Arm MUCH: Full-spectrum hardware-event-based Armv8 application profiler

Andrea Misuraca
Technical University of Munich
andrea.misuraca@tum.de

Andrea Bastoni
Technical University of Munich
andrea.bastoni@tum.de

***Abstract*—Profiling on Arm architecture using the Performance Monitoring Unit (PMU) gives developers low-overhead access to Hardware Events Monitors (HEM). These events are available on every Armv8-based platform and provide detailed information on the execution of applications, including multicore-related interference and usage of shared hardware resources. Prompt access to such information is fundamental for mixed-criticality systems in order to manage and regulate interference. Despite board configuration providing dozens of different HEMs (typically 30 to 50), the PMU only allows the simultaneous monitoring of a limited number of them (generally between 6 and 8). A simultaneous full-spectrum hardware profiling based on HEMs is therefore not possible. Recently, a methodology (MUltiCorrelation HEM reading and merging approach *MUCH*) has been proposed to statistically reconstruct a coherent HEM-based execution context from multiple application runs. MUCH has been validated only in a bare-metal environment for an NXP T2080 PowerPC platform. Given the widespread adoption of the Arm architecture for embedded systems, in this paper we implemented the MUCH approach for the Armv8 architecture on Linux to assess the viability of such an approach for complex systems. Our results on a Raspberry Pi 3 Model B confirm the practicality of the approach on complex Armv8-based systems. Furthermore, we explored how to leverage the obtained full statistical profile to derive properties of the analyzed application, for example, the sufficient set of HEMs to simultaneously monitor at runtime with minimal information loss, and machine learning-based models for HEMs prediction on a future set of applications.**

***Index Terms*—Arm, profiling, PMU, HEM, pairwise-correlation**

I. Introduction

Software profiling is a dynamic program analysis technique where a program's behavior is modeled using data monitored at runtime. Hardware-based software profiling enables developers to better understand the execution of a program, monitoring how the hardware layer reacts to the application. When integrating applications with different criticalities on complex multi-processor systems, monitoring of hardware performance counters is the basic mechanism used by several techniques (*e.g.*, [14], [23]) to manage hardware-related interference (especially in the memory subsystems) and to achieve higher isolation among otherwise independent applications.

The Arm architecture exposes Performance Monitor Units (PMUs) as architecture-specific, on-chip solution for low-overhead hardware event-based profiling. In this context, Hardware Event Monitors (HEMs) constitute the hardware events to be observed, and Performance Monitor Counters (PMCs) represent the actual architectural counter where the monitored information is stored. The Armv8 architecture defines a standard, common set of HEMs that should be present in any implementation, and a set of HEMs that could be optionally implemented by manufacturers [6].

Compared to other software profiling and debugging solutions (*e.g.*, Arm CoreSight [8]), Arm hardware-event-based profiling has low overhead and generates less execution interference in terms of *e.g.*, resource usage and bus accesses. Unfortunately, extensive application profiling and monitoring using HEMs is limited by the low number of PMCs that are typically available on Arm boards. In fact, PMCs are often a magnitude order fewer than the HEMs that could possibly be monitored. Therefore, despite the standard availability of PMCs and HEMs on Arm-based platforms, monitoring the *full hardware execution context* observing one single run of an application (or a benchmark) is not possible.

MUltiCorrelation HEM reading and merging (MUCH) is a recent approach [20] that relies on statistical analysis to reconstruct the full-hardware application context across multiple runs of an application or benchmark. The approach has been developed to target explicitly complex multiprocessor systems-on-chip (MPSoC), where HEM monitoring and profiling is becoming progressively more important to master interference among mixed-criticality (real-time) applications [15]. In [20], the approach has been validated in a bare-metal setup (without operating system) on a PowerPC NXP T2080. Given the increasing relevance of the Arm architecture for complex MPSoC used in safety critical automotive, industrial, and avionics domains, this paper proposes Arm MUCH, an application profiler for the Armv8 architecture that adopts the MUCH-approach and runs on a complex operating systems such as Linux. With Arm MUCH we:

