Performance Data Visualization of Linux Events on Multicores

Claudio Scheer, Renato B. Hoffmann, Dalvan Griebler, Isabel H. Manssour, Luiz G. Fernandes

[1] School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre – RS – Brazil

`claudio.scheer@edu.pucrs.br`

Abstract. *Profiling tools are essential to understand the behavior of parallel applications and assist in the optimization process. However, tools such as Perf generate a large amount of data. This way, they require significant storage space, which also complicates reasoning about this large volume of data. Therefore, we propose VisPerf: a tool-chain and an interactive visualization dashboard for Perf data. The VisPerf tool-chain profiles the application and pre-processes the data, reducing the storage space required by about 50 times. Moreover, we used the visualization dashboard to quickly understand the performance of different events and visualize specific threads and functions of a real-world application.*

1. Introduction

With the advent of multicore processors, application developers faced the challenge of handling the parallel programming paradigm. Instead of executing application instructions on a single processing unit, developers could concurrently run parts of the same application (threads) in multiple processing units. This way, it became possible to increase application performance by leveraging the resources of the inherently parallel hardware. However, efficiently exploiting the computational resources of parallel hardware requires a deep understanding of the underlying system and architecture.

Achieving linear speed-ups with traditional multicore parallel programs is often unfeasible. For instance, the memory wall is a well-known deterrent to performance scalability, where each memory access may take hundreds or even thousands of CPU cycles [Solihin 2016]. For that, manufacturers created the ingenious design of multiple cache levels. Related to the ubiquitous instruction pipeline technology, it is essential to minimize pipeline flushes with branch-aware programming. Another common example is the NUMA (Non-Uniform Memory Access) architecture, which introduced the concept of memory locality. In turn, programmers interested in extracting maximum performance had to start taking into account data access patterns, cache hits/misses, branch predictions, and thread placement.

There is a large variety of multicore chips available on the market with unique architectural details. When considering efficient parallel applications, many variables may impact the performance. The cache and memory architecture are examples of such variables. Customizing such features is a challenge since the optimal configuration is often architecture dependent. For example, the heterogeneous systems have processors with different capabilities [Pacheco 2011]. Thus, exploiting the maximum performance of a specific multicore may require precise tuning of these variables. As a solution, one may use a profiling tool for extracting different application runtime metrics and characteristics. These metrics express the performance achieved by the application running under a tuning proposition or particular set of variables. For instance, one common metric is the execution time. In addition, other metrics may be considered, such as cache misses, thread assignment, latency, and CPU utilization. All these metrics guide developers in the decision process for optimizing their applications under a specific multicore architecture. For this purpose, there are profiling tools available such as Perf [Linux Kernel Organization 2020] and PAPI [Terpstra et al. 2010]. This work focuses on the Perf profiling tool since it is integrated with the Linux kernel.

Raw data captured from application monitoring may be large, hard to read, and difficult to reason about it quickly. One of the ways to address this problem is to use visualization techniques that help obtain meaningful insights from the raw data. There already exist works that address this problem, as we discuss later in Section 5. However, these works focus on visualizations related to an application's communication bottlenecks between the spawned threads.

This work addresses the gap of allowing visual analysis of profiling metrics, other than communication patterns, through the development of a visualization dashboard and tool-chain called VisPerf[1]. Our goal is to easily reason about the differences between each variable configuration per CPU or even per thread spawned by the application, via interactive plots. In summary, our contributions are threefold: 1) a tool-chain to reduce the storage space required for `perf_event`; 2) a visualization dashboard for understanding and comparing the performance metrics captured with `perf_event`; 3) A case study with the Person Recognition parallel application using VisPerf.

The organization of this paper is the following: Section 2 explains the Perf profiling tool, Section 3 discusses VisPerf tool-chain and the visualization dashboard, and Section 4 shows a case study using VisPerf. Finally, Section 5 discusses the works related to this paper and Section 6 discusses future work directions for VisPerf.

2. Perf Profiling Tool

The `perf_event` is a Linux subsystem to monitor and profile the performance of an application or system [Linux Kernel Organization 2020]. Introduced in version 2.6.31 of the Linux kernel, it can collect information through the multicore chip's PMU (Performance Monitoring Unit) or from the application itself. Cache miss, the count of times that a memory address was not present in the cache, is an example of a hardware event that the `perf_event` monitors. An example of a software event is page misses, that is, the count of times a data was not found in main memory and had to be loaded from the disk.

Events monitored with `perf_event` are read from a file descriptor created by calling the system call `perf_event_open`. Each event opens its file descriptor, used to capture data in two profiling modes: counting and sampling. Counting aggregates the occurrence of specific events, while the sampling mode measures the event periodically. `perf_event` can record a sample after a specific number of events or based on a specific frequency, such as samples per second.

