In 29th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2021), September 10–13, 2024, Padova, Italy

Predicting Cache Behaviour of Concurrent Applications

***Abstract*—Modern digital solutions are built around a variety of applications. The continuous integration of these applications brings advancements in technology. Therefore, it is essential to understand how these applications will behave when they run together. However, this can be challenging to interpret due to the increasing complexity of the execution details. One such fundamental detail is the utilization of shared cache as it goes hand in hand with the computation capacity of computer systems. Since cache utilization behavior is not simple enough to translate with few assumptions we have investigated if this complex behavior can be predicted with the help of machine learning. We trained the deep neural network with enough examples that represent the cache behavior when applications were running alone and when they were running concurrently on the same core. The Long Short-Term Memory (LSTM) network learns the entire execution period of each application in the training set. As a result, without running two applications together in reality, provided with the L1 cache misses of two applications (running alone), it can predict how the cache will look like if two applications wish to run together. The model returns a time series that reflects the cache behavior in concurrency.**

***Index Terms*—Performance monitoring counters, L1 Cache, Long Short-Term Memory Network, Machine Learning**

I. Introduction

Regardless of the technical complexities introduced by the integration of more applications, industries seek modern and real-time solutions to satisfy their operational needs and to improve the user experience. This continuous integration of applications is driving the need for a thorough understanding of how they will influence each other if they share computational resources. In general, there are more tasks to run to get a job done and it is not always possible to find a free separate core for each application. So concurrency is always the case and one needs to understand how the two applications will impact each other's execution if they wish to run on the same CPU core at the same point in time.

To achieve better performance levels, there is plenty of work around algorithm improvements and code optimization [1] but resource contention can still happen when multiple applications are competing for the shared resource(s). In concurrency, it can be experienced at the L1 cache level as applications are running on the same core. That being the case, resource contention at L1 cache level would cause more cache pollution than contention at the Last Level Cache (LLC) level. Consequently, it has a leading impact on cache management techniques such as cache partitioning [2] and cache-aware scheduling [3]. Therefore, we aim to predict the behavior of L1 cache misses which indicates either the requested data was not already loaded or it has been polluted by the other applications. One way to capture the L1 cache misses is through Performance Monitoring Counters (PMCs), and we use them in this study. However, understanding this behavior is not that straightforward due to continuous advancements and resource sharing.

Concurrent applications share the execution resources such as processing unit, registers, L1 cache and in some models even the L2 cache (in particular, we are interested in L1 cache). In principle, the operating system (OS) is responsible for memory management and it cannot just statically allocate the demanded memory to one application, at once at startup [4]. Here, OS ensures the demand optimization as per the system and task model. Since tasks may block and sleep their elapsed time and memory utilization behavior can vary depending on the scheduling technique and hardware specifications. If the scheduled time slot (quantum) is large, an application would have lower elapsed time and fewer cache misses. Also, if there are more tasks to run the quantum will be shared accordingly among all. Not to forget, instructions for only one of the concurrent applications are processed at a time. Here, performance remains a metric of interest which is highly sensitive to the number and type of applications running simultaneously. Despite all the considerations, performance easily gets affected by the way applications are programmed, the environment in which they are running, multithreading, out-of-order execution and resource contention due to the hierarchical cache subsystems (also shown in Figure 1). The way forward is to adopt resource optimization strategies. However with optimization techniques, the execution behavior of applications can easily be affected by numerous factors such as workload, hardware specification, application characteristics, compiler optimization techniques, scheduling algorithms and operating system policies. Therefore, to avoid performance degradation, industrial applications are tested before deployment in terms of computational cost and memory requirements. Yet it is highly unlikely to include and quantify all described factors (among others) even by the experts.

Nevertheless, applications get their execution done not just randomly but they are served as per the predefined rules and systematic procedures of the operating system. This implies that there is an underlying execution pattern. That being the case, machine learning (ML) can identify these patterns by learning from past experiences. Finding these patterns through traditional methods requires manual analysis and explicit pro-

Acknowledgements: This work was supported by the Knowledge Foundation in Sweden through the projects ACICS and XPRES.

gramming for various factors which is time-consuming yet uncertain. In contrast, machine learning can predict patterns that may not be easily apparent otherwise. So, our main contribution is to use ML to predict the L1 cache misses of two applications if they run concurrently. The model trains itself on the data that was collected when the applications were running alone and when they were running concurrently on the same core.

In this paper, we start by presenting a technical background in Section II for a better reading experience and to understand the contribution made through this work. Next, we present the predictive modeling approach in Section III describing the method used to achieve the goal of the study. The implementation details and the experimental details are then outlined in Section IV. Following the implementation details, results are discussed in Section V. The state-of-the-art and related work to our study is then presented in Section VI. Finally, the anticipated future work followed by the conclusion wraps up the paper in Section VII.

II. Background

We use the Performance Monitoring Counters (PMCs) to measure the L1 cache misses by the applications. The ultimate L1 cache behavior is predicted using the deep neural network called the Long Short-Term Memory (LSTM) network.

