PerfFlowAspect Documentation

Release 0.1

Dong Ahn, Stephanie Brink, James Corbett, Stephen Herbein, Fra

Apr 24, 2024

BASICS

PerfFlowAspect is a tool to analyze cross-cutting performance concerns of composite scientific workflows.

CHAPTER

ONE

INTRODUCTION

High performance computing (HPC) researchers are increasingly introducing and composing disparate workflow-management technologies and components to create scalable end-to-end science workflows. These technologies have generally been developed in isolation and often feature widely varying levels of performance, scalability and interoperability. All things considered, optimizing the end-to-end workflow amidst those considerations is a highly daunting task and thus it requires effective performance analysis techniques and tools.

Unfortunately, there still is a paucity of techniques and tools that can analyze the end-to-end performance of such a composite workflow. While a myriad of analysis tools exist for traditional HPC programming paradigms (e.g., a single application running at scale), there has been a lack of studies and tools to understand the effectiveness and efficiency of this emerging workflow paradigm.

Enter PerfFlowAspect. It is a simple Aspect-Oriented Programming-based (AOP) tool that can cast a cross-cutting performance-analysis concern or aspect across a heterogeneous set of components (e.g, combining Maestro and a custom workflow pipeline with Flux along with microservices running on on-premises Kubernetes machines) used to create a modern-day composite science workflow.

PerfFlowAspect will provide multi-language support, particularly for those most relevant in HPC workflows including Python. It is designed specifically to allow researchers to weave the performance aspect into critical points of execution across many workflow components without having to lose the modularity and uniformity as to how performance is measured and controlled.

CHAPTER

TWO

PERFFLOWASPECT PROJECT RESOURCES

Online Documentation https://perfflowaspect.readthedocs.io/

Github Source Repo https://github.com/flux-framework/PerfFlowAspect.git

Issue Tracker https://github.com/flux-framework/PerfFlowAspect/issues

CHAPTER

THREE

CONTRIBUTORS

- Dong H. Ahn (NVIDIA)
- Stephanie Brink
- James Corbett
- Stephen Herbein (NVIDIA)
- Aliza Lisan (University of Oregon)
- Daniel Milroy
- Francesco Di Natale (NVIDIA)
- Tapasya Patki
- Jae-Seung Yeom
- Hariharan Devarajan

CHAPTER

FOUR

PERFFLOWASPECT DOCUMENTATION

4.1 Basic Tutorial

PerfFlowAspect is based on Aspect-Oriented Programming (AOP). PerfFlowAspect relies on annotated functions in the user's source code and can invoke specific performance-analysis actions, a piece of tracing code, etc. on those points of execution. In AOP, these trigger points are called join points in the source code, and the functionality invoked is called *advice*. To learn more about AOP and associated terminology, please refer to our presentation slides here.

The python package `perfflowaspect` contains the PerfFlowAspect tool for the Python language. The file `src/python/perfflowaspect/aspect.py` contains a key annotating decorator. Users can use the `@perfflowaspect.aspect.critical_path()` decorator to annotate their functions that are likely to be on the critical path of the workflow's end-to-end performance. These annotated functions then serve as the join points that can be weaved with PerfFlowAspect to be acted upon. The decorator accepts the following pointcut values at the join points:

- `before`: The advice is invoked only before the join point.
- `after`: The advice is invoked only after the join point.
- `around`: The advice is invoked both before and after the join point.

The asynchronous versions of these pointcut values are also supported in PerfFlowAspect, which are: `before_async`, `after_async`, and `around_async`.

Note: The default pointcut value is `around`.