Since each multicore chip has its PMU, `perf_event` provides an abstraction layer to facilitate performance monitoring. Moreover, `perf_event` allows monitoring events on specific threads or processing units. The `perf_event` subsystem also provides the Perf user space utility. In addition, Perf provides several subcommands to gather information from `perf_event`. For example, `perf list` to list all events that Perf can monitor in the current kernel and multicore chip available, and `perf record` to run an application and record the performance counter information. In this paper, we use the terms `perf_event` and Perf interchangeably.

[1]VisPerf website: `https://gmap.pucrs.br/visperf`

3. VisPerf

Profiling performance of parallel applications on multicores with Linux Perf tool may generate a large amount of data, which becomes a challenge for users to analyze and reason about the information, aiming to identify performance bottlenecks. To this end, we propose VisPerf, a tool-chain that provides different visualization techniques to facilitate the analysis of Linux Perf's recorded/captured data. As shown in Figure 1, VisPerf works in three steps: data capture, data processing, and data visualization.

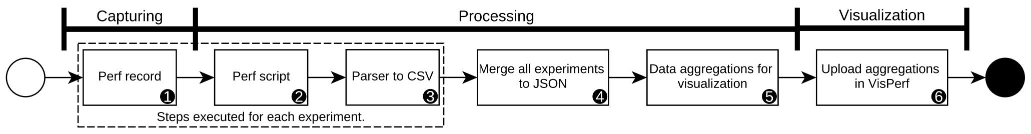

Capturing
Processing
Visualization
Perf record ❶
Perf script ❷
Parser to CSV ❸
Merge all experiments to JSON ❹
Data aggregations for visualization ❺
Upload aggregations in VisPerf ❻
Steps executed for each experiment.

Figure 1. VisPerf workflow.

Initially, we describe how the user interacts with the VisPerf tool-chain and show how simple it is. For that, we provide a base script named `run-base` that defines how to setup the experiments and use the `perf-capture` tool for profiling them. The application developer is free to customize the execution of the experiments fully and inside the script, call the `perf-capture` tool that is responsible for running and profiling the experiment. Listing 1 shows an example of how to call `perf-capture`. Later, in Section 3.2 we discuss the `perf-capture` tool in greater details and in Section 3.3 we describe the VisPerf interactive dashboard.

Listing 1. `perf-capture` tool usage example.

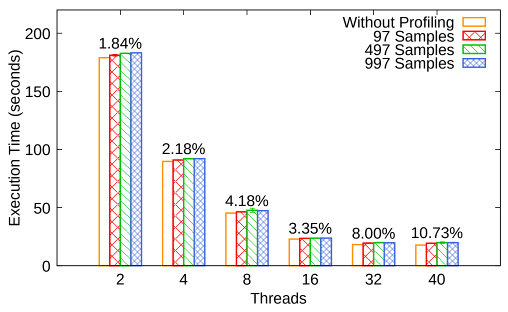
```
./perf_capture.sh --name "Experiment Name" --runs 3
    --frequency 997 --output ./data/experiment
    --command "sleep 5"
```

3.1. Capturing Data

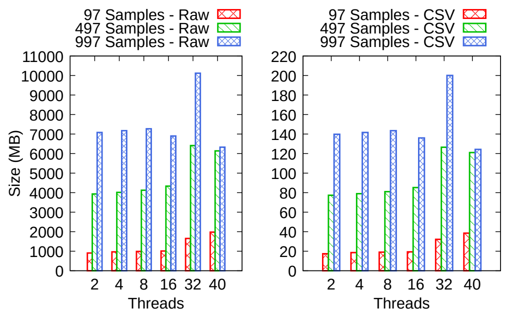
In this step, applications are instrumented to perform profiling with `perf record` using a set of parameters (step 1 in Figure 1). Users may not worry about the specification of the parameters. However, the user may add custom profiling records. By default, we set the sampling frequency per second in 997 because it provides a good trade-off between data generation (size and precision) and performance overhead in our applications test-case. Higher sampling frequencies may add some overhead in the application execution time. It is up to the user to choose the best option for its profiling requirements.

Figure 2a provides an overview of the introduced overhead for different sampling numbers per second. As the number of processing units used to execute the application increases, the overhead also increased (behavior shown by the label on each number of threads profiled). One of the reasons for this behavior is the number of active file descriptors to monitor the events, which is one per processing unit and one per event ($threads \times events$). Although the number of files is the same for each number of threads tested, the ones with fewer threads have fewer data to store (I/O operations). The amount of data generated when profiling the parallel applications is also an important concern since it may surpass even the terabyte-scale for more extended periods of profiling. A higher amount of data entails a higher profiling precision.

Choosing the best number of sampling per second depends on the problem addressed. For example, if the goal is to understand how a specific function behaved during application execution, many samples may be required. However, as shown in Figure 2, more overhead is added to the application execution, and data size increases. Therefore, to reduce the disk space used, as shown in Figure 2b, we convert the raw data captured by Perf into CSV files. With that, we were able to reduce the data size up to ≈ 50 times. On the other hand, if the goal is to have a more general understanding of the application performance, the application developer may reduce the number of samples.



(a) Perf overhead at different sampling frequencies.



