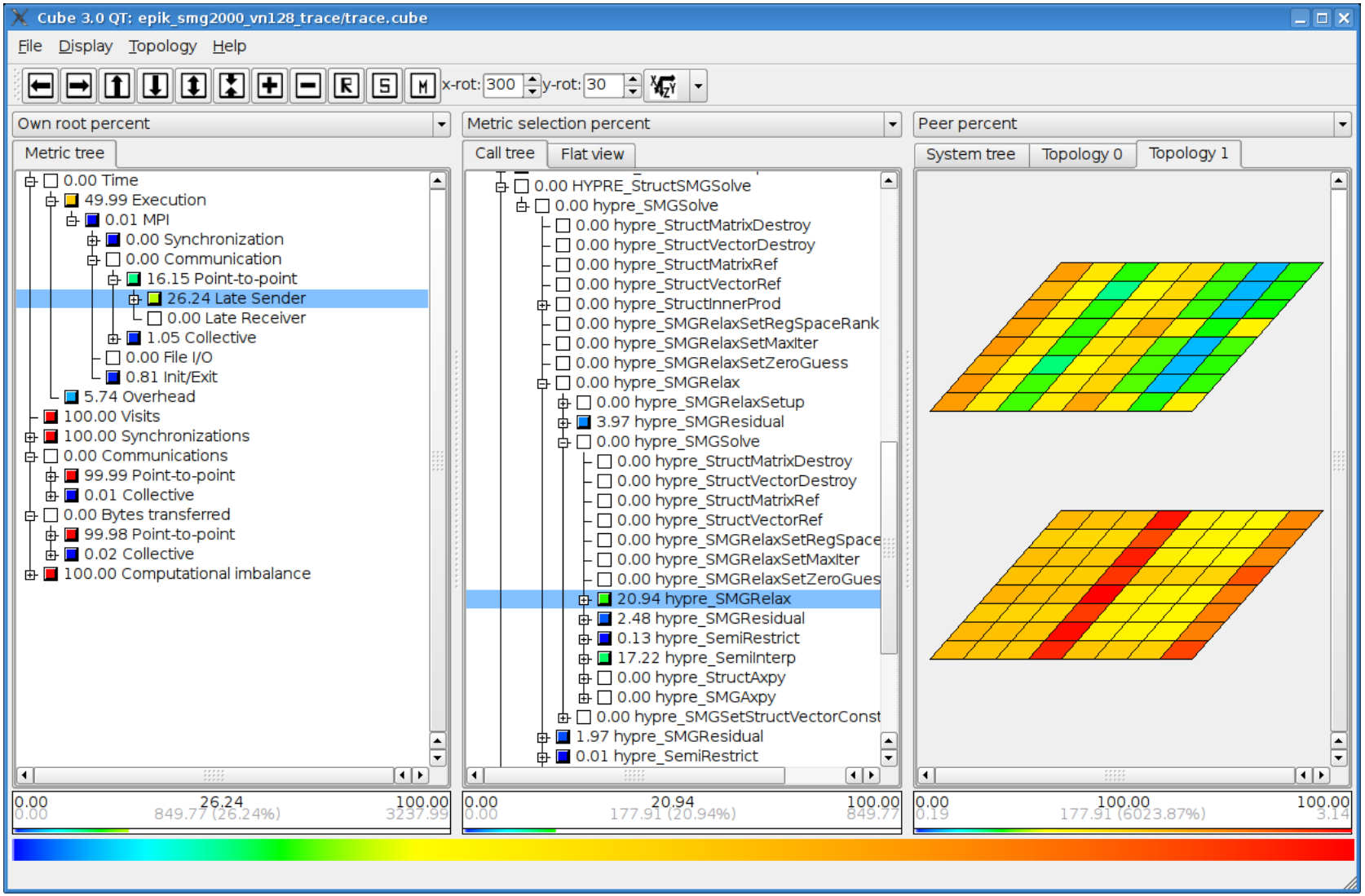
- Performance analysis with Scalasca
- Filtering
  - Motivation
  - Code structure based filters
- Configurable binary instrumenter
- Evaluation
- Future plans

