Performance Analysis and Source-Code Instrumentation Toolsuite (PASCIT)

Anja Gerbes[1] (gerbes.anja@gmail.com), Julian Kunkel[2]

[1]*Goethe University Frankfurt,* [2]*University of Reading*

PASCIT automates profiling tools usage

With PASCIT we provide a single interface for analyzing scientific code using a variety of different kinds of profiler tools, creating a single, versatile tool. We hereby improve usability and tool selection to automated performance profiling.
PASCIT's command line interface facilitates performance analysis and source-code instrumentation of scientific softwares by finding hotspots. It allows using module files as well as git repositories to satisfy an applications requirements. PASCIT analyzes profiler output and provides insights into HPC computes dwarfs via a correlation between certain hardware counters summarized in tables and graphs.
PASCIT additionally evaluates profilers, analyzing the correlation between certain hardware counters summarized in tables and graphs. The knowledge of profilers and compilers is not required.

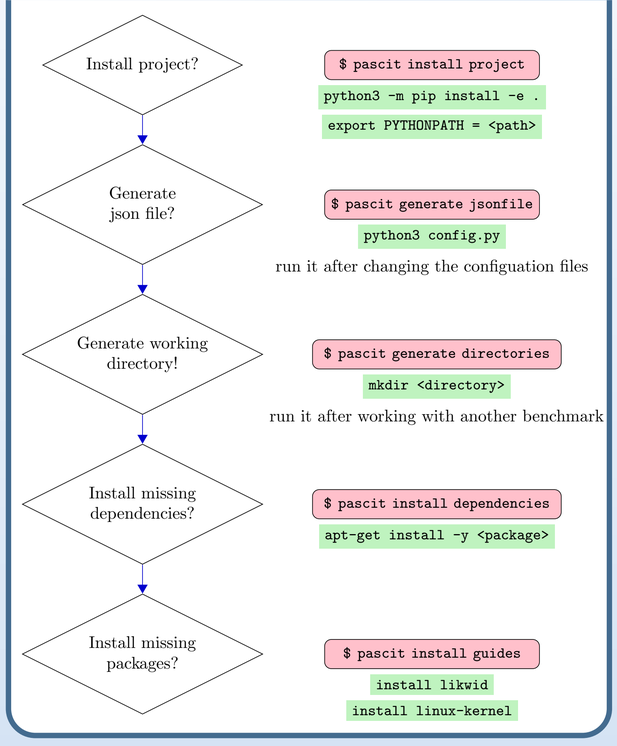
PASCIT Toolsuite & Benefits

Optimizing code, particularly 3rd party application, is a cumbersome process. Our solutions targets scientific applications including but not limited to the seven HPC compute dwarfs known as key algorithmic kernels: dense linear algebra, spectral methods, N-body methods, (un-)structured grids and Monte Carlo Methods.

Some implementations do not adequately use a given CPU instruction set. PASCIT helps to identify the theoretical optimum and the *de facto* loss of a given implementation. It guide programmers to optimize parallel applications via automated support.
In order to offer users of HPC centers help to optimize their code, we proposed to automate the steps for performance analysis and source-code instrumentation combined in one toolsuite. PASCIT's goal is to analyze scientific code with different independent profiler tools.

- Performance Analysis
  - Identify bottlenecks automatically
  - Identify HPC code patterns suitable for optimization
- Source-Code Instrumentation
  - Instrument & profile scientific code automatically
- Understanding of Compilers
  - Study the compiler for deficits in terms of performance when translating HPC applications
  - Study the limitations of why compilers can not make the necessary optimizations
  - Incorporate insights into the compiler for a future automatic compiler optimization
  - Analyze compilers theoretically & systematically
- Compiler Optimization
  - Select compiler options in an intelligent way to optimize compile time & performance
  - Optimize LLVM translation of HPC relevant patterns
  - Optimize typical HPC code structures at the compiler-level

Configuration



Satisfying Dependencies

Option 1: Module files
Option 2: Clone from github/gitlab (in process)
Option 3: Singularity support (in process)
Option 4: Generate dockerfile

```
$ pascit generate dockerfile --approach <single|multi|compose>
```

- `single` all packages dump in a single Dockerfile
- `multi` multi-stage build Dockerfile
- `compose` docker-compose version

} = --approach

```
$ pascit docker build
DOCKER_BUILDKIT=1 docker build . --target=base --tag pascit_image
$ pascit docker run
docker run --rm -it --privileged -v <path>:<path> pascit_image /bin/bash
```

Download

```
$ pascit download repository --recursive --benchmark <name>
Starting with https://github.com/user/repo.git
```

parse git repository
download git repository
benchmark configuration

classic version:
`git clone <repo.git> --recursive = False`
recursive version:
`git clone --recursive <repo.git> --recursive = True`
→ download subdirectories directly

Benchmark Configuration

- locate & modify Makefile
  `find . -name 'Makefile' -exec sed -i s/oldstring/newstring/' {} \;`
- update software file of benchmark → docker usage
- create install guide of benchmark
- get Makefile suffix → copy Makefile with new suffix → Makefile
- execute Makefile via `make`

compiler generates object file
`nm` list symbols from object files
extract function names

Details of benchmark stored in nested dictionary, like files and function names via analysis of object files.