(b) Raw data on the left and CSV files on the right.

Figure 2. Comparing experiments' general performance.

The Perf tool allows capturing many events. We focused on those events expressing parallel application performance executing on multicores such as cache memory, branches, instructions, and CPU cycles. The user can choose other events according to the multicore architecture available by specifying the event names in a configuration file named `events.txt`. `perf-capture` reads this configuration file and customizes Perf to sample only events present in `events.txt`.

In addition, for some cases, such as stream processing applications[2], it may be needed to visualize the performance of specific threads. Since Perf can only capture the thread id, we expect the application developer to inform the corresponding name of each thread. Therefore, each time the application launches a new thread, the developer captures the thread id and logs it into a file to be read by the subsequent second stage of VisPerf, discussed in Section 3.2. In VisPerf, users can see the performance of each individual application region. We consider a region a specific thread or the scope of specific functions. Applications must compile with the debug flag to capture the function names present in the CPU stack trace. In case of `g++` compiler, it is possible to use the `-g` flag. Furthermore, we can still use optimization flags such as `O3`.

3.2. Data Processing

Raw data from Perf, captured with `perf record`, are usually large. For example, an experiment executed for about 2 minutes and capturing 997 samples per second generates a raw file with about 12 GB. With a higher number of experiments or the execution time, Perf requires considerably larger amounts of space in the storage disk. To address this issue, we used `perf script` (step 2 in Figure 1). This subcommand of Perf allows filtering from the raw data only the trace of the events specified for capture. Using the example cited before, this operation reduced the file size from approximately 12 GB to 800

[2]Stream processing applications are composed of stages, where each stage operates over its data and send the processed data to the next stage, up to a sinking stage.

MB. On average, `perf script` can reduce the file size by about 15 times. In addition to reducing the size of captured data, `perf script` generates a structured text file, which is easy to read and parse for further analysis. Figure 3 shows the structure of each sample captured with `perf record` and transformed with `perf script`.

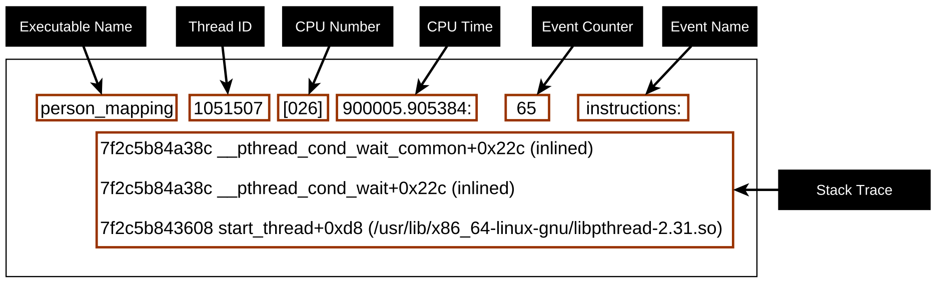

Executable Name
Thread ID
CPU Number
CPU Time
Event Counter
Event Name
person_mapping
1051507
[026]
900005.905384:
65
instructions:
7f2c5b84a38c __pthread_cond_wait_common+0x22c (inlined)
7f2c5b84a38c __pthread_cond_wait+0x22c (inlined)
7f2c5b843608 start_thread+0xd8 (/usr/lib/x86_64-linux-gnu/libpthread-2.31.so)
Stack Trace

Figure 3. Structure of a data block from `perf script`.

The `perf record` sub-command captures the stack trace from the function executing on a CPU for each individual sample. However, this can pose some problems for the visualization since the stack trace may include functions from low-level libraries such as threads initialization. For most of these libraries, Perf captures the function name as `[unknown]`. Therefore, as shown in Figure 3, we limited the stack trace to only three levels deep, which greatly reduces the file size with little impact on profiling accuracy.

After executing the `perf script`, another parsing step is required. Since CPU time does not start from zero, it is difficult to analyze the application's execution time from a developer's perspective. Therefore, we pick the execution time from the first sample captured and subtract it from the CPU time for each captured subsequent sample. This way, the last sample captured has the complete application's execution time. Moreover, we only need the function name to allow VisPerf users to filter captured events by specific functions by filtering the stack trace data. Furthermore, as discussed in the previous section, we also allow custom thread names, which helps reason about multi-threaded programs, especially with applications with multiple parallel independent stages organized in a pipeline structure. For these cases, VisPerf conveniently organizes the data based on the thread's name instead of the numerical id.

In the third step (in Figure 1), we read the output from `perf script` and produce a CSV file structured for further aggregations. Each row in the CSV file represents a sample of the application made by Perf. Beyond helping with the visualization, an inherited benefit of parsing raw data into a CSV file is that we have reduced the file size (Figure 2b) and have structured data that can serve analysis beyond VisPerf. One such example is serving as training datasets for machine learning algorithms to help automatically configure the best variables for executing parallel stream processing applications [Hirzel et al. 2014, Nakada et al. 2020, Nguyen et al. 2017].

