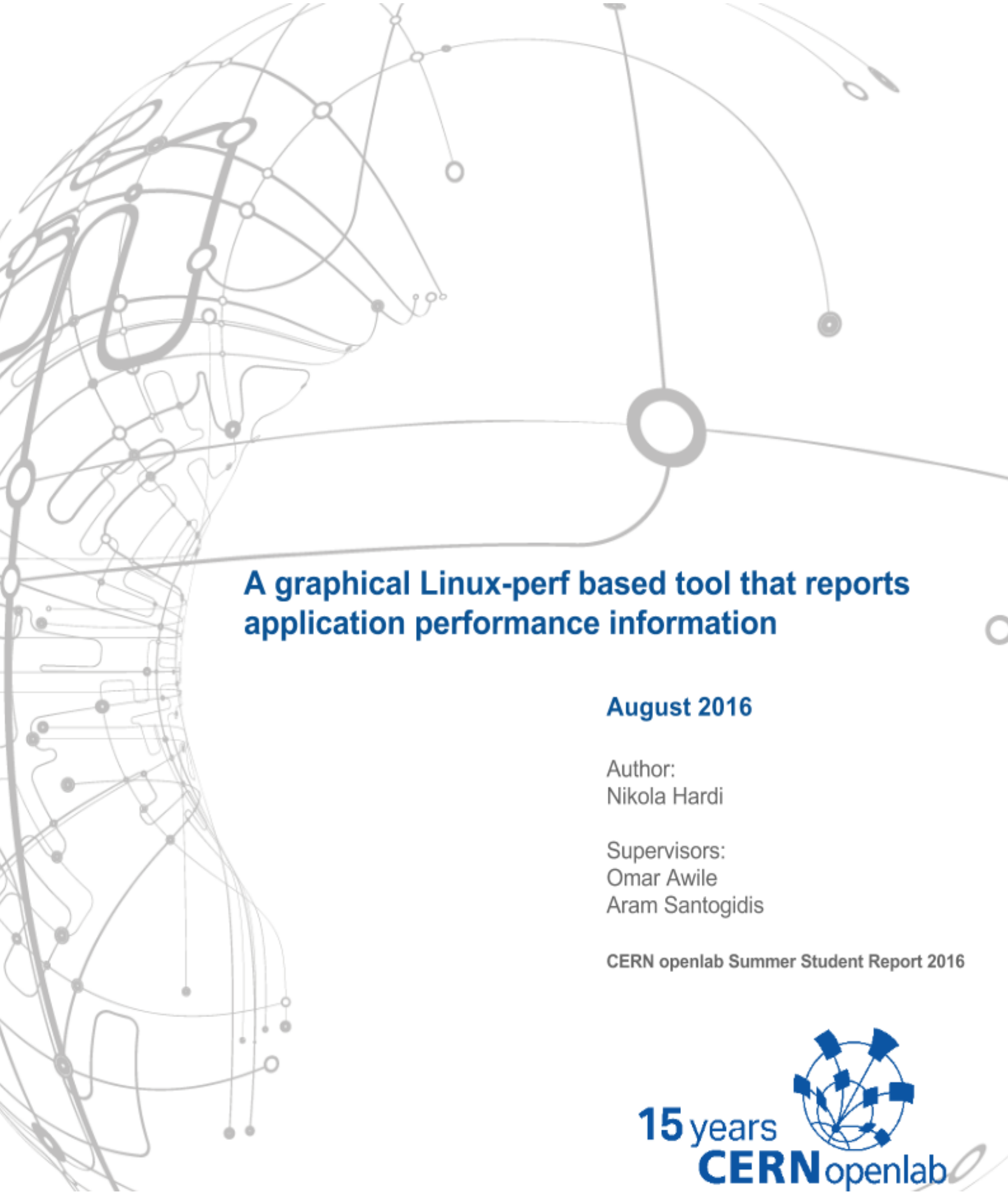

A graphical Linux-perf based tool that reports application performance information

August 2016

Author:
Nikola Hardi

Supervisors:
Omar Awile
Aram Santogidis

CERN openlab Summer Student Report 2016

15 years CERN openlab

Project Specification

Application performance is often assessed using the Performance Monitoring Unit (PMU) capabilities present in modern processors. One popular tool that can read the PMU's performance counters is the Linux perf tool. Perf, however, requires in-depth knowledge of performance counters, the supported events on the platform's CPU and their meaning. Also, perf returns measurements in simple plain text (or binary) format and leaves the task of visualizing, analyzing and interpreting the results up to the user.

The aim of this project is to develop a tool that allows the user to more easily profile an application using perf. This tool shall act as a front-end to perf, abstracting the specific details of supported PMU events on the target architectures. It will also return simple and clear graphs and summaries helping the user to interpret the results. Such a tool can significantly boost the productivity of application developers who are engaged with application performance monitoring at CERN and elsewhere.

In the scope of this project we would like to focus on the visualization and analysis of the application profile over time, further differentiating this tool from other existing profilers.

Abstract

The aim of this project was to implement a tool for visualizing data from the PMU (Performance Monitoring Unit). The implemented tool is built on top of the Linux perf system and pmu-tools. The implemented solution integrates perf tools more tightly. It provides an interactive graphical user interfaces and extends already available tools with completely new features such as data visualization, storing benchmark data and collaborative work. The tool is implemented as a web application which can be run remotely and accessed by multiple users. It is built using modern technologies such as AngularJS, Flask, Bokeh, Pandas etc. At the end of this programme the solution can be used for running benchmarks and visualizing the recorded data, as well as reviewing past benchmark runs. It provides access to the perf list, perf stat, perf record and perf script tools. Output of the perf stat tool can be streamed and analyzed in real time. In this report I describe the motivation behind this project as well as my contributions that address the requirements outlined above. Finally, some ideas for future work are proposed.

