POSEIDON: Analyzing the secrets of the Trident Node monitoring

AUGUST 2018

AUTHOR(S):
Leonardo Xavier Kuffo Rivero

IT-DI-WLCG: Understanding Performance

SUPERVISOR(S):
Servesh Muralidharan.
David Smith.

CERN openlab

PROJECT SPECIFICATION

Trident is a novel node load characterization tool that can look at various metrics with respect to the Core, Memory and I/O. Trident uses a three-pronged approach to analyze the node's utilization and understand the stress on different parts of the node based on the given workload with the goal to optimize the overall performance. Trident is different from other profiling tools by focusing only on fundamental hardware and software counters where monitoring does not induce any significant overhead to the execution of the application. It is also different from other tools that work at the application level which only look at limited metrics such as CPU usage and memory consumption. Currently core metrics such as memory bandwidth, core utilization, active processor cycles, etc., are being collected with minimal overhead. The interpretation of this data requires often a deep understanding of the system at hardware and software level.

POSEIDON aims to provide automated data analysis on the data monitored by Trident to construct feedback over the applications that can be understood and exploited by the average developer. In addition to this, POSEIDON will be able to profile an application workflow under several metrics over a timeline by computing similarity measures between them and specialized benchmarks. Poseidon achieves its goals by using multi-variate time series analysis methodologies and machine learning techniques.

ABSTRACT

Improving the performance of an application is an important objective carried out from the application conception until its deprecation. Developers are constantly trying to improve the performance of their applications, either by using more computationally efficient algorithms, migrating to distributed processing platforms or doing intensive computation tasks on modern GPU's. This is no surprise, given that achieving even a slight improvement in the performance of an application can be translated into the saving of time and economical resources. Every component of a system is responsible for the running time of an application, which is the most tangible representation of performance. Therefore, monitoring the activity of these components on application runtime is essential in order to understand and optimize the application. Trident is a profiling tool which does the latter by monitoring the relevant *hardware and software* counters throughout the execution of an application at the node level, without inducing significant overhead. However, data produced by Trident often requires a deep understanding of computer systems at a hardware and software level to be correctly interpreted. The present study introduces our framework *Poseidon* which aims to profile and describe a running application based entirely upon specialized benchmarks, turning trident data into knowledge for developers. Poseidon use a novel approach of Multi-variate time series classification denominated WEASEL + MUSE (Word ExtrAction for time SEries cLassification + Multivariate Unsupervised Symbols and dErivatives), to measure the degree of similarity between the Trident monitored metrics of these benchmarks and the running applications. Applying our framework, we were able to profile our ground-truth data with a 96.36% of accuracy. Moreover, we satisfactorily discovered different workflow phases embedded in real applications and profiled their Trident data based upon the knowledge present on benchmarks.

TABLE OF CONTENTS

1. INTRODUCTION

Understanding the performance of an application can be a challenging task. Many factors such as the application complexity, code design and node infrastructure have direct influence on its performance. In fact, the smallest performance improvements can be a great accomplishment, taking into account that this enhancement may represent costs reduction in time, money and resources. Therefore, it is no surprise that developers are constantly eager to improve the performance of the applications they develop. Either by using more computationally efficient algorithms, migrating to distributed processing platforms or doing intensive computation on modern GPU's instead of CPU's.

Performance can be described as simple as the running time of an application. Which is, the most tangible representation of good/bad performance. On the other hand, we can dig deeper to analyze the performance of each of the components of a modern system (Figure 1).

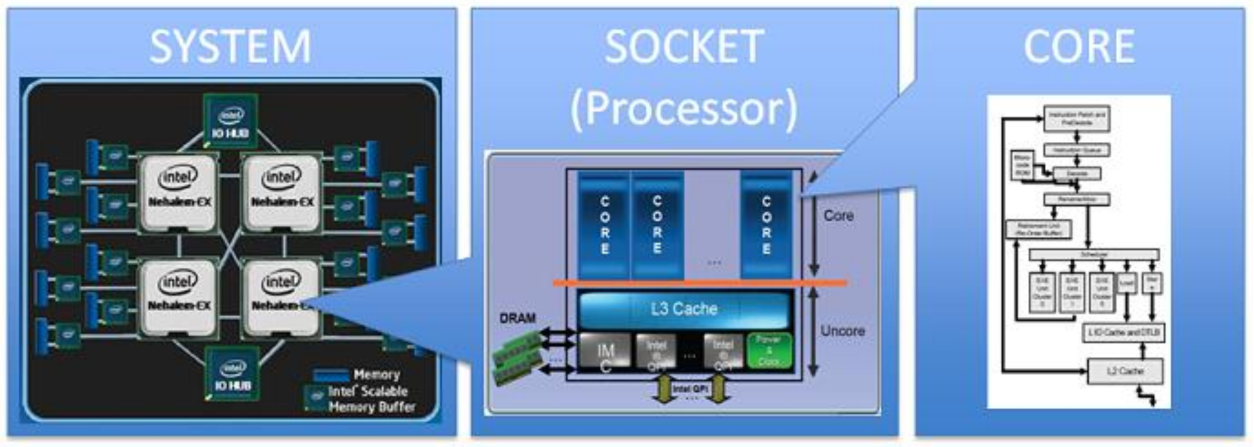

*Figure 1. Modern multi-processor and multi-core system. **(Image taken from Willhalm et al. 2012 [1]).***

When an application is being executed on a system, each of the system components reacts in a particular way. Despite the complexity of these systems, it is possible to monitor its components "reactions" at application runtime. This is advantageous to profile the behavior of an application, resulting into possible fault-detections and further software optimization [2]. In addition to this, monitoring can also help us do in-depth performance analysis [2].

As exposed in [2], a considerable amount of applications runtime monitoring tools have been developed over the years. Each one of these tools have unique characteristics and use monitoring to accomplish specific goals, such as system fault detections, monitoring oriented programming or overhead optimization. Most of them are attached to a specific programming language and are focused on software counters. Moreover, each of them has different implementations (e.g. single processor, multi-processor), placement (e.g. in line, off line, synchronous), monitoring points (i.e. application stage where the motoring will start), etc.

Nonetheless, the exposed constraints, usage limitations and induced overhead of these tools is an obstacle to overcome for the CERN IT-DI-WLCG: Understanding Performance team. Hence, they have developed a novel node load characterization tool: **Trident**. Trident uses a three-pronged approach to analyze the node's utilization and understand the stress on different parts of it. Trident can look at various metrics with respect to the Core, Memory and I/O by focusing only on fundamental hardware and software counters where monitoring does not induce any significant overhead to the execution of the application. However, Trident monitored data does not represent useful information for the average developer, since the correct interpretation of this data often requires a deep understanding of the system at the hardware and software level.

In this report, we propose a framework called **Poseidon.** Poseidon aims to mitigate the presented problem by transforming the raw data of trident into information that can be utilized by the average developer. Poseidon use Multi-variate Time Series classification techniques to compute similarity measures between trident data obtained from specialized **benchmarks** and trident data obtained from the developed **applications**. Hence, Poseidon proceeds to profile and describe the performance of the latter at a high-level based on these benchmarks. In doing so, we make the following contributions:

- We present a novel strategy to analyze Trident data from different workflows in an optimal and quick fashion. This information allows us to compare different workflows and transform Trident data into knowledge for applications developers.
- We propose a method to find the different phases or segments embedded in a long-running application workflow.