To address capturing and pre-processing data from Perf, we created a tool named `perf-capture`. This tool takes as input the experiment to be executed, the number of times to execute this experiment, and where to save output parsed files. With `perf-capture`, we abstracted the complexity of managing large files and understanding specific parameters of Perf subcommands in a single command. As output, each call to `perf-capture` returns a JSON file informing the location of the CSV file for each experiment execution. By performing steps 2 and 3 of Figure 1 for each experiment, we can greatly reduce the size of the file generated by `perf record`. From that point onward, VisPerf only works with the CSV file since it occupies much less space than raw data.

After all experiments finish, we merge all JSON files from `perf-capture` into a single JSON file (step 4 in Figure 1). In addition, we added to the merged JSON file details about the multicore architecture used to execute the experiments. These details include the number of processing units and the events captured. This step is essential because it defines which experiments and events are aggregated to generate the final visualization. The user can change the JSON file to create only visualizations for specific events since some experiments may have large amounts of less relevant data.

To perform the data aggregation for visualization (step 5 in Figure 1), the user must pass as input the JSON file defining the CSV files' location and multicore architecture details. For this step, we provide another tool that will read all CSV files given as input and generate a final JSON file that the user has to upload to the visualization dashboard. The final JSON file does not have any external dependencies. All data required for the visualizations are embedded into it. This feature is vital to provide an easy way to share experiments between different researchers.

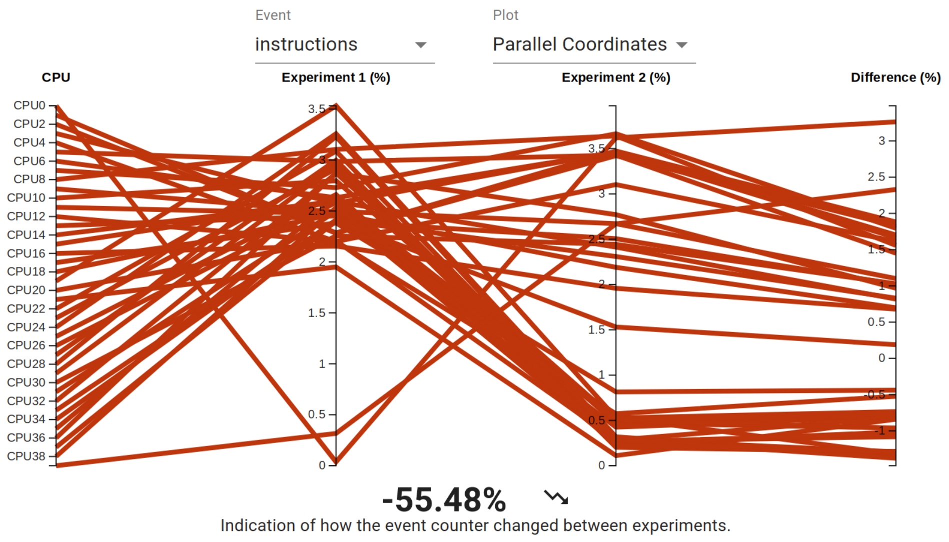
We aggregate data from three different types of events: by CPU, by time, and other events. When aggregating by CPU, we average the counter of the same event from all captured samples. This aggregation gives a general idea of how each event performed for a specific experiment or varied between experiments. When aggregating data by time, we create a timeline of events covering all the experiment execution. The aggregation is made by second: we average the event counter of all samples captured in each exact second. Moreover, we make a sub aggregation per CPU for each second, enabling the VisPerf user to understand how each CPU performed or how processes migrated during the application execution. This is an essential feature for analyzing load balancing between parallel threads. Furthermore, each CPU aggregation is related to a set of threads and stack trace functions, making it possible to filter for only specific regions or functions.

Finally, the last aggregation combines different events sampled. So far, we only have one of these aggregations: Instructions Per Cycle (IPC). IPC measures how many instructions the multicore architecture was able to execute on each CPU cycle. Similar to the aggregation by time, here we also aggregate counters by second. The difference is that for this aggregation, we only consider the number of instructions divided per the number of CPU cycles the application had in any given second. From this aggregated data, VisPerf may generate the visualization for each event.

3.3. Interactive Visualization Dashboard

Aside from the tool-chain for capturing and processing data from Perf previously explained, we provide an interactive visualization dashboard. The VisPerf dashboard is a client-side web application developed using React JS. VisPerf requires a web server only to deliver the HTML, JavaScript, and CSS files. Any processing beyond the pre-processing stage uses the resources of the computer running the web browser. To create the visualizations and allow users to interact with them, we used D3.js. With D3, it is possible to manipulate SVG elements, which, combined with HTML, JS, and CSS provides a customizable tool for creating interactive visualizations. The first step to use VisPerf is uploading the JSON file (step 6 in Figure 1) generated in the pre-processing stage. We used JSON files because it is easy to read in JavaScript, avoiding external libraries. As discussed before, we do not upload this file to our web server. Instead, we perform the file processing only on the client-side. As soon the user loads an experiment file, VisPerf automatically reads the file and generates the plots.