Event-based performance analysis

- Instrument e.g. enter and exit points of functions
- Instrumentation is mostly done automatically by tools
- Most tools instrument every function
- Record performance data on events



Cube display



Overhead

- Inserting calls to a measurement system inserts overhead
- Runtime overhead grows with density of events
- Traces can grow large

- For many functions, enter/exit events are not needed for targeted analysis
  - E.g., communication analysis requires only call-path to MPI functions

- Target: Reduce overhead by excluding less relevant functions from instrumentation

Filtering

- Existing tool provide blacklist/whitelist approaches
  - Name-based filters are application dependent
  - Require several test runs to create appropriate filters
- Idea: Usage of code structure metrics provides application independent criteria
  - Criteria depend on analysis goal

Filter criteria

- Approaches for communication analysis:
  - MPI functions and functions on a call path to MPI functions
    - *May not be sufficient to pinpoint the source of computational imbalance that causes communication delays*
    - *Tradeoff between overhead reduction and information loss*
  - Try to avoid frequently called small functions
    - *High overhead*
- Need access to code structure

Configurable Binary Instrumenter (Cobi)

- Dyninst provides easy access to code structure
- Developed a binary instrumenter
  - uses the Dyninst binary rewriting features for instrumentation
  - Uses the information provided by Dyninst to apply filter rules
  - The inserted code can be configured by tool developers
  - The filters can be specified by the user based on a set of code structure criterias and name matching patterns

Cobi – instrumentation configuration

- XML file which specifies the code snippets inserted at instrumentation points
  - Subset of C
- Provided by the tool developer
- Can instrument functions and loops
- Definable points are: Before, enter, exit, after, initialize, finalize
- Specify additional shared libraries to be linked against the executable
- Special context variables: E.g., @ROUTINE@

Instrumentation example

```
<adapter>
  <dependencies>
    <library name="mlib.so" />
  </dependencies>
  <code name="functions">
    <enter> enterFunc(@ROUTINE@, @FILE@); </enter>
    <exit>  exitFunc(@ROUTINE@, @LINE@);  </exit>
  </code>
</adapter>
```

Cobi - filters

- Are specified in a separate XML file
- Start with all or no function
- Include/exclude functions with filter rules
- Possible rules are:
  - Are on call path
  - Lines of source code
  - Cyclomatic complexity
  - Number of instructions
  - Number and nesting level of loops
  - Number of function calls
  - Depth in call tree
  - Name matching
  - Prefix
  - Suffix
- Rules can be combined by logical operators

Filter example

```
<filter name="mpicallpath"
  instrument="functions=functions" start="none">
<include>
  <property name="path">
      <functionnames match="prefix"> MPI mpi
      </functionnames>
  </property>
</include>
</filter>
```

Costs of binary instrumentation

- Instrumented empty function with enter/exit calls to the Scalasca measurement system
- 1,000,000 executions in a loop
- Scalasca uses floating point registers
  - Need to save floating point registers on every event

instrumentation method	runtime / s
no instrumentation	0.02
compiler instrumentation	0.67
Cobi w/o saving floating point register	1.26
Cobi with saving floating point registers	2.61