- Validate whether the MUCH approach can be applied to real-world Arm architectures together with a complex operating system such as Linux. Our results on a Raspberry PI 3 with Linux 5.9.93 confirm the applicability of the MUCH methodology to the Arm architecture even with a complex operating system.
- Contribute a framework for implementing MUCH on Armv8. In particular, we automated the HEM allocation

Andrea Bastoni was supported by the Chair for Cyber-Physical Systems in Production Engineering at TUM and the Alexander von Humboldt Foundation.

at application runtime, the gathering of the retrieved data and performing the statistical MUCH methodology.
- Interface the Arm MUCH framework with multiple profiling technologies for Arm Linux, namely *perf*, *eBPF*, and inline assembly for manual HEMs allocation.
- Explore how to derive the minimal set of HEMs that best characterize an application. This enables confidence in capturing the key properties of an application despite the limited number of monitored PMCs.
- Implement AI-based HEM-prediction systems that use the statistical data retrieved by MUCH to reconstruct the *complete set of HEMs inside one benchmark execution* by forecasting all non-monitored HEMs.

The rest of this paper is organized as follows. Section II introduces the concepts of MUCH and discusses previous work. Section III presents the architecture of Arm-MUCH, while Section IV discusses its implementation and our experimental results. Section V concludes.

II. Background and Related Work

A. The MUCH Approach

The core concept underlying the MUCH approach [20] is to *rearrange and merge individual and independent HEM readings* into one single coherent dataset as if each all HEMs were all measured in the same run. MUCH employs Multi-Variate Gaussian Distributions *(MVGD)* to preserve all pair-wise HEM correlations simultaneously. The goal is to use measured data to generate full-spectrum HEM vectors (merged from multiple runs) of size nh in accordance with the MVGD model $X \sim N_{nh}(\hat{\mu}, \hat{\Sigma})$, where $\hat{\mu}$ and $\hat{\Sigma}$ are the empirical expected value and empirical covariance matrix obtained in multiple runs of the experiments for the same HEM h^i. The empirical covariance matrix $\hat{\Sigma}$ can be obtained from the empirical *correlation* matrix $\hat{S}$ that expresses the correlation $\hat{\rho_{ij}}$ for each pair of empirical HEMs vectors $(\{h^i\}, \{h^j\})$. Each empirical covariance value $\hat{\sigma_{ij}}$ in $\hat{\Sigma}$ is computed as: $\hat{\sigma_{ij}} = \hat{\rho_{ij}} \cdot \hat{\sigma_i} \cdot \hat{\sigma_j}$, where $\hat{\sigma_i}$ is the variance associated with an empirical vector of HEMs $\{h_i\}$.

In order to use Multi-Variate Gaussian Distributions, the correlation between two different HEMs must be correctly evaluated. Each HEM needs to be measured in the same sub-experiment, namely a *benchmark run*, at least once with every other HEM present in the machine. Each HEM's "true" value is statistically modeled with a *Gaussian distribution*. Therefore, by the *central limit theorem*, each sub-experiments should run multiple times for a sufficient amount of time. We run each sub-experiment for at least 10 seconds and repeat every experiment 50 times.

From a mathematical point of view, the challenge is to reorder the grouped sample vector $\{h_i\}$ for each HEM h_i, such that for each pair of HEMs $\{h_i\}$ and $\{h_j\}$, the empirical correlation $\hat{\rho_{ij}}$ of the grouped samples is close to ρ_{ij}, namely the Pearson's empirical correlation between $\{h_i\}$ and $\{h_j\}$, calculated in the sub experiment in which they are both allocated.

B. Previous Work

Despite the risks of non-precise and context-dependent event accounting, the low overhead and widespread availability of HEMs have been exploited in a host of tools to gain insights into application's behavior [10]. In the embedded real-time field, the usage of HEMs have been leveraged in multiple works to monitor and regulate MPSoC interference at both cache [13], [14], interconnect [19], [23], and DDR level [17], [22]. Recent architecture-level features such as Arm MPAM [7] or Intel RDT [12] extend the concept of HEMs to provide quality of service throughout the memory subsystem. The usage of HEMs for profiling purposes in the GPU and accelerators domain is a standard practice supported *e.g.*, by tools such as the NVIDIA Nsight Systems [16]. The Arm v8.2 architecture also introduced a dedicated infrastructure for statistical profiling [9], but its usage and adoption is still limited. The Arm Coresight [8] infrastructure can deliver both hardware assisted application profiling and debugging capabilities. Its overheads for data collection and exporting hinder nonetheless its usage for low-overhead hardware profiling.