Each JSON file loaded in VisPerf may contain multiple experiments, making it impossible to compare all experiments at once. Therefore, we allow the user to select up to two experiments for comparison at the same time. All visualizations and comparisons inside VisPerf consider these two selected experiments. For instance, the VisPerf user may select two experiments using two different load balancing techniques to compare which experiment promotes a better CPU utilization or fewer cache misses. Each time the user changes the selected experiment, the data from the new experiment is loaded, and the visualization dashboard is updated. Furthermore, VisPerf has a help section describing each of the monitored events since Perf does not present straightforward documentation.


Event: instructions
Plot: Parallel Coordinates

-55.48%
Indication of how the event counter changed between experiments.

Figure 4. Number of instructions executed on each experiment (40 threads).

VisPerf visualization divides into three sections. The first section, shown in Figure 4, compares specific events captured while executing the two selected experiments. The user can select between two visualizations, parallel coordinates [Inselberg 2009] and heatmap [Ward et al. 2010], to analyze all processing units available in the multicore architecture. The parallel coordinate plot allows to compare individual experiments and identify patterns in the data. This type of graph can show the variation even when experiments have different ranges. In VisPerf, we used parallel coordinate plots to understand how the selected event varied between the two experiments selected, as shown in Figure 4. To avoid over-cluttered and improve readability when the CPU number increases and there are a high number of lines, in VisPerf, when hovering a specific line, we highlight it and decrease the opacity of other lines. The second visualization shows the processing units' variation using color scales without the need for user interaction. The Figure 6 show examples of this visualization.

In addition to the two visualization options, the first section of VisPerf allows selecting which event to visualize. As soon as an event is selected, we load the data and update the visualizations. The interpretation of these plots depends on the event selected since the selected event will define if it is better to increase or decrease the counter of the event. For example, increasing the number of cache misses decreases the application performance, while increasing the number of cache references increases the application performance. Moreover, the first section of VisPerf also shows an averaged summary of the event's performance regardless of the processing units. This is shown in the lower part of Figure 4. In this example, the second experiment executed, on average, 55.48% fewer instructions to finish its execution when compared with the first experiment. Later, we discuss in Section 4 more details about the experiment shown in Figure 4.

In the second section of VisPerf, show in Figure 5, we allow the user to explore the application performance at a specific point in the application's execution timeline. We show a timeline where the user can scroll through all of the application execution. This can be seen at the lower part of Figure 5. Moreover, it is possible to select multiple simultaneous functions and threads to the visualization. In this case, named threads are kept as specified by the application developer, and the other ones are displayed as "Other threads" in the dashboard.

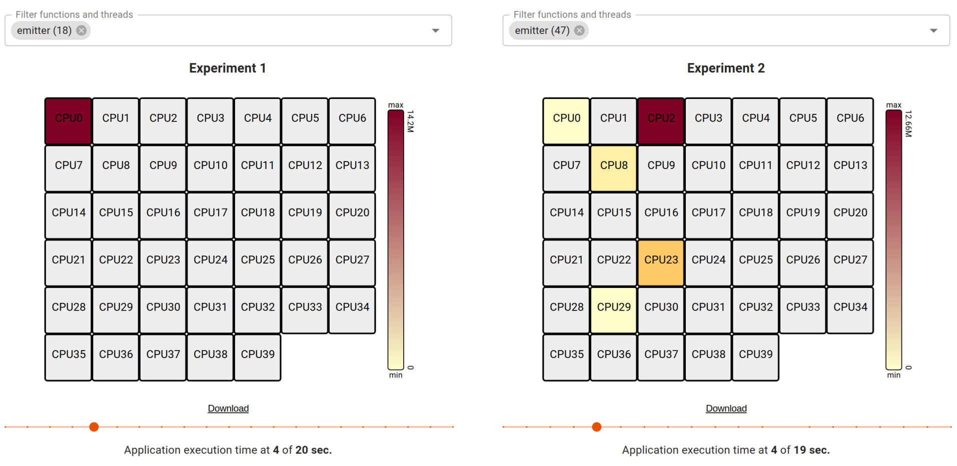

Filter functions and threads
emitter (18)
Experiment 1
CPU0 CPU1 CPU2 CPU3 CPU4 CPU5 CPU6
CPU7 CPU8 CPU9 CPU10 CPU11 CPU12 CPU13
CPU14 CPU15 CPU16 CPU17 CPU18 CPU19 CPU20
CPU21 CPU22 CPU23 CPU24 CPU25 CPU26 CPU27
CPU28 CPU29 CPU30 CPU31 CPU32 CPU33 CPU34
CPU35 CPU36 CPU37 CPU38 CPU39
max
14.2M
0
min
Download
Application execution time at 4 of 20 sec.
Filter functions and threads
emitter (47)
Experiment 2
CPU0 CPU1 CPU2 CPU3 CPU4 CPU5 CPU6
CPU7 CPU8 CPU9 CPU10 CPU11 CPU12 CPU13
CPU14 CPU15 CPU16 CPU17 CPU18 CPU19 CPU20
CPU21 CPU22 CPU23 CPU24 CPU25 CPU26 CPU27
CPU28 CPU29 CPU30 CPU31 CPU32 CPU33 CPU34
CPU35 CPU36 CPU37 CPU38 CPU39
max
12.66M
0
min
Download
Application execution time at 4 of 19 sec.