The following shows a simple snippet that annotates two functions.

```
import perfflowaspect.aspect


@perfflowaspect.aspect.critical_path()
def bar(message):
    time.sleep(1)
    print(message)


@perfflowaspect.aspect.critical_path()
def foo():
    time.sleep(2)
    bar("hello")


def main():
    foo()
```

Once annotated, running this python code will produce a performance trace data file named `perfflow.<hostname>.<pid>`. It uses Chrome Tracing Format in JSON so that it can be loaded into Google Chrome Tracing to render the critical path events on the global tracing timeline, using the Perfetto visualization tool. Details on these can be found at the links below:

- **Chrome Tracing Tool:** https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/
- **Perfetto Visualizer:** https://perfetto.dev/

To disable all PerfFlowAspect annotations, set the `PERFFLOW_OPTIONS="log-enable="` to `False` at runtime.

```
PERFFLOW_OPTIONS="log-enable=False" ./test/smoketest.py
```

4.1.1 PerfFlowAspect CLI Options

PerfFlowAspect options can be set with the `PERFFLOW_OPTIONS` environment variable. Separate multiple variables with a colon as follows:

```
PERFFLOW_OPTIONS="<var1>=<val1>:<var2>=<val2>" <executable>
```

Variable	Description	Default Value	Supported Values
name	Name of this workflow component	generic	
log-filename-include	Customize name of log file	host-name,pid	name,instance-path,hostname,pid
log-dir	Directory where log file is created	./	
log-enable	Toggle annotations on/off	True	True, False
cpu-mem-usage	Collect CPU and memory usage metrics	False	True, False
log-event	Collect B and E events (verbose) or single X event (compact)	Verbose	Verbose, Compact

4.1.2 Visualization of PerfFlowAspect Output Files

There are two types of logging allowed in PerfFlowAspect trace files which are `verbose` and `compact`. Either can be enabled by setting `PERFFLOW_OPTIONS="log-event="` to `compact` or `verbose`, respectively. The logging is `verbose` by default. `Verbose` logging uses B (begin) and E (end) events in the trace file as shown below:

```
[
  {"name": "foo", "cat": "/PerfFlowAspect/src/c/test/smoketest.cpp", "pid": 3134, "tid":␣
↪3134, "ts": 1679127184455376.0, "ph": "B"},
  {"name": "bar", "cat": "/PerfFlowAspect/src/c/test/smoketest.cpp", "pid": 3134, "tid":␣
↪3134, "ts": 1679127184456525.0, "ph": "B"},
  {"name": "bas", "cat": "/PerfFlowAspect/src/c/test/smoketest.cpp", "pid": 3134, "tid":␣
↪3134, "ts": 1679127184457610.0, "ph": "B"},
  {"name": "bas", "cat": "/PerfFlowAspect/src/c/test/smoketest.cpp", "pid": 3134, "tid":␣
↪3134, "ts": 1679127184457636.0, "ph": "E"},
  {"name": "bar", "cat": "/PerfFlowAspect/src/c/test/smoketest.cpp", "pid": 3134, "tid":␣
↪3134, "ts": 1679127184457657.0, "ph": "E"},
  {"name": "foo", "cat": "/PerfFlowAspect/src/c/test/smoketest.cpp", "pid": 3134, "tid":␣
↪3134, "ts": 1679127184457676.0, "ph": "E"},
  ...
]
```

The above trace file is generated for three functions with `around` pointcut annotations. The same trace file will be reduced to half the lines with `compact` logging which uses a single X (complete) events, as can be seen below:

```
[
  {"name": "bas", "cat": "/PerfFlowAspect/src/c/test/smoketest.cpp", "pid": 2688, "tid":␣
↪2688, "ts": 1679127137181517.0, "ph": "X", "dur": 600.0},
  {"name": "bar", "cat": "/PerfFlowAspect/src/c/test/smoketest.cpp", "pid": 2688, "tid":␣
↪2688, "ts": 1679127137179879.0, "ph": "X", "dur": 2885.0},
  {"name": "foo", "cat": "/PerfFlowAspect/src/c/test/smoketest.cpp", "pid": 2688, "tid":␣
↪2688, "ts": 1679127137177783.0, "ph": "X", "dur": 5532.0},
  ...
]
```

The visualization of both types of logging in trace files will be the same in Perfetto UI. An example visualization is shown below:

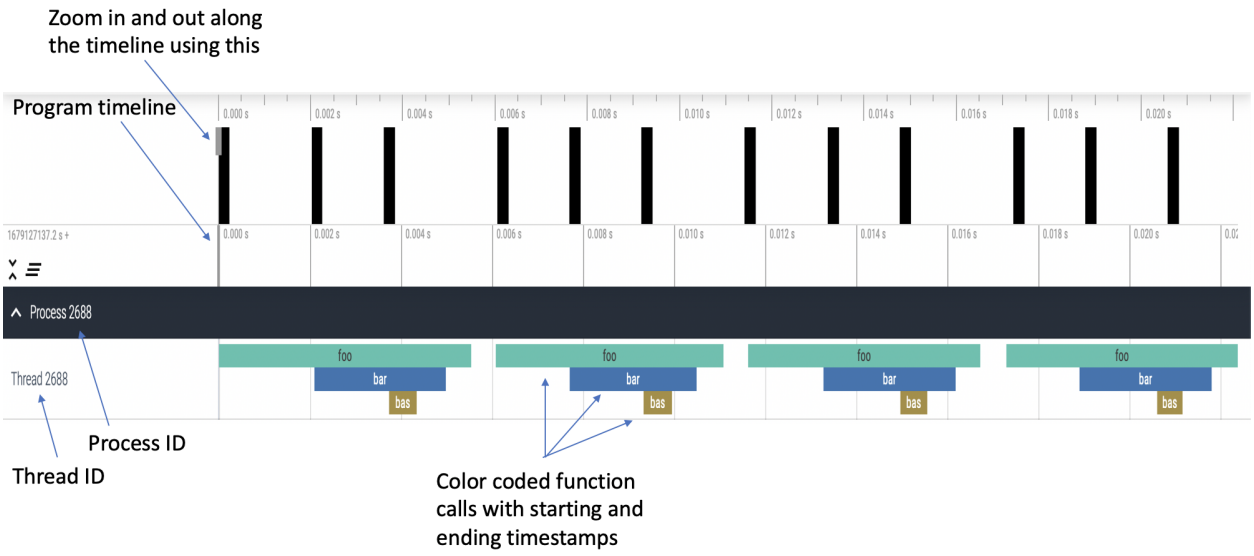

Fig. 1: Fig. 1: Visualization of a single process, single thread program in Perfetto UI

The visualization in Fig. 1 is of the following python program:

```
#!/usr/bin/env python

import time
import perfflowaspect
import perfflowaspect.aspect


@perfflowaspect.aspect.critical_path(pointcut="around")
def bas():
   print("bas")


@perfflowaspect.aspect.critical_path(pointcut="around")
def bar():
```

(continues on next page)

(continued from previous page)

```
   print("bar")
   time.sleep(0.001)
   bas()


@perfflowaspect.aspect.critical_path()
def foo(msg):
   print("foo")
   time.sleep(0.001)
   bar()
   if msg == "hello":
      return 1
   return 0


def main():
   print("Inside main")
   for i in range(4):
      foo("hello")
   return 0


if __name__ == "__main__":
   main()
```