This document is structured as follows: Section 2 describe and illustrate the data collected by Trident. Next, in Section 3 we present with high granularity the methodology used to conduct the experiments to analyze the data. Section 4 present the results of the conducted experiments. Section 5 comprise the conclusions of our work, the methodology limitations and motivations for further research. Finally, we provide a description on the implementation of Poseidon in Section 6.

2. TRIDENT TOOL DATA

Trident is a monitoring and load characterization tool, designed to record the relevant hardware and software counters produced by the systems components throughout the execution of an application. Trident novelty relies on the completeness of the retrieved information and the absence of significant overhead in the applications that run on par with it.

Trident is currently collecting 94 metrics (refer to Appendix A.) for any workflow being executed at the node level on par with it. Each of these metrics can be described as a time series (TS). Formally, a time series T is defined as a *sequence* of $n \in \mathbb{R}$ real values being recorded in chronological order, $T = (t_1, t_2, t_3, \dots t_n), t_i \in \mathbb{R}$. Thus, when we retrieve Trident data for any workflow, we are in front of 94 time series. Since each of these time series are being recorded in parallel over time (i.e. Trident retrieving hardware and software counters every ~10ms), we can describe the obtained data as Multi-variate Time Series (MTS). Figure 2 shows 6 dimensions (i.e. metrics) of Trident data plotted in area charts.

As of the writing of this report, there is no approach to profile a running application just by analyzing the monitored Trident data. In order for Poseidon to describe these running applications, we built knowledge using **benchmarking workflows**, and recorded the Trident data obtained from them. Benchmarking workflows differ from applications workflows in the knowledge embedded in them. **Benchmarks** are application of which the team holds enough knowledge to describe them at a high granularity.

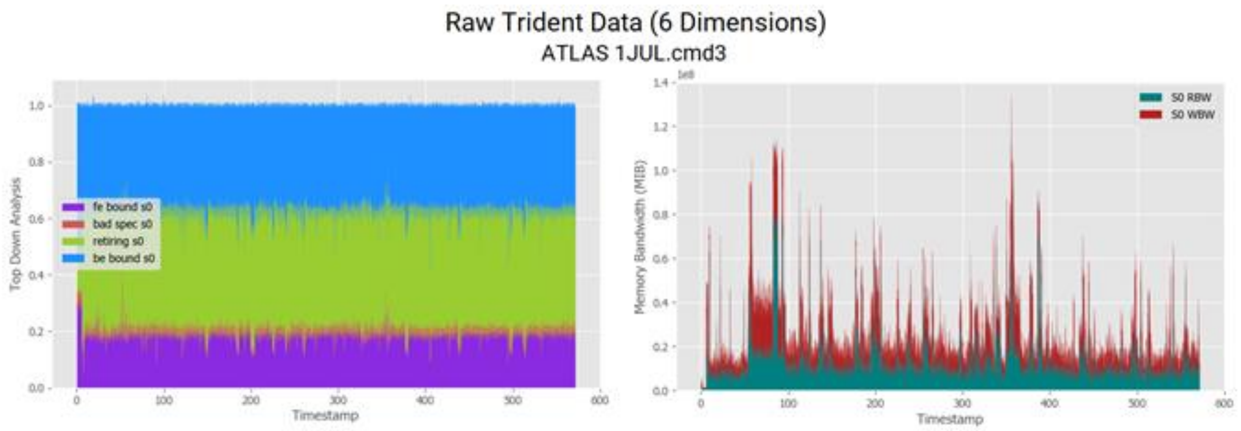

Figure 2. MTS raw Trident data measured on ATLAS 1JUL cmd3 experiment (6 metrics).

3. METHODOLOGY

In order to profile applications at a high level we propose a methodology which is entirely built upon benchmarks. Poseidon measures at which extent these benchmarks share similarity with applications, to then proceed on profiling them based on the knowledge present in the benchmarks.

Data recorded by Trident has some unique characteristics. One of them, is the high fluctuation of the monitored metrics in a small-time window (Figure 3). In addition to this, trident TS length can be variable between applications, and benchmarks. Hence, it would be impractical to analyze this type of data with whole series-based analysis methods (i.e. point wise comparisons). Hence, we transform the raw trident data from benchmarks and applications following a Bag of Patterns (BOP) approach, as described in [3], to obtain adequate features from the raw Time Series.

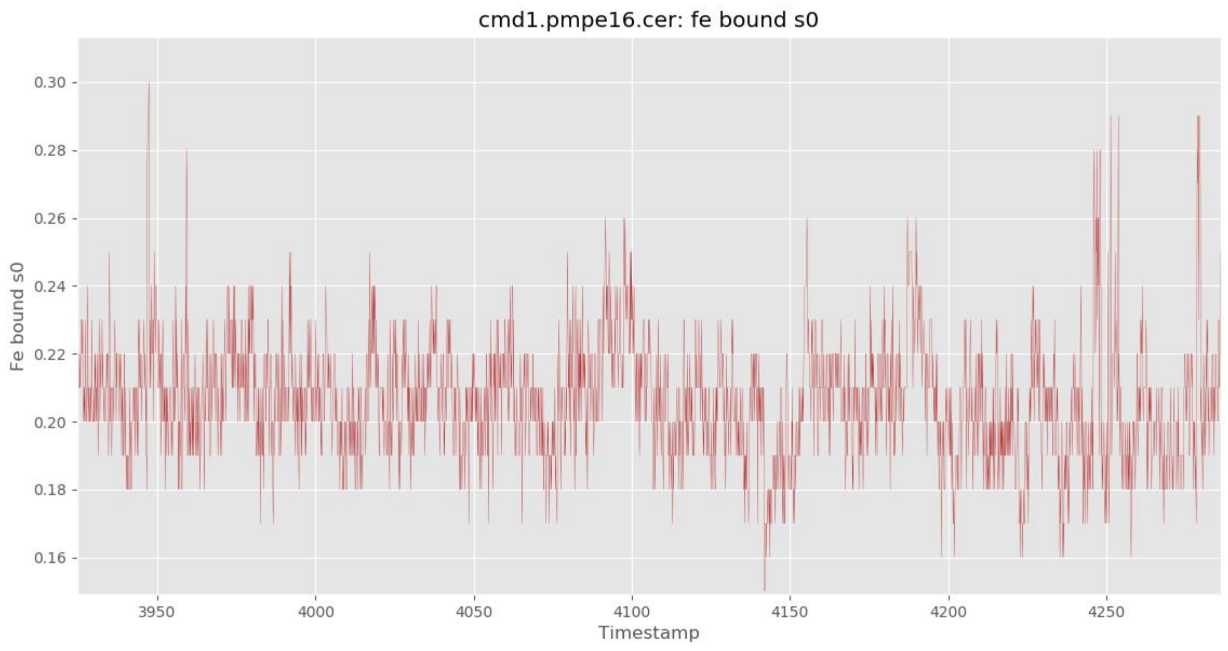

Figure 3. Application ATLAS CMD1 fe bound S0 time series plot showing high fluctuation of values.