Alternative approaches to MUCH (*e.g.*, [11], [18], [21]) to merge HEMs have been shown [20] to be not applicable to HEMs or inferior to the MUCH approach.

III. Architecture

Our Arm MUCH framework is divided in two main components:

- **Profiler Middleware** that will access and manage the run-time progress of the profiled application. Currently, Arm MUCH supports both *perf subsystem* and *kprobes* in order to evaluate and obtain HEMs data at run-time from the profiled application. We note that any profiler compatible with the perf output format could be used for the data acquisition.
- **Data Analysis Framework** that performs the data processing. This includes data collection, loading and writing benchmark sessions to disk, and the statistical evaluations associated with MUCH.

The framework has been split into two components to provide a wider set of profiling tools that the end-user can leverage to validate the data collected. The implementation abstracts away details of the *e.g.*, Linux *perf* API and provides a unified interface for data processing that facilitated the validation of the experimental data.

A. Profiling Middleware

The profiling layer supports two ways to allocate at run time the selected set of HEMs to be monitored.

The first approach uses Linux *perf* to select the HEMs from bash's command line. This approach is the easiest one, as we do simply need to spawn a correct instance of *perf* with the application we choose to monitor and the right sets of *HEMs* right after the *-e argument*. Despite its easiness of use, with this approach we can only monitor the entire application from start to end. Specifically, we cannot monitor particular

sections of the application or have warm-up phases *e.g.*, to avoid measuring memory allocation and initialization.

The second supported method for selecting HEMs is a lightweight API for code instrumentation. The API can directly allocate, enable, and disable *hardware events*. The API wraps the `perf_event_open()` syscall to access PMUs for writing and reading purposes. This method provides better integration and function-level granularity with the trade-off of rebuilding the application against the profiling middleware, inserting the right calls for PMU activation, and reading the data through a file descriptor.

B. Data Analysis Framework

The data analysis framework processes the data generated by the profiler—any profiler could be used as long as the traces are in perf-format—and supports the statistical MUCH analysis. The framework consists in a main application that supports different modes, with regard to what the inputs and outputs should be, including *e.g.*, processing previous benchmark iterations using the *-l* flag, or exporting benchmark sessions to disk using *-w* argument. The statistical evaluation is performed in *python* using *numpy* [1], *scipy* [3], and *sklearn* [4]. Plots and graphical evaluation are visualized using *matplotlib*. HEMs vector results are exported as binary file or printed on *stdout*.

The application implements different phases to performs the MUCH analysis of event counters.

1) **Empirical correlation matrix:** $\hat{S}(i,j)$ is calculated between all pair of HEMs h^i and h^j. Each cell of the matrix is defined as *Pearson correlation coefficient* between HEMs at a given column and row. Hence, this matrix will be symmetrical with unitary values on the main diagonal.

$$\hat{S}(i,j) = \begin{Bmatrix} 1, & \text{for } i = j \\ \hat{\rho}_{ij}, & \text{for } i \neq j \end{Bmatrix}$$

2) **MVGD mapping:** Using *copula theory* and *Percent Point Function*, each *HEM counter measurement* $\{h^i\}$ is mapped to its relative value as it was part of normally distributed linear space.
3) **MVGD-mapped covariance matrix:** $\hat{\Sigma}_0(i,j)$ is calculated between all pair of HEMs, using *MVGD values* with covariance and correlation values from the experiments. Each cell of the matrix is defined as follows:

$$\hat{\Sigma}_0(i,j) = \begin{Bmatrix} \hat{\sigma}_i^2, & \text{for } i = j \\ \hat{\sigma}_i \times \hat{\sigma}_j \times \hat{\rho}_{ij}, & \text{for } i \neq j \end{Bmatrix}$$

4) **Random samples extraction:** This step reconstructs empirical data order statistics from the *MVGD-mapped covariance matrix*, using the multivariate-normal function.
5) **MVGD-based correlation matrix:** This step calculates the correlation matrix between all between all pairs of HEMs using the sorting order defined by the previous steps.