A. Performance Monitoring Counters

The processor is one of the main information sources for observing the performance of computing devices. There are tools for performance monitoring such as perf for Intel, CodeXL for AMD and DS-5 Development Studio fro ARM. These tools provide insights into CPU utilization and cache behavior by using Performance Monitoring Counters (PMCs). PMCs are special-purpose, hard-wired registers present at the core of PMU [5, 6, 7]. They can record hardware events (also known as PMU events) such as instructions retired, cache misses, branch missprediction and many more. However, the number of these registers is limited because they are fast and expensive memory components. PMCs are not just used by developers and systems administrators, they are used by the OS itself and task scheduling algorithms for their operations.

PMCs and PMU events are limited and vendor-specific. Vendors design their PMU differently so the availability of PMU events is also varied across different platforms. However common events are available on each platform. L1 cache misses is one such PMU event. The naming convention is not standard so the same event can be present under two different names on two platforms. An effort has been made by the developer of the Performance Application Programming Interface (PAPI) to standardize the events names [8]. PAPI allows the collection of platform-specific events hence we use PAPI to measure L1 cache misses caused by the test application.

B. Memory Hierarchy

To perform any operation, data needs to be in the working memory. However, due to storage capacity limitations, it is not possible to load all the data at once at startup. Therefore to achieve the desired goal, memory is structured in a hierarchical manner which ensures the availability of required data as per the need [9, 10]. The hierarchy is maintained under the principle of locality. It says that if a memory location is referenced once, it is more likely it will be referenced again in the near future or it is expected that the adjacent memory will be referenced shortly i.e., temporal locality and spatial locality respectively [4]. Therefore, multiple levels of hierarchy are maintained from slowest and least expensive (at the bottom) to fastest and most expensive (at the top, near CPU), also shown in Figure 1.

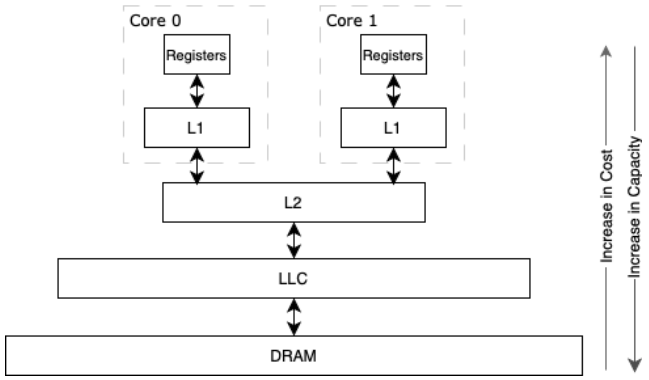


Fig. 1. Example of the memory hierarchy in a dual-core system.

The registers are located inside the CPU to hold the instruction and data on which the processor is currently working. Moving one level up is cache which itself has three levels as L1 cache, L2 cache and Last Level Cache (LLC or L3). When a processor does not find the required information at one level, the request is made to the next level (from top to bottom). Each reference to cache is either cache hit, miss or a prefetch. Starting from L1 to L3, the L1 cache is closest to registers and it is usually shared among core(s) whether it is a uni-core system or a multi-core system.

C. Concurrency

Concurrency is managing multiple tasks that are ready to be executed simultaneously, i.e., without waiting for a task to complete starting the other one. This is particularly required for quick response time in the case of real-time systems. Effective management of applications ensures system performance and user satisfaction. If concurrency does not involve shared resources, then it is true parallelism in time, and requested resources are promptly assigned to the application that is requesting. In comparison, when applications require shared resources, the operating system manages them through time-slicing or preemptive techniques. The scheduler, an operating system component, divides CPU time between the concurrent applications through context-switching. Context switching is preempting the CPU from the running application to the other. Understanding memory hierarchy is not only important in hardware design and software development, it influences performance management and even the decision-making process.

D. Cache Pollution

Concurrent tasks overlap in time to share resources so we need better management of applications. It can cost energy, grow complexity, increase programming efforts, limit scalability and cause cache pollution. In cache pollution, when an application is sharing time, valuable cache space is already overwritten by the other application which leads to more cache misses [11]. Therefore, the processor has to spend more cycles waiting for the data to be fetched from the higher memory levels. Consequently, this undermines the cache efficiency and the elapsed time of applications is also expands due to context switching.

Not just the cache pollution, resource contention can also happen. Resource contention is a battle for the shared resources such as for processor, registers and L1 cache which means interference [1].

E. Prediction Using Machine Learning

Prediction is a process of projecting the unseen using machine learning models. A machine learning model learns from the past examples. It identifies patterns using statistical methods, algorithms and computational techniques. A 'statistical method' analyzes the statistical properties of data such as continuous outcome (regression), categorical value (classification) or probability (bayesian). The algorithms provide rules for making a decision such as Support Vector Machines (SVM), Decision Trees, Ensemble Methods, and Neural Networks. Furthermore, Computational techniques contribute to artificial intelligence (AI) by training on a large data set. In this study, we aim for regression using the deep neural network. A deep neural network computes patterns and dependencies in a large training set by applying many layers of computation iteratively [12, 13].