PerfFlowAspect also allows the user to log CPU and memory usage of annotated functions by setting `PERFFLOW_OPTIONS="cpu-mem-usage="` to `True` at runtime. The trace file, in that case, will have the following structure with `compact` logging enabled:

```
[
  {"name": "bas", "cat": "/PerfFlowAspect/src/c/test/smoketest3.cpp", "pid": 44479, "tid
↪": 44479, "ts": 1679184351167907.0, "ph": "C", "args": {"cpu_usage": 0.0, "memory_usage
↪": 10944}},
  {"name": "bas", "cat": "/PerfFlowAspect/src/c/test/smoketest3.cpp", "pid": 44479, "tid
↪": 44479, "ts": 1679184351168628.0, "ph": "C", "args": {"cpu_usage": 0.0, "memory_usage
↪": 0}},
  {"name": "bas", "cat": "/PerfFlowAspect/src/c/test/smoketest3.cpp", "pid": 44479, "tid
↪": 44479, "ts": 1679184351167907.0, "ph": "X", "dur": 721.0},
  {"name": "bar", "cat": "/PerfFlowAspect/src/c/test/smoketest3.cpp", "pid": 44479, "tid
↪": 44479, "ts": 1679184351167127.0, "ph": "C", "args": {"cpu_usage": 11.
↪980575694383594, "memory_usage": 10944}},
  {"name": "bar", "cat": "/PerfFlowAspect/src/c/test/smoketest3.cpp", "pid": 44479, "tid
↪": 44479, "ts": 1679184351170287.0, "ph": "C", "args": {"cpu_usage": 0.0, "memory_usage
↪": 0}},
  {"name": "bar", "cat": "/PerfFlowAspect/src/c/test/smoketest3.cpp", "pid": 44479, "tid
↪": 44479, "ts": 1679184351167127.0, "ph": "X", "dur": 3160.0},
  {"name": "foo", "cat": "/PerfFlowAspect/src/c/test/smoketest3.cpp", "pid": 44479, "tid
↪": 44479, "ts": 1679184351165193.0, "ph": "C", "args": {"cpu_usage": 98.
↪625834450525915, "memory_usage": 14976}},
  {"name": "foo", "cat": "/PerfFlowAspect/src/c/test/smoketest3.cpp", "pid": 44479, "tid
↪": 44479, "ts": 1679184351505085.0, "ph": "C", "args": {"cpu_usage": 0.0, "memory_usage
↪": 0}},
```

(continues on next page)

(continued from previous page)

```
  {"name": "foo", "cat": "/PerfFlowAspect/src/c/test/smoketest3.cpp", "pid": 44479, "tid
↪": 44479, "ts": 1679184351165193.0, "ph": "X", "dur": 339892.0},
  ...
]
```

Following is the visualization for the python program above with CPU and memory usage logging enabled:

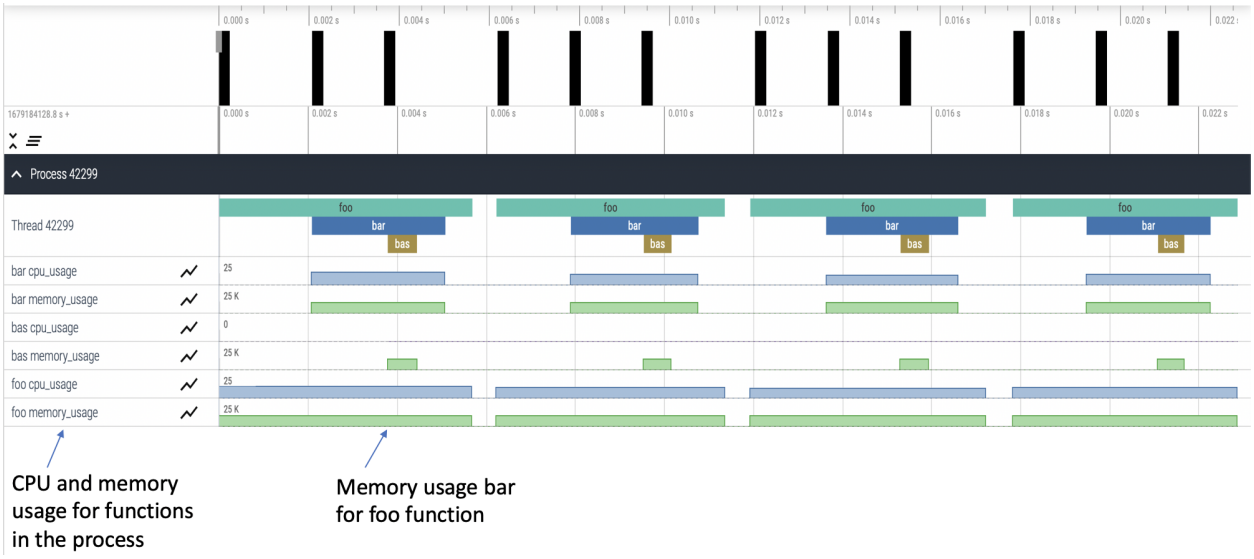

Fig. 2: Fig. 2: Visualization of a single process, single thread program with CPU and memory usage

4.2 Release Information

NOTE: The interfaces are being actively developed and are not yet stable. The GitHub issue tracker is the primary way to communicate with the developers.

