An Accurate Tool for Modeling, Fingerprinting, Comparison, and Clustering of Parallel Applications Based on Performance Counters

V. Ramos[1], C. Valderrama[1], P. Manneback[2] and S. Xavier de Souza[3]

Abstract— **The analysis of application performance is essential to better exploit its potential on High-Performance Computing (HPC) architectures. Access to performance counters, available in modern processors, allows collecting key information about program behavior to provide the most appropriate HPC execution strategy. In this context, we have developed an accurate tool based on performance counters, which facilitates modeling, fingerprinting, behavior comparison and clustering of applications. It provides a high-level Python API for accessing and configuring performance counters; while execution and counters data gathering is performed by a C++ module to reduce overhead. Indeed, the accuracy of this multiplatform tool was also compared to existing alternatives. Key features, such as performance counters collection, post-processing, and comparison, enable fingerprinting of applications, an important step in understanding program behavior for later classification and optimization according to the parameters characterizing the target HPC platform. For demonstration purposes, the tool was used in the clustering of Polybench applications, a frequently used benchmark set for kernels monitoring. This clustering facilitated the identification of applications with similar and comparable behaviors in terms of input size, data access and transfer, resource utilization and computation, which facilitates the creation of test sets for a given environment based on specific measurement parameters.**

I. INTRODUCTION

Hardware Performance Counters are special registers available on most modern processors capable of counting micro-architectural events such as instructions executed, cache-hit, branches miss-predicted, energy estimation and much more. In new architectures, there are hundreds of hardware events that can be monitored, and more are added to each new generation. Performance counters were initially introduced for debugging, but since then they have provided a lot of useful information about running applications without slowing down the execution. They have been used in several other areas, such as software profiling [1], [2], [3], CPU power modeling [4], dynamic frequency and voltage scaling, vulnerability research and malware defense [5].

[1]Electronics and Microeletronics Dept., Faculty of Engineering, University of Mons, 7000 Mons, Belgium vitor.ramos|carlos.valderrama@umons.ac.be

[2]Computer Science Unit, Faculty of Engineering, University of Mons, 7000 Mons, Belgium pierre.manneback@umons.ac.be

[3]Computer Engineering and Automation, Dept. Federal University of Rio Grande do Norte,Natal, Brazil

Exploiting Performance Monitoring Units (PMU) requires an intimate knowledge of the micro-architecture and kernel API, as well as an awareness of an ever increasing complexity. Otherwise, the measurement performance and accuracy will be seriously affected. Although many tools have been developed using performance counters, programmable interfaces capable of providing good accuracy are still lacking, especially for high-level programming languages. Indeed, apart from PAPI [6], [7] and Perfmon [8], [9] there are only a few APIs allowing access to these counters, and many others are poorly documented, unstable, or designer for a specific purpose.

Performance metrics may have different definitions and programming interfaces on different platforms. Therefore, besides gathering information, post-processing modules are also needed. Such modules will overcome the lack of precision of counters on some architectures, as indicated in [10], [11], [12]. Events that must be precise and deterministic (such as retired instructions) show a variation on run-to-run and over-count on x86_64 machines, even in strictly controlled environments. These effects are almost always non-intuitive to casual users and pose problems when strict determinism is desirable.

To meet the above mentioned requirements, our strategy combines low-level, efficient and accurate access to PMUs, facilitated by high-level programmable interfaces. This allows the user to perform all configurations and post-processing in Python, while the underlined architecture details and information gathering are supported by a C++ module, thus preserving accuracy with limited overhead. In order to understand the behavior of programs for future comparisons and classification, the proposed tool makes another contribution: the ability to fingerprint and cluster applications.

The definition of reference parameters, such as input size of programs, and performance measures uniformization, allow us to cluster benchmarks such as PolyBench [13], a collection of numerical computations with static control flow extracted from various application domains, with interesting results. In addition to contributing to the standardization of kernel execution and monitoring, this clustering has identified applications with similar and comparable behavior in terms of input size, data transfer and access, resources used and computation; which facilitates the creation of test sets for a given environment, according to specific

measurement parameters.

The rest of this article is organized as follows. Section 2 provides the related work regarding tools available and requirements. Section 3 presents the performance counters and motivation. Section 4 presents our clustering and application analysis tool, architecture and components. Section 5 shows evaluation results: the comparison of existing API and the clustering of the PolyBench benchmarks. Finally, Section 8 concludes this article.

II. RELATED WORK

There are only a few APIs allowing access to performance counters. PAPI [6], [7], one of the most used libraries for accessing hardware performance counters, was originally developed to provide portable access to the counters found on a diverse collection of modern microprocessors. Rather than learning and writing a new performance infrastructure every time it is ported to a new machine. Measurement code can be written in the PAPI API, which hides the underlying interface. PAPI was developed on C and a few non-official libraries were ported to Python. The main problem we found using PAPI was the Python version of has a considerable overhead, it also does not have an easy way to create raw events or low-level control without having to use a special driver. And as our tests will show later, counters sampling over time does not produce good results either.

There are also available a set of interfaces using their own drivers, mainly because the counters are only accessible in kernel mode (ring 0) to control the events for which the counter must be started or stopped. Some events are fixed and others require the development of a dedicated kernel driver.

Perfctr [14] supports per-kernel-thread and system-wide monitoring for most major processor architectures. It is distributed as a stand-alone kernel patch. The interface is mostly used by tools built on top of the PAPI performance toolkit.

The Intel VTUNE [15] performance analyzer comes with its own kernel interface, implemented by an open-source driver. The interface supports system-wide monitoring only and is very specific to the needs of the tool.

The problem with the approach of a tool and its own kernel interface is dangerous because, as mentioned on [8], there is clearly code duplication, but more importantly, there is no coordination between the various interfaces that may coexist sharing access to the same PMU resource. To solve this problem, Perfmon2 [9] offers a standard interface that all tools can use. Unfortunately, it has not been widely adopted, just supported by a few architectures like the IA64. Instead, Linux comes up with a performance counters subsystem which provides a complete set of configurations.

III. READING PERFORMANCE COUNTERS

Although hundreds of events are available for monitoring, only a limited number of counters can be used simultaneously. Therefore, the events to be monitored should be carefully selected and configured using the available counters.

The number of counters available varies between processor architectures, e.g., modern Intel CPUs [16] support three fixed and four programmable counters per core. Fixed counters monitor events such as Instruction Retired (how many instructions were completely executed), logical cycles, and reference cycles, while programmable counters can be configured to monitor architectural and non-architectural events. However, if additional counters are needed, the available ones must be multiplexed. The configuration of the counters is done by writing in Model-Specific Registers (MSR), only accessible in ring 0 (kernel mode) as indicated previously.

Operating systems provide an abstraction of these hardware capabilities to access counters and MSRs. On the Linux system, where our work was developed, there is a performance monitoring subsystem which provides per-task and per CPU counters, counter groups, and related event features. All events are seen as 64-bit virtual counters, regardless of the width of the underlying hardware counters. They are accessible via special file descriptors, one file descriptor per virtual counter, opened via the perf_event_open() system call. Counter events can be processed by interrupt, polling or on time. The interrupt operates by hooking a user-defined function to a specified event, such as a counter overflow, and whenever this event happens, a signal will be generated passing the control to the designated handler function. With polling, whenever an event occurs on the system, the counter value is queued by the operating system and the user can read from this queue using a system call. The last option is to sample over time reading counters every n second.

Ideal hardware performance counters provide exact deterministic results. Real-world PMU implementations do not always live up to this ideal [10], [11], [12], [17]. Events that should be exact and deterministic (such as the number of executed instructions) show run-to-run variations and over counts on x86 64 machines, even when running in strictly controlled environments. These effects are non-intuitive to casual users and cause difficulties when strict determinism is desirable, such as when implementing deterministic replay or deterministic threading libraries. Because of that, we have implemented a methodology to reduce the noise and over counts on performance counters.

IV. IMPLEMENTATION

This tool is composed of 5 modules the are Profiler, Events, Workload, Analyzer and libpfm4. The Workload and libpfm4 module are developed on C/C++ and

interfaced with python using Python C API with SWIG (a software development tool to connects programs written in C/C++ with a variety of high-level programming languages). The libpfm4 module, developed by [18], is used as an auxiliary library.

A. ARCHITECTURE

In figure 1 we can see how these modules interact with each other.

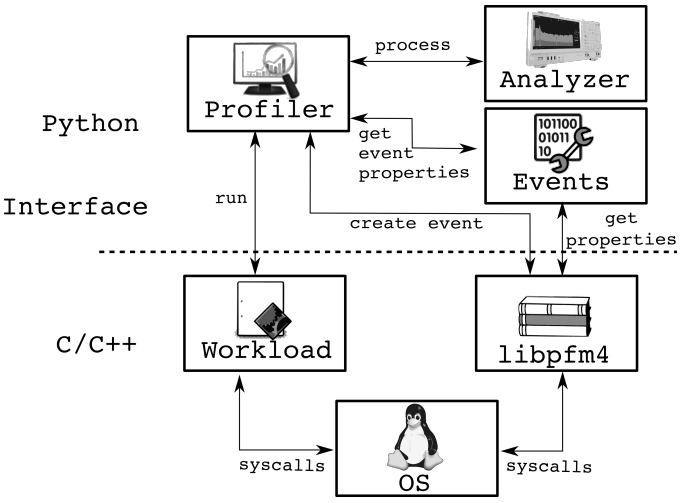

Fig. 1: Modules interconnection

The Profiler is the user interface for configuring and creating events. It is also responsible for calling the Workload module to run the application and retrieve the data after the execution is complete.

The Workload module, developed on C++, is the core of the library. It is responsible for creating the application and the sampling process. It provides a precisely synchronized start, it halts the program before the execution of the first instruction, using the debug interface on Linux (place), and launches the application once the counters have been properly reset and ready to run. This module, built-in as a Python module using the Python C API, defines a set of functions, macros, and variables to access most aspects of the Python runtime system. The Event module provides a description of events, events parameters, configurations and PMUs available.

System calls for reading and event creation are done directly by the Workload module or through the libpfm4 Python link. The libpfm4 is also used to find events encodings and convert an event name (expressed as a string) to the corresponding event encoding, either as a raw event number (as documented by the hardware vendor) or the OS-specific encoding. In the latter case, the library is able to prepare the OS-specific data structures needed by the kernel to setup the event.

The Analyzer module is responsible for the post-processing of the data. It takes the data from several runs of the application and provides a set of functions to remove outlines, interpolate, filter and compare.

B. POST-PROCESSING

With the data collected from multiple runs of the application, the first goal is to obtain a single curve that minimizes the noise caused by the operating system and inaccuracies of the counters. In figure 2 we can see the raw result of multiple runs of the Polybench 2mm program measuring the number of executed instructions. For better visualization, the horizontal axis has been normalized over the interval from 0 to 100. This implies that we no longer analyze the program on the time scale, but in the interval in which it took place.

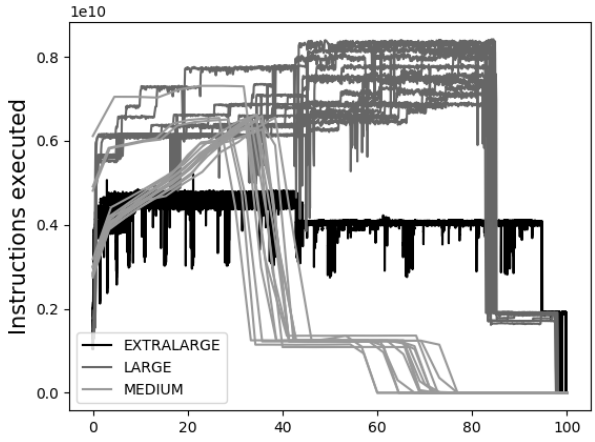

Fig. 2: Multiple executions of the Polybench 2mm program with different input sizes

We apply a median filter to each set of runs, sorting the values and removing edge values. Then we calculate the average curve, whose final result can be observed on the figure 3.

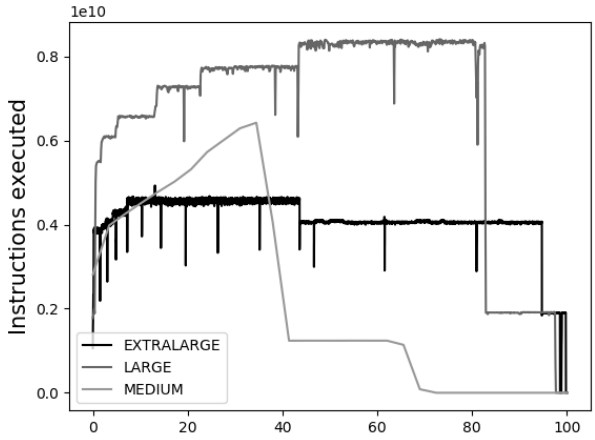

Fig. 3: Single instance representation

After that, we interpolate the curve using the B-spline [19] algorithm to get the same number of points to all curves. The result of this step can be observed in figure 4.

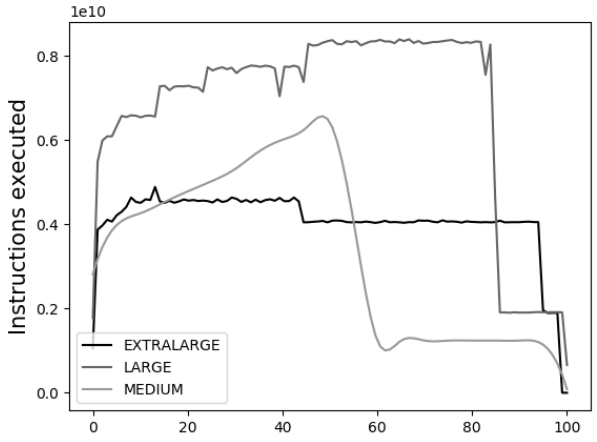

Fig. 4: Interpolation

Finally, we use the Savgol filter [20] in order to smooth the data to increase the signal-to-noise ratio without too much distortion. The result is shown in figure 5.

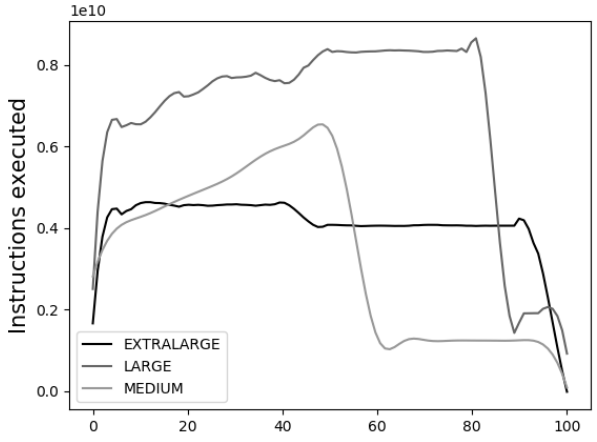

Fig. 5: Filtering

V. RESULTS

In this section, we first show the comparison between our tool and others already established, as well as the clustering process.

A. ACCURACY COMPARISON

To validate the tool, we compared the results of the counters obtained with different APIs. We used the hand-crafted assembly benchmark from [11], designed to test determinism and accuracy of PMUs. We compared the values obtained from the Linux API, PAPI on C and PAPI on Python. The events used for this comparison were instructions retired, branch instructions, memory read, memory load, and arithmetic operations. We ran the benchmark 30 times and calculated the mean and standard deviation as shown in table I. Some events could not be measured using PAPI because the tool does not accept raw events and there are no equivalent events.

Since the benchmark was hand-crafted with assembly, we know exactly the value for some counter events. For this reason, the number of instructions, branch instructions, and conditional branch are pin. However, some other events are architecture-specific and there is no pined value. In the latter case, we can still compare to the Linux API, which should be closest to the reality.

The differences using the Linux low-level API, PAPI, and our tool are negligible (the average percentage distance is less than 0.01% in all the cases). As expected, PAPI on Python had the largest difference (with an average distance of 0.25%) mainly due to an unsynchronized start that resulted in the loss of some instructions at the beginning of the execution. This can be an important problem if the application contains a small number of instructions.

The standard deviation of the 30 executions shows that our tool has the smallest variation on a run-to-run on most events. On the contrary, PAPI on Python shows a big variation compared to the others.

B. CLUSTERING

To cluster applications first we need to have a way to compare two programs, for that we define a new variable that tries to compute a fingerprint to the program, this variable has to look similar when we execute the same program with different conditions and inputs. Thinking in the simplest model of a program as a Turing machine as everything can be done with a tap of memory and set of rules we empirically define the variable input size as described on the equation 1. On real computers this is not too far from reality, most of the computations and input and output operations somehow pass through the memory, so analyzing the relationship of the total number of instructions and memory instructions can give us a good fingerprint.

$$I_{sz} = \frac{I}{I_m} \tag{1}$$

Where I_{sz} we called input size, I the number of instructions executed, I_m number of memory instructions execute. We observe that this variable demonstrate to have the proprieties that we are looking for, produce similar results to the same program with different input size, and environment. This can be used to identify programs but also to see the similarities between different applications.

TABLE I: Comparison

Average*10^{-6}						Standard deviation			
Counters	Pined values	Linux API	PAPI	PAPI Python	MyPerf	Linux API	PAPI	PAPI Python	MyPerf
INSTRUCTIONS_RETIRED	226.99	227	227	225.9	227	396	133	337763	175
BRANCH_INSTRUCTIONS_RETIRED	9.24	9.25	9.25	9.24	9.25	297	208	8485	91
BR_INST_RETIRED:CONDITIONAL	8.22	8.22	8.22	8.21	8.22	0	0	3383	0
MEM_UOP_RETIRED:ANY_LOADS		2484.18			2484.16	37399			38953
MEM_UOP_RETIRED:ANY_STORES		189.96			189.96	1513			687
UOPS_RETIRED:ANY		12291.08			12290.9	345246			333298
PARTIAL_RAT_STALLS:MUL_SINGLE_UOP		0.6			0.6	1222			521
ARITH:FPU_DIV		5.8			5.8	1760			1544
FP_COMP_OPS_EXE:X87		48.79			48.79	1283			3311
INST_RETIRED:X87		17.2			17.2	4			3
FP_COMP_OPS_EXE:SSE_SCALAR_DOUBLE		5.4			5.4	1547			2097

To compute the distance between two programs we use the Canberra metric [21], described on the equation 2.

$$d(p,q) = \sum_{i=1}^{n} \frac{|p_i - q_i|}{|p_i| + |q_i|} \tag{2}$$

Where p and q are n-dimensional vectors.

We compute the input size for all 30 applications of the Polybench with 3 different inputs. Running each program 15 times, with a sampling rate of 0.01 seconds collecting the total number of instruction, number of loads and write instructions, number of floating point operations, besides some software counters. After applying the post-processing to the data collected we compute the input size and perform a Hierarchical clustering using the linkage method of Ward [22], that minimizes the total within-cluster variance. The results of the clustering can be seen on the figure 7 and on the dendrogram on figure 6.

From this dendrogram, we can have an idea of how close two applications are. We choose the number of clusters that maximize the number of hits of the same program with different input sizes on the same cluster, in this case, was 5 clusters.

It was observed that as input size grows the program behavior tends to a specific curve, but for small input sizes some have variation, so in some cases, the same program has been classified in more than one cluster. This can also happen if specific parts of the program are triggered with specific inputs, in which case it will also belong to more than one cluster.

In order to have an overall classification of each program, we can pick up the frequency in which each appeared in the clusters and classified it in the cluster in which it appeared more often. In this case, the clusters are:

- Cluster 1: 2mm, 3mm, cholesky, correlation, covariance, floyd-warshall, gemm, gramschmidt, lu, ludcmp, nussinov, symm
- Cluster 2: deriche, doitgen, syrk
- Cluster 3: adi, fdtd-2d, jacobi-2d, syr2k
- Cluster 4: atax, bicg, durbin, gemver, gesummv, mvt, trisolv, trmm
- Cluster 5: heat-3d, seidel-2d

On the figure, 7 we display the clusters using the spring force algorithm where each program with a specific input is a node and the edge weight is the Canberra distance. To a better visualization, the name of the inputs was replaced by numbers, the EXTRALARGE is 3, LARGE 2 and MEDIUM 1.

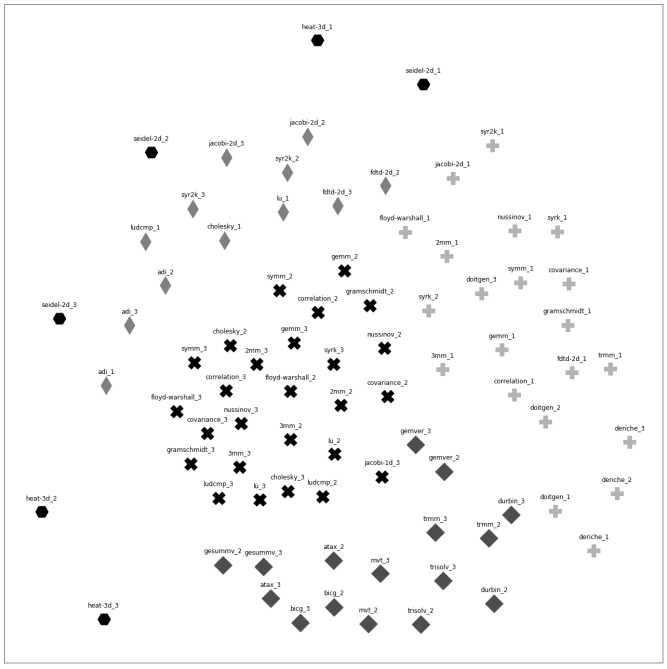

Fig. 7: Spring force of Canberra distance

The figure above shows a different way to observe in a reduced space the distance between clusters and applications and how they are organized. From this graph, we see that clustering it's well partitioned and we can clearly separate each cluster. It is also interesting to note that the applications of the cluster formed by the circle symbol are more separated, which may indicate that they were classified in this way because they did not fit into any other cluster.

To have an idea of how the behavior of the variable input size for each clusters is, it was plotted for the

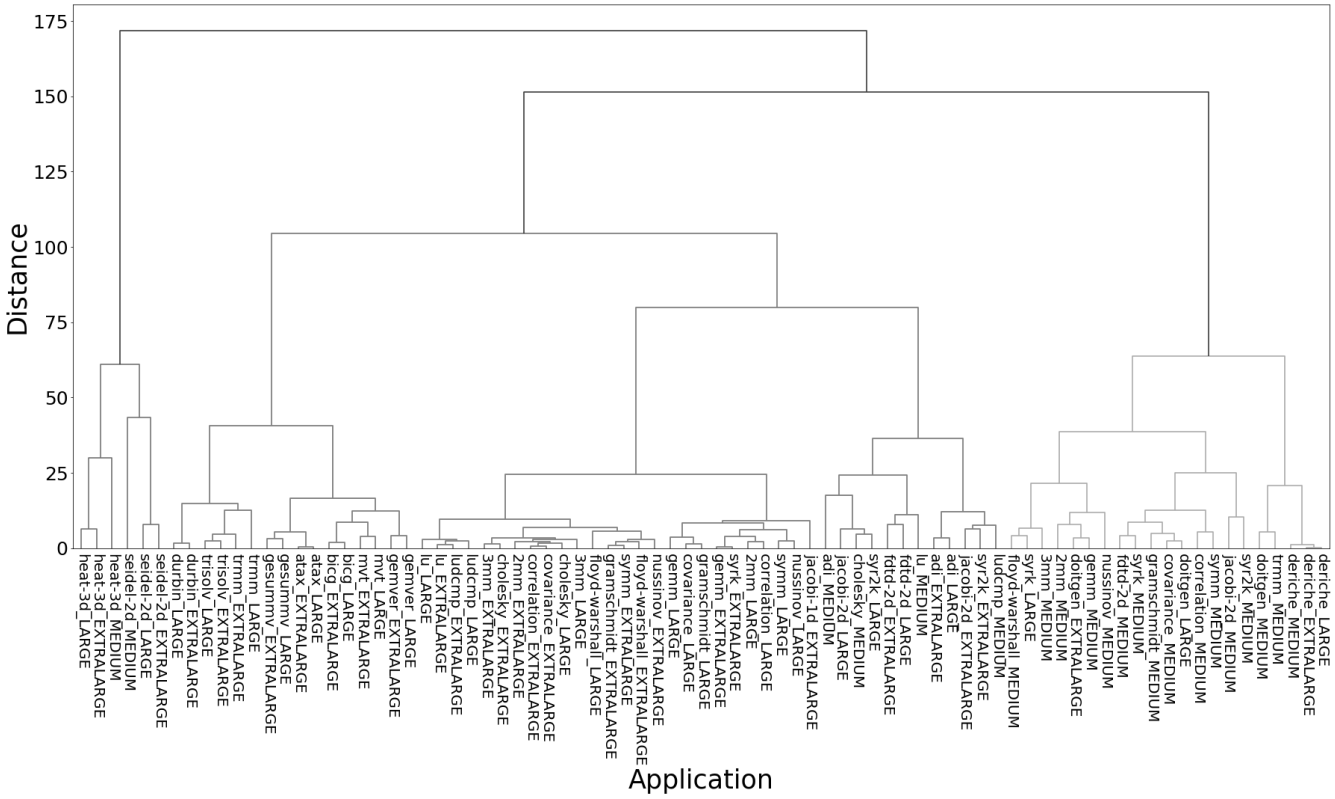

Fig. 6: Dendrogram

applications of clusters 1 and 2 shown on the figures 8, 9.

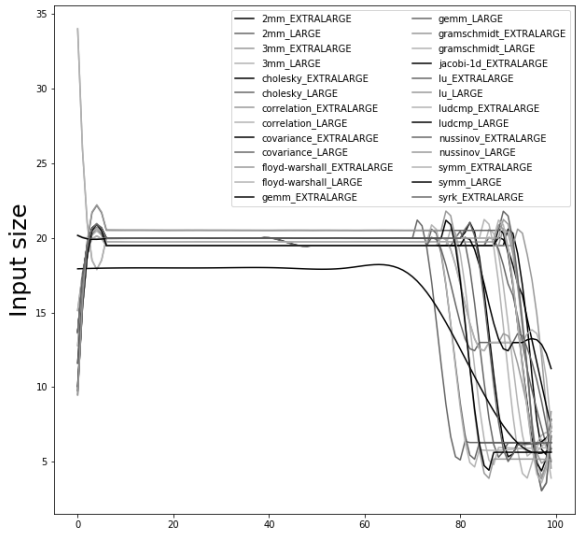

Fig. 8: Input size - Cluster 1

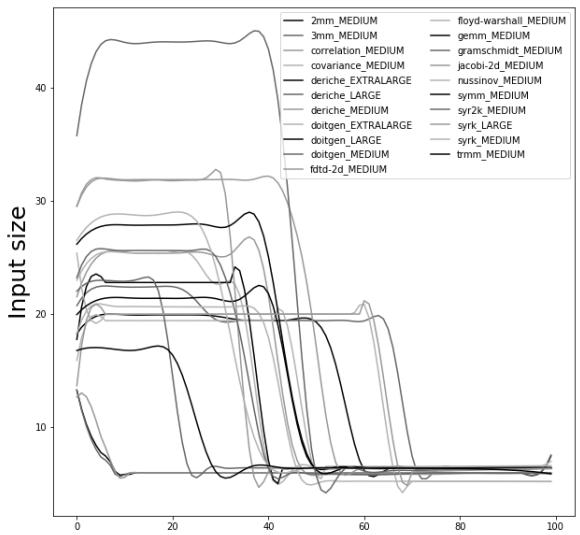

Fig. 9: Input size - Cluster 2

From these figures, we can have an idea of what behavior was classified as the same class. On cluster 1 on the figure 8 all applications showed the same behavior with approximately the same amplitude.

On figure 9 we observe that curves that have similar shape regardless of scale on vertical and horizontal axis

have also been classified as the same clusters. This is the wanted comportment for the classification because we are only interested in the overall shape of the curve.

VI. CONCLUSIONS

From the results, we observe that the API present overhead similar or lower to other low-level APIs, with the advantage of being in high abstraction and simplified configuration with a few lines of code, is possible to configure and gather counter data.

The tool developed provide also provided a way to fingerprint programs and compute similarities between different programs or the same program with different inputs. This can be useful to reduce applications spaces for benchmarks as was done in Polybench clustering but also to analyze the behavior of a parameter providing insights to the programmer to find a possible bottleneck.

We also provide a precise definition to input size that made possible fingerprint the programs, but it also can help programmers of benchmark applications to better create inputs with more precise growth of a particular parameter.

VII. FUTURE WORK

We pretend to use this tool to create a data set of applications behavior and automatically classify any program as belonging to a cluster or a set of clusters. The idea is to have a set of clusters that can describe most applications in this way we can know specific behaviors of the applications. This can be applied to various areas such as voltage scaling and frequency of the processor, once knowing the behavior of a particular program we can have an idea of what is the best strategy to control the frequency in order to save energy or increase performance.

VIII. APPENDIX

This tool and all the analysis are available on github: `github.com/VitorRamos/performance_features`

IX. ACKNOWLEDGMENT

This research was developed in the super-computing centers of the University of MONS in collaboration with the center of the Federal University of Rio Grande do Norte.

References

[1] A. C. de Melo, "The New Linux 'perf' Tools," *Linux Kongress*, 2010.

[2] R. Kufrin, "Perfsuite: An accessible, open source performance analysis environment for linux," *Dans Presented at The 6th International Conference on Linux Clusters: The HPC Revolution*, vol. 151, no. April, p. 5, 2005.

[3] A. Knüpfer and C. Rössel, "Score-P A Joint Performance Measurement Run-Time Infrastructure for," pp. 1–12, 2011.

[4] R. Zamani and A. Afsahi, "A study of hardware performance monitoring counter selection in power modeling of computing systems," *2012 International Green Computing Conference, IGCC 2012*, 2012.

[5] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, p. 559, 2013.

[6] V. M. Weaver, D. Terpstra, H. McCraw, M. Johnson, K. Kasichayanula, J. Ralph, J. Nelson, P. Mucci, T. Mohan, and S. Moore, "PAPI 5: Measuring power, energy, and the cloud," *ISPASS 2013 - IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 124–125, 2013.

[7] P. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," *Proceedings of the department of defense HPCMP users group conference*, vol. 32, p. 710, 1999.

[8] S. Eranian, "Perfmon2 a exible performance monitoring interface,"

[9] S. Eranian, I. Systems, and M. Unit, "The perfmon2 interface specification," *Development*, vol. 200, pp. 0–132, 2005.

[10] V. M. Weaver and S. A. McKee, "Can hardware performance counters be trusted?," *2008 IEEE International Symposium on Workload Characterization, IISWC'08*, pp. 141–150, 2008.

[11] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," *ISPASS 2013 - IEEE International Symposium on Performance Analysis of Systems and Software*, no. April, pp. 215–224, 2013.

[12] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security," *Proceedings - IEEE Symposium on Security and Privacy*, 2019.

[13] "PolyBench/C version 3.2."

[14] Mikael Pettersson., "The Perfctr interface.,"

[15] Intel Corp., "The VTune TM performance analyzer.."

[16] Intel, "Intel® 64 and IA-32 Architectures Software Developers Manual, Volume 3 (3A, 3B & 3C): System Programming Guide," vol. 3, no. 253665, pp. 1–1386, 2013.

[17] N. Mc Guire, P. Okech, and G. Schiesser, "Analysis of Inherent Randomness of the Linux Kernel," *Eleventh RealTime Linux Workshop*, 2009.

[18] S. Eranian, "Perfmon2: a standard performance monitoring interface for Linux," *Slides, perfmon2 overview*, 2008.

[19] H. Hang, X. Yao, Q. Li, and M. Artiles, "Cubic B-Spline Curves with Shape Parameter and Their Applications," *Mathematical Problems in Engineering*, vol. 2017, pp. 1–8, 2017.

[20] J. Luo, K. Ying, P. He, and J. Bai, "Properties of Savitzky-Golay digital differentiators," *Digital Signal Processing: A Review Journal*, vol. 15, no. 2, pp. 122–136, 2005.

[21] G. Jurman, S. Riccadonna, R. Visintainer, and C. Furlanello, "Canberra distance on ranked lists," *Proceedings, Advances in Ranking-NIPS 09 Workshop*, pp. 22–27, 2009.

[22] F. Murtagh and P. Legendre, "Ward's Hierarchical Clustering Method: Clustering Criterion and Agglomerative Algorithm," no. June, pp. 1–20, 2011.