BOP is the analog of Bag of Words approach in text mining problems. This BOP representation let us use feature-based methods to analyze trident data [8] which let us work freely with time series of different length, being invariant to "noise", and providing us with high computational scalability for long time series analysis. BOP model construction can be described in the following steps: 1) The TS is subject to a windowing process using a sliding window. 2) Each window is normalized. 3) Every window is converted to a Symbolic Aggregate Approximation (SAX) string representation. This step is called discretization. Discretization objective is to produce a lower dimensional representation of a TS transforming the raw data into a symbolic word (Figure 4). 4) The set of symbolic words is converted to a word-sequence matrix. In which every row denotes each obtained SAX string, every column represents each time series in the dataset, and every cell contains the number of times that a SAX string appeared among the different time series. Finally, this word-sequence matrix, which is the BOP model, can be used for further classification tasks.

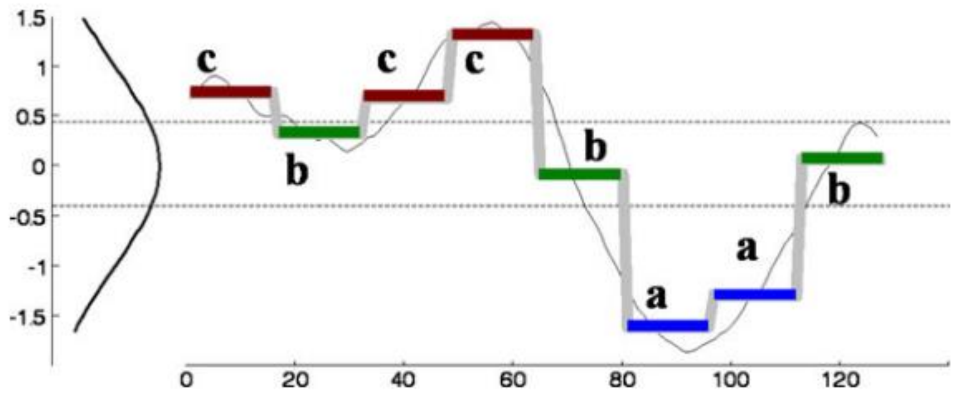

Figure 4. Time Serie discretization illustration. The discretization process encodes the string 'cbccbaab', reducing the dimensionality of the TS from 128 to 8. ***(Image taken from J. Lin et al. 2012 [8]).***

Once the benchmarks and applications trident data has been preprocessed (i.e. transformed to a BOP model), we use the benchmarks data to train a multi-variate TS classification model in which the labels to predict (i.e. classes) are each benchmark. Then, we use the trained model to find the degree of similarity of the application with each benchmark and obtain the one with the highest probability of sharing similar properties. Finally, we profile the application by mapping the results of the classification with the knowledge embedded in the benchmarks. It is important to highlight an additional step in the process, in which we divide the application in *segments*. Each of these steps is explained in detail in the following subsections. Entire pipeline is presented at a high level in Figure 5.

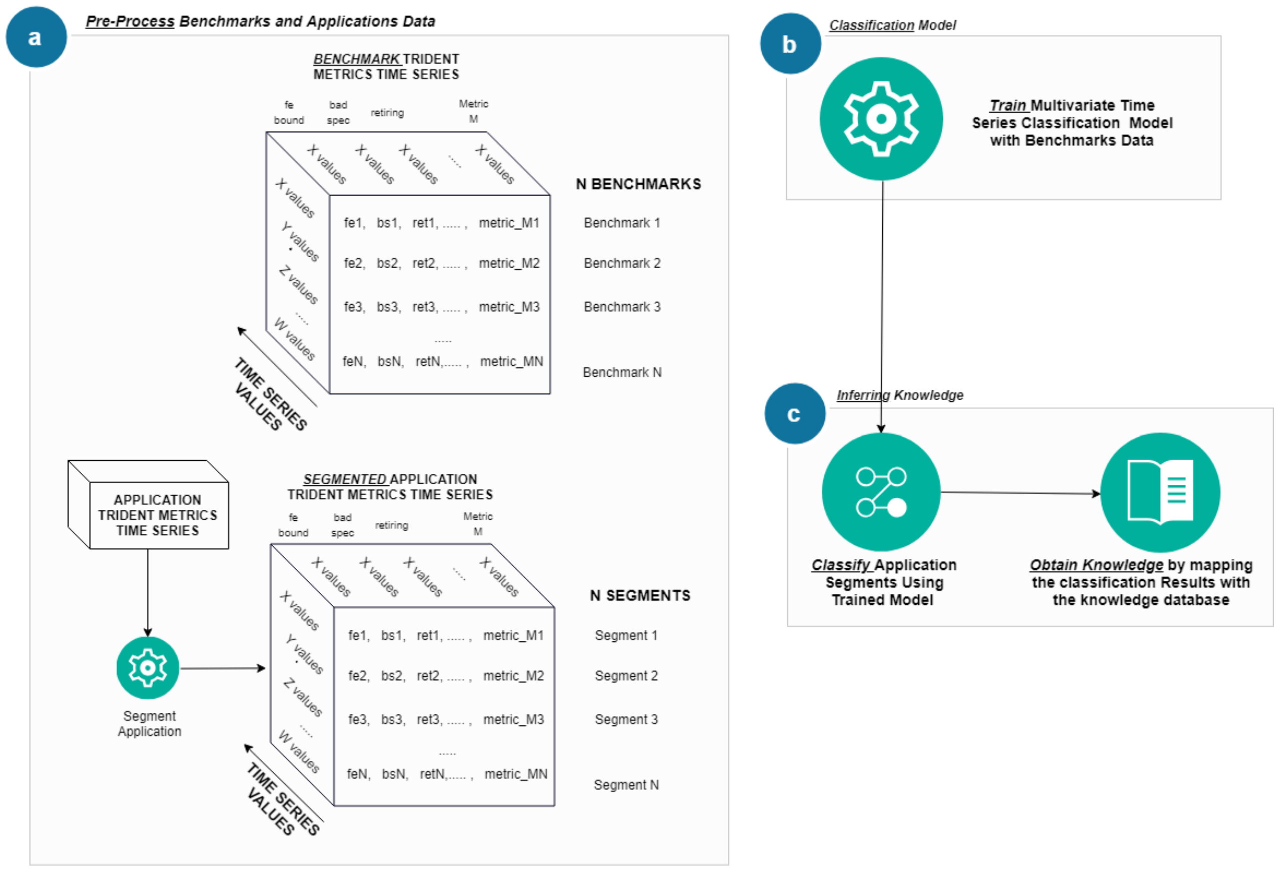
a
Pre-Process Benchmarks and Applications Data
BENCHMARK TRIDENT METRICS TIME SERIES
fe bound
bad spec
retiring
Metric M
X values
Y values
Z values
W values
N BENCHMARKS
fe1, bs1, ret1, ..., metric_M1
fe2, bs2, ret2, ..., metric_M2
fe3, bs3, ret3, ..., metric_M3
feN, bsN, retN,....., metric_MN
Benchmark 1
Benchmark 2
Benchmark 3
Benchmark N
TIME SERIES VALUES
APPLICATION TRIDENT METRICS TIME SERIES
Segment Application
SEGMENTED APPLICATION TRIDENT METRICS TIME SERIES
N SEGMENTS
Segment 1
Segment 2
Segment 3
Segment N
b
Classification Model
Train Multivariate Time Series Classification Model with Benchmarks Data
c
Inferring Knowledge
Classify Application Segments Using Trained Model
Obtain Knowledge by mapping the classification Results with the knowledge database