Table of Contents

Project Specification
Abstract
Table of Contents
Introduction
    The importance of application performance profiling
    Existing tools for accessing PMUs on Linux
    Proposed solution
Perf tools
        MSR registers and raw event codes
        Third party tools
Use Cases
    Typical use case
    Examples of web ocperf use cases
        Offline stat
        Offline record
        Streaming stat
        Streaming record
Architecture of the solution
        Technologies
    Architecture of the backend
        Architecture of the frontend
        Role of the utility modules
Results
        Perf list widget
        Perf tool chooser
        Workload widget
        Plot widget
        Benchmark runs history
        Sessions screen
Future work - what next
References

Introduction

The Importance of application performance profiling

Although choosing suitable algorithms and data structures should be the developer's highest priority, one has also to be able to efficiently utilize the hardware in order to achieve good performance. Optimising too early can lead to various problems during development so the common approach is to implement a straightforward solution, and then trace and measure program execution to determine bottlenecks and hotspots.. Sometimes a single (or few) hotspot can be found and improved, but this is not always the case. Even if execution inefficiencies or scalability issues cannot be traced to a single bottleneck or hotspot, it is often still possible to further optimize the code, albeit at a higher cost. The process of analyzing program execution efficiency is called profiling and tools for doing it are called profilers.

There are two common profiling techniques, *instrumentation* and (*statistical) sampling*. Instrumentation based approaches are able to precisely keep track of the executed instructions, taken branches or monitor accessed memory locations But as they assume inserting additional instructions into the observed program, they are systematically adding overhead to the results. The overhead doesn't only slow down the execution and add systematic errors, but also distorts recorded results as there are more instructions to be cached, the program execution trace can be changed and even bounds can be shifted from memory to CPU.

Sampling based approaches are present since the early days of computing at least in the form of sampling the program counter or stack frame [1]. Conventional tools generally provide a rough estimate of how much of execution time is spent in which part of the code. This information may be insufficient for an in-depth analysis. In the last 20 years there is an expansion of hardware support for this kind of profilers [2]. Modern CPUs come with builtin support for periodically counting various hardware events and provide infrastructure for monitoring and reading these counters. This feature is the so called *Performance Monitoring Unit* (PMU) and is integrated inside the core of a CPU or outside in which case it is used for monitoring uncore events. PMUs provide insight into CPU internals which was previously unobservable. It is common for the CPUs to issue multiple instructions concurrently and out of order, pipelines are deep (more than 5 stages) and cache structures are becoming increasingly complex. To understand how the program interacts with such complex system, we need both hardware support and tools for accessing it.

Existing tools for accessing PMUs on Linux

Support for PMU events exists in the Linux kernel since version 2.6.x. The support is present as a kernel subsystem (perf_event), a system call for accessing it from user space and user space tools. All of these components are described in greater detail further in the text.

Alternatively there are proprietary tools such as Intel Amplifier XE (also known as VTune). VTune Amplifier provides a rich set of performance insight into CPU & GPU performance, threading performance and scalability, bandwidth, caching and much more. Analysis is faster and easier because VTune Amplifier understands common threading models and presents information at a higher level

that is easier to understand. It also offers powerful analysis tools to sort, filter and visualize results on the timeline and in the source code.

Perf is a set of open source tools which are part of the Linux kernel. These tools are mature and are regularly used in many software projects. However, some issues have been identified as major obstacles for efficiently using perf, which is due to the fact that the set of tools it provides are not well integrated. They are suitable for scripting and command line work, but users often are forced to implement their own tools on top of perf in order to overcome some of its limitations. . There are already community projects which are built on top of the Linux perf, for example pmu-tools [3], which this work is based on.

Proposed solution

In this project we propose a solution to the aforementioned problem by creating a web-interface for the perf tools. It provides better integration of these tools, adds interactivity and provides real-time data visualization for perf benchmarks. Our solution parses raw data into the Pandas [4] DataFrame format which is a more powerful, user friendly and modern format for data analysis. Also, as running benchmarks involves repetitive tasks and experimentation, recorded data quickly accumulates. Our web application provides also features for profiling data storage and management.

Perf tools

The perf suite of tools or simply perf is spread through the system, including interrupt handlers, system calls, kernel support and user-space tools.

Getting PMU data involves configuration steps and periodical readouts. Here is an example of these steps, starting from the configuration. Perf tools execute similar steps.

1. **configuration** :: The perf event subsystem in Linux kernel is exposed through the POSIX API as file descriptor. This file descriptor is obtained by issuing a system call with a configuration as argument. This configuration is then transferred to the perf event subsystem of the Linux kernel and propagated down to configuration registers inside PMUs. It is important to mention that PMUs can be configured to keep track of counters for only one thread at a time, process or CPU as well as the whole system altogether.
2. **counting** :: Although perf can track both software and hardware events, , the ones that are sampled by PMUs are hardware events. PMUs can be abstracted as black boxes with a counting register, and a control register. The control register is used for selecting which events we are counting, and the counter register counts occurrences of those events.
3. **interrupts** :: Each time an event occurs, it will increase the value in the counter. This may not be the case for more frequent events but generally holds true. When the counter reaches a certain threshold value or a timer interrupt is triggered, the value of the counter is recorded and the counter is reset. The interrupt routine copies the value from the counter into a buffer, and notifies the reader process in user-space that there are new samples ready. The reader will be notified through the above mentioned file descriptor obtained during initialization and configuration phase.

4. **reading values** :: When the reader is notified that new samples are ready, it will fetch them. Depending on the tool and implementation, the new sample will be stored or presented.

Detailed description of perf tools (with usage and output)

- **list** :: Prints the list of events with symbolic names. There may exist some events that are supported by hardware but not listed by this tool. These events can be chosen by raw event code (more details later).
- **stat** :: Perf stat is a simple tool which produces useful output without any configuration except command of process to be monitored. In the case that events list is not specified, perf stat will configure PMUs to keep track of some common events, execute given process, wait until it terminates and produces overview of event counts in human readable textual format. Users can also provide list of events to be recorded both in symbolic or raw codes format. Also, there is the possibility to report the counts periodically, based on an interval chosen by the user. It is important to keep in mind that this interval cannot be shorter than 100 ms. Perf stat supports also outputting data in CSV format.
- **record** :: Perf record captures more information than perf stat and is considered the more advanced tool. With perf record it is possible to capture more information about each recorded sample. For example, the value of the instruction pointer. This tool can record at frequencies of up to 25 KHz, while perf stat samples at frequencies of up to 10 Hz. Perf record will store all samples in a binary "perf.data" file, which stands for the default filename but is also a synonym for the file format. The structure of this file format is described in the "perf file format" report [5]. Note that perf record is only used for recording samples and doesn't produce any output besides the "perf.data" file.
- **report** :: Perf record is a tool for reading and presenting the data stored in a "perf.data" file. It offers multiple user interfaces (text, ncurses, gtk). Perf report can show profiling data as flat profiles which allows exploring the target application's callgraphs and allows even code annotation.
- **script** :: perf comes with multiple python and perl scripts for profiling data analysis. The list of all available scripts can be obtained with `perf script -l`. By default perf script will print out a chronological list of all recorded samples. The output is not in CSV format, but is tabulated and can be easily parsed.