Figure 5. Interactive timeline for the number of instructions executed.

One important feature of this comparison over time is the visualization of the behavior of threads during application execution. As seen in Figure 5, VisPerf enables the user to visualize the performance of specific threads or functions easily. This example shows that the experiment on the right schedules the emitter thread to several different processing units while the experiment on the left pins the emitter to a single CPU. In the upper part of Figure 5, the user can easily select the event of interest. For instance, it would be possible to observe cache misses for the emitter thread in each CPU. In this experiment, the cache miss counter is higher for the second experiment that executes the same thread in multiple CPUs since it migrates the thread five times in that specific second. The user can visualize any supported event in a similar manner.

Customizing the sampling frequency of Perf is an essential aspect, especially for this analysis over time. For instance, a low number of samples might not accurately express all the migration made by a thread. Furthermore, it may not even capture some functions for visualization since the functions are extracted from the stack trace present in the processing unit when capturing a sample. However, a higher sampling frequency adds overhead to the profiling execution and increases the data file sizes.

Finally, the last section of VisPerf shows the combination of events sampled by Perf. In the current VisPerf version, we only support one metric: IPC (Instructions Per Cycle). We show an example of IPC visualization over the experiment execution time in Figure 8. In summary, IPC indicates the average number of instructions executed on each CPU cycle. This section has two observable dimensions: over time or by CPU. The visualization over time shows how IPC varies over the application execution, and the visualization per CPU shows the IPC variation between the CPUs used to run the application.

4. Case Study

In this Section, as a case study of VisPerf, we compare the performance of two experiments using a Person Recognition application written in C++ [Griebler et al. 2017b]. In summary, this application detects faces and recognizes a person of interest in a specific image using the support of OpenCV video processing library. As input, we used a 15 seconds, MPEG-4, 450 frames, and 640x360 resolution video. The goal of this study is to identify bottlenecks when comparing two thread mapping policies. A mapping policy defines which processing unit runs the threads launched by an application. Thus, we profiled the Person Recognition application using the default mapping policy of the Linux operating system and the mapping policy proposed by FastFlow [Aldinucci et al. 2017]. Furthermore, towards exploiting the parallelism of this application, we generated the source code for the FastFlow library using SPar [Griebler et al. 2017a]. Such analysis can assist us in proposing a better thread mapping policy for FastFlow.

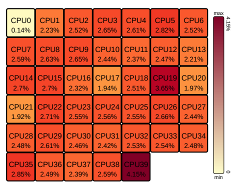
The main difference between the two mapping policies is that the Linux policy is dynamic, while FastFlow has a static mapping policy. A static mapping policy pins each thread to a processing unit until the application execution finishes. The thread may be moved to another processing unit in a dynamic mapping policy while the application is running. The Linux mapping policy, for example, tries to keep fairness and balanced work between all processing units.

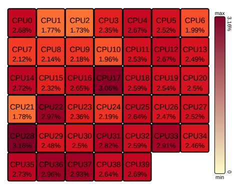
The Person Recognition application has three parallel stages: emitter, parallel workers, and collector. The emitter extracts each frame from the videos and sends it to the workers. The workers process the frame and send it to the collector responsible for merging all the processed frames. We ran the experiments on a machine eqquipped with 64 GB of RAM and two Intel(R) Xeon(R) Silver 4210 CPU 2.20GHz Xeon Silver processor totaling 20 physical processing units and 40 with hyper-threading, with the Linux kernel version `5.4.0-74-generic`. Using the FastFlow mapping policy, we pin the emitter to the first processing unit and the collector to the last processing unit. The other processing units in the middle execute the parallel workers. For both mapping policies, we ran the experiments with 18 and 38 workers. This way, we try to explore the multicore processor resources as much as possible.

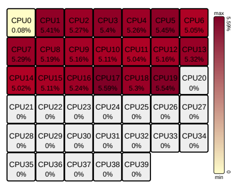
In our first analysis, obtained from Figure 4, we can compare both thread mapping policies with respect to the load balancing. CPU0 and CPU39 on experiment 1 executed only a few instructions. On the other hand, in experiment 2 using the Linux thread mapping policy, both processing units executed more instructions. Another important piece of information shown in this image is that experiment 2, when compared to experiment 1, required $\approx 55\%$ fewer instructions to execute the experiment. One of the reasons for this expressive reduction is that, in experiment 1, we allocated the collector thread on a different socket than some workers threads. This way, experiment 1 also had $\approx 50\%$ more cache misses, which requires threads to load data and instructions from the main memory. This results in more CPU cycles, and, consequently, more instructions.

