Modelling Application Cache Behavior using Regression Models

Jakob Danielsson[1],Janne Suuronen[1], Marcus Jägemar[1,2], Tiberiu Seceleanu[1], Moris Behnam[1], Mikael Sjödin[1]
[1] Mälardalen University, Västerås, Sweden
[2] Ericsson AB, Stockholm, Sweden
jakob.danielsson@mdh.se

Abstract—**In this paper, we describe the creation of resource usage forecasts for applications with unknown execution characteristics, by evaluating different regression processes, including autoregressive, multivariate adaptive regression splines, exponential smoothing, etc. We utilize Performance Monitor Units (PMU) and generate hardware resource usage models for the L_2-cache and the L_3-cache using nine different regression processes. The measurement strategy and regression process methodology are general and applicable to any given hardware resource when performance counters are available. We use three benchmark applications: the SIFT feature detection algorithm, a standard matrix multiplication, and a version of Bubblesort. Our evaluation shows that Multi Adaptive Regressive Spline (MARS) models generate the best resource usage forecasts among the considered models, followed by Single Exponential Splines (SES) and Triple Exponential Splines (TES).**

I. Introduction

Cache memories in multi-core systems are prone to resource contention, most notably the last-level cache since it is commonly shared across multiple cores and allows for simultaneous usage [13]. The cache is a small, finite memory storage area and will evict data when its' capacity limit is met; the evictions are called cache misses. In contrast, references to a cache memory block are called cache accesses. Resource contention typically occurs when two memory-intensive applications execute on different cores, continuously executing cache accesses to new memory blocks. The cache memory will become full at some point during execution and, therefore, needs to evict cache lines to make space for new data requests. A vicious cycle can in the worst cases occur, where the applications' cache accesses continuously triggers cache evictions from each others' data, leading to performance degradation's and execution time fluctuations. One popular way to avoid such a scenario is to disqualify simultaneous usage of certain cache blocks through page coloring, also known as cache partitioning [30]. Page coloring removes resource contention through assigning specific cache blocks to specific processes at the cost of overhead performance penalties [9].

The execution characteristics of applications different depending on the application functionality. Applications are typically split into several phases [23], such as cache-heavy phases, arithmetic-heavy phases and floating-point heavy phases. A cache-heavy phase means the majority of the instructions leads to an access in the cache memory. In contrast, an arithmetic phase means the majority of the instructions causes an operation within the Arithmetic Logic Unit (ALU), etc. The most vicious scenario for cache contention is when two applications run simultaneously on different cores while executing their most cache heavy phases and stresses the cache to the capacity limit. We should not run applications that execute their most cache heavy phases simultaneously because of resource contention. Instead, we should schedule applications according to their shared resource usage, so the resource-specific phases (e.g., cache-heavy phases) never collides with each-other, thus mitigating the resource contention to a small degree. Making such a schedule requires modelling techniques that estimate the applications' resource usage trends for offline scheduling. To further adapt the methodology for reactive, run-time scheduling, we also need the model to predict the future resource usage.

Regression modelling is a mathematical process used to analyze trends in time-varying processes such as stock prices. In this paper, we benchmark different regression models with respect to the computing realm. We aim to create suitable regression models that can formalize an application's resource usage and create a prediction model for future resource usage.

We use the computer's performance counters to generate resource usage models. Our models look at what hardware is triggered by a software application and estimate its future resource usage. Performance counters are widely used in modern computers, making the modelling approach scale-able to all hardware that has the performance counter utility. In this paper we exemplify the modelling process using a set of three applications, including Bubblesort (non-cache heavy), matrix multiplication (cache heavy) to serve as synthetic workloads, to show resource forecasting applicability. We also use the Feature detection algorithm Scale Invariant Feature Transform (SIFT) [17] to serve as a realistic workload for resource forecasting.

Our contributions are:

- Resource forecast modelling on the three previously mentioned algorithms using the forecast models available in the Statsmodel module [22].
- A comparison evaluation on the accuracy of the different forecast models using the Root Mean Square Predicted Error (RMSPE) as a comparison metric.

The rest of this paper is organized as follows: Section II presents relevant notions, including measurement strategy, computer resource usage, and regressive performance analysis. Section III introduces our method for evaluating different regression processes. In section IV we show the comparison results of the nine regression processes and discuss the results. Section VI concludes the paper and presents opportunities for future work.