MSR registers and raw event codes

There is a set of standard events which are present in almost all CPU models, but also there are some events that are not common. Some of these events are cycles, instructions, page-faults, branch instructions, or branch misses. Each generation of CPUs introduces some new types of events and more performance counters. In example, Pentium III had only 2 performance counters while Pentium 4 had 18. Each event type is identified by it's raw code. This raw code is stored in the PMU's control register during the configuration procedure. Perf comes with list of symbolic names for many events and it translates these symbolic names into raw codes. Some events are not known to perf and don't have their symbolic names, but they may be supported by given hardware. Because of this, perf lets users request events by their raw codes. The codes can be obtained from the Intel Software Developer Manuals Vol 3B, Section 19. The syntax for specifying raw event code is similar to the syntax for using symbolic names but instead of using the name of the event, the format rNNN is used where

NNN is hexadecimal number representing the event code. There are 3rd party tools like ocperf which address this problem of translating symbolic names to raw codes.

Third party tools

The pmu-tools is a collection of 3rd party tools developed by Andi Kleen [3]. In this report ocperf is particularly important, because it provides symbolic names for machine specific events. Ocperf uses modules from pmu-tools for downloading emaps or information about machine specific events. An emap is a dictionary which translates symbolic name of an event into it's raw event code. Descriptions of the events are also available.

Use Cases

Typical use case

There are multiple workflows for using perf. Usually, it involves running perf list and choosing list of events and/or reading the CPU datasheet and finding raw codes of events.

Example perf list output:

```
$ perf list

List of pre-defined events (to be used in -e):
  cpu-cycles OR cycles                                [Hardware event]
  instructions                                        [Hardware event]
  cache-references                                    [Hardware event]
  cache-misses                                        [Hardware event]
  branch-instructions OR branches                     [Hardware event]
  branch-misses                                       [Hardware event]
  bus-cycles                                          [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend     [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend       [Hardware event]
  ref-cycles                                          [Hardware event]

  cpu-clock                                           [Software event]
  task-clock                                          [Software event]
  page-faults OR faults                               [Software event]
  context-switches OR cs                              [Software event]
  cpu-migrations OR migrations                        [Software event]
  minor-faults                                        [Software event]
  ...
```

After choosing the events, a user can run some of tools for recording (stat/record). In case of perf stat, no further action is needed, but in case of perf record it is. It's because perf record stores recorded samples into the perf.data file. When perf stat is called periodically, it will output timestamps too.

Example output of perf stat (overview, non-periodic):

```
$ perf stat -- sleep 5

 Performance counter stats for 'sleep 5':

          0.718446 task-clock (msec)         #    0.000 CPUs utilized
```

```
             1 context-switches          #    0.001 M/sec
             0 cpu-migrations            #    0.000 K/sec
           185 page-faults               #    0.258 M/sec
     1,441,830 cycles                    #    2.007 GHz
     1,083,729 stalled-cycles-frontend   #   75.16% frontend cycles idle
       902,449 stalled-cycles-backend    #   62.59% backend  cycles idle
       691,362 instructions              #    0.48  insns per cycle
                                         #    1.57  stalled cycles per insn
       134,220 branches                  #  186.820 M/sec
         8,029 branch-misses             #    5.98% of all branches

   5.001196571 seconds time elapsed
```

Example of perf stat (periodic):

```
$ perf stat -I 1000 -- sleep 5
#           time             counts events
     1.000246678           0.367131 task-clock (msec)
     1.000246678                  1 context-switches
     1.000246678                  0 cpu-migrations
     1.000246678                176 page-faults
     1.000246678            909,502 cycles
     1.000246678            609,994 stalled-cycles-frontend
     1.000246678            495,718 stalled-cycles-backend
     1.000246678            557,346 instructions
     1.000246678            105,021 branches
     1.000246678              6,018 branch-misses
```

Perf.data file format is described in the previous openlab report [5]. It can be parsed by perf report or perf script. Perf script is a suitable tool for our purpose because it's output provides timestamp and value and even PC value for each sample. This lets us visualize recorded samples as a line plot.
Now when underlying tools are described they can be combined and new tools can be built on top of them. We are describing only implemented workflows and leave the rest for the "future work" section.

Offline perf stat plot

In this workflow a user interacts with the perf list and perf stat tools through dedicated widgets. After the setup process (choosing events, sampling period etc) the job will be spawned on the machine running the tool. While the samples are recorded periodically they will be combined and visualized only after the job ends. This means that the user has to wait for the job to finish to see any data.

Offline perf record plot

Similar to the previous one, this workflow lets the user interact with the perf list for listing and choosing events, but for recording the perf record tool is used instead of the perf stat. A new job will also be spawned and recorded data will be stored into a file. The difference between using perf record and perf stat is that perf record stores data in binary form and the perf script has to be used. When the job finishes and after the recorded data is stored into the file and read with the perf script, a plot can be produced. The user has to wait for the plot until the end of the execution of the job.

Streaming the perf stat output

A major disadvantage of the two approaches above is that the user needs to wait for the job to finish before the visualization is generated. For long running tasks there may be useful data even before the end of execution of the job. In this workflow the output of perf stat is presented on the plot as it is produced by the perf stat. The obvious advantage of this approach is that the user gets data in real-time and doesn't have to wait for the process to finish.

Architecture of Solution

Web ocperf is a web application that consists of 4 main modules, as it is illustrated in the figure 1. The web page is an interface for backend modules. Backend and frontend communicate through HTTP requests as any other standard web application. The backend controls the benchmark process, parses and transforms data and controls the plotting server. The plotting server serves raw data for plots. Plot widgets on client side and plotting server communicate through the websockets [6].