Figure 5. High-level abstraction of Poseidon pipeline.

a. PRE-PROCESS BENCHMARKS & APPLICATIONS TRIDENT DATA

We can categorize the approaches to conduct Time Series analysis into two groups: Whole series-based methods (i.e. point-wise time series comparison) and feature-based methods (i.e. computing invariant features from time series) [8]. As we previously stated, the unique characteristics of trident data prevents us from using whole series-based methods to conduct our analysis. In addition to this, it has been proven that the use of common distance measures as Euclidean Distance or Manhattan Distances on point-wise time series comparison methods (i.e. whole series-based methods), is meaningless in Time Series classification and clustering problems [5, 6]. Moreover, it has been shown that common techniques such as Dynamic Time Warping and 1-NN Euclidean Distance for whole series-based TS classification and clustering fail with noisy or long TS, among many computational complexity issues [3].

In order to make Trident data digestible and useful for any model, we transform our benchmarks and applications TS using **WEASEL** (Word ExtrAction for time SEries cLassification). WEASEL is a novel approach which follows the Bag of Patterns (BOP) model previously introduced. Since WEASEL approach is based on BOP, it is robust to noise, present high scalability, and deal with variable lengths and offset on time series. WEASEL implements improvements over each step of the BOP approach. The most relevant improvement of WEASEL is the use of Symbolic Fourier Approximation (SFA) [13] instead of SAX (Symbolic Aggregate Approximation) in the discretization step. SFA approximates the raw time series data using discrete Fourier Transformations and quantization. WEASEL also makes use of co-occurrent SFA words (i.e. bigrams), sliding windows of different length and the use of an aggressive Chi-Squared test for filtering only the most relevant features that distinguish each class on the classification model.

i. SEGMENTING APPLICATION

Applications can usually run for many hours, or even days. In addition to this, they are not limited to run isolated tasks but many different tasks, one after another. This characteristic makes them harder to be profiled as a whole. Therefore, we propose a simple but effective methodology to divide an application in *segments* before transforming them into a BOP model with WEASEL. We define a *segment* as a group of points within the time series whose statistical properties differs from the group of points surrounding them. To find these segments we use a Change Point Detection (CPD) technique, called Wild Binary Segmentation (WBS) [7].

As described in [7], WBS select random data subsamples from a TS and calculate the CUmulative SUM (CUSUM) statistic for each of the subsamples. After maximizing each CUSUM statistic, the largest of the entire set of CUSUMs is chosen as a possible candidate for being a change point. If the candidate passes a simple test against a previously defined threshold, it is chosen as change point. After this, the TS is divided in two segments by this change point (i.e. the segments at the left and right of the point), and the process is repeated recursively with each of the segments until no candidate pass the threshold test (Figure 8).

b. CLASSIFICATION MODEL (MUSE + WEASEL)

Benchmarks data is going to be used to train a Multi-variate time series classification model. To obtain the most relevant features for classification from the MTS, we are going to use a multi-variate extension of the WEASEL approach we introduced in the previous subsection, called WEASEL + MUSE (Multivariate Unsupervised Symbols and Derivatives) [4]. WEASEL + MUSE follows the same approach of WEASEL (i.e. building an enhanced BOP model from the Time Series), with every TS *on every dimension*. To distinguish the SFA strings of different dimensions, identifiers are concatenated to them. In addition to this, when building the classifier, it considers the interplay of dimensions (i.e. giving higher weights if two features from different variables co-occurrence is relevant to the likelihood of a class) and it consider each dimension TS derivatives into the analyzed samples [4]. Entire pipeline of WEASEL + MUSE transformation is presented in Figure 6. Finally, a logistic regression classifier is used to train the model using the features obtained from WEASEL + MUSE.

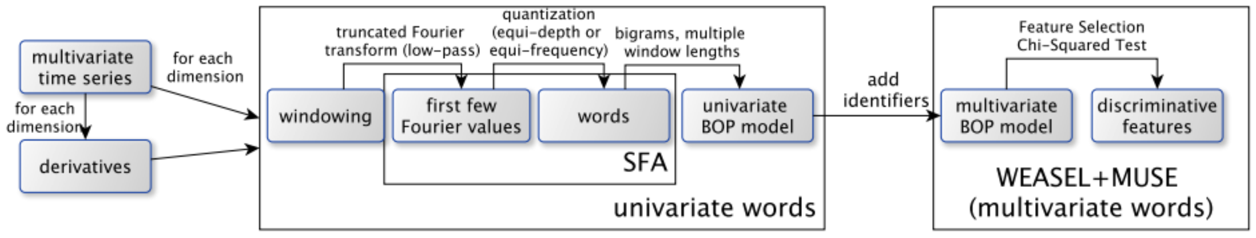

Figure 6. WEASEL + MUSE pipeline. [(Image taken from Schäfer et al. 2017 [4]).]

c. INFERRING KNOWLEDGE

We use the trained MTS classification model to predict the degree of similarity of each of the application segments with each available benchmark. We pick the benchmark with the highest probability of sharing similar properties with every segment. Using the benchmarks knowledge and the probability of a segment belonging to the class (i.e. benchmark) given by the model, we generate a report to the user.

4. RESULTS

Benchmarking applications runs are usually executed a certain number of times (i.e. iterations of the same process). In our case, most of the benchmarks had *three iterations*. Due to many factors, the speed they are executed can diverge, resulting on these iterations being out of phase from each other. When trying to find the most representative value of the three iterations (i.e. an average signal) the latter can be a problem. However, since all of the iterations are from the same nature, we used the Dynamic Time Warping (DTW) [10] algorithm to find an optimal match for every point in the three iterations (Figure 7). Using the matching points, we can calculate an average signal with the most representative values for the given benchmark (Figure 8). Nevertheless, we *did not* follow this approach in Poseidon methodology, given that it reduces to one the number of samples within a benchmark, which is counterproductive for the model training.