Machine learning models have features and hyperparameters. Features come from data whereas hyperparameters are configuration variables for the model. There are two types of parameters; model parameters and hyperparameters. Model parameters are estimated from the data (like a Sigmoid coefficient, which is not assigned by the scientist) but hyperparameters are to be set before running an algorithm (such as the number of neurons in a deep learning model which can affect the accuracy of the model). Hyperparameters are estimated before training the model and the process is called hyperparameter tuning.

Predictions are inherently probabilistic however higher confidence can be achieved by training on high-quality data and tuning the hyperparameters and architecture of the AI model. For behavior analysis, getting the right measurements for training & evaluation is a foundation for a trustworthy prediction model. This means that the targeted behavior should be available in the training data of the model.

F. Long Short-Term Memory Networks (LSTM)

LSTM is a specialized form of recurrent neural network (RNN) to capture long-range dependencies in a sequence of data. LSTMs are not just feedforward networks but they can flow back and have long-term memory [12, 13]. Its sophisticated nature for remembering past data is suitable for recognizing complex patterns that might otherwise be overlooked in a long duration. Hence makes it ideal in the case of predicting L1 cache misses which evolves over time.

LSTM controls information by using gates. These gates allow us to learn long-term relationships in the data [12]. Lower time gaps in data points enable better predictions. A typical LSTM block, also shown in Figure 2, has a memory cell to carry the relevant information, an input gate to update the memory cell using activation function(s), an output gate to carry forward the information for the next memory cell and a forget gate to decide which information to discard from the memory cell. Each gate involves several neurons which are the computational units responsible for managing cell state and gate functionalities.

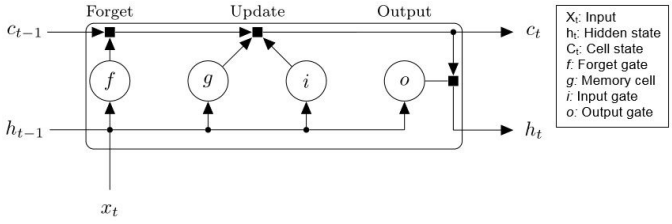


Fig. 2. Architecture of LSTM block at point t [12].

III. Prediction Approach

A shared cache can reduce the performance, so we aimed to predict L1 cache misses to avoid degraded response time and to improve the performance. We set up the experiment to see, without actually running two applications on a given hardware, if we can predict cache behavior when two applications wish to run together. Figure 3 shows the captured cache behavior in each scenario. Our target is to predict Figure 3c. The following sections present the learning architecture and prediction model.

A. Learning Architecture

Given two applications $p1, p2$, performance event e is defined as a measure of performance. For an e under observation, a time-ordered series m is collected at frequency, f. Here $m(p1)$ and $m(p2)$ represent the behavior when $p1$ and $p2$, respectively, were running alone on a CPU core c. In contrast, $m(p1, p2)$ presents the behavior when $p1$ and $p2$ were running concurrently on c. In particular, regardless of the case for running the applications, alone or together, identical core c affinity is ensured for each measurement.

Corresponding to the observed behavior, it is rare to find $|m(p1)| = |m(p2)| = |m(p1, p2)|$. In fact, $|m(p1, p2)| > |m(p1)|$ and $|m(p1, p2)| > |m(p2)|$ are always the case due to cache sharing and context switching between concurrent applications on c. Context switching extends the elapsed time of concurrent applications due to the overhead and time involved while preempting resources from each other.

(a) SIFT (running alone) (b) FAST (running alone) (c) SIFT and FAST (running Concurrently)

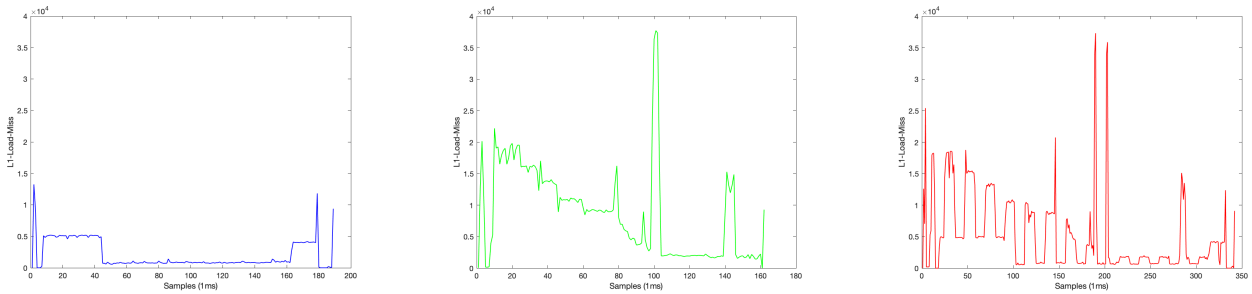


Fig. 3. Cache behavior of two applications when they were running alone and when they were running concurrently

B. Prediction Model

Providing the known behavior of applications running alone, the criteria for evaluation is the ability to predict the behavior if they execute concurrently. Let $x1$ and $x2$ be the indicators to target Y when $x1 = m(p1)$, $x2 = m(p2)$ and $Y = m(p1, p2)$.