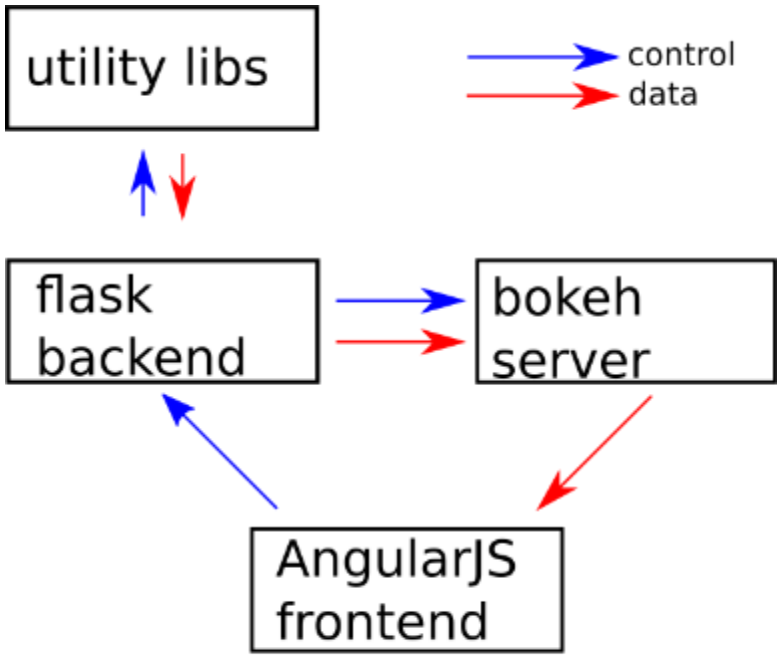

Figure 1: architecture overview

When the user creates a new benchmark and runs it in the frontend, an HTTP request is issued to the backend. The backend will spawn a new workload process upon receiving the request. New processes are created using utility functions. Creating a new workload process also creates a new session on the plot server. After the session is created, an HTML tag representing the plot is sent to the frontend as a response. Consecutively the bootstrapping process of the plot widget takes place which will connect to the plot server and request the data to required for the plot. Communication between the plot widget and the plot server is done through a websocket.

Technologies

- **pmu-tools/ocperf,** a tool built on top of perf. It is used for obtaining the emaps, extended list of the PMU events [3].
- **Pandas,** a data analysis framework for Python. It is used for parsing the CSV output into the DataFrames and later as input for plots [4].
- **Bokeh modules,** are part of plotting framework for Python. are used for plotting, embedding plot widgets in the client and controlling the plotting server (bokeh server) [7].
- **Flask (web microframework)** is used for dispatching the HTTP requests to the handlers [8].
- **Sqlite (database), peewee (ORM) and marshmallow (serialization)** are used for the data persistency and serialization [9], [10], [11].
- **bokeh server**, implements real-time streaming plots [12].
- **AngularJS & Bootstrap** :: Frontend is based on the AngularJS and Bootstrap frameworks [12], [13].

Architecture of backend

The backend module implements handlers for two types of endpoints:

- API
- Static files

Description of API endpoints:

Emap endpoint is rather simple, it returns list of all available events with their descriptions on the given machine on which the web_ocperf is running. Implements only GET request.
Session endpoint implements both POST and GET requests. On GET request it responds simply with a list of session objects in the database. On POST request, it will create and run a new benchmark. Depending on the state object, it will run necessary tools (perf stat or perf record) and communicate with the bokeh server to create a new session and document.
Benchmark endpoint implements only GET request handler and depending on the chosen format, it will return javascript object with frontend state and bokeh autoload_server script, the html page with only bokeh plot widget for the given benchmark run or raw perf data.

Although the session endpoint starts the benchmark run, the majority of implementation of setting up tools and running the job is delegated to utility functions in utils modules.

Architecture of frontend

The frontend is implemented as a very simple AngularJS application, without any project builders, package managers. All code resides in a single JavaScript file as there is no benefits of advanced code organization. Built-in angular router ties together the homepage with the session screen template and controller, and session/<uuid> shows the benchmark screen with its controller for that particular session.

Role of the utility modules

Ocperf utils module implements the function for communicating with ocperf and further with perf. It provides following functionalities:

- Building perf command

- Getting list of supported events (ocperf + native perf)
- Running perf (both stat and record)
- Parsing output of perf stat and perf.data
- Handler for parsing the streaming output of perf stat

Plot utils module relies on the bokeh library for creating the plot from the recorded and parsed perf data. The container for data to be visualized differs depending on whether the recorded data will be streamed to the plot widget in real time or all data will be recorded and visualized afterwards at once. In the first case a ColumnDataSource is used and new samples are appended to it while in the second case ordinary Pandas' DataFrame is used. In both cases each row contains the timestamp, event type and value. When perf record is used, there is additional information like the value of instruction pointer.

Streaming utils module abstracts out the process of creating the streaming results benchmark. The entry point of this module is the blocking_task() function which will start perf as a subprocess and read lines on stdout. For each line, it will parse it and schedule callback run on next bokeh tick. That callback will append new samples to the ColumnDataSrouce which is attached to the current plot as a source. Pushing data from backend to ColumnDataSource will actually push the data to bokeh server, and then to the frontend. This means that a connection must be maintained between a bokeh session thread on the bokeh server. For this reason a new subprocess is spawned for every new session created. When the session is closed, this thread dies.

Persistent storage. Raw perf.data and and logs for running perf stat are stored as files under the logs directory. Metadata for benchmarks (frontend benchmark screen state, session, uuid, date) and sessions (title, uuid, date) are stored in an sqlite database and accessed through the peewee models. The python marshmallow library is used for serializing the model objects and transforming them into json objects.

Results

The web application has two main screens, the benchmark and session management screens, represented in the figures 1 and 2.

Typical perf record and perf stat commands consist of 3 variable parts:

1. Tool name (stat or record) with additional arguments like frequency, CSV output, etc.
2. List of events
3. Workload string

A user can choose values for these parts through the widgets of web interface. Tool choice, event list and workload command are marked in Figure 1.