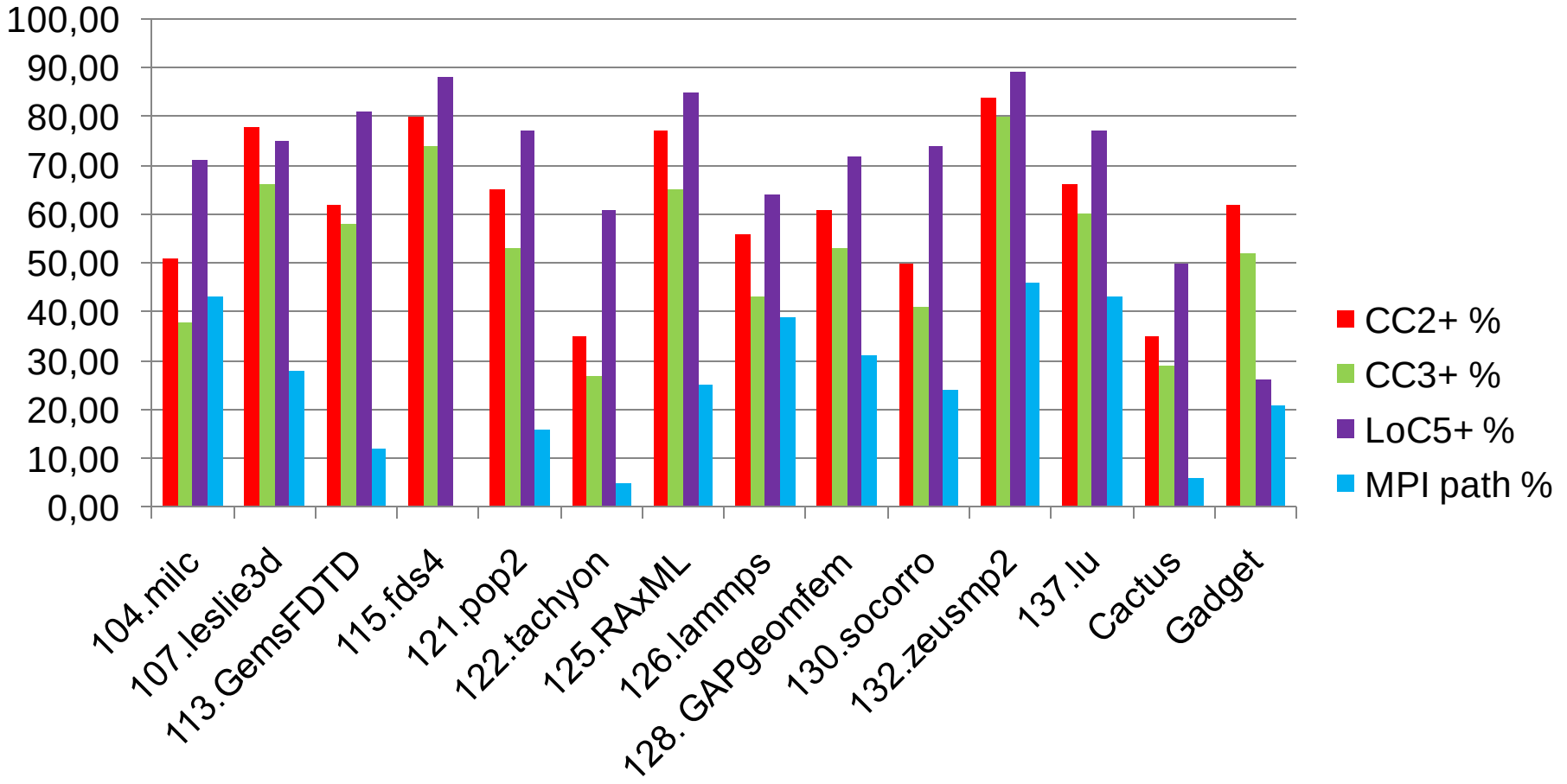
Evaluation method

- Apply filter to real world applications
  - MPI 2007 SPEC benchmarks
  - Cactus Carpet
  - Gadget
- Filters
  - MPI callpath
  - Lines of Code 5+
  - Cyclomatic complexity 2+
  - Cyclomatic complexity 3+
- Measure runtime and fraction of instrumented functions