This becomes a case for sequence-to-sequence regression modeling. However, there may not be enough data points in sequence $x1$ and $x2$, as explained in Section III-A. This discrepancy is handled by truncating each sequence to $min(|x1|, |x2|)$. There are two arguments to support this strategy. First, in observation formulation, if there is no information exists, training with fabricated data will not supply trustworthy forecasting. Especially in the case of a short application running concurrently with a long application, excessive padding can obscure the detection of genuine patterns since there will be more padded data points than actual data points. Second, within a batch, observations come from various applications of different execution lengths. If padded to the highest sequence length instead of truncated to the shortest sequence length, data will go through another cycle of padding to have batches of same-length observations. Besides, sorting observations as per their sequence length is also a good precedence to control unnecessary padding at the batch level.

Furthermore, applying zero mean and unit variance standardizes the data. This kind of standardization maintains balance in analysis and is represented as $\mathcal{N}(\mu, \sigma^2)$ such that $\mu = 0$ and $\sigma^2 = 1$. Once the data is normalized, the regression equation representing each observation will be:

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \epsilon \tag{1}$$

Here β_0 is intercept term, a constant to adjust the line of best fit. β_1 and β_2 are slope coefficients to represent the change in the dependant (target) variable Y. $x1$ and $x2$ are independent (indicators) variables to predict Y. Lastly, ϵ is the error term.

The observations are sorted, as described already, in descending order before splitting into the training and test sets. The sorting also gives an advantage in defining the batch size for the deep neural network.

Following the data preparation, neural network architecture is defined as the Input layer, LSTM layer, fully connected layer, dropout layer, output layer and regression layer. The Input layer does not perform any computation and just takes $x1$ and $x2$ as input features. The LSTM layer defines the most important information and is responsible for computation. It defines the number of neurons (the computational units in the LSTM block as described in Section II-F) and provides output as a sequence using the activation function. An activation function is used to identify the complex patterns in data. Then the fully connected layer defines the number of neurons whose each neuron connects with all the outputs. It is recommended to add neurons in the form of 2^n where n is positive integer, such as 64 or 128 or 256. Before moving to the output layer a penalty is also applied through the dropout layer to prevent overfitting. And finally, the regression layer provides the predicted sequence. The architecture is also shown in Figure 4.

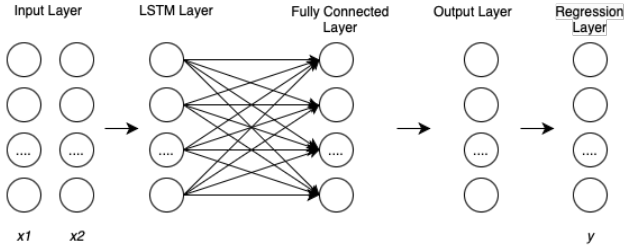


Fig. 4. LSTM Prediction Model for L1 Cache Misses.

To get the best-fit model, hyperparameter tuning is performed which is custom to the requirements and the problem to be solved. To name a few, a suitable batch size and learning speed are important for designing a good prediction model. We present those details in Section IV.

IV. Implementation and Experiments

Experiments are performed in two steps; data collection and analysis. We present the implementation details of the set up.

Hardware and Tools Specification: We collect data from an Intel processor running Linux OS *Ubuntu 4.13.0-21-generic* and *g++ 7.2.0*, detailed hardware specifications are listed in Table I. Performance profiling tool *PAPI library version 5.7.0.0* was used to measure the L1 cache misses. To restrict and control the CPU affinity of the applications we use *taskset* utility, available in Linux. Since it is a command line utility, we

secure the simultaneous execution of two applications on the same core using the shell script. The scheduler responsible for the execution multiple tasks is Completely Default Scheduler (CFS) [14]. Instead of even share of CPU time between all processes CFS allocates the CPU time between concurrent based on processes' priority and amount of time they have already consumed. Overall, for prediction, visualization and analysis purposes we use *Matlab version R2024a* which has advanced deep learning packages for extensive analysis.

TABLE I
HARDWARE SPECIFICATIONS INTEL® CORE™ I5 7200U

Feature	Hardware Component
Core	2xIntel® Core™ i5-7200U CPU (Kaby Lake) 2.5 GHz
L_1cache	8 KB 8-way set assoc. I-cache/core + 8 KB 8-way set assoc. D-cache/core
L_2cache	128 KB 4-way set assoc. cache/core
L_3cache	3 MB 12-way set assoc. Inter-core shared cache

Test Applications: Here, we characterize 5 applications namely 2×2 matrix multiplication, *SUSAN* (an image processor to find corners), *SIFT* (a complex feature detection algorithm to detects objects rather than just corners), *FAST* (a corner detection algorithm in images)and *SORT* (an insertion sort with average $\mathcal{O}(n^2)$ quadratic complexity). We targeted these applications not only because they are commonly used in industry but also because, depending on the workload (such as image size or length of input), their execution can significantly vary. So, by changing the load we can replicate various resource utilization demands. For example, for image processing applications we run them with load varying from $64KB$ image (normal load) to $8MB$ images (huge load). In short, these applications satisfy the requirements for experiments by affecting system performance in terms of computational and memory.