PMUs Evaluation

PMU Name	# Events	# Run
br_immed_retired	13553467	0
br_mis_pred	585430	0
br_pred	14568163	0
bus_access	181166	0
bus_cycles	74048498	0
cid_write_retired	0	1
cpu_cycles	74361580	1
exc_return	427	1
exc_taken	427	1
l1d_cache	50613980	2
l1d_cache_refill	36431	2
l1d_cache_wb	50720	2
l1d_tlb_refill	1101	2
l1i_cache	72381213	2
l1i_cache_refill	846215	3
l1i_tlb_refill	769	3
l2d_cache	973143	3
l2d_cache_refill	50931	3
l2d_cache_wb	7456	3
ld_retired	37788240	4
mem_access	52275877	4
memory_error	0	4
pc_write_retired	9318972	4
st_retired	11483425	4

Fig. 1: Output example of the PMU statistical evaluation.

6) **Mean Squared Error (MSE):** Since multiple sorting orders could exist, this step selects the best reordering—among multiple iterations—that minimizes the *mean squared error* between the *empirical correlation matrix* and the *MVGD-based correlation matrix*.

In addition to the standard MUCH analysis, the framework can automatically: i) *cluster* HEMs to identify the smallest set of HEMs that could reflect the key characteristics of the profiler application, and ii) make prediction on values of HEMs that were not allocated in a specific run.

Since the maximum number of PMUs that can be traced simultaneously in one run is limited (*e.g.*, the Arm Cortex-A53 supports simultaneous tracing from only six PMUs), clustering the most "meaningful" HEMs helps reducing the number of runs to identify application behaviors. Clustering of n HEMs is determined by maximizing the sum of the *maximum absolute correlation values* with regards to every other HEMs within one experiment. Correlation values are filtered using the *python Pandas Dataframe* [2].

Furthermore, the framework implements three machine learning algorithms (Linear Regression, Multi-layer Perceptron, and Random Forest) to predict values for HEMs that were not allocated in a specific run. The models are trained on the full set of all HEMs.

IV. Implementation and Evaluation

The framework, comprising the profiling middleware and the data analysis component, has been developed in Python and C. The profiler middleware automates the activities of profiling applications and benchmarks using both *perf* and our

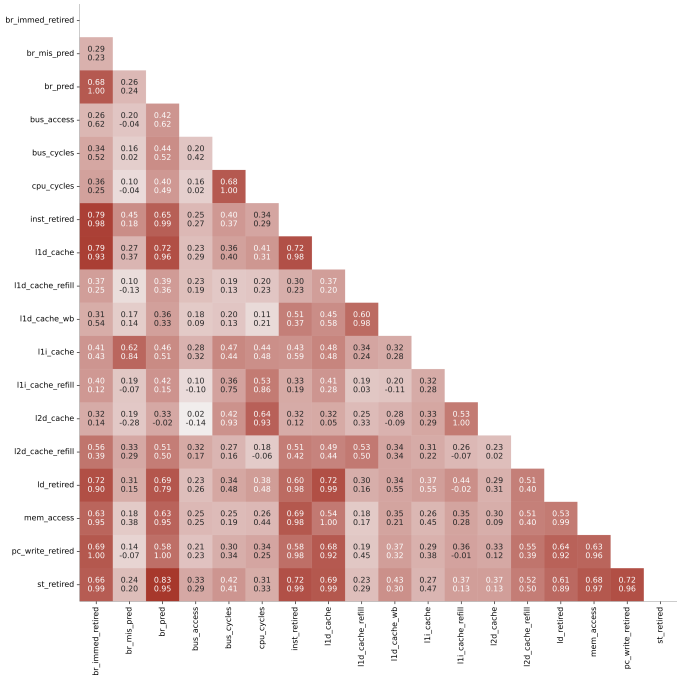

Fig. 2: Single-core experiment correlation matrix for pairs of HEMs. In each box, the upper value is the empirical correlation, the lower value the MVGD-correlation.

