Link-Time Call-Graph Analysis to Facilitate User-guided Program Instrumentation

An LLVM based approach

TECHNISCHE UNIVERSITÄT DARMSTADT

exaFOAM

https://exafoam.eu/

SCIENTIFIC COMPUTING

Exploring Application Performance

Survey Measurement
→ Initial overview, hotspot identification

Focus Measurements
→ Analysis of critical kernels

Empirical Modeling
→ Prediction of scaling behavior

Accurate & reliable measurements needed

OpenFOAM

- Computational fluid dynamics toolbox
- Variety of solvers
- ~1.2M LOC

Low-overhead Instrumentation

Code Instrumentation is a reliable method for collecting accurate performance data:

- e.g. `-finstrument-functions` flag in GCC/Clang

```
void kernel(double* A, int n) {
    for (int i = 0; i < n; i++) {
        …
    }
}
```

```
void kernel(double* A, int n) {
    __cyg_profile_func_enter(&kernel, ...)
    for (int i = 0; i < n; i++) {
        …
    }
    __cyg_profile_func_exit(&kernel, ...)
}
```

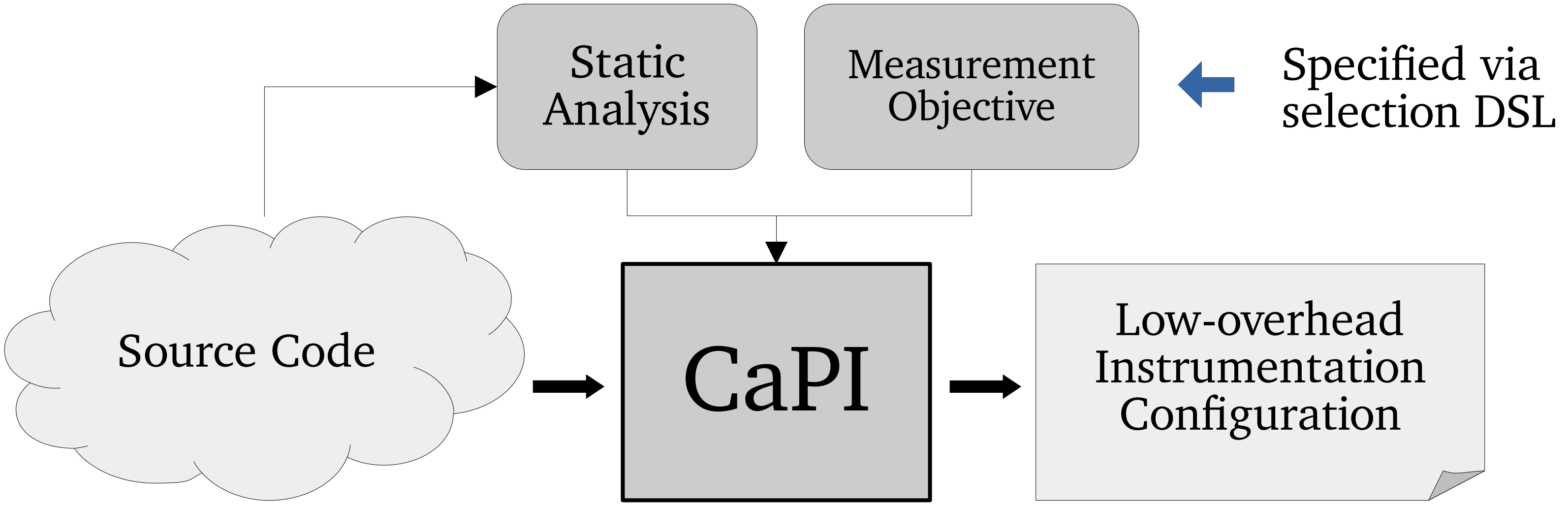
Profiling Tool

May increase runtime by orders of magnitude!

Selection mechanisms:

- Profile-based filtering (manual or tool-assisted, e.g. `scorep-score` [6])
- Call-graph based approaches:
  - **PIRA**: Automatic iterative refinement [1]
  - **CaPI**: User-defined selection specification [2]

CaPI: Compiler-assisted Performance Instrumentation



Selection Example

"I want to record all call-paths that contain **MPI communication**. Additionally, I want to measure functions that **contain loops** with **at least 10 floating point operations**. I don't care about **system headers** or **inlined** functions."