Measurements: For this study, our measurement approach is inspired by the method proposed in [15]. As we target only L1 cache misses so multiplexing is not required to capture more performance events. Instead, the work is focused on characterizing multiple applications. However, with PAPI it is challenging to record collective L1 cache misses from two separate processes at once. This is because $PAPI\_attach()$ wrapper function captures events for only one process at a time [16]. Characterizing two applications at different times does not reflect concurrent behavior. So we programmed a shell script that runs two applications simultaneously. Characterizing shell script will also include L1 cache misses from its child processes which in our case are two test applications. The script also ensures the core affinity by using *taskset* utility. This enabled us to sample the PMC that is recording L1 cache misses for each scenario i.e., running alone and running together. Each measurement collects L1 cache misses until the application(s) finish execution. It is important for machine learning models to consider if the data is coming from the same distribution therefore sampling frequency was 1 ms in each scenario and experiments were performed on the same hardware.

Data Preparation: We run test applications 40 times alone and 40 times with another application, in different combinations. Data is then divided into 50% training data set and 50% test data sets. The machine learning model learns the pattern observation by observation within batches. An observation is composed of $x1$, $x2$ and y. Whereas a batch is a set of observations. In total, 240 observations of different lengths (depending on execution time) were supplied, as a result in total 106885 records each were used for training and test. Test data was not used during the training to ensure the model's ability to generalize new data. This separation is necessary to avoid overfitting because otherwise, the model will learn the data very well that it may not be able to perform well when new data is arriving.

Prediction Model: In the prediction model, we have 2 input features (applications running alone) and 1 output feature (applications running concurrently). Generally, selecting and tuning all the hyperparameters is time-consuming. Selection and tuning of the most effective ones comes from domain knowledge and expert opinion. Here, we present some important ones. First is batch size. Sorting performed on observation as per their sequence length (response time) not only reduced the amount of padding, it enabled us to find an optimal batch size. Next is hidden nodes(neurons), quite a few were tested during the hyperparameter tuning and the model performed well with 70 neurons. Learning rate was also tuned and the final model is working well at 0.01 learning rate.

Results: In Figure 5, we present 6 examples of predicted L1 cache misses for two concurrent applications. The vertical coordinate shows L1 cache misses and the horizontal coordinate shows samples collected at every 1 millisecond. Without running two applications together, the model has predicted L1 cache misses, shown as '.-' line (in red). The '–' line (blue) shows the actual L1 cache misses when we have been running two applications in reality. In most of the cases, predicted behavior is moderately closer to the actual behavior, shown in Figure 5d and Figure 5f, which shows the model is not overfitted. But there are cases, when the model's prediction was not good i.e., in case of Figure 5e. The goodness of fit is also presented in Figure 6 with the help of a regression line. The dotted line labeled 'Y = T' is a reference line where predicted values are the same as original.

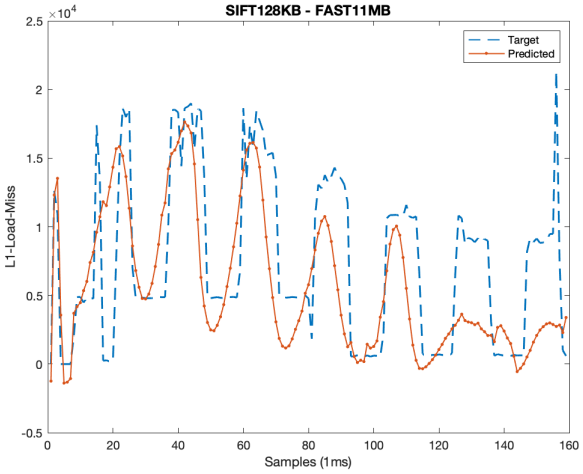
Evaluation: For regression models, R-squared (R) and Root Mean Squared Error (RMSE) are considered as metrics of evaluation. To quantify the accuracy, a lower value of RMSE is considered good. Our model demonstrated a prediction error of 0.5421 (RMSE) and 0.88766 (R). That means predicted values are 0.5421 units away from the actual values and 88.8% of variability in target can be explained with this model.

In Figure 6, the data points are close to the regression line that confirms consistent prediction can be made for a range of values. We see some points are distant from the regression line, this is where the model did not perform well, may be due to higher peaks or outliers.