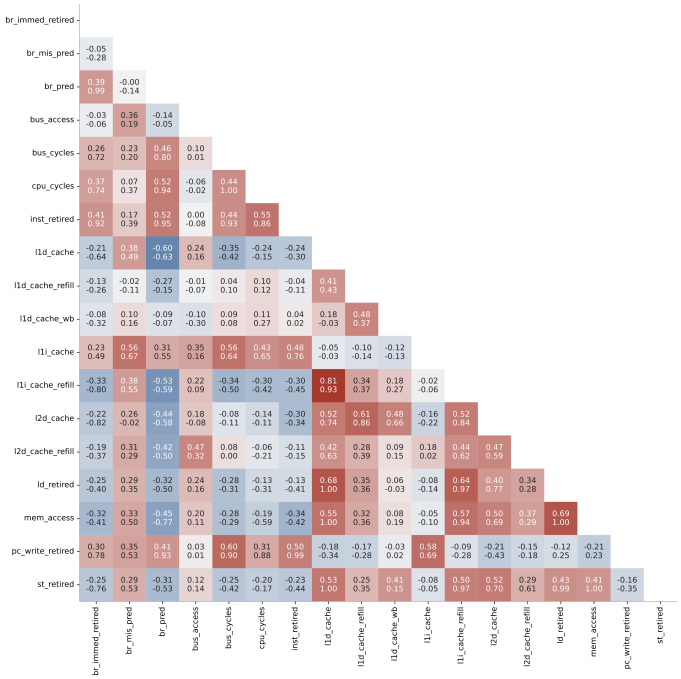

Fig. 3: Multi-core experiment correlation matrix for pairs of HEMs. In each box, the upper value is the empirical correlation, the lower value the MVGD-correlation.

lightweight API for code instrumentation. The data analysis frameworks implements the phases described in Sec. III-B.

In order to enable profiling support in the Linux kernel, the kernel configurations related to `BPF` and `IKHEADERS` must be enabled. The framework requires a version of the *perf* profiling tool that matches the compiled kernel version (*perf* can be found in the Linux kernel sources, under `./tools/perf`). Furthermore, the framework needs Python $>$ 3.9.2 (including pip).

The framework has been validated with Linux on a Raspberry Pi 3 Model B (Arm Cortex-A53 cores).[1] We used *Armbian Buster* as Linux distribution, running a 5.9.93 Linux kernel with enabled profiling-configuration.

A. HEM Correlation Matrices

We evaluated pairwise correlation of HEMs as well as the Arm MUCH approach in a single- and multi-core setup.

The single-core setup also serves as validation and uses a CPU-intensive benchmark that computes the *discrete logarithm* for random natural numbers. Each benchmark-run execute 90000 iterations. The multi-core setup uses the *Sysbench* [5] tool. Sysbench is a scriptable *multi-threaded benchmark tool* that creates complex time-based workloads. In our testing, we used $t = 10$ in order to generate multithreaded benchmarks of 10 seconds lengths. Fig. 1 presents the output of the tool with multiple runs of the benchmarks.

We performed testing runs using both setups and identified 17 statistically-relevant HEMs, *i.e.*, whose values were orders of magnitude larger than the remaining HEMs. Specifically, we focused on: br_mis_pred, br_pred, bus_access, bus_cycles, cpu_cycles, instr_retired, l1d_cache, l1d_cache_wb, l1d_cache_refill, l1i_cache, l1i_cache_refill, l2d_cache, l2d_cache_refill, ld_retired, mem_access, pc_write_retired, and st_retired. These HEMs have been allocated to 21 different sub-experiment, and each HEM is part of *5 sub-experiments*. Hence, there are 50×5 measurements for each of the hardware monitors.

The matrices in Fig. 2 and Fig. 3 underline the correlation between couples of HEMs $\{h_i\}$ and $\{h_j\}$. In each correlation *box*, the upper value is the empirical correlation $\hat{\sigma_{ij}} = \hat{\sigma}_i \times \hat{\sigma}_j \times \hat{\rho}_{ij}$ and the lower one is the MVGD-based correlation after the MUCH data processing.