4.3 License Information

GNU LESSER GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/> Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates

the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, “this License” refers to version 3 of the GNU Lesser

General Public License, and the “GNU GPL” refers to version 3 of the GNU General Public License.

“The Library” refers to a covered work governed by this License,

other than an Application or a Combined Work as defined below.

An “Application” is any work that makes use of an interface provided

by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A “Combined Work” is a work produced by combining or linking an

Application with the Library. The particular version of the Library with which the Combined Work was made is also called the “Linked Version”.

The “Minimal Corresponding Source” for a Combined Work means the

Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The “Corresponding Application Code” for a Combined Work means the

object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License

without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a

facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

> a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or
>
> b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from

a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

> a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.
>
> b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that,

taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

> a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.
>
> b) Accompany the Combined Work with a copy of the GNU GPL and this license document.

c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.

d) Do one of the following:

0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the

Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.

b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions

of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the

Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide

whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

4.4 Build Instructions

4.4.1 Python Install

The minimum Python version needed is 3.8. You can get PerfFlowAspect from its GitHub repository using this command:

```
$ git clone https://github.com/flux-framework/PerfFlowAspect
```

This will create a directory called `PerfFlowAspect`.

To use PerfFlowAspect, you will need to update your `PYTHONPATH` with the path to the PerfFlowAspect python directory:

```
$ cd src/python
$ export PYTHONPATH=$PWD:$PYTHONPATH
```

4.4.2 C Build

Host Config Files

To handle build options, third-party library paths, and other environment-specific configurations, PerfFlowAspect relies on CMake's initial-cache file mechanism.

These initial-cache files are called host-config files in PerfFlowAspect, since we typically create a file for each platform or specific system if necessary.

Example configuration files can be found in the `host-configs/` directory. Assuming you are in a `build/` directory, you can call the host-config file as follows:

```
$ cmake -C host-configs/{config_file}.cmake ../
```

Build Dependencies and Versions

redhat	ubuntu	version
clang	clang	>= 6.0
llvm-devel	llvm-dev	>= 6.0
jansson-devel	libjansson-dev	>= 2.6
openssl-devel	libssl-dev	>= 1.0.2
cmake	cmake	>= 3.10
flex	flex	>= 2.5.37
bison	bison	>= 3.0.4
make	make	>= 3.82

Building PerfFlowAspect

PerfFlowAspect uses CMake and requires Clang and LLVM development packages as well as a `jansson-devel` package for JSON manipulation. It additionally requires the dependencies of our annotation parser code: i.e., `flex` and `bison`. Note that `LLVM_DIR` must be set to the corresponding LLVM cmake directory which may differ across different Linux distributions.

```
$ module load clang/10.0.1-gcc-8.3.1 (on LLNL systems only)
$ cd PerfFlowAspect/src/c
$ mkdir build && cd build
$ cmake -DCMAKE_CXX_COMPILER=clang++ ../
$ make (note: parallel make (make -j) not supported yet)

$ find . -print | grep lib # successful build produces 3 libraries
./build/parser/libperfflow_parser.so
./build/runtime/libperfflow_runtime.so
./build/weaver/weave/libWeavePass.so
```

4.5 Source Code Annotations

Users can annotate their workflow code to get end-to-end performance insights. Currently, three techniques are available for this:

- Critical path annotation
- Synchronous events annotation
- Asynchronous events annotation

For critical path annotation, the user can provide pointcut and scope information for the annotated region. Currently, valid pointcut values are `before`, `after`, `around`, `before_async`, `after_async`, and `around_async`. When no pointcut is specified, the default assumption is `around`. We show an example of this below:

```
#!/usr/bin/python3

import time
import perfflowaspect
import perfflowaspect.aspect


@perfflowaspect.aspect.critical_path()
def foo(msg):
    print("foo")
    time.sleep(1)
    if msg == "hello":
        return 1
    return 0


def main():
    print("Inside main")
    for i in range(4):
        foo("hello")
```

(continues on next page)