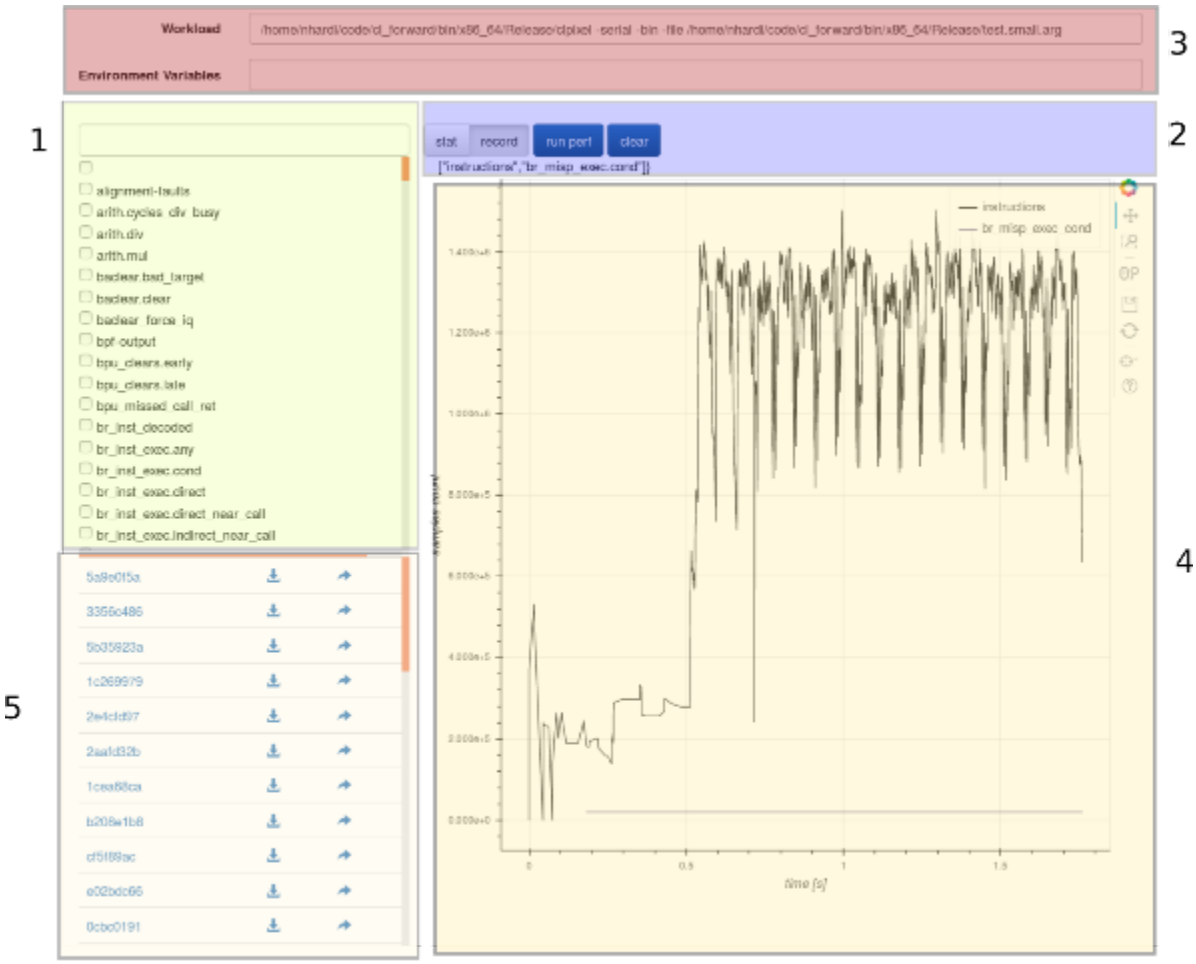

Figure 2: the benchmark screen

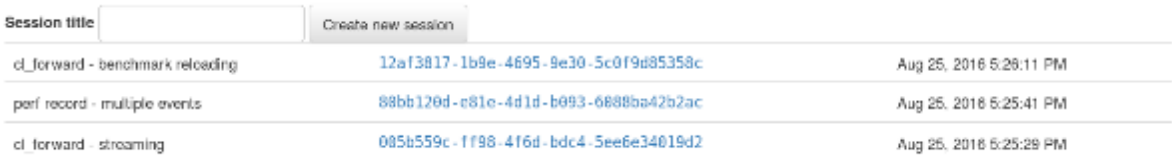

Figure 3: the session screen

Perf list widget

This widget connects perf list, stat and record commands. It consists of list of the events with checkboxes. On hover, the description will pop-up. Multiple events can be chosen for a certain benchmark. Above the list there is search field which will filter out events in the list by the entered search term. It will consider both names of the events and their descriptions.