II. Background

It is challenging to predict bottlenecks for a particular hardware resource (such as L_1-cache, L_2-cache or similar) since the hardware resource usage may vary significantly during the execution time. For demonstrative purposes, we show the L_3-cache usage for the SIFT algorithm using an 8MB image executed on a single CPU, Figure 1. The y-axis plots the total number of L_3-cache accesses, while the x-axis shows the measurement points over the entire execution.

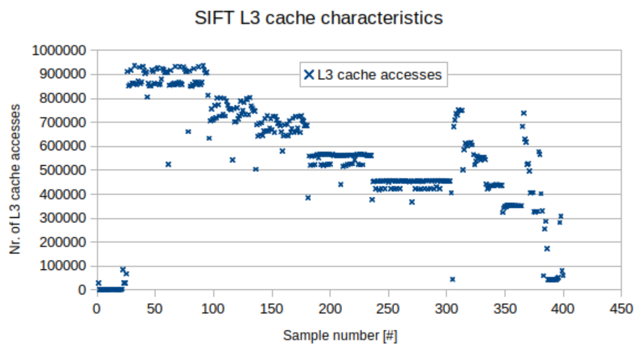

Fig. 1: Illustration of SIFT using an 8MB image.

The L_3-cache usage of the SIFT algorithm varies by a significant amount over time. The L_3-cache usage is low at the start of the algorithm and rapidly increases after 0.5 seconds. The L_3-cache accesses count is significantly reduced at the 6-second mark.

The SIFT application takes 9.1 seconds to execute, which itself is not a very long time. Still, a complete software system often consists of 10's to 1000's of tasks, which could have similar execution time to that of SIFT. Accurately forecasting resource usage can significantly decrease run-time applications' testing time since they do not need to run for their full duration.

Forecasting and predicting the hardware resource usage also helps system designers in making three significant decisions, listed as follows:

- Hardware evaluation: The system designer will be able to distinguish sooner if a specific platform has enough capacity to run the software.
- System scheduling: Forecasting will enable scheduling the system run-time in a resource-efficient way so that hardware resources can be utilized at their maximum capacity without interference from other tasks.
- Resource bottlenecks: Accurate resource usage forecasting can also indicate if an application will be affected by resource capacity limits in the future casing resource bottleneck.

In the following sections, we discuss how different resources affect an application's performance and how to measure interesting resources using the Performance Monitor Unit (PMU). We also discuss different regressive models for forecasting application resource usage.

A. Computer resource usage

Computers consist of a vast set of resources, such as cache's, memory management unit (MMU), main memory (DRAM), I/O's, etc. These resources add functionality to the processor, such as memory access speed through temporary memory storage areas (cache's), increase instruction-level parallelism (processor pipeline), increase process parallelism (processor cores), etc. All applications utilize at least one computing resource during execution and are therefore dependent on this resource to complete their execution - we call this resource boundness [10]. However, many applications are often complex and thus bound to several resources simultaneously - a memory-bound application, for instance, typically utilizes the entire spectra of the memory chain: TLB's, cache's, DRAM, and instruction memory.

In this paper, we mainly investigate how to generate and forecast cache resource usage models. We limit ourselves to constructing resource usage forecasting models of cache for memory-bound applications, and, at the moment, we exclude other resource boundness situations.

B. Measurement strategy

Measuring the resource usage of applications can be done using the Performance Monitor Unit (PMU), which is included in most modern hardware. The PMU hosts a large set of counters - the Performance Monitor Counters (PMC), which are event-triggered hardware counters that trace the various resource usage within a computer. We use here the Performance API (PAPI) [18], a performance counter library that utilizes the built-in Linux *perf* headers [26] for measuring the performance counters. Further, we take a sampling-based approach to generate resource forecasts of applications. This means that we continuously measure the performance counters during runtime of an application with a certain frequency, instead of measuring the total count. Figure 2 depicts our measurement strategy.

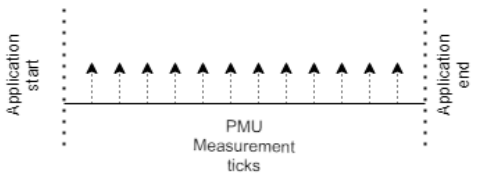

Fig. 2: Periodic measurement of performance counters.