As expected, data from the predictable synthetic single process CPU-intensive benchmark presents very different correlation than the sysbench multiprocessing benchmark.

The single-core benchmark (Fig. 2) manifests a strong correlation between HEMs, mostly describing partial linear relationships between *HEMs* belonging to the *same context*, such as *cache misses*, or *branch predictions*. Instead, the multi-core benchmark (Fig. 3) presents less correlated data, including negative relationship between pairs of HEMs.

B. Accuracy Evaluation

We focus on the accuracy of the framework with respect to the empirical correlation of sub-experiment data and to potential errors in predicting HEM values in later benchmark executions.

We assessed the distribution of the delta between empirical pairwise correlation and MVGD sampled pairwise correlation

[1]The framework was also tested on Huawei Taishan Servers using a proprietary OS. We cannot disclose information on this setup.

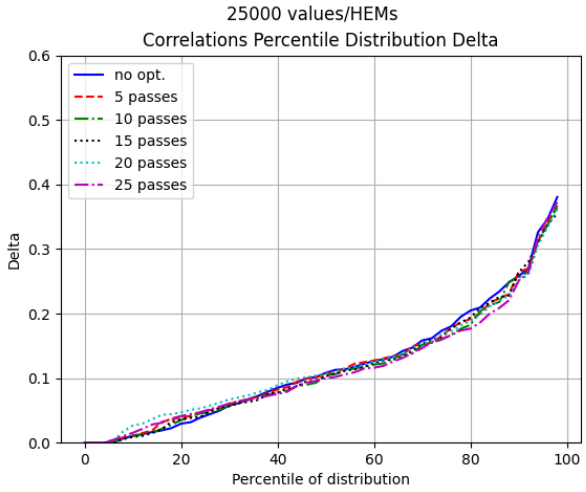

Fig. 4: Correlation delta as probability density function for single-core benchmark and 25000 values/HEM

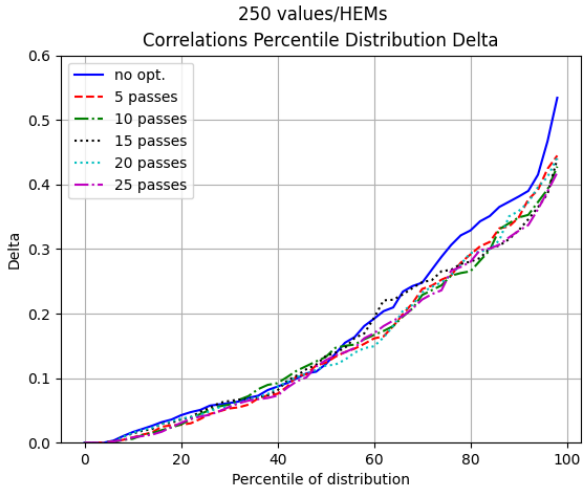

Fig. 5: Correlation delta as probability density function for single-core benchmark and 250 values/HEM

between couples of HEMs. This quantifies how much the reconstructed samples differ from the actual data after MVGD processing. Fig. 4 and Fig. 5 show the distribution of the correlation deltas for the single-core benchmark and 25000 and 250 samples/HEM respectively. Overall, we found the correlation delta to depend on the number of values sampled for each HEM and reaching 0.4 (0.6) for 25000 (250) samples of each HEM. As noted in [20], we also observed that the experimental and the reconstructed values depends on the number of iterations (*optimization steps*) performed during sorting to obtain the re-arranged full-wide HEMs vectors ((step (5) in Sec. III-B). In our experiments, the optimization step had a lower impact than the number of collected samples per HEM. This is visible in Fig. 5 that shows a higher variability of the correlation delta when only 250 samples/HEM are profiled.