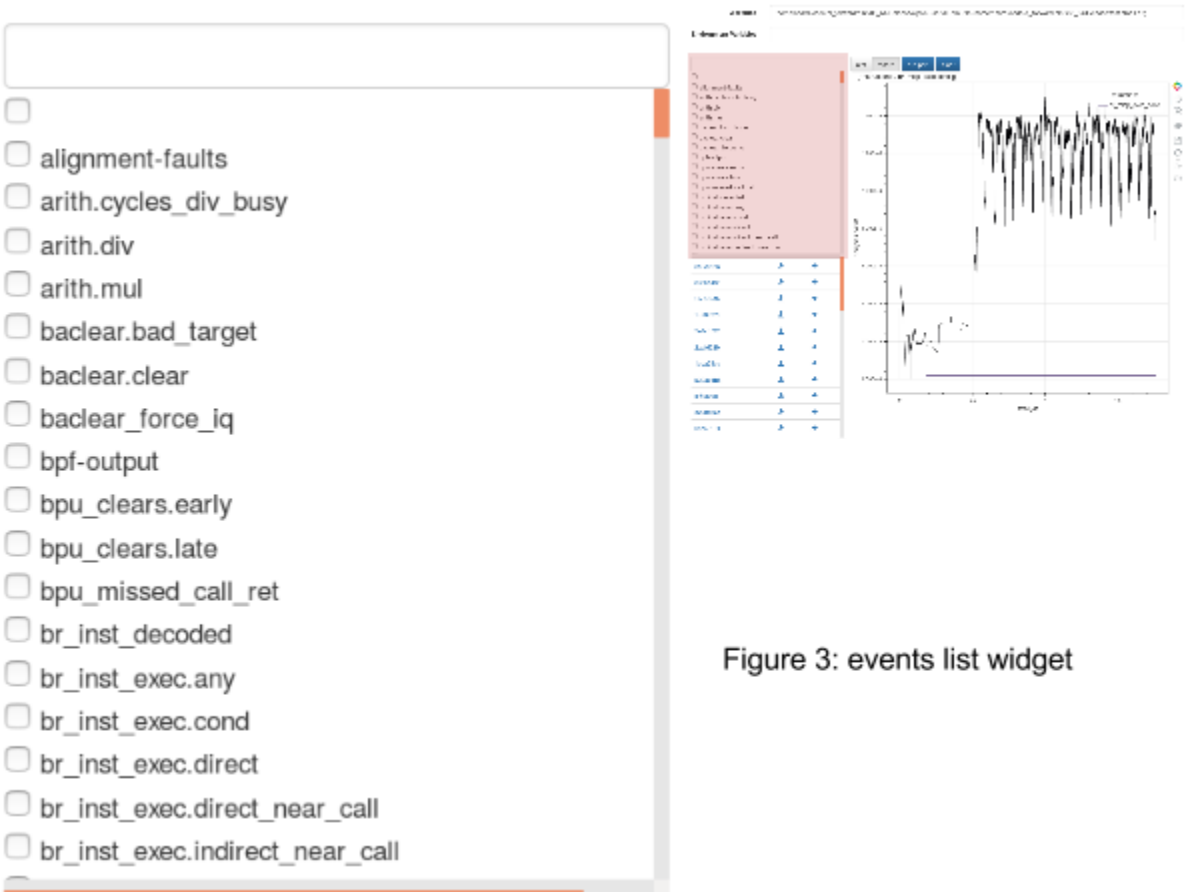

Figure 3: events list widget

Perf tool chooser

This widget provides user with the ability to choose between perf stat and perf record. As perf stat supports real-time plot updates, the “streaming” checkbox is also present.

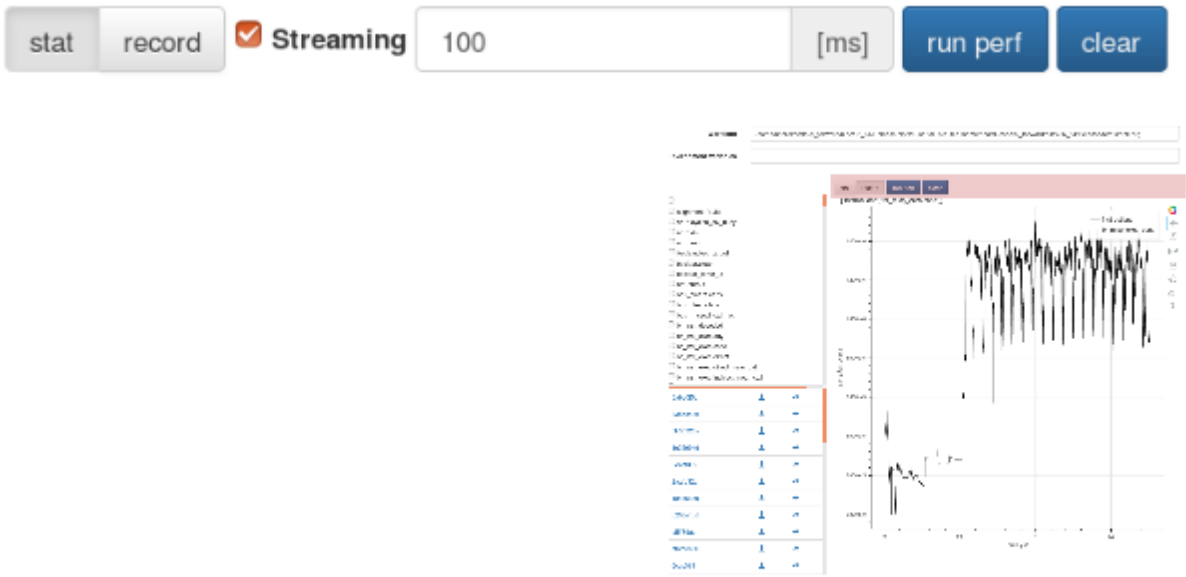

Figure 4: perf tool widget

Workload widget

A workload consists of two input fields. The first one is for entering command, and the second one is for setting environment variables. The value of the second field will be prepended to the final perf command and a new subprocess will be spawned through bash. The format for setting environment variables is the same as for inline environment variables in the bash. An example of setting an environment variable is given in the figure 5 where the *COUNT* variable is set to value 7. Multiple variables can be set inline (e.g. `COUNT=7 THREAD_NUM=12` will be assembled in the final command to `COUNT=7 THREAD_NUM=12 ./workload`).

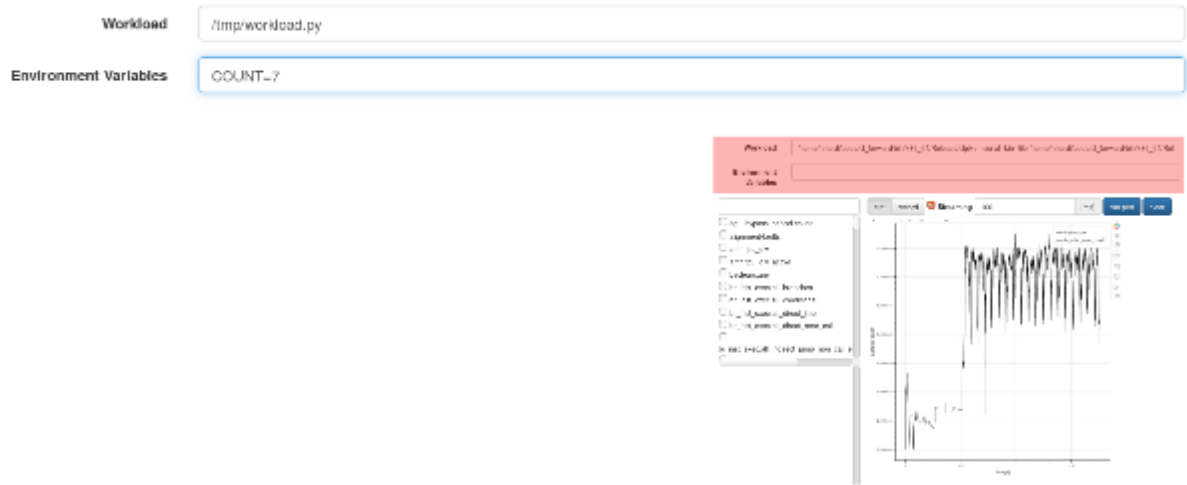

Figure 5: workload widget

Plot widget

The bokeh plot takes the most space of the benchmark screen. It is a standard bokeh plot which means that it's an interactive widget. Standard set of tools for exploring the plot is present. This set includes pan, block zoom, scroll zoom, hover tooltips, and saving bitmap plots of the image, as in Figure 6.