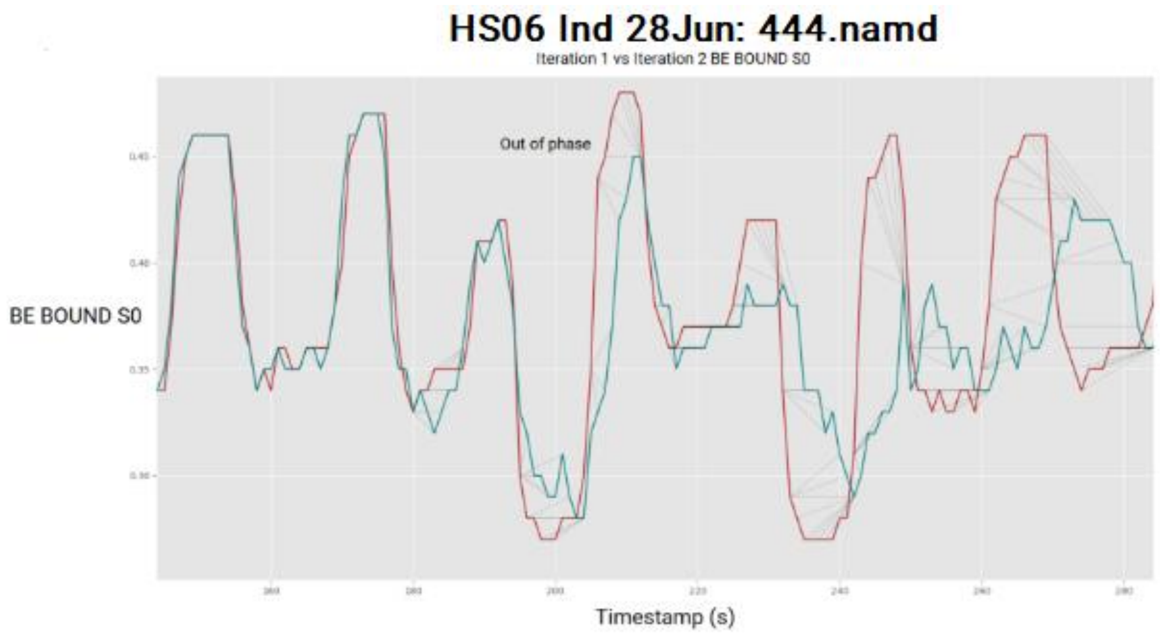

Figure 7. Two Iterations of a HS06 28 Jun 444 Benchmark be bound metric. Using Dynamic Time Warping the best matching points from the two signals are computed (black connection), even though they are out of phase.

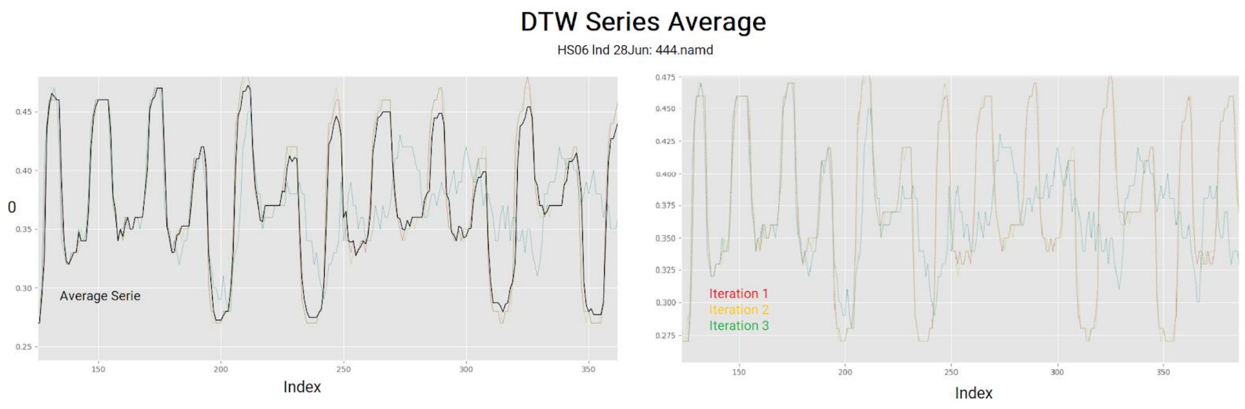

Figure 8. The figure in the right comprise the three iterations of HS06 28JUN 444 benchmark be bound s0 metric. The figure on the left, shows the average signal computed by applying DTW to the three benchmarks iterations and obtaining an arithmetic average of the resulting matches.

We conducted our first experiments of the proposed methodology using two of the three iterations to train the MTS classification model and the last iteration to evaluate it. Using 8 dimensions (i.e. top down analysis on s0 & s1) an accuracy of 100% was achieved. Afterwards, we repeated the same experiment, but randomly subsampling the evaluation data (i.e. the last iterations), to simulate the segmentation process. We use these random segments to evaluate the model. Using 8 dimensions (i.e. top down analysis on s0 & s1) an *accuracy of 96.36%* was achieved.

Before testing Poseidon with real applications workflows, we start by applying our segmentation process to them. When tested on CMD3 ATLAS 1JUL experiment, WBS performed as expected. Figure 9 shows the process of segmentation. Figure 9.a & Figure 9.b shows how WBS find different phases in the application workflow without being sensitive to noise.

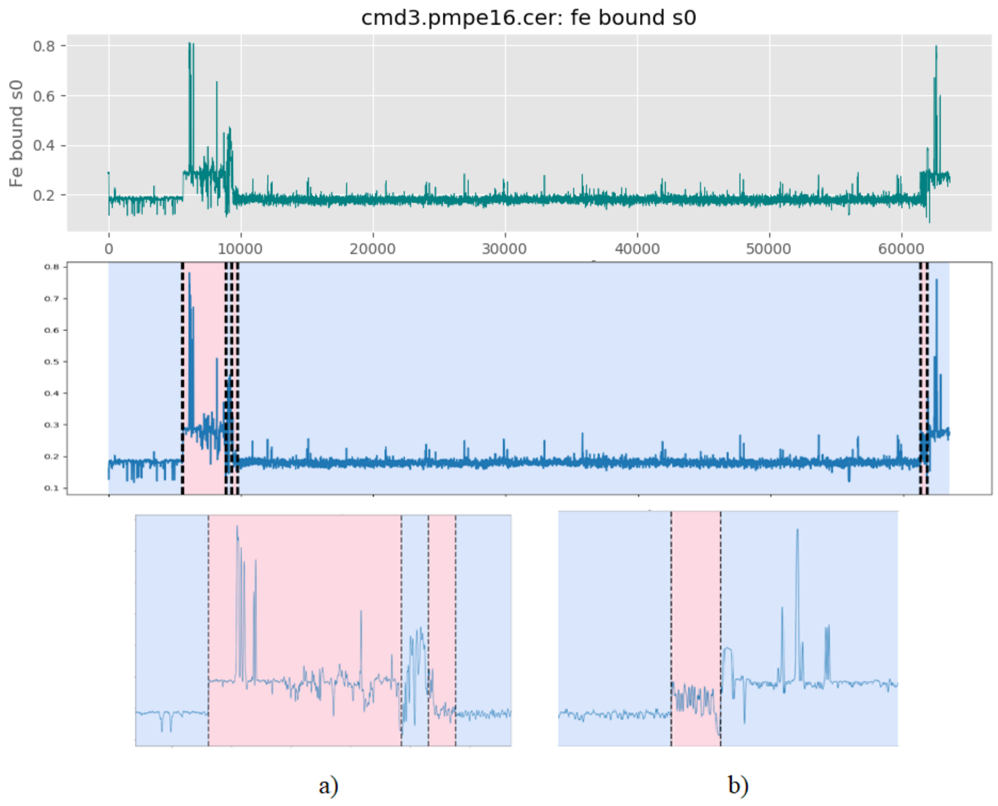

Figure 9. Wild Binary Segmentation [4] automatically finding segments in CMD3 ATLAS 1JUL experiment on fe bound s0 metric. a) and b) are expanded images of the first and last section on the plot.