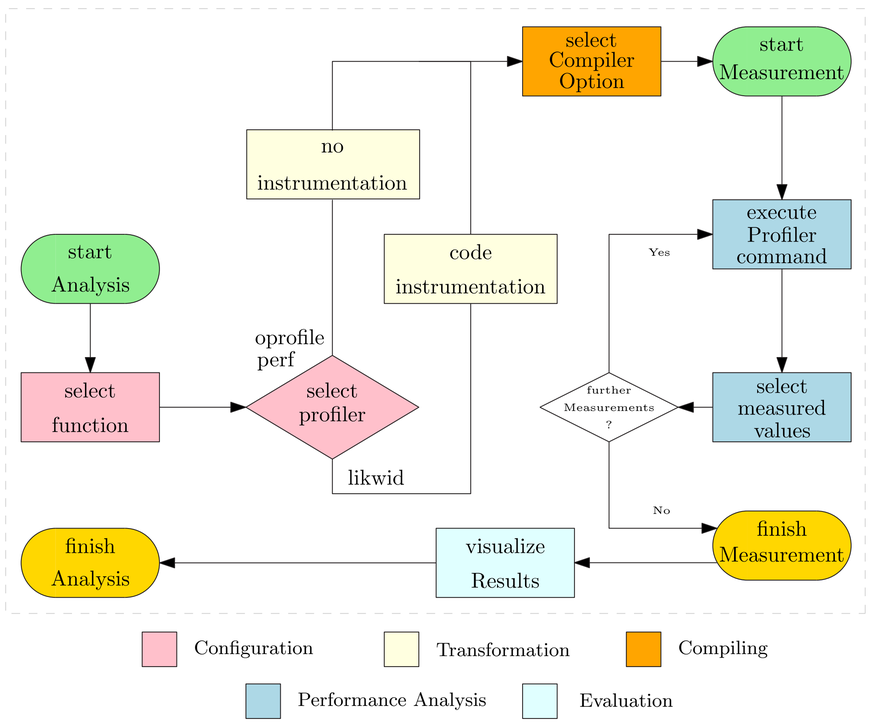
Compile

```
$ pascit compile code --compiler <name> --benchmark <name> --optlevel <level>
```

choose file

create library object files	`<compiler> <cflags> -c <file>.c -o <file>_<compiler><optlevel>.o <optlevel> -lm`
object file → executable	`<compiler> <file>_<compiler><optlevel>.o -o <file>_<compiler><optlevel> -lm`
compile code	`<compiler> -std=c99 -Wall <file>.c -g -o <file>_<compiler><optlevel> <optlevel> -lm`

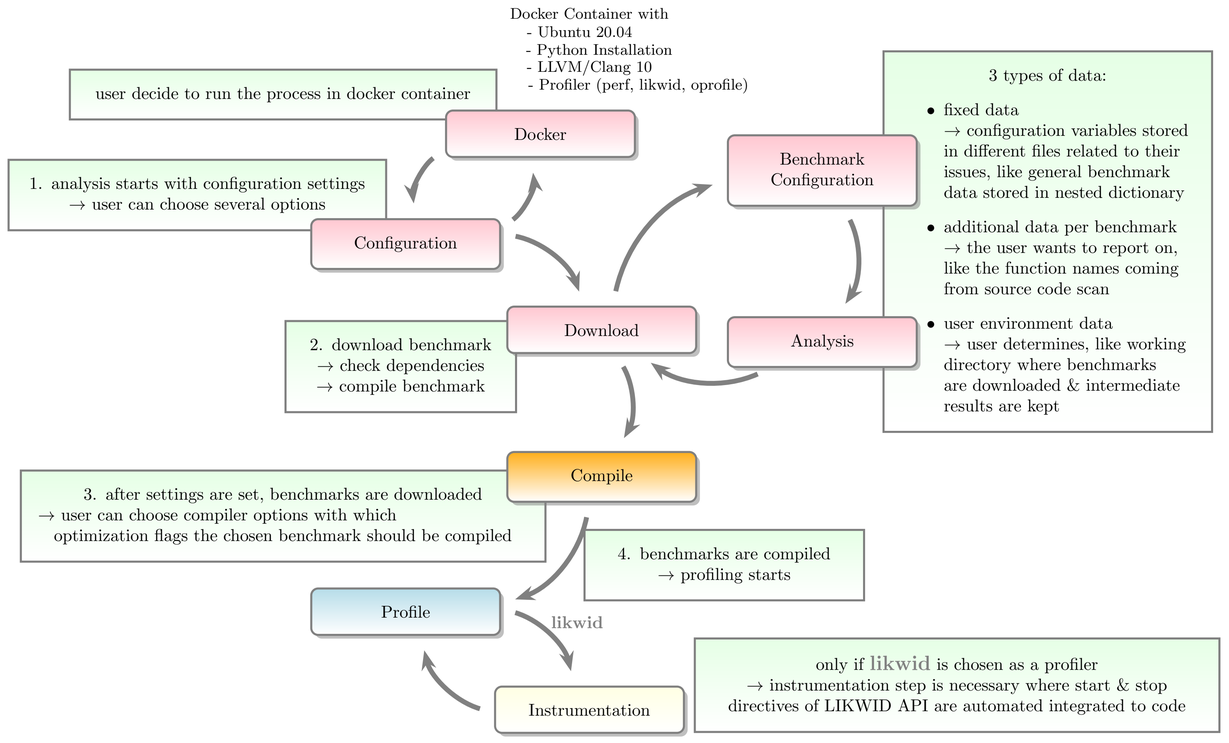
PASCIT Workflow



On the left hand side is a flow chart of PASCIT where you can get an high-level overview of PASCIT's key steps.