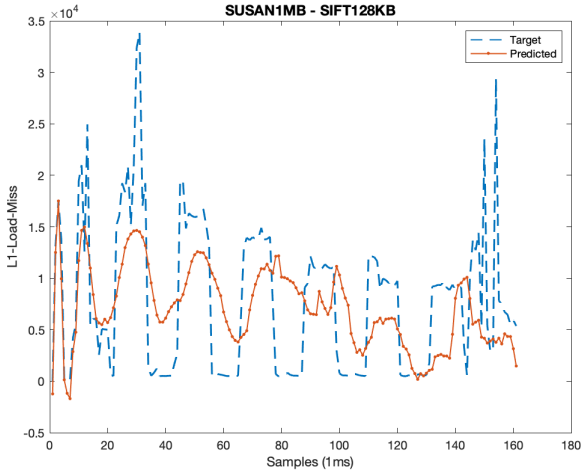
V. DISCUSSION

As per the main goal of the study, we have seen that by using the L1 cache misses of individual applications it is possible

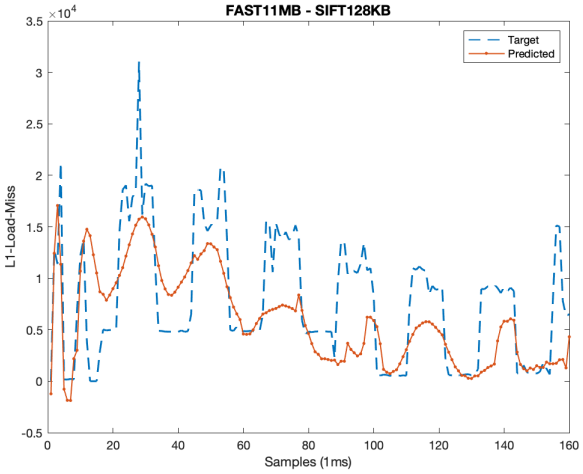
(a)



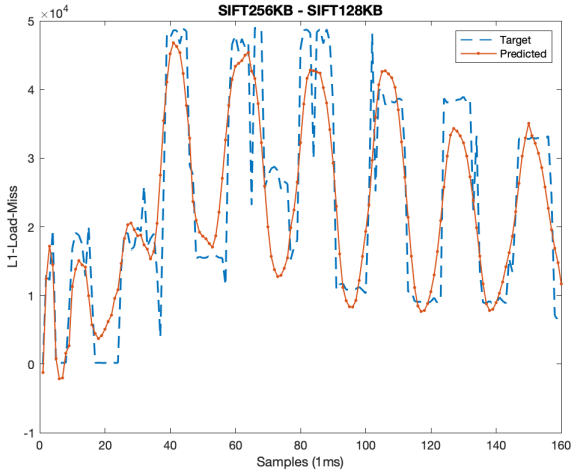
(b)



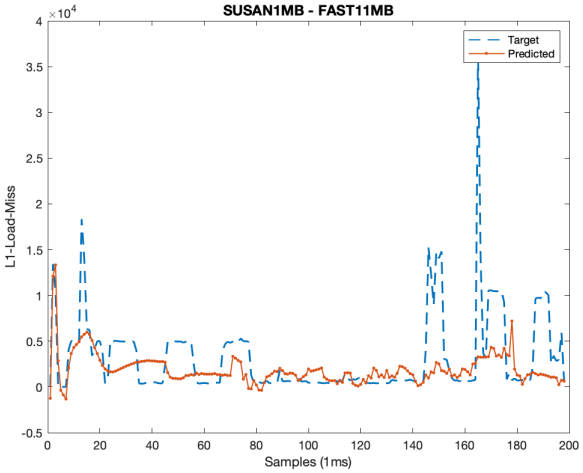
(c)



(d)



(e)



(f)

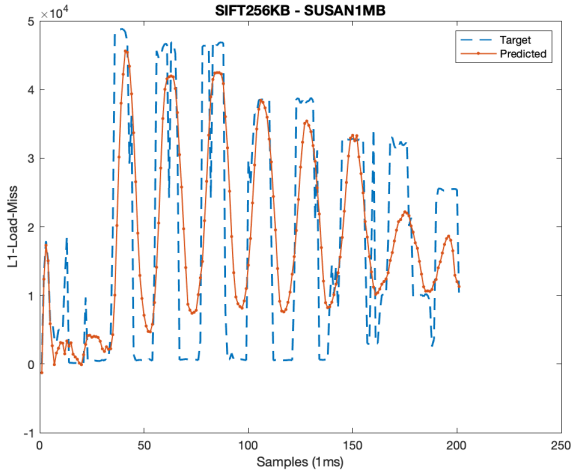


Fig. 5. Predicted L1 Cache Misses

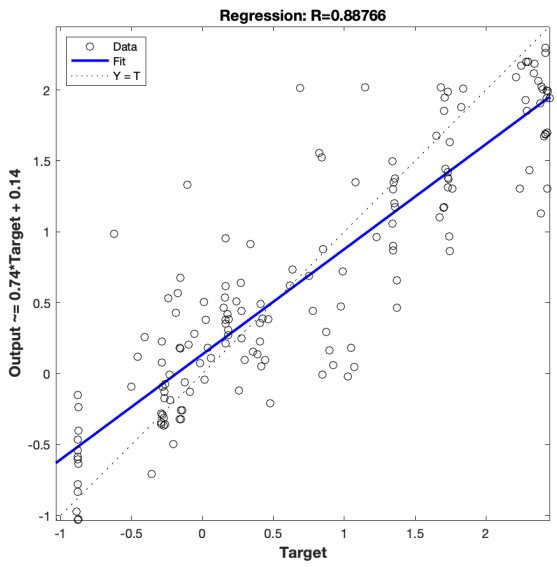


Fig. 6. Performance of LSTM Prediction Model for L1 Cache Misses for SIFT(with 256KB load) and SIFT(with 28KB load).

