VisPerf: visualize and compare performance of stream processing applications executed on CPU

Claudio Scheer* Dalvan Griebler† Isabel Harb Manssour‡

Pontifical Catholic University of Rio Grande do Sul - PUCRS
Brazil

ABSTRACT

Tracking performance metrics of applications is indispensable to identify bottlenecks or even where resources should be applied to fix performance issues. Perf is a tool that captures performance metrics from the hardware where the application was executed. However, data captured by Perf can be hard to read and, in some cases, impossible due to the large size of the generated files. Therefore, we propose in this paper a visualization dashboard for data captures using Perf: VisPerf. VisPerf implements a data pipeline from capturing to visualizing data. The dashboard allows the user to compare specific experiments and to see how hardware events varied between these experiments. VisPerf is still in the prototype phase. Therefore, requires some validations, especially with application developers who may take benefits from VisPerf.

Index Terms: Human-centered computing—Visualization—Visualization techniques—Treemaps; Human-centered computing—Visualization—Visualization design and evaluation methods

1 INTRODUCTION

2 PERF

Perf is a profiling tool included in the Linux kernel. It can be used to capture CPU performance counters, tracepoints, kprobes[1], and uprobes[2] [2]. Perf also supports software events, such as page faults[3]. In turn, tracepoints are placed in the source code to collect timestamps and stack traces, and performance counters are hardware registers that count events of the CPU, such as instructions executed and cache misses. Perf has low overhead, as it is integrated into the kernel. Moreover, Perf supports counters in different architectures. Therefore, it can be used to profile applications in different Intel architectures or even in AMD CPUs.

There are several subcommands available in Perf. Below we describe some of the subcommands:

- `perf stat`: gather performance counter statistics from a command;
- `perf record`: run a command and record its profile for later analysis. By default, profile is saved in a file named `perf.data`;
- `perf report`: read `perf.data` and display the profile recorded in this file;
- `perf script`: read `perf.data` and display the trace recorded in this file;
- `perf top`: display performance counter profile in real time;
- `perf list`: list events available to be captured in the current architecture;

`perf list` shows events from several sources, such as hardware and software. Hardware events are provided by the processor and its PMUs (Performance Monitoring Unit), which may vary among different CPU providers. It is important to note that our experiments were executed on a Intel Xeon architecture. Next, we discuss the counters captured in our experiments [1].

- **cpu-cycles**: number of fetch-decode-execute cycles the CPU executed;
- **instructions**: number of instructions the CPU executed;
- **cache-misses**: number of cache-misses;
- **cache-references**: number of references found in cache;
- **L1-dcache-load-misses**: number of misses when loading data from L1 cache;
- **L1-dcache-loads**: number of data loads from L1 cache;
- **L1-dcache-stores**: number of data stores made in L1 cache;
- **L1-icache-load-misses**: number of misses when loading instructions from L1 cache;
- **L1-icache-loads**: number of instructions load from L1 cache;
- **L1-icache-stores**: number of instructions store made in L1 cache;
- **LLC-loads**: number of references loaded from LLC (Last Level Cache, usually L3);
- **LLC-load-misses**: number of references missed when loading from LLC;
- **LLC-stores**: number of references store made loading in LLC;
- **mem-stores**: number of stores made in the main memory;
- **mem-loads**: number of loads made from the main memory;
- **branch-misses**: number of mispredicted speculative branches;
- **branch-instructions**: number of correct speculative branches;
- **bus-cycles**: number of CPU cycles needed to fetch or write data to the main memory, for example;

*e-mail: claudio.scheer@edu.pucrs.br
†e-mail: dalvan.griebler@pucrs.br
‡e-mail: isabel.manssour@pucrs.br

[1] `https://www.kernel.org/doc/html/latest/trace/kprobes.html`

[2] `https://www.kernel.org/doc/html/latest/trace/uprobetracer.html`

[3] Page faults occur when virtual memory is not mapped to physical memory. One of the reasons for this is when data must be 'swapped'.

For this paper, we profiled the person recognition application. The main goal of this application is to recognize whether there is a person in a specific image using OpenCV[4]. To characterize a stream processing application, this is, it is impossible to predict when data will stop entering the application, we used a video as input.

Towards exploiting parallelism, we use the FastFlow[5] library and compared two mapping policies. In summary, the mapping policy defines in which core threads launched by the application will be placed. Therefore, we profiled the person recognition application using the default mapping policy of the Linux operating system and the mapping policy proposed by FastFlow.

3 VisPerf

Since Perf outputs a lot of data, we propose VisPerf to easily visualize and compare the data captured. The VisPerf pipeline can be seen as three stages: capture, processing and visualization.

In the capture stage, the person recognition stream application is executed using the two mapping policies discussed before and profiled with Perf. We configured Perf to sample 997 times each second the application is running. The raw data from Perf is converted to a CSV file for posterior analysis.

In the second stage, the CSV files are processed and merged. In summary, the processing stage groups captures made within the same second, extract the three top functions in the stack executing on the CPU, and also label the threads. This stage generates a JSON file that is used in the visualization stage. The JSON file contains the structured data of each experiment and also the details of the hardware where experiments were executed, allowing quick data reading on the VisPerf dashboard. In addition, generating a single JSON file with all experiments makes it simple to share data for analysis between researchers.

After these two stages end, VisPerf dashboard can be accessed to analyze captured data. In Section 3.1, we discuss the visualizations and interactions we propose to VisPerf dashboard.