(continued from previous page)

```
    return 0


if __name__ == "__main__":
    main()
```

For synchronous event annotation, the user can provide a pointcut, name, and category for the annotated region. Valid pointcut values are `before`, `after`, and `around`. The name represents a way to identify the current function being annotated, and the category can be a filename. An example of this is shown below:

```
#!/usr/bin/python3

import time
import os.path
from perfflowaspect import aspect


def foo():
    aspect.sync_event("before", "foo", filename)
    time.sleep(2)
    print("hello")
    aspect.sync_event("after", "foo", filename)


def main():
    aspect.sync_event("before", "main", filename)
    foo()
    aspect.sync_event("after", "main", filename)


if __name__ == "__main__":
    filename = os.path.basename(__file__)
    main()
```

For asynchronous event annotation, the user can provide a pointcut, name, category, and scope for the annotated region. An example of this is shown below with the help of futures and thread pools:

```
#!/usr/bin/python3

import os.path
import time
import logging
import threading
from perfflowaspect import aspect

from concurrent.futures import ThreadPoolExecutor
from time import sleep

pool = ThreadPoolExecutor(3)


def bar(message):
```

(continues on next page)

(continued from previous page)

```
    aspect.async_event("before", "bar", filename)
    sleep(3)
    aspect.async_event("after", "bar", filename)
    return message


def foo():
    aspect.sync_event("before", "foo", filename)
    time.sleep(2)
    future = pool.submit(bar, ("hello"))
    while not future.done():
        sleep(1)
    print(future.done())
    print(future.result())
    aspect.sync_event("after", "foo", filename)


def main():
    foo()


if __name__ == "__main__":
    filename = os.path.basename(__file__)
    main()
```

4.6 Upcoming Features

Upcoming features include the ability to specify categories while tracing, a connector for Hatchet, and allow for collection of other statistics, such as GPU utilization. Additionally, the team plans to provide examples of how various benchmarks and workflows have been annotated with PerfFlowAspect. Please follow our GitHub issues page for learning about upcoming features, as well as for suggesting new features for PerfFlowAspect.

4.7 Developer's Guide

This is a short developer guide for using the included development environment. We provide both a base container for VSCode, and a production container with PerfFlowAspect you can use for application development outside of that. This is done via the .devcontainer directory. You can follow the DevContainers tutorial where you'll basically need to:

1. Install Docker, or compatible engine
2. Install the Development Containers extension

Then you can go to the command palette (View -> Command Palette) and select `Dev Containers: Open Workspace in Container.` and select your cloned PerfFlowAspect repository root. This will build a development environment. You are free to change the base image and rebuild if you need to test on another operating system! When your container is built, when you open `Terminal -> New Terminal` you'll be in the container, which you can tell based on being the "vscode" user. You can then proceed to the sections below to build and test PerfFlowAspect.

Important the development container assumes you are on a system with uid 1000 and gid 1000. If this isn't the case, edit the `.devcontainer/Dockerfile` to be your user and group id. This will ensure changes written inside the container are owned by your user. It's recommended that you commit on your system (not inside the container)

because if you need to sign your commits, the container doesn't have access and won't be able to. If you find that you accidentally muck up permissions and need to fix, you can run this from your terminal outside of VSCode:

```
$ sudo chown -R $USER .git/
# and then commit
```

4.7.1 Installing PerfFlowAspect

Once inside the development environment, you can compile PerfFlowAspect:

```
export PATH=/usr/local/cuda-12.1/bin/:$PATH
cd src/c
mkdir build
cd build
cmake -DCMAKE_CXX_COMPILER=clang++ -DLLVM_DIR=/usr/lib/llvm-10/cmake ..
make
sudo make install
```

If you want to run tests, cd to where the tests are, and run a few!

```
cd src/c/build/test
./t0001-cbinding-basic.t
```

Note that if you don't have a GPU, these probably will error (I do not)! Here is how to run Python tests:

```
cd src/python/test
./t0001-pybinding-basic.t
```

You'll again have issues without an actual GPU. And that's it! Note that we will update this documentation as we create more examples.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search