After computing the segments for an application, we proceed to pre-process and transform the metrics (WEASEL + MUSE) in order to build the classification model. The results obtained from Poseidon reflected segments of the applications which were not similar at all with benchmarks. However, there were other segments which the model reflected a high probability of belonging to a benchmark class. Figure 10 & Figure 11 reflects the Poseidon results of both cases. It is important to highlight, that we only used 12 dimensions for these results (Top down analysis S0 & S1, and RBW / WBW S1 & S0).

POSEIDON - 16:17:54 | August 21, 2018

Application No.1 | ATLAS_1JUL.cmd3

1th segment out of 7 | Length: 5610 timestamps {0 - 5610} | 8.833% of the total application.
POSEIDON computed this application segment is similar to HS06_Ind_28Jun.483, with a 70.67% of probability.

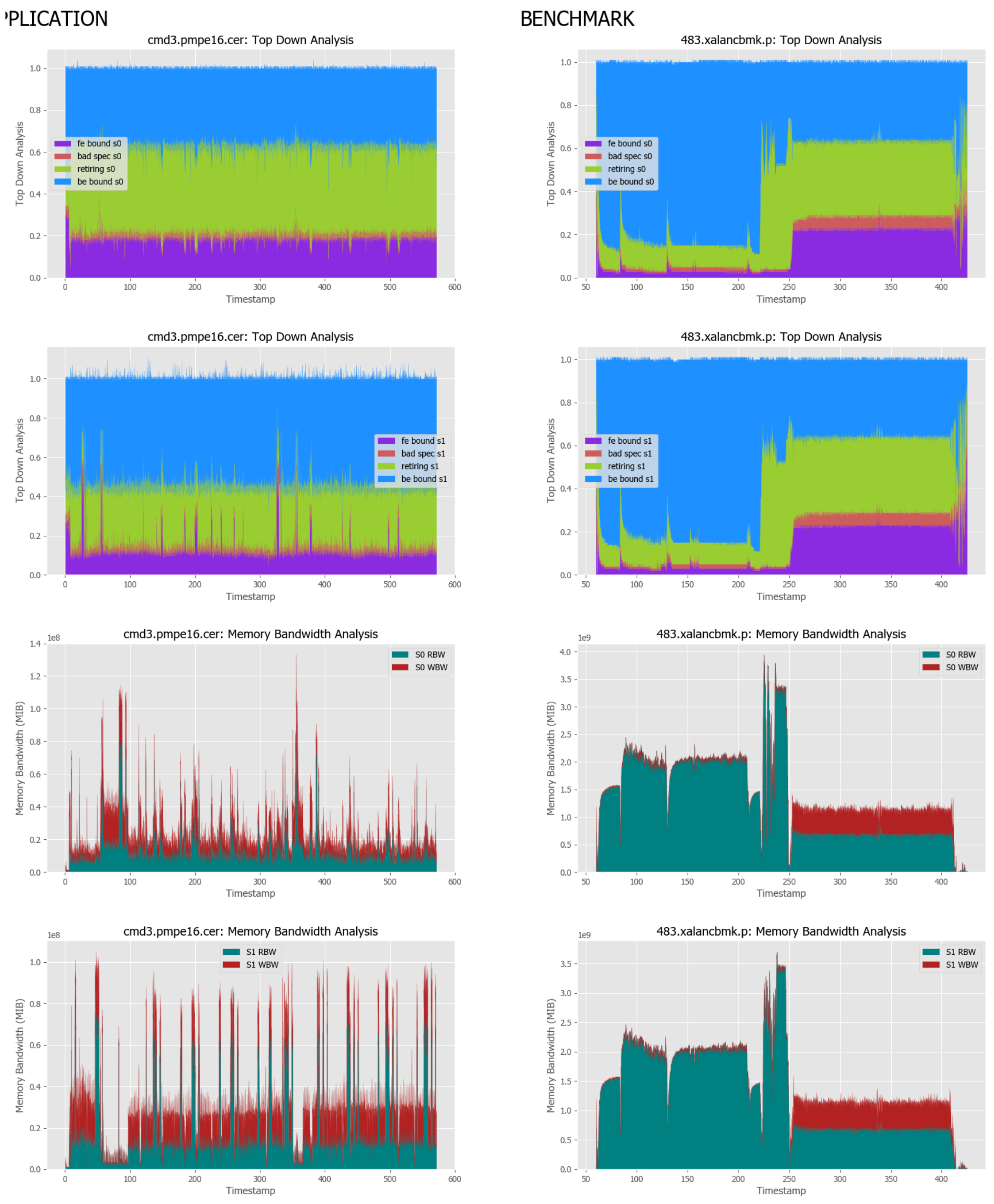

*Figure 10. Poseidon report for a segment which reflected **high** probability of belonging to a benchmark class.*

POSEIDON - 16:30:56 | August 21, 2018

Application No.2 | ATLAS_1JUL.cmd2

18th segment out of 25 | Length: 420 timestamps {81630 - 82050} | 0.452% of the total application.
POSEIDON computed this application segment is similar to SP17_Ind_Run.541, with a 8.893% of probability.