Set of all functions

```
!import("mpi.capi")
excluded = join(inSystemHeader(%%),inlineSpecified(%%))
kernels = flops(">=", 10, loopDepth(">=", 1, %%))
join(subtract(%kernels, %excluded), onCallPathTo(%mpi_comm))
```

➔ Reduces the number of instrumented functions by 74% (OpenFOAM)

Streamlining the Call-Graph Analysis

CaPI relies on a statically generated whole-program call-graph
- Currently generated on the source level by **MetaCG** [3]
- Can be cumbersome for complex applications
  - Requires separate analysis step
  - Manual merging of local call-graphs

In this talk, we:
- Highlight differences of **generating call-graphs at different stages**
- Introduce the **CAGE compiler plugin for LTO call-graph embedding**
- Elaborate how it can be used to **streamline the CaPI user experience**

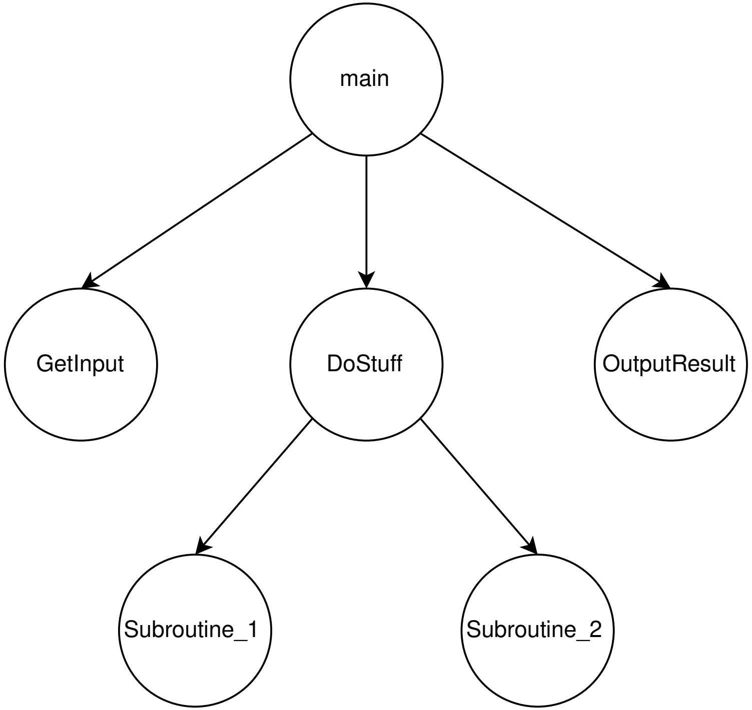
Whole-Program Call-Graph

Central data structure for CaPI selection
- Allows for named identification
- Allows for Path calculations

- Metadata can be attached
  - Instruction composition
  - Local/global loop depth
  - Instruction count
  - → Used to make instrumentation decision

- Can be generated at different stages
  - Source code
  - Intermediate representations
  - Machine Code

Example call-graph



Source Code

- MetaCG can generate call-graphs from source code
  - Generates graph for each translation unit (TU)
  - Merges separate sources to whole-program call-graph

✔ Information gathered maps cleanly to source code
✔ Is what the programmer wrote
✔ Readily available tools exist

✗ Is unaware of code transformations
✗ Is unaware of other TUs
  - Manual merge necessary
  - Might not perfectly emulate linker behavior

Compiled Machinecode

Radare2 [4] or Ghidra [5] can generate call-graphs from object files
- Requires no access to source code

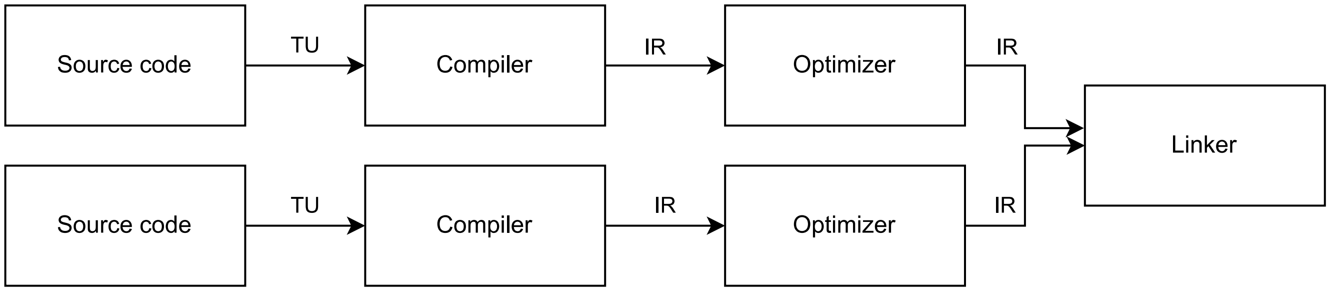
✔ Represents what is actually run on the CPU
✔ No code transformation will happen later