In this way, we can generate resource usage forecasts on the individual application, since we have multiple sampling points of the cache resource usage.

C. Regressive performance analysis

Regressive analysis is a method for modeling relationships between dependent and independent variables through a statistical process. Dependent variables are what a regressive model tries to predict or model. Independent variables are factors that have an impact on the dependent variables we are investigating. For example, a dependent variable can correspond to the execution time of any given process. A potential independent variable is the number of cache misses within that process, which may negatively impact that process's execution time.

There exists a broad spectrum of different regression tests, e.g., *autoregressive* and *spline modeling* processes [19]. Regressive modeling requires a dataset to construct models and approximate dependent variables by estimating independent variables' functions alongside an error term. The result of this procedure is an estimation model of the relationship between different variables of interest. The final product of estimating a mathematical function is the ability to forecast dependent variables.

We list here the major steps to construct a regressive model:

1) Select a modelling process suitable for the observed patterns in the data
2) Fit the parameters of a model to training data according to what is dictated by the modelling process.
3) Evaluate the prediction accuracy, for example by examining Root Mean Square Prediction Error (Equation 4), of the fitted model using a validation dataset.

There are two types of models that regressive modeling processes estimate:

- *Parametric models* consist of a finite number, specified before the model is generated, of parameters.
- *Non-parametric models* can theoretically include an infinite number of model parameters.

We limit this work only to include first-order parametric models. Which means the number of parameters (i.e., the amount of variables within the model) is limited to only one. The quality of the generated model is measured by evaluating how the polynomial function of the model "fits" the measured dataset. There exists three outcomes, *good* fits, *overfits* and *underfits*. Good fits accurately describe the shape of the dataset and can also detect the trends in the dataset. Therefore, they can forecast future values of the dataset. Overfits means the model has generated too many parameters and will be an almost exact representation of the dataset. Overfits also means that it is impossible to make any forecasts of the dataset since the model is an exact representation of the values in the current dataset. Underfit means the model has too few parameters to make any forecast prediction. Sharma et al. [24] exemplify over-, under- and good-fitting in Fig. 3:

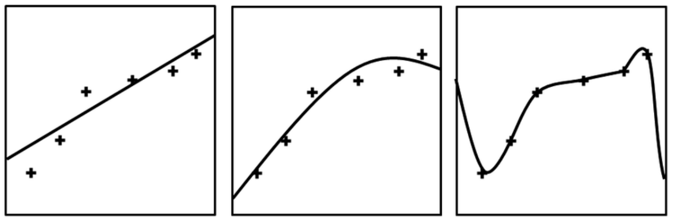
Fig. 3: Example of under- (left), good- (middle) and over-fitting (right) [24]

Other than the problems of over-and under-fitting curves, adopting regressive analysis to construct forecasting models of hardware resource usage is a relatively straightforward process. Regressive models require datasets to be available at the moment of construction. Thus, we sample the cache resource usage of processes before initiating the model construction step and feed the dataset to our regressive models once the sampling is finished. Our regressive resource usage models are, therefore, offline representations of resource usage.

D. Related work

Existing research addresses methods that investigate how to create hardware resource usage models and how they can be applied to predict application performance. Several studies evaluate statistical regression models as suitable candidates for building performance forecasts [25, 14]. These works show how different performance values can be predicted using different regressive models. Other related work which is closest to our investigates resource usage through a novel autoregressive model called *Threshold Autoregressive* (TAR) [6]. The authors show that it is possible to create resource usage forecasts of any given application using regressive models, with a relatively low prediction error. Our research expands on this topic, and we evaluate the applicability of resource forecasting on different regressive models in the Statsmodel module [22] and *pyearth* for MARS models. Other results on predicting software performance and resource usage using autoregressive models are presented by Courtois and Woodside [8].

While some of the previously listed works employ similar regressive models to ours, we introduce the use of hardware performance counters' in conjunction with regressive models. To the best of our knowledge, no other research works utilize performance counters as a mean for forecasting and detecting hardware capacity bottlenecks.