to predict the L1 cache misses of two applications running at the same time. When two applications are sharing CPU time then it is logical to expect increased elapsed time to finish both. The notion was confirmed during the experiments and is shown in Figure 3c. Furthermore, the study considers that if two applications, $p1$ and $p2$, start running together then they will interfere with each other only during the execution of the shorter application. Having this in mind, truncating the time series to the shortest becomes valuable to avoid unnecessary padding.

Another facet of this approach is generality. It can include various applications of different lengths and still can accurately predict the L1 cache misses. Anticipating a sequence without the need to physically run them on a system enables systems administrators and developers to simulate their demands.

Having quality data is a foundation for good analysis and predictive modeling. To supply the right data for the learning process, the simultaneous release of tasks has been ensured to maximize the execution interference between them. Besides, it is not problem to execute applications simultaneously, challenge is to filter L1 cache misses for two applications, in particular out of system wide L1 cache misses. The ability is tool dependant and it is not that straightforward using PAPI. Nevertheless, using PAPI on top of *taskset* cleared the obstacle and enabled us to satisfy the assumptions in the captured behavior.

With the availability of the right data, the deep neural network was configured to effectively minimize the prediction errors. Initially training with 200 neurons gave almost the same accuracy as with 128 neurons. The reason could be that the function has already reached the minimum value (loss) and adding more neurons was not improving prediction accuracy. Since neurons are the main computational units so it was inappropriate to add computation cost if there is no more loss to minimize. Instead, continuous training over new examples may improve the performance of the model. All in all, hyperparameter tuning is a fuzzy approach and one need to see where to stop.

Eventually, the regression analysis shows positive correlations between target and predicted output. Most of the data points are close to the regression line indicating that on average the model is valid. Data points distant from the regression line could be considered outliers but this is not the case for us because those points are evenly distributed which implies the fact that the model is not performing well in capturing the peaks. This can be confirmed with the Figure 5e that the model has not performed well in that case. Overall, the results demonstrate model is effectively forecasting L1 cache misses and in the future, more improvements can be provided.

VI. Related Work

Recent advancements in predictive modeling have focused on enhanced memory management techniques. Predictive modeling for cache has been used for many purposes such as for enhanced application performance [17], dynamic resources allocation, energy management [18], prefetching [18], cache sizing, fault detection, security monitoring [19] and real-time system performance [20]. Many researchers have studied interference caused by the applications using shared resources. A similar study with a similar focus was performed by researchers in [17]. They studied the applications that cause cache pollution when they are running in the same environment. The proposed mechanism was able to improve the application performance by taming the unused instructions responsible for cache management. This memory management technique did improve the performance through application classification and by utilizing non-temporal memory accesses.

As mismanagement of the cache can significantly degrade the performance, a proactive approach can improve the situation. Jalili has also proposed a cache-level predictions method to enhance the prefetching [18] of data which is one of the indicators for cache misses. They complemented the prefetcher with memory load information so that memory is loaded earlier than required. But a load can be subject to cache flushes. In those situations cache preemption delay significantly affects scheduling schemes [3]. Including cache-related preemptive delays for fixed-priority schedulers can avoid missing the deadlines. Another study focused on memory hierarchy and the impact of concurrent applications on memory behavior in [21]. They successfully showed how the performance can be optimized by predicting an accurate cache miss rate. Whereas, its been also investigated that there is a lack of performance models to predict L2 cache misses due to cache sharing between threads [22]. Regardless of the use case, most of the studies have mainly focused on higher levels of memory such as L3 and DRAM. It is good to mention that these traditional methods are time-consuming, explicit programming dependant and require a lot of effort and domain knowledge.

Compared to approaches developed around collecting low-level supportive details, modern approaches like machine

learning got more attention. Advanced approaches empower efficient solutions and reduce effort. One such case is for cache prediction. The field is advancing with predictive tools and machine learning algorithms [23]. A study has explored cache behavior by using performance monitoring counters since they provide real-time insights into system usage and performance. Not only that, studies have been performed to detect side-channel attacks through the prediction of cache [19]. By applying various cache configurations and partitioning techniques machine learning can help in saving time for static profiling of caches [24]. This saves energy and improves performance. An additional strength of deep neural networks for sequence-to-sequence prediction has been proven effective in [25]. Inspired by the approach, we aim to target L1 cache misses.