✗ Does not necessarily reflect what the user wrote
✗ Does not contain certain information
- Inlining
- Virtualness (Override/Final)
- Pointer Type information
- Constness

LLVM-IR at Link-Time

Best of both worlds

- ✔ Is close to what will be run on the Machine
- ✔ Is also close to what the programmer wrote
- ✔ Contains information about inlining, constness, virtualness, type-information
- ✔ Is not limited to TU, but can view linking context



We developed: CAGE-Plugin

- **C**a**ll-G**raph **E**mbedding LLVM plugin

- Call-graph creation as a LLVM plugin
  - Either as part of OPT
  - Or as part of ld.lld (custom fork)

- Can also do:
  - VTable analysis
  - Metadata annotation

- Embeds result into the created binary
  - Enables dynamic augmentation

Constructing the CG at Link-Time

TECHNISCHE UNIVERSITÄT DARMSTADT

Structural Information

- Call Hierarchy
  - Call Path
  - Call Depth
  - Number of Children

  - Virtual Function Calls
    - Partly meta-information

Meta Information

- Instruction composition
  (FLOPS, IOPS, MEMOPS)
- Local and global loop depth

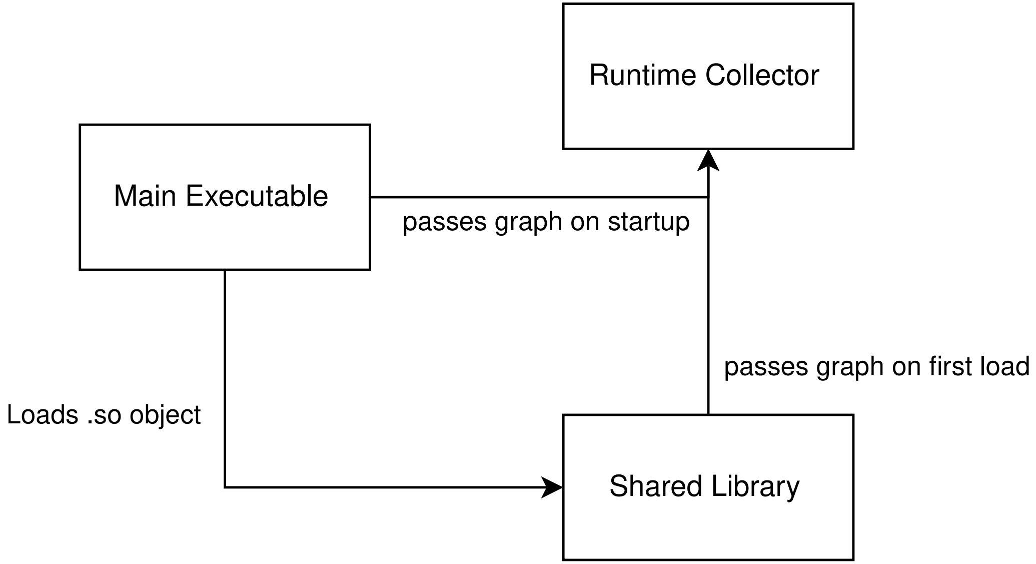
- Inlining Information
  - Partly structural information