TABLE I: MAPE: CPU-intensive single-process benchmark

MAPE: CPU-intensive single-process benchmark			
HEM name	MLR	MLP	RF
br_pred	0.000545	0.164585	0.000460
bus_cycles	0.002273	0.009775	0.006071
inst_retired	0.000349	0.012472	0.000330
l1d_cache	0.000454	0.011996	0.000347
l1d_cache_wb	0.036503	51.105616	0.014636
l1i_cache	0.007588	0.014341	0.003902
l1i_cache_refill	0.433256	4.051663	0.459776
l2d_cache	0.350933	2.164955	0.674850
l2d_cache_refill	0.037171	28.199508	0.027867
ld_retired	0.000535	0.065614	0.000350
mem_access	0.000633	0.011904	0.000549
pc_write_retired	0.000478	0.098950	0.000509
st_retired	0.000262	0.012920	0.000346
	0.870979	85.924299	1.189993

C. HEM Clustering

We used the framework to identify cluster of HEMs that can best characterize an application, despite the limited number of monitored PMCs in one run. We defined HEM clusters to contain the n HEMs that maximize the sum of *maximum absolute correlation values* with regards to every other HEMs in an experiment.

For the single core benchmark experiment, we discover that branch predictions, L1 cache counters, and CPU cycles counters (br_immed_retired, br_mis_pred, br_pred, cpu_cycles, l1d_cache_refill, l1i_cache_refill) are the most suited to capture the behavior of the benchmark with a maximum absolute correlation value of ~ 7.866.

For the sysbench multi-core benchmark, the best cluster includes the bus access and the write-retired counters (br_immed_retired, bus_access, l1d_cache_refill, l1i_cache, l1i_cache_refill, pc_write_retired) with a maximum absolute correlation value of ~ 6.874.

D. AI-based HEM Prediction

Arm MUCH provides a full statistical analysis of the correlation between HEMs. We used this *data set* to train three machine-learning models—Linear Regression (MLR), Multilayer Perceptron (MLP), Random Forest (RF)—and to evaluate the capability of the framework in *predicting* non-monitored HEM values.

After training on the set of re-arranged HEM vectors, *i.e.*, after applying MUCH, we fed the networks with subsets of HEM values coming from the HEM clustering experiments, and evaluated the accuracy of the networks in generating *non-monitored* HEMs. We used the full values of the HEM-clustering experiments as *empirical reference*. The accuracy of the *forecasted* values has been assessed using the *Mean Absolute Percentage Error (MAPE)*.

Tables I and II report the MAPE accuracy for MLR, MLP, and RF networks in predicting HEM values that were not measured during a benchmark run. In the tables, the best values are marked in blue color, while the ones exhibiting more than 10% MAPE error are marked in red. The values are the average of 10 experiments.

TABLE II: MAPE: Sysbench multithreaded benchmark

MAPE: Sysbench multiprocess benchmark			
HEM name	MLR	MLP	RF
br_pred	0.396886	0.003298	0.697910
bus_cycles	0.390227	0.161784	0.697271
cpu_cycles	0.299356	0.534736	0.697193
inst_retired	0.447805	0.234720	0.696650
l1d_cache	0.575274	0.330672	0.595609
l1d_cache_refill	47.127874	0.799111	0.350560
l1d_cache_wb	86.262104	10.248902	0.314104
l1i_cache	0.130094	0.004722	0.696297
l2d_cache_refill	40.071846	186.542361	0.291218
ld_retired	11.226796	1.327011	0.585759
mem_access	10.733564	0.425685	0.596006
pc_write_retired	0.341011	0.027340	0.696904
st_retired	4.933629	0.431589	0.607128
	202.936466	201.071931	7.522609

The Sysbench multithreaded benchmark (Table II) clearly shows that RF outperforms MLP and MLR. Notably, MLR predictions exceed five times the threshold (10) and MLP can produce very high errors (186.54 for l2d_cache_refill predictions). The CPU-intensive benchmark (Table I) confirms the trends. The results are preliminary, as more training and experiments will be needed to fully assess the capability of the framework. Nonetheless, the approach seems promising.

V. Conclusion

In this paper, we have presented Arm MUCH, a framework for the Armv8 architecture that adopts the MUCH [20] approach to overcome the limitations of modern PMUs that only allow a reduced number of HEMs to be monitored simultaneously.