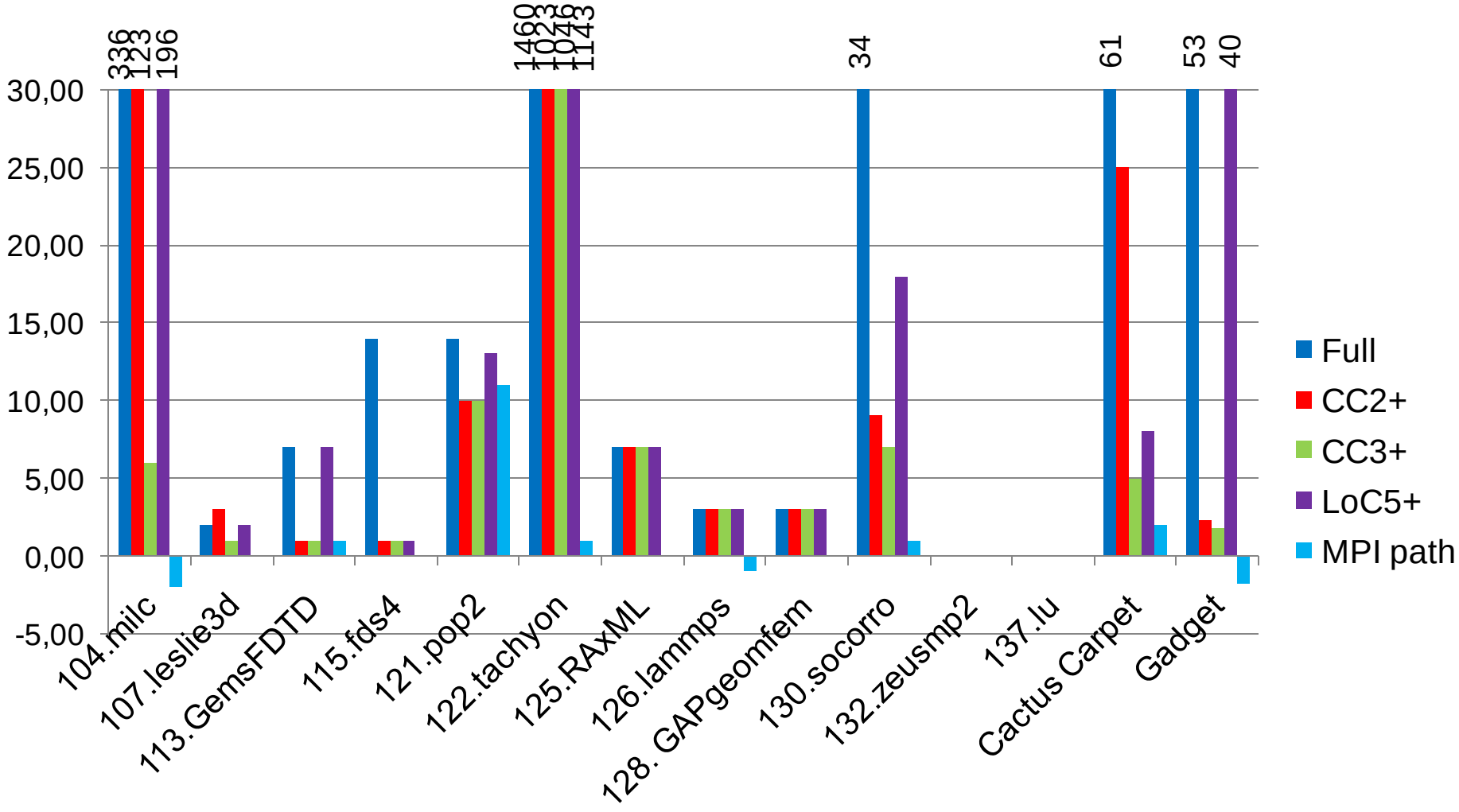
Technical environment

- Applications were compiled with gcc 4.3.4
- Patched Dyninst 6.1
  - Patches are included in Dyninst 7
- Instrumented test applications with Cobi
- Measured with Scalasca 1.3.2
- Runs on Juropa with 32 ranks
  - Except RAxML ran on 64 ranks

Instrumented fraction of functions with filters



Runtime overhead in percent



Conclusions

- MPI path is most aggressive and results in least overhead
- In most cases MPI path and CC3+ result in reasonable overhead
- In case of Gadget, CC3+ instruments much more functions than LoC5+, but has lower runtime overead
  - More accurate selection of overhead prone functions

Future plans with Cobi

- Want to distribute Cobi as a separate package
  - Coming soon
  - Need reliable detection of all entry/exit points for a function
- In future versions of the measurement system
  - Want to support Cobi as an experimental feature