Other relevant works investigate how to use PMU's to evaluate application performance models without in-depth knowledge of application code [27, 2, 7, 15, 12, 21]. However, these current methodologies strictly rely on measured data, which requires an application to be run until completion at least once to completion. Resource bottlenecks can only be discovered offline after application execution. Since our paper targets forecasting, our additions to this domain enable us to discover potential hardware resource bottlenecks before they occur.

III. Method

A. Model System Behaviour

We primarily focus on constructing forecast models of an application L_2-cache and L_3-cache usage. We start by gathering hardware resource usage samples during the application runtime to generate forecast models. We use a time slot sampling strategy which is similar to a frequency based measurement strategy. One PMU measurement sample is taken at the end of each sampling timeslot. We determine the time-length of a time slot according to Equation 1, where T is the time slot length in a time unit, app_e is an applications' execution time and s is the desired number of samples.

$$T = \frac{app_e}{s} \tag{1}$$

Assuming an application execution time (app_e) of 1 second and the desired number of samples (S) is 100, the timeslot length (T) is equal to 10 milliseconds, which means a sampling rate of 100Hz. We denote the set of all measurement samples as y_c. Here, y denotes the application, and c denotes the performance counter. In this paper, we focus only on L_2-cache and L_3-cache accesses, thus, c will indicate either the L_2-cache or L_3-cache accesses. We furthermore denote the individual performance counter sample of an application y as t, we can have the complete execution characteristics of c, with 100 samples, given in Equation 2.

$$y_c = \{t_0, .., t_{100}\} \tag{2}$$

Next, we use y_c set to generate the forecast model $\hat{y_c}$, see Equation 3, where model represents a regressive model process.

$$\hat{y_c} = model(y_c) \quad (3)$$

The populations y_c and $\hat{y_c}$ describes the actual (y_c) and modelled ($\hat{y_c}$) cache resource usage at specific time points. We access data within the respective population using discrete time points t. $y_c(t)$ returns the actual cache resource usage at timepoint t and $\hat{y_c}(t)$ returns the modelled cache resource usage at timepoint t.

We exemplify a resource usage forecasting scenario using matrix multiplication as a test application and a MARS model as forecasting generation model, Figure 4. The first 50 samples in the model show a tendency of overfitting but still generates a close estimation of the measured L_2-cache usage in the last 100 samples. The y-axis shows the amount of L_2-cache accesses, and the x-axis shows the sample number. One blue dot corresponds to the data samples at a specific time point (t_i), which means all blue dots builds the population y_c where c is equal to L_2-cache accesses. The green line depicts the MARS model's fit, and the red line depicts the forecast model produced by MARS ($\hat{y_c}$). The figure visualizes the purpose of resource forecasting, i.e., the ability to forecast the L_2-cache usage.

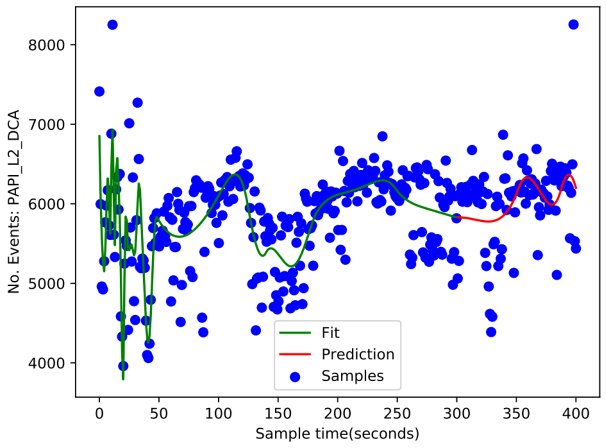

Fig. 4: Example of an L2 cache usage function estimated using a MARS model [19].

The figure shows the model generation of MARS, where the red line depicts the actual forecast model. We then evaluate the model's applicability using RMSPE, see Section III-B. In this work, we investigate the applicability of multiple different regression models, including Auto-regressive, Auto-regressive Moving Average, Auto-regressive Integrated Moving Average, and Spline (Natural, B- and MARS) Regressive modeling processes [19] and their applicability for forecasting applications' usage of L_2-cache and L_3-cache memory. We list all the modelling that we consider in Table II.