APPLICATION

BENCHMARK

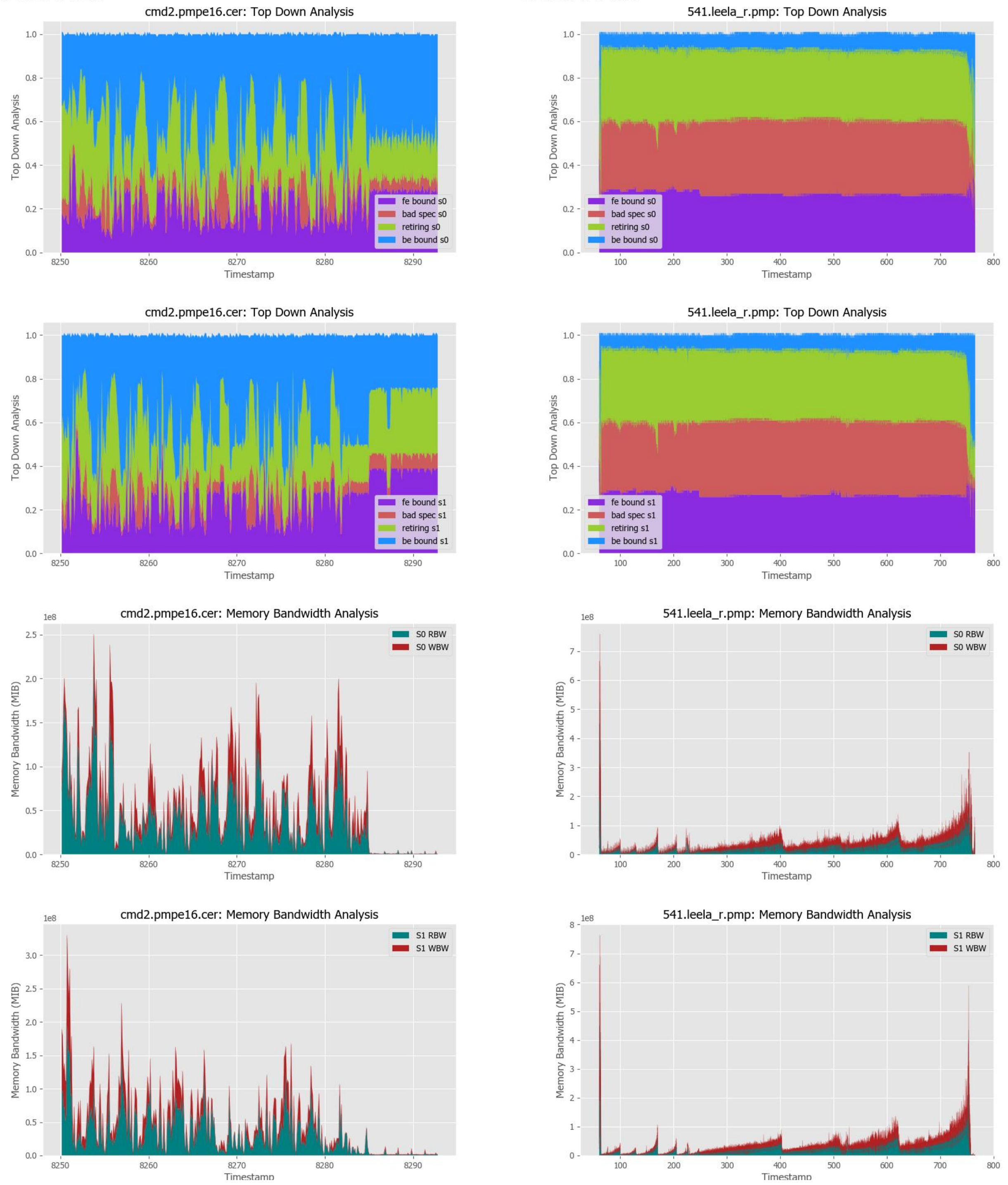

Figure 11. Poseidon report for a segment which reflected **low** probability of belonging to a benchmark class.

5. CONCLUSIONS & FUTURE WORK

This report exposed the first phase of Poseidon, in which we successfully analyzed data monitored by Trident to describe applications workflows based on the knowledge embedded on specialized benchmarks using WEASEL + MUSE novel MTS classification approach. Furthermore, we successfully used Wild Binary Segmentation (WBS) to find the different segments embedded on large application workflows. We tested our proposed framework with ground-truth data, obtaining a 96.36% of accuracy. Moreover, we tested Poseidon on real applications workflows obtaining preliminary but promising results. As expected, most of the segments obtain a low highest probability of belonging to a benchmark class. However, a few of them did achieve a high highest probability ($> 70\%$) of being similar to a benchmark. The latter have to be analyzed in-depth in future research.

However, these results are currently limited by the richness of the benchmarks repository and the amount of knowledge present on it. In addition to this, the features generation and weighting of WEASEL + MUSE leave us clueless of the most relevant Trident metrics for the model. This happens since WEASEL + MUSE generated features are transformed discretizations of the original metrics.

Nevertheless, Poseidon can start to grow upon our work. We encourage future work to try and contrast WEASEL + MUSE with other techniques of MTS classification, such as the one described in [9], in which Short – Long Term Memory Fully Convolutional Networks (LSTM - FCN) are used for high-dimensionality MTS classification. In addition to this, we highly recommend the construction of a ground-truth applications repository, in order to correctly compute performance measures on the trained classifiers or more basic methodologies as *baselines*. Moreover, *find the most relevant trident metrics for the model is still an important unfinished task*. Finally, we encourage future works to take Poseidon further beyond by adding more functionalities to the framework such as, the measurement of the variation in execution “speed” of two equal applications running in different systems architectures using Dynamic Time Warping algorithm [10] or the generation of synthetic benchmarks starting from application segments that could not be profiled by Poseidon.

6. IMPLEMENTATION

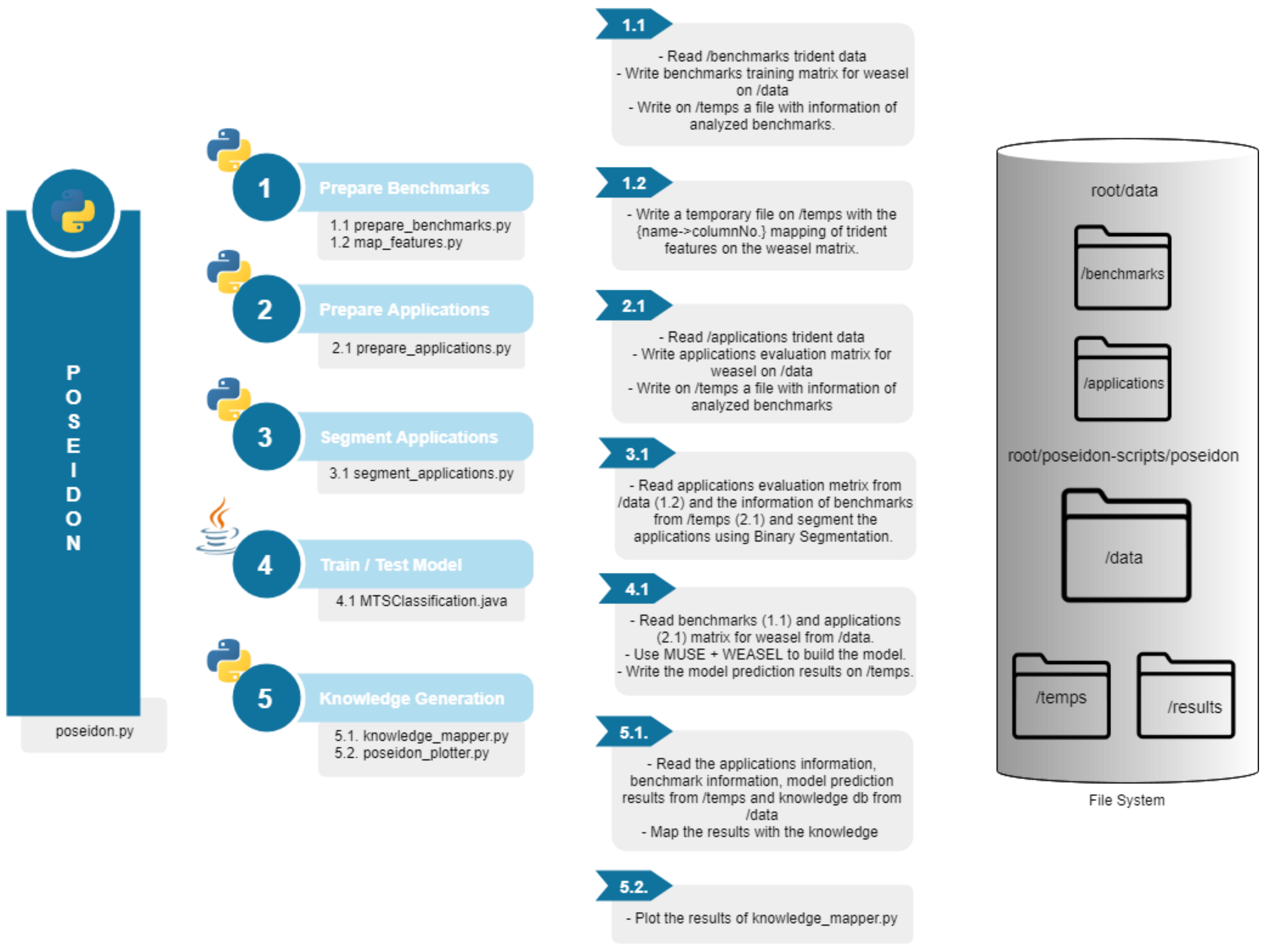
P O S E I D O N
poseidon.py
1 Prepare Benchmarks
1.1 prepare_benchmarks.py
1.2 map_features.py
2 Prepare Applications
2.1 prepare_applications.py
3 Segment Applications
3.1 segment_applications.py
4 Train / Test Model
4.1 MTSClassification.java
5 Knowledge Generation
5.1. knowledge_mapper.py
5.2. poseidon_plotter.py
1.1
- Read /benchmarks trident data
- Write benchmarks training matrix for weasel on /data
- Write on /temps a file with information of analyzed benchmarks.
1.2
- Write a temporary file on /temps with the {name->columnNo.} mapping of trident features on the weasel matrix.
2.1
- Read /applications trident data
- Write applications evaluation matrix for weasel on /data
- Write on /temps a file with information of analyzed benchmarks
3.1
- Read applications evaluation metrix from /data (1.2) and the information of benchmarks from /temps (2.1) and segment the applications using Binary Segmentation.
4.1
- Read benchmarks (1.1) and applications (2.1) matrix for weasel from /data.
- Use MUSE + WEASEL to build the model.
- Write the model prediction results on /temps.
5.1.
- Read the applications information, benchmark information, model prediction results from /temps and knowledge db from /data
- Map the results with the knowledge
5.2.
- Plot the results of knowledge_mapper.py
root/data
/benchmarks
/applications
root/poseidon-scripts/poseidon
/data
/temps
/results
File System