SCIENTIFIC COMPUTING

Dynamic Augmentation

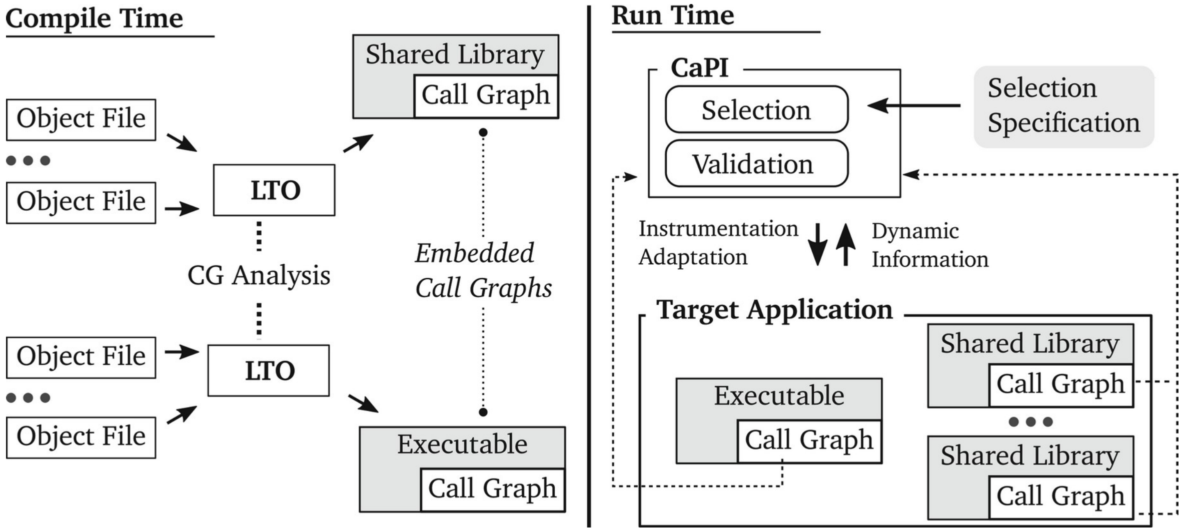
- Each object file contains its own call-graph
- The call-graphs are aggregated at runtime
- Same merging rules as for TU approaches
- Can attach runtime data and export it

→ May be used to improve CaPI selection



CaPI Integration

- CaPI runtime receives embedded call-graph at program start
  - Call-graphs of shared libraries merged in-memory
  - Runs selection and performs dynamic instrumentation



Summary

- **CaPI**: Instrumentation selection tool based on call-graph analysis

- New **CAGE plugin** generates call-graph at link-time:
  - Whole-program visibility, dynamically augmentable
  - Allows embedding into object files

- **CaPI + CAGE**
  - Selection and instrumentation at program start, using embedded call-graph
  - Improvement of CaPI usability due to full integration into compilation process
  - In active development

https://github.com/tudasc/CaPI

References

[1] J.-P. Lehr, A. Hück, and C. Bischof, “**PIRA: Performance instrumentation refinement automation**”, in AI-SEPS 2018 – Proceedings of the 5th ACM SIGPLAN International Workshop on Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems, Co-located with SPLASH 2018. New York, NY, USA: Association for Computing Machinery, Inc, nov 2018, pp. 1–10. https://dl.acm.org/doi/10.1145/3281070

[3] J.-P. Lehr, A. Hück, Y. Fischler, and C. Bischof, “**MetaCG: Annotated call-graphs to facilitate whole-program analysis**”,

in TAPAS 2020 - Proceedings of the 11th ACM SIGPLAN International Workshop on Tools for Automatic Program Analysis, Co-located with SPLASH 2020. New York, NY, USA: ACM, nov 2020, pp. 3–9. https://dl.acm.org/doi/10.1145/3427764.3428320

[2] S. Kreutzer, C. Iwainsky, J.-P. Lehr, and C. Bischof, “**Compiler-assisted instrumentation selection for large-scale c++ codes**”, in High Performance Computing. ISC High Performance 2022 International Workshops, H. Anzt, A. Bienz, P. Luszczek, and M. Baboulin, Eds. Cham: Springer International Publishing, 2022, pp. 5–19. https://link.springer.com/chapter/10.1007/978-3-031-23220-6_1

[4] Radare2. https://rada.re/n/

[5] Ghidra - Software Reverse Engineering Framework. https://github.com/NationalSecurityAgency/ghidra

[6] Knüpfer, Andreas, et al. "**Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir.**" Tools for High Performance Computing 2011: Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, September 2011, ZIH, Dresden. Springer Berlin Heidelberg, 2012. https://link.springer.com/chapter/10.1007/978-3-642-31476-6_7