The visualization shown in Figure 5 may help understanding how each mapping policy schedules the emitter and collector threads. For Instance, in it, it can clearly be seen that FF mapping from the Experiment 1 pins the thread to the CPU while the Linux mapping from Experiment 2 constantly migrates it during runtime execution. In this case, this was expected. However, this feature may be very important to understand how any given part of the code is scheduled in the multicore CPU at any given time. This may help the developer to understand which part of the application runtime their impact is more relevant, helping to identify exact possible bottlenecks.

Figure 6 shows the number of branch misses for each processing unit. In Figure 6a and Figure 6c, we can see that CPU39 and CPU19, respectively, had a high number of branch misses. This fact shows that, in most cases, the FastFlow thread mapping policy does not help the processor "guess" the source code path and execute instructions speculatively. There may be some differences between each execution of the same experiment. However, the Linux thread mapping policy had $\approx 15\%$ more branch misses when considering all executions. This is expected since Linux mapping switches threads more often, deteriorating the branch predictor module. This way, the developer could reason that optimizing cache misses is more relevant than optimizing branch predictions. One possible decision would be to optimize FastFlow thread mapping to try allocating the maximum number of worker threads in the same socket of the collector and emitter threads.


(a) FastFlow (40 threads)


(b) Linux (40 threads)


(c) FastFlow (20 threads)

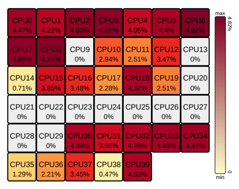
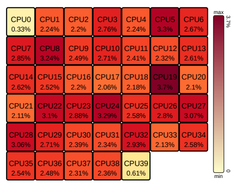
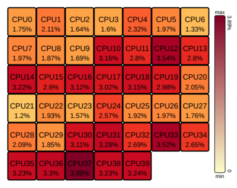
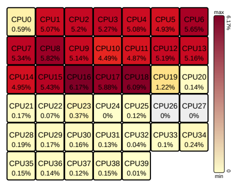

(d) Linux (20 threads)

Figure 6. Branch misses for different thread mapping policies.


(a) FastFlow (40 threads)


(b) Linux (40 threads)


(c) FastFlow (20 threads)

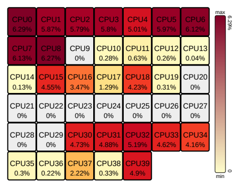
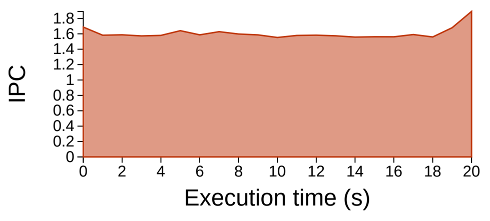
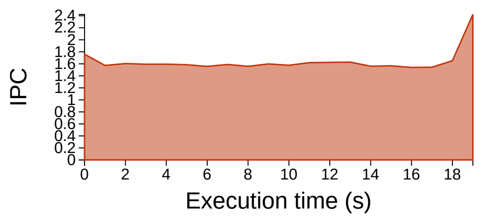

(d) Linux (20 threads)

Figure 7. Cache misses for different thread mapping policies.

When profiling applications, there is a need to specify a warm-up time in milliseconds time to avoid inaccurate data. In Figure 7c, we used 100 ms warm-up time, which reduces but still indicates cache misses for unscheduled CPU's. Finally, in the last section of the VisPerf visualization dashboard, we see how the IPC varied during the experiments' execution. Both experiments had similar IPC, as shown in Figure 8. However, the Linux thread mapping had a slightly better performance at the end of the application execution.



(a) FastFlow (40 threads)



(b) Linux (40 threads)

Figure 8. Instructions Per Cycle (IPC) for both thread mapping policies.

With the use of VisPerf we could identify some bottlenecks in the thread mapping policies and conclude that the Linux thread mapping policy had a better performance for the Person Recognition application tested in this case study.

5. Related Work

In the literature, other works already proposed visualization tools for parallel applications. For example, Pajé [de Kergommeaux et al. 2000] is a visualization tool for parallel applications composed of a large number of communicating threads. Pajé aims to allow programmers to interact with the data and be extensible beyond thread communication. Traces from the observed application are captured using the programming model ATHAPASCAN. Pajé uses a space-time diagram for visualization, which combines state and communications performed by each thread. Also, Pajé has the source code click-back functionality allowing users to find the statement that spawned the thread quickly.

The authors of [Weidendorfer 2008] present Callgrind and KCachegrind. These tools aim to capture and visualize information about simulated cache performance. The tools focus on the performance of sequential programs since, according to the authors, we can identify bottlenecks and the need for more complex parallelization strategies based on them. KCachegrind has three types of visualizations: a call graph, a Callee Map, and an annotated view that maps to the application source code, differentiating instructions that are either conditional or not.