In general, a fair amount of work has been around L3 cache usage and prediction, the unique challenge of predicting the L1 cache behavior of concurrent applications is been addressed less frequently.

VII. Conclusion and Future Work

Memory management is a complex task and many factors can impact system performance. The complexity is increased due to memory hierarchy, cache-sharing techniques and OS policies during resource management. Despite all these facts, the study has provided a predictive model for L1 cache misses such that without the need to run the applications on a system. It can visualize how the L1 cache misses will be affected for a length of period in a shared environment. A deep neural network, LSTM, is trained on various applications of different lengths and types. This method ensures that our predictions remain accurate and relevant to the actual usage scenario. The deep neural model can predict a time series with RMSE, $= 0.5421$, and R, $= 88.8\%$.

In the future, studying complex scenarios involving more sources (such as memory bus, prefetches, stalls) can be performed to understand interference between more than 2 applications. It can help target high-tech infrastructures. The idea is to build a model that can predict the behavior of applications when they run together without the need to know the details of the hardware.

References

[1] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," in *IEEE micro*, vol. 28, no. 3, 2008, pp. 42–53.

[2] J. Danielsson, "Automatic Characterization and Mitigation of Shared-Resource Contention in Multi-core Systems," Ph.D. dissertation, Mälardalen University, 2021.

[3] R. J. Bril, S. Altmeyer, M. M. van den Heuvel, R. I. Davis, and M. Behnam, "Integrating cache-related pre-emption delays into analysis of fixed priority scheduling with pre-emption thresholds," in *Real-Time Systems Symposium*. IEEE, 2014, pp. 161–172.

[4] M. Jägemar, "Utilizing Hardware Monitoring to Improve the Quality of Service and Performance of Industrial Systems," Ph.D. dissertation, Mälardalen University, 2018.

[5] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual," 2022.

[6] AMD, "Open-Source Register Reference For AMD Family 17h Processors Models 00h-2Fh," 2018.

[7] ARM, "ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile," 2017.

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

[9] S. Manegold, "Memory Hierarchy," in *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu, Eds. New York, NY: Springer, 2018.

[10] T. Kilburn, D. B. Edwards, M. J. Lanigan, and F. H. Sumner, "One-level storage system," in *IRE Transactions on Electronic Computers 2*, 1962, pp. 223–235.

[11] S. Noll, J. Teubner, N. May, and A. Böhm, "Accelerating concurrent workloads with CPU cache partitioning," in *IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 437–448.

[12] Matlab, "Long Short-Term Memory (LSTM)," 2024. [Online]. Available: https://se.mathworks.com/discovery/lstm.html

[13] R. DiPietro and G. D. Hager, "Deep learning: Rnns and lstm," in *Handbook of Medical Image Computing and Computer Assisted Intervention*. Academic Press, 2020, pp. 503–519.

[14] J. Corbet. (2007) The CFS scheduler. Accessed on: 2024-05-03. [Online]. Available: https://lwn.net/Articles/230574/

[15] S. Imtiaz, J. Danielsson, M. Behnam, G. Capannini, J. Carlson, and M. Jägemar, "Automatic Platform-Independent Monitoring and Ranking of Hardware Resource Utilization," in *26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2021, pp. 1–8.

[16] Linux man pages. (2024) papi_attach. [Online]. Available: https://linux.die.net/man/3/papi_attach

[17] A. Sandberg, D. Eklöv, and E. Hagersten, "Reducing cache pollution through detection and elimination of non-temporal memory accesses," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11.

[18] M. Jalili and M. Erez, "Reducing load latency with cache level prediction," in *IEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 648–661.

[19] J. Cho, T. Kim, S. Kim, M. Im, T. Kim, and Y. Shin, "Real-time detection for cache side channel attack using performance counter monitor," *Applied Sciences*, vol. 10, no. 3, p. 984, 2020.

[20] F. Marković, J. Carlson, and R. Dobrin, "Cache-aware response time analysis for real-time tasks with fixed preemption points," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2020, pp. 30–42.

[21] A. M. Mohamed, N. Mubark, and S. Zagloul, "Performance aware shared memory hierarchy model for multicore processors," Scientific Reports, pp. 7313–7313, 2023.

[22] D. Chandra, G. Fei, S. Kim, and S. Yan, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *1th International Symposium on High-Performance Computer Architecture*. IEEE, 2005, pp. 340–351.

[23] H. Choi and S. Park, "A survey of machine learning-based system performance optimization techniques," *Applied Sciences*, vol. 11, no. 7, p. 3235, 2021.

[24] A. Ahmed, Y. Huang, and P. Mishra, "Cache reconfiguration using machine learning for vulnerability-aware energy optimization," in *ACM Transactions on Embedded Computing Systems (TECS)*. ACM, 2019, pp. 1–24.

[25] A. Saxena, K. Goebel, D. Simon, and N. Eklund, "Damage propagation modeling for aircraft engine run-to-failure simulation," in *International conference on prognostics and health management*. IEEE, 2008, pp. 1–9.