3.1 Visualizations and Interactions

The first step inside VisPerf is to upload the JSON file generated in the processing stage. VisPerf reads this file and generates the plots. We used D3[6] library to create the visualizations as it is very customizable and has many features to allow users to interact with the plots. Since the JSON file file may contains many experiments, we allow the user to select two experiments they want to compare.

VisPerf visualization is divided into three sections. The first section, showed in Figure 1, is named "Comparing experiments". This section compares specific events captured while executing the two experiments selected. As shown in Figure 1, the user can select the event to be compared and the plot. For this section, there are two plots available: the parallel coordinates and another plot with a grid representing the CPUs available in the processor where experiments were executed.

The first section in VisPerf also shows a summary of how the event selected changed between the experiments. As shown in Figure 1, the second experiment required 41.59% fewer instructions to finish execution. Another important metric this plot shows is that the second experiment had a better balance between the CPUs. CPU 12, for example, executed $\approx 9\%$ of the instructions, while in experiment two most of the CPUs executed $\approx 5\%$ of the instructions. The last column in the parallel coordinates plot shows the absolute difference between CPUs.

As the CPU number increases, the parallel coordinate plot starts to be a bit confusing because of the high number of lines. However, when hovering a specific line, this line is highlighted and the opacity

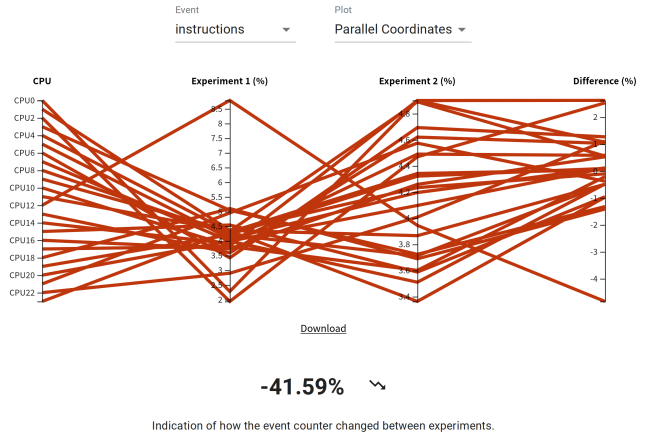

Figure 1: VisPerf section "Comparing experiments".

of other lines decreased. Another feature that is present in all plots is the download button. Here, the plots are downloaded in SVG format, which can be converted to any other format by the user.

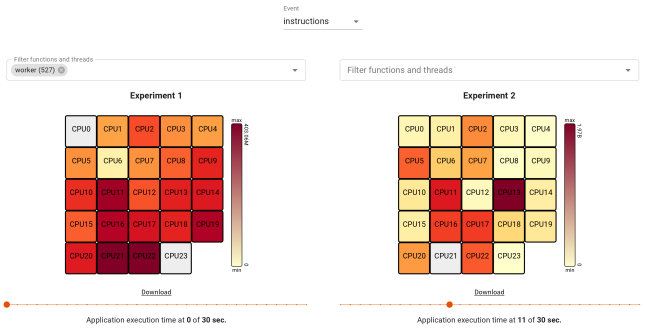

Figure 2: VisPerf section "Comparing experiments over execution time".

The second section, named "Comparing experiments over execution time" and shown in Figure 3, allows the user to explore the performance of specific functions or threads executed on the application. Since Perf only captures the thread id, we modified the source code of the application to inform the function being executed on the specific threads. Therefore, thread ids that are not present at the program output are named as "Other threads". In addition, in the second section, the user can view the metrics of the PMUs at specific points of the application execution. Navigating in the experiment one, shown in Figure 3, we can see that CPUs 0 and 23 were not used. While in experiment two, all CPUs are used, and at some points, some threads migrated of CPU while running the person recognition application. This behavior is correct because experiment one is using a fixed mapping policy, proposed by FastFlow, and experiment two the default operating system mapping policy.

The last section of VisPerf has metrics that use the events captured by Perf. For now, we only have one metric: IPC (Instructions Per Cycle). This metric indicates the average number of instructions executed at each CPU cycle. This section has two view dimensions: over time or by CPU. Over time visualization shows how IPC vary over the application execution time and by CPU visualization shows the IPC variation between the CPUs used to run the application.

[4] https://opencv.org
[5] https://github.com/fastflow/fastflow
[6] https://d3js.org

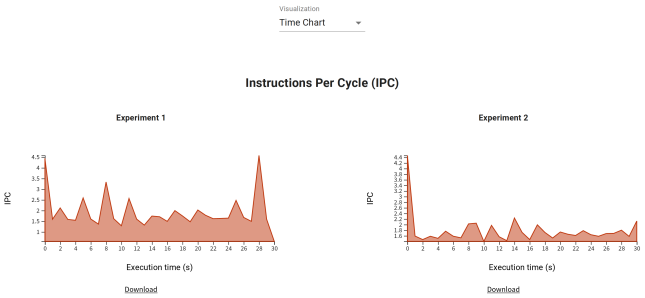

Figure 3: VisPerf section “Performance evaluation metrics”.

4 Conclusion

Through VisPerf we could achieve our goal that was to visualize and compare data captured by Perf. The work is yet a prototype. There are some visualizations missing, such as how to view which functions executed in specific CPUs. Moreover, we are missing a validation of VisPerf from other developers.

References

[1] collectd Wiki. *Plugin:Intel PMU - collectd Wiki*, June 2021.
[2] Linux Kernel Organization, Inc. *Perf Wiki*, May 2021.