A new visualization for application stack traces was proposed in [Gregg 2016]. The visualization can be created using data from any profiling tool containing the application's stack trace, such as Perf. In summary, the visualization shows how deep the stack trace is and which functions appear the most on the stack trace. Moreover, the visualization also shows the most common functions on top of the stack trace. These functions are the ones that are using resources from the processing unit. Therefore, since these functions are wider in the visualization, the user knows where to focus the optimization. Compared to these related works, we provide visualizations with different perspectives to facilitate the understanding of the data captured and focus exclusively on understanding the performance of parallel applications.

Different from [Gregg 2016], we focus on the visualization of all metrics that `perf_event` supports, instead of only focusing on the stack trace. Moreover, while [de Kergommeaux et al. 2000] focus only on metrics related to each thread, we also focus on metrics expressing the performance of each CPU logical core and specific functions of the application. Different from all previous work, we also provide a way to compare the variations of the captured metrics between different experiments.

6. Conclusion

This paper provided a solution for easily profiling, capturing, visualizing, and reasoning about the performance of parallel applications. For that purpose, we used Perf (part of the Linux Kernel) to extract data from applications. From that, we created VisPerf to aggregate this data and generate an interactive dashboard that permits isolating specific functions and threads through a timeline. This way, developers can understand performance implications in their parallel applications and take hints on addressing them. We also demonstrated a use-case of VisPerf with a parallel Person Recognition application, where we compared two sets of experiments and reasoned about performance considerations that our solution helped assess. Additionally, VisPerf reduced about 50 times the profiling data log size. Finally, we identify possible future works: (1) include the flame graph visualization; (2) visualization of multicore architectures with multiple sockets; (3) Perf profiling for distributed parallel applications.

Acknowledgements: This research is partially funded by FAPERGS 05/2019-PQG project PARAS (N^{o} 19/2551-0001895-9), FAPERGS 10/2020-ARD project SPAR4.0 (N^{o} 21/2551-

0000725-7), CNPq Scholarship - Brazil (N^o 308456/2020-3), Universal MCTIC/CNPq N^o 28/2018 project SPARCLOUD (N^o 437693/2018-0), and MCTIC/CNPq call 25/2020 (N^o 130484/2021-0).

References

Aldinucci, M., Danelutto, M., Kilpatrick, P., and Torquati, M. (2017). *Fastflow: High-Level and Efficient Streaming on Multicore*, chapter 13, pages 261–280. John Wiley & Sons.

de Kergommeaux, J. C., de Oliveira Stein, B., and P.E., B. (2000). *Pajé, an interactive visualization tool for tuning multi-threaded parallel applications*.

Gregg, B. (2016). The flame graph: This visualization of software execution is a new necessity for performance profiling and debugging. *Queue*, 14(2):91–110.

Griebler, D., Danelutto, M., Torquati, M., and Fernandes, L. G. (2017a). SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters*, 27(01):1740005.

Griebler, D., Hoffmann, R. B., Danelutto, M., and Fernandes, L. G. (2017b). Higher-Level Parallelism Abstractions for Video Applications with SPar. In *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing*, ParCo'17, pages 698–707, Bologna, Italy. IOS Press.

Hirzel, M., Soulé, R., Schneider, S., Gedik, B., and Grimm, R. (2014). A catalog of stream processing optimizations. *ACM CSUR*, 46:46.

Inselberg, A. (2009). *Parallel Coordinates: Visual Multidimensional Geometry and Its Applications*. Springer, New York, NY.

Linux Kernel Organization, I. (2020). *Perf Wiki*. Linux Kernel Organization, Inc.

Nakada, T., Yanagihashi, H., Imai, K., Ueki, H., Tsuchiya, T., Hayashikoshi, M., and Nakamura, H. (2020). An energy-efficient task scheduling for near real-time systems on heterogeneous multicore processors. *IEICE Trans. Inf. Syst.*, 103-D(2):329–338.

Nguyen, V. A., Hardy, D., and Puaut, I. (2017). Cache-Conscious Offline Real-Time Task Scheduling for Multi-Core Processors. In Bertogna, M., editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:22, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Pacheco, P. (2011). *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Solihin, Y. (2016). *Fundamentals of Parallel Multicore Architecture*. CRC Press.

Terpstra, D., Jagode, H., You, H., and Dongarra, J. (2010). Collecting performance data with papi-c. In Müller, M. S., Resch, M. M., Schulz, A., and Nagel, W. E., editors, *Tools for High Performance Computing 2009*, pages 157–173, Berlin, Heidelberg. Springer.

Ward, M. O., Grinstein, G. G., and Keim, D. A. (2010). *Interactive Data Visualization - Foundations, Techniques, and Applications*. A K Peters.

Weidendorfer, J. (2008). Sequential performance analysis with callgrind and kcachegrind. In Resch, M., Keller, R., Himmler, V., Krammer, B., and Schulz, A., editors, *Tools for High Performance Computing*, pages 93–113, Berlin. Springer Berlin Heidelberg.