B. Evaluation Methodology

We use the Root Mean Square Prediction Error (RMSPE) to evaluate our regressive models' accuracy. The RMSPE metrics describe the difference between predicted values and actual observation values of the data set. This paper uses RMSPE as a model comparison metric; the lower RMSPE value means the difference between the actual data and the forecast model is smaller and is preferable over a high RMSPE value. Equation 4 describes the RMSPE calculation, which is the root square value of the difference between all values in population $\hat{y_c}$ and y_c divided by the number of samples n.

$$RMSPE = \sqrt{\frac{\hat{y_c} - y_c}{n}} \quad (4)$$

Equation 4 gives the prediction error by squaring the sum of the averaged difference between predicted ($\hat{y_c}$) and actual (y_c) values, where n is the number of samples considered. In this paper, we exclusively use 400 as the number of samples, and therefore, n will always be equal to 400 in our experiments.

IV. Experiments

We generate resource usage forecast models and evaluate the RSMPE value using the platform in Table I.

TABLE I: Hardware specifications Intel® Core™ i5 8250U

Feature	Hardware Component
Core	4xIntel® Core™ i5-8250U CPU (Kaby Lake) 1.6 GHz
L_1-cache	32 KB 8-way set assoc. I-cache/core + 32 KB 8-way set assoc. D-cache/core
L_2-cache	256 KB 4-way set assoc. cache/core
L_3-cache	6 MB 12-way set assoc. Inter-core shared cache

In addition to the hardware setup, we set the desired number of samples (s) to 400 for all experiments. We use 400 for all applications since it provides the best trade-off between over- and under-fitting of the curves for our test applications. In the following subsections, we discuss the applications put under test, the software execution environment and also the different regressive models that we use.

A. Execution scenario

We use three different applications including, a traditional bubblesort of an array, a conventional matrix multiplication of two randomly generated matrices, and finally, an application containing the SIFT algorithm [20] for detecting features within an image. We use a matrix multiplication and bubblesort due to the simplicity in following their execution characteristics. Furthermore, we use SIFT to display resource forecasting usage in a more realistic non-synthetic scenario. In the following subsections, we discuss the basic mechanics and the resource usage of our applications.

1) BUBBLESORT: The BUBBLESORT algorithm compares two adjacent values within an array, the left-hand side value, and the right-hand side value. If the right-hand side value is lower than the left-hand side value, these values swap location within the array. The bubble sort application's main mechanic utilises comparisons mainly, which means it is a heavily branch-predictor dependent application.

2) Matrix multiplication: We use a standard *ijk* matrix multiplication, famous for loading the cache in a very suboptimal way. Our matrix multiplication multiplies the columns of one matrix A with the row of matrix B. The result value is stored in matrix C. The procedure of loading values from a matrix and storing new values into another matrix is very memory intensive, which means its a memory-bound application, including caches and DRAM.

3) SIFT: SIFT is a complex feature detection algorithm containing several mathematical operations such as the difference of Gaussian, nearest neighbor, hough transform voting, linear least squares, and more. The mathematical operations mean the SIFT application performs multiple steps and may depend on several different resources during the algorithm's different phases.

B. Environment

We collect data using the platform specified in Table.I) running the 64-bit desktop version of Ubuntu 18.04 LTS in an unmodified state with Linux kernel version 5.3.0-46-generic. As a measure to lessen stalls due to user-related interface interaction, we disable the graphical interface. We reboot our test platform for each test run to clear cache levels and ensure each test runs with a cold cache and comparable circumstances. Our experiments run on an as-is Intel® Core™ i5 8250U(Kaby Lake architecture) with four homogeneous cores clocked at a base frequency of 1.60 GHz and a three levelled cache hierarchy. We list all the details on our test platform in Table I.

C. Execution

For each test application, we collect L_2-cache and L_3-cache resource usage data at a sampling rate specified in Equation 1. The resulting three datasets are each individually split according to a 75/25% ratio which yields data subsets of 300(75%) and 100(25%) measurements for model training and testing. The values of hyperparameters is a crucial factor to consider when fitting regressive models. Regarding Auto-Regressive models(AR, ARMA, ARIMA), the adjustable hyperparameters are the Auto-Regressive(AR) and Moving Average(MA) orders. In our case, all Auto-Regressive-based models are of the first order. The Exponential Smoothing models(SES, DES and TES) are tunable by modifying the weights applied to previous observations(α), trends(β) and seasonality(γ). We optimize α in SES and α, β, γ in TES using maximum log-likelihood. α and β in DES are set to 0.8 and 0.2 respectively as this procured better results compared to maximum log-likelihood optimization. Finally, the Spline-based models are tunable by the number of knots, Degree of Freedom(DF), and the maximum polynomial degree of each spline. Both B-spline and N-spline DFs are set to 10, whilst the maximum polynomial degree is set to five. In MARS, only the maximum polynomial degree is adjustable by design and is set to 3.