We have validated the applicability of MUCH to Arm in contexts that include a complex operating system (Linux) and non trivial benchmark applications. Furthermore, we have investigated extensions of MUCH to i) derive minimal sets of HEMs that can best characterize the runtime behavior of an application, and ii) predict values of non-monitored HEMs using AI-networks trained on the full set of statistical data.

Our results confirm that Arm MUCH can capture the correlation between HEMs observed in different runs with adequate accuracy. Our experiments on clustering and prediction presented promising initial results that are worth investigating in future works, potentially in combination with tools to detect and analyze hardware noise,[2] to better understand whether the currently achieved accuracy can be improved. Additionally, the Random Forest approach to predict HEM values would benefit from a larger training set and more iterations.

In the future, we would also like to investigate the integration with custom eBPF programs and user-defined *uprobe* hooks to easily and precisely access performance counters.

References

[1] Python Numpy: documentation. https://numpy.org/doc/stable/reference/index.html#reference.

[2] Python Pandas: Dataframe. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html.

[3] Python Scipy: documentation. https://docs.scipy.org/doc/scipy/.

[4] Python Sklearn: documentation. https://scikit-learn.org/stable/modules/classes.html.

[5] Sysbench: repository. https://github.com/akopytov/sysbench.

[6] ARM. Arm Architecture Reference Manual for A-profile architecture. https://developer.arm.com/documentation/ddi0487/latest/ Accessed: 2023-05-01.

[7] ARM. Arm Architecture Reference Manual Supplement. Memory System Resource Partitioning and Monitoring (MPAM) for Armv8-A. https://developer.arm.com/docs/ddi0598/latest Accessed: 2023-05-01.

[8] ARM. Arm CoreSight Architecture. https://developer.arm.com/Architectures/CoreSight Architecture Accessed: 2023-05-01.

[9] ARM. Arm Statistical Profiling Extension. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/statistical-profiling-extension-for-armv8-a Accessed: 2023-05-01.

[10] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 20–38, 2019. doi:10.1109/SP.2019.00021.

[11] Trevor Hastie, Rahul Mazumder, Jason D. Lee, and Reza Zadeh. Matrix completion and low-rank svd via fast alternating least squares. *J. Mach. Learn. Res.*, 16(1):3367–3402, jan 2015.

[12] Intel. Resource Director Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html Accessed: 2023-05-01.

[13] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna. Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 1–14, 2019.

[14] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 45–54, 2013.

[15] Laurence H. Mutuel, Xavier Jean, Vincent Brindejonc, Anthony Roger, Thomas Megel, and E. Alepins. Assurance of Multicore Processors in Airborne Systems. Technical Report DOT/FAA/TC-16/51, FAA and Thales Avionics, 2017.

[16] NVIDIA. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems Accessed: 2023-05-01.

[17] Xing Pan and Frank Mueller. Controller-aware memory coloring for multicore real-time systems. SAC '18, page 584–592, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3167132.3167196.

[18] Benjamin Recht. A simpler approach to matrix completion. *J. Mach. Learn. Res.*, 12(null):3413–3430, dec 2011.

[19] Ahsan Saeed, Dakshina Dasari, Dirk Ziegenbein, Varun Rajasekaran, Falk Rehm, Michael Pressler, Arne Hamann, Daniel Mueller-Gritschneder, Andreas Gerstlauer, and Ulf Schlichtmann. Memory Utilization-Based Dynamic Bandwidth Regulation for Temporal Isolation in Multi-Cores . In *2022 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 133–145, 2022.

[20] Sergi Vilardell, Isabel Serra, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. Much: Exploiting pairwise hardware event monitor correlations for improved timing analysis of complex mpsocs. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, page 511–520, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3412841.3441931.

[21] Sergi Vilardell, Isabel Serra, Roberto Santalla, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. Hrm: Merging hardware event monitors for improved timing analysis of complex mpsocs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3662–3673, 2020. doi:10.1109/TCAD.2020.3013051.

[22] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 155–166, 2014.

[23] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms. *IEEE Transactions on Computers*, 65(2):562–576, 2016.

[2] https://www.kernel.org/doc/html/latest/trace/osnoise-tracer.html