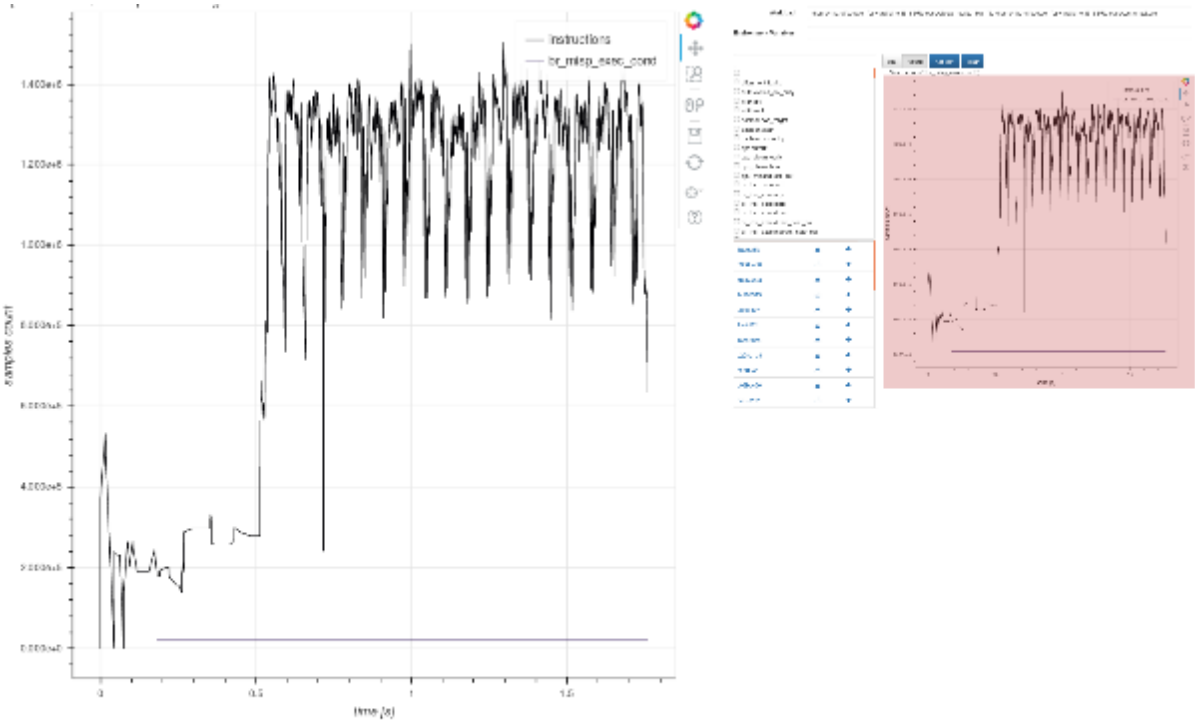

Figure 6: plot widget

Benchmark runs history

This widget doesn't occupy much space, but provides a lot of functionality. Each time a new benchmark is executed, a new entry will be added to the list in this benchmark history. Each entry in the list has links for reloading that plot, download raw "perf.data" and permanent link to full screen plot. Reloading old benchmarks will not only reload the plot widget but it will also reload the whole state of the benchmark screen including selected events, workload string and environment variables. After the benchmark screen state is reloaded, a new benchmark based on the previous run can be executed.

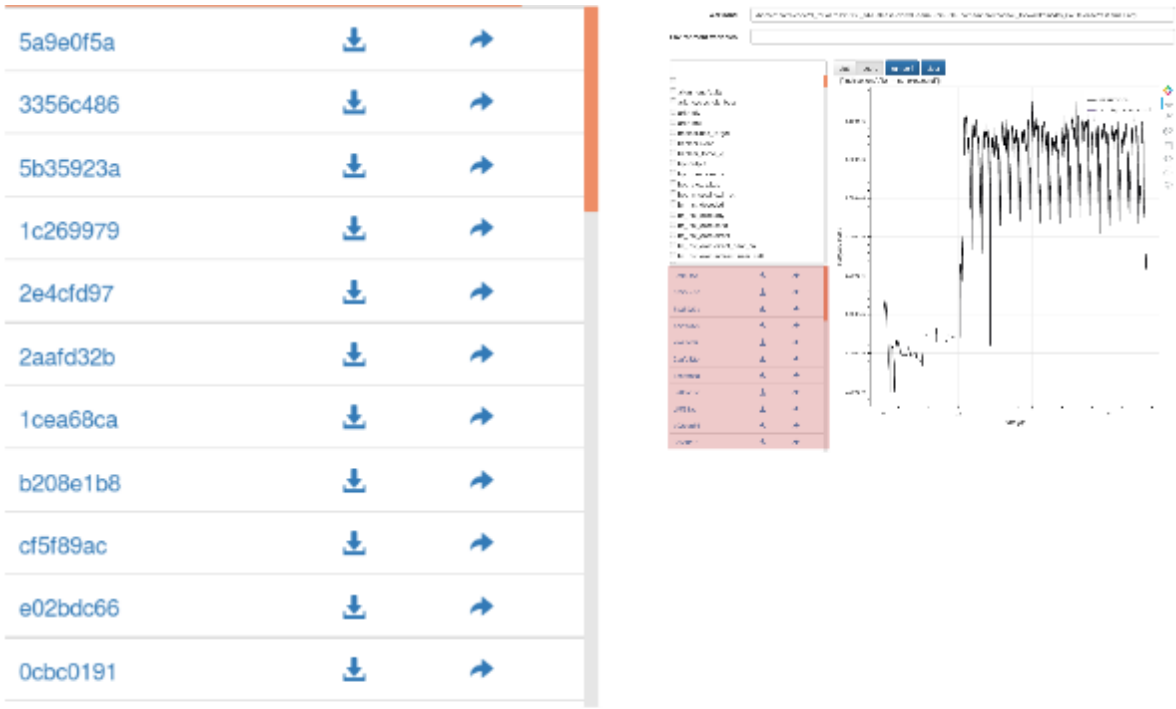

Figure 7: benchmark history widget

Sessions screen

A benchmark always belongs to a session. A session consists of multiple benchmarks. On the session screen new sessions can be created, and previous ones can be searched through and accessed. The session screen is represented in Figure 2.

Future work

Although the web interface already provides some nice features, it has the potential to be extended and encompass more use case scenarios. A short list of problems and proposed solutions follows:

Asynchronous communication between the web frontend and the backend once the benchmark process is run.
Given a benchmark is being run, the web interface becomes non-responsive. The web interface waits on a HTTP response from the backend server after the benchmark task is finished. When benchmark tasks are short (few seconds), this isn't a big issue, but if the benchmark task takes a few hours, there should be at least some way of interactivity. For example, the user should be informed which link to follow to get results once the benchmark is finished.