A detailed description of what is happening in the background of each developement step can be found on the right hand side.

The color code of both pictures are the same.



Profiling

There are already many performance analysis tools for Linux distributions. We are taking a look into the open-source performance monitoring tools, **perf**, **oprofile** and **likwid** for evaluating CPU performance on Linux machines. We are concentrating here on profilers which facilitates access to performance counters. Performance counters are useful to study usage of hardware architecture. There are native, derived and software counters. We can get information about CPU cores, cache and memory. Working with performance counters is easy, they are already there, but it is processor-specific, not each counter is available at an architecture and we have limited amount of counter registers.

perf Linux' on-board performance tool and part of Linux kernel
oprofile profiler for dynamic program analysis and uses Linux kernel performance events subsystem
likwid set of command line utilities that completely bypasses Linux kernel by accessing MSRs directly

Manual Use of Profiler

The following tables provides equivalent mappings for the profiler subcommands of `perf` and `oprofile`. Likwid is a performance monitoring and benchmarking toolsuite of different command line applications, such as `likwid-perfctr` which examines hardware performance counters. Likwid's Marker API allows us to measure named regions of your code. For using the LIKWID Marker API it is necessary to link the code against the LIKWID library. PASCIT automates commands such as

perf	oprofile	likwid
`sudo perf record --event counter ./executable`	`sudo operf --events=counter:high number:methode ./executable`	`gcc -O3 -fopenmp -pthread -o executable file.c -DLIKWID_PERFMON -IPATH_TO_LIKWID/include -LPATH_TO_LIKWID/lib -llikwid -lm`
`sudo perf report --dsos=executable --sort=srcline --full-source-path`	`sudo opreport ./executable --debug-info --long-filenames --symbols`	`likwid-perfctr -C cores -g group [ -o file.json|-O ] -m ./executable`
`objdump --source --source-comment=SRC --line-numbers --all-headers ./executable`	`objdump --source --source-comment=SRC --line-numbers --all-headers ./executable`	
`sudo perf annotate`	`sudo opannotate --assembly --source`	
`sudo perf stat --field-separator , --event counter ./executable`	`sudo ocount --events counter ./executable`	

Perf

```
$ pascit profile benchmark-suite --benchmark <name>
```

choose: perf → command → counters → benchmark options

- `perf record` gather hardware performance counter
- `perf report` analyzed saved performance data
- `objdump` displays information about object files
- `perf annotate` displays an annotated version of code
- `perf stat` run program & report performance data

Oprofile

```
$ pascit profile benchmark-suite --benchmark <name>
```

choose: oprofile → command → counters → benchmark options

- `operf` operf is used to control profiling
- `opreport` produce symbol or binary image summaries
- `objdump` displays information about object files
- `opannotate` produce source or assembly annotated with profile
- `ocount` count native hardware events

Likwid

1

```
$ pascit instrument benchmark-suite --benchmark <name>
```

2

```
$ pascit profile benchmark-suite --benchmark <name> --format <format>
```

choose: likwid → command → counters → benchmark options

`json` == JSON format
`csv` == CSV format
`table` == Table format

} = --format

Instrumentation

```
$ pascit instrument benchmark-suite --benchmark <name>
```

choose file → choose function

With LLVM and the Clang AST we can instrument the benchmark with the LIKWID directives to use their marker API, which make it possible to record only the hardware performance counters on specific regions.

- replace function name
- generate makefile
- instrument code
- run makefile
- compile code

Utilize the Clang RecursiveASTVisitor class
- Performs depth-first traversal on AST
- Visits each node

Specify the AST nodes of interest
- Add Decl Code
  - FunctionDecl
- Add Stmt Code
  - ReturnStmt
  - ForStmt

Instrumentation of
- the whole file
- all functions in file
- specific function
- a for loop
- a specific for loop

Conclusion

We want to identify the best possible execution to help programmers in their code optimization for parallel applications via automated support. PASCIT's goal is to analyze scientific code with different independent profiler tools.
The poster's aim is to illustrate the key aspects of PASCIT's workflow, written in Python, and to draw a comparison of the profiler commands and profiler features. PASCIT's multifaceted tool stands out for

- an user friendly command-line interface
- the ease of usage of accessing various profiler tools, it automates the step for performance analysis and source-code instrumentation
- an intelligent compiler selection
- providing required dependencies, such as missing packages and dockerfiles

All features are combined in a single interface.