A key aspect concerning regressive models is their data demands to avoid over-and underfitting. Thus we set the sampling frequency to capture enough measurements without compromising the significance of the observed usage. Adopting a higher sampling rate could provide unrepresentative usage data since the overhead of measuring the performance counters becomes overwhelming compared to the actual measurement samples.

We run the SIFT algorithm on an 8MB image. BUBBLESORT sorts a 6MB array of randomly generated values. The MATMULT workload multiplies two matrices summing up to a workload size of 1MB. We sample the L_2-cache, and L_3-cache accesses during each application's execution. Figures 5, 6 and 7 plots the execution profiles for the SIFT, MATMULT and BUBBLESORT respectively. Orange dots mark the quantity of L2 accesses and blue crosses, which marks the number of L3 cache accesses over 400 measurements.

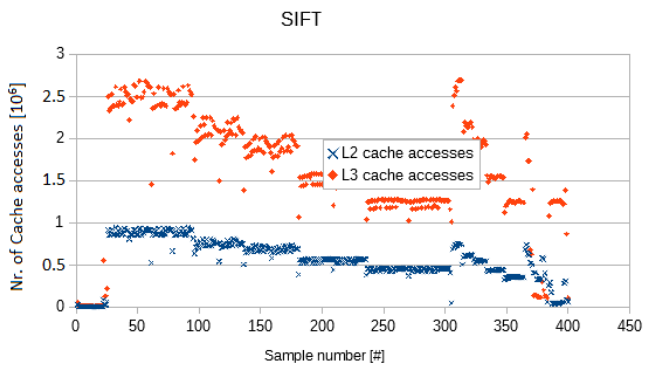

Fig. 5: Memory usage illustration of SIFT using an 8MB image.

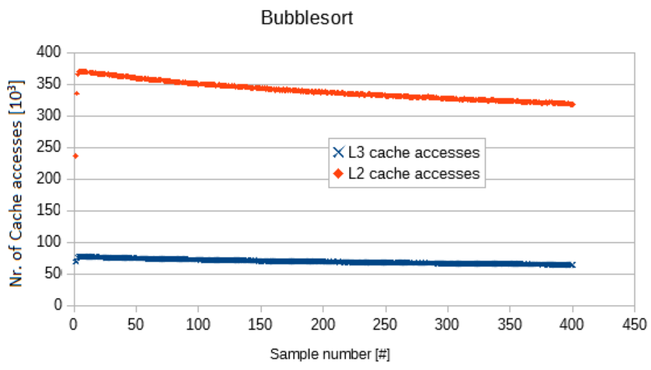

Fig. 6: Memory usage illustration of a matrix multiplication using a 1MB dataset.

ma

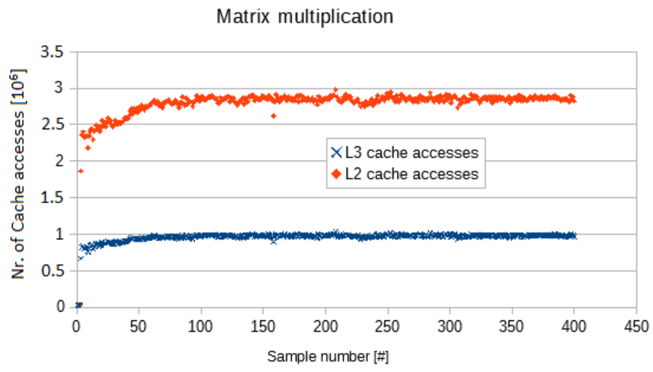

Fig. 7: Illustration of SIFT using an 8MB image.

All three applications show very different execution profiles. The BUBBLESORT application shows a contiguous decrease in both