Attaching to already running processes

The perf tools are able to "attach" to running processes. For example, a database server or a web server are long running processes and a user should be able to get PMU data for them without the need to start or stop them. This can be achieved by passing the process ID or thread ID flag and the ID value of the target process instead of the workload command. This feature is supported by perf stat and perf record commands and is missing only in the web interface.

Removing both sessions and benchmarks from the web interface

Once a session is created, it can't be deleted through the web interface. The same holds true for created benchmarks in a session. The recorded data can grow big and there may be need to remove benchmarks which are not needed. At the moment of writing this report there is no easy way for removing benchmarks from the system because both raw data and metadata from the SQLite database need to be removed.

Downloading recorded benchmark data in the HDF5 format

Currently benchmark data is permanently stored either as a log file in textual form or in binary "perf.data" format. The pandas' DataFrame format is used for internal data processing. The data in this DataFrame format is suitable for later data analysis and can be valuable to the users. As this format is already available in the system there is only small overhead for implementing the feature which would let users download the recorded data as DataFrames. HDF5 format is a binary container format which is well supported by many tools, including libraries as Pandas and other data analysis frameworks. HDF5 is a hierarchical format so it can store the data for all benchmarks from a session in a single file.

Viewing stdout and stderr output from the web interface

An observed program can crash because of invalid arguments or because of any other reason. In that case the program usually reports error by writing an error message to standard error output (stderr). In the current implementation there is no error log reporting in the web frontend, although the error log will be printed during the execution in the terminal running the backend and the plot servers.

Most used events

The list of available PMU events becomes longer and longer with each generation of CPUs. Some of these events are used more frequently than others. Keeping track of the most used events can make the events list widget more usable. The list of selected events is already stored for each benchmark as part of the web frontend state. This data can be used to count how many times was each event selected in benchmarks of one session. The most used events could be presented at the top of the list in the events list widget.

Reloading only data in the plot widget instead of reloading whole plot widgets

In the current implementation the old plot widget is removed from the page and the HTML tag for new plot is fetched upon each new benchmark run or old benchmark reload. When a new plot tag is fetched and embedded into the page it will go through the bootstrapping process and it will fetch the data from the plotting server afterwards. The bootstrapping process of the plot widget takes significant time. The Bokeh library supports pushing new data from the plot server to the plot widget. This

feature can be used to avoid the bootstrapping process of the plot widget and make the web interface more responsive to changes.

Benchmark runs queueing
A benchmark run can take more than few hours or even days. Waiting for one benchmark run to be finished so the next one can start represents a problem. Letting users schedule multiple benchmark runs which will then be executed serially is a significant improvement to the user experience. Also, as simultaneous benchmark runs should not be allowed introducing the queueing mechanism would solve the issue of multiple users trying to run benchmarks simultaneously.

Histogram plots
The simplest way to visualize a time series data is to plot a dot for each data record. Although such visualization is valuable for data analysis, there are many other ways to visualize the same data. For example, histogram plots are a common way to take a different point of view on a data set. Bokeh and Pandas support straightforward way for producing histogram plots.

Streaming perf record
The current implementation supports streaming only for perf stat. Supporting streaming plots for perf record would take significant time to be implemented. The main difference between these tools is that perf stat naturally streams data to the standard output while perf record stores data into a binary file for later processing. By default, perf record will buffer the recorded data and flush it periodically. There is the -D (--no-delay) flag for perf record which will prevent this buffering effect but the problem of parsing this file on the fly remains unsolved. Running perf script while perf record is writing to the perf.data file is not working at the time of writing this document. The pmu-tools repository has an implementation of perf.data parser written in Python. There may be possibility of implementing an asynchronous reader and parser of perf.data file. Limitations of the file the perf.data file format for such feature were not investigated. For example, the data in the header of this file may be needed for parser but is written and available only after perf record exits.

Conclusion

This tool provides better integration of the perf tools and lets users visualize recorded data right in the same place and in real time. Although there are plenty of features proposed for the future work, it already meets the main goals. This tool can make the process of program profiling with Linux perf faster and easier. Having visualization for each benchmark makes spotting interesting behaviour in profiled programs easier. I hope that some of proposed features will eventually be implemented soon.

While working on this project I had the opportunity to learn more about hardware supported profiling techniques and software infrastructure for reading PMU data. During this project I also had opportunity to learn about history of CERN, experimental physics and research in scientific computing. A special thanks goes to my supervisors Omar Awile and Aram Santogidis who led me through this project and kindly shared their large experience and knowledge with me. I also want to thank to openlab staff without who this program could not be so successful.

References

1. Graham S, Kessler P, McKusick M. gprof. *ACM SIGPLAN Notices*. 2004;39(4):49. doi:10.1145/989393.989401.
2. Methisen T. Pentium Secrets. *Byte*. July 1994;(19 vol 7).
3. Kleen A.. *GitHub - pmu-tools*. 2016. Available at: https://github.com/andikleen/pmu-tools.
4. pandas: Python Data Analysis Library. *pandas.pydata.org*. Available at: http://pandas.pydata.org/.
5. Fassler U. *Perf File Format*. 1st ed. Geneva: openlab, CERN; 2016. Available at: https://openlab-mu-internal.web.cern.ch/openlab-mu-internal/03_Documents/3_Technical_Documents/Technical_Reports/2011/Urs_Fassler_report.pdf.
6. RFC 6455 - The WebSocket Protocol. *IETF*. 2011. Available at: https://tools.ietf.org/html/rfc6455.
7. Bokeh. *bokeh.pydata.org*. Available at: http://bokeh.pydata.org/.
8. Flask. *flask.pocoo.org*. Available at: http://flask.pocoo.org/.
9. SQLite. *sqlite.org*. Available at: http://sqlite.org/.
10. peewee. *GitHub - peewee*. Available at: https://github.com/coleifer/peewee.
11. marshmallow. *GitHub - marshmallow*. Available at: https://github.com/marshmallow-code/marshmallow.
12. AngularJS. *angularjs.org*. Available at: http://angularjs.org.
13. Twitter Bootstrap. *getbootstrap.org*. Available at: http://getbootstrap.org.