Figure 12. Poseidon framework code implementation.

Figure 12 shows the software architecture behind Poseidon. The framework follows a mediator design pattern, in which we have a mediator (poseidon.py) which orchestrate all the other components of the framework (1 - 5) for them to work correctly. This implementation let us have each step of the framework isolated from the others, resulting in an independent and modularized code in which each individual step can be executed individually. First, we prepare the datasets in order to be digestible for the transformation step (i.e. 1, 2), next we segment the application (i.e. 3) and finally we use WEASEL + MUSE to transform the raw data to build the logistic regression model and generate knowledge from the model prediction results (i.e. 4, 5). Each of the components is constantly reading and writing to the file system. The implementation of Poseidon is mainly on Python 3.7, being WEASEL + MUSE transformation and training and evaluation of the model implemented in Java 8 [11].

7. REFERENCES

[1] Willhalm, T., Dementiev, R., & Fay, P. (2012). Intel performance counter monitor-a better way to measure cpu utilization. Dosegljivo: https://software.intel.com/en-us/articles/intel-performance-counter-monitor-a-better-way-to-measure-cpu-utilization. [Dostopano: September 2014].

[2] Delgado, N., Gates, A. Q., & Roach, S. (2004). A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on software Engineering*, *30*(12), 859-872.

[3] Schäfer, P., & Leser, U. (2017, November). Fast and accurate time series classification with weasel. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management* (pp. 637-646). ACM.

[4] Schäfer, P., & Leser, U. (2017). Multivariate time series classification with weasel+ muse. *arXiv preprint arXiv:1711.11343*.

[5] Keogh, E., & Lin, J. (2005). Clustering of time-series subsequences is meaningless: implications for previous and future research. *Knowledge and information systems*, *8*(2), 154-177. [6] Making subsequence time series clustering meaningful

[7] Fryzlewicz, P. (2014). Wild binary segmentation for multiple change-point detection. *The Annals of Statistics*, *42*(6), 2243-2281.

[8] Lin, J., Khade, R., & Li, Y. (2012). Rotation-invariant similarity in time series using bag-of-patterns representation. *Journal of Intelligent Information Systems*, *39*(2), 287-315.

[9] Karim, F., Majumdar, S., Darabi, H., & Harford, S. (2018). Multivariate LSTM-FCNs for Time Series Classification. *arXiv preprint arXiv:1801.04503*.

[10] Salvador, S., & Chan, P. (2007). Toward accurate dynamic time warping in linear time and space. *Intelligent Data Analysis*, *11*(5), 561-580.

[11] Symbolic Fourier Approximation, WEASEL & WEASEL + MUSE. Github Repository: https://github.com/patrickzib/SFA

[12] Truong, C., Oudre, L., & Vayatis, N. (2018). A review of change point detection methods. *arXiv preprint arXiv:1801.00718*. GitHub repository: https://github.com/deepcharles/ruptures

[13] Schäfer, P., & Högqvist, M. (2012, March). SFA: a symbolic fourier approximation and index for similarity search in high dimensional datasets. In *Proceedings of the 15th International Conference on Extending Database Technology* (pp. 516-527). ACM.

APPENDIX A. *TRIDENT METRICS BEING COLLECTED*

Metric	Description
S0/S1/S2/.../SN:	Represent a Socket
C0/C1/C2/C3:	Represent a Core
P1/P2/P3/.../PN:	Represent a Port
UOPS:	uOp, or micro-op, is a low-level hardware operation. The CPU Front-End is responsible for fetching the program code represented in architectural instructions and decoding them into one or more uOps.
READ BW (MIB):	Quantity of Read Bandwidth in MIB
WRITE BW (MIB):	Quantity of Written Bandwith in MIB
ACT COUNT:	DRAM Activate Count (# of times the DRAM became active)
PAGE ACT COUNT:	DRAM Page Activate Count (# of times the Page DRAM became active)
PRE_COUNT.PAGE_MISS:	DRAM Precharg events due to page Miss (i.e. page conflict)
INST	Total number of instructions retired
CYC	Total number of cycles
IDQ UPS NOT DELV CORE	This event counts the number of uops not delivered to Resource Allocation Table (RAT) per thread adding "4 – x" when Resource Allocation Table (RAT) is not stalled and Instruction Decode Queue (IDQ) delivers x uops to Resource Allocation Table (RAT) (where x belongs to {0,1,2,3}). Counting does not cover cases when: a. IDQ-Resource Allocation Table (RAT) pipe serves the other thread; b. Resource Allocation Table (RAT) is stalled for the thread (including uop drops and clear BE conditions); c. Instruction Decode Queue (IDQ) delivers four uops.
UOPS ISSUED	This event counts the number of Uops issued by the Resource Allocation Table (RAT) to the reservation station (RS).

UOPS RETIRED	This event counts all actually retired uops. Counting increments by two for micro-fused uops, and by one for macro-fused and other uops. Maximal increment value for one cycle is eight.
INT MISC RECOVERY CYCLES	Core cycles the allocator was stalled due to recovery from earlier clear event for any thread running on the physical core (e.g. misprediction or memory nuke).
S[N] UOPS EXEC P[M]	Cycles of core N when uops are dispatched to port M.
slots	4 * cpu_clk_unhalted
fe bound (top down analysis)	idq_uops_not_delivered / slots
bad spec (top down analysis)	(uops_issued - uops_retired_slots + 4*recovery_cycles) / slots
retiring (top down analysis)	uops_retired_slots / slots
be bound (top down analysis)	1 - fe_bound - bad_spec - retiring
RBW	Total read bandwidth
WBW	Total written bandwidth
CY	Cycles
IN	Instructions
IPC	Instructions per cycle (IN / CY)
RATIO	Ratio of the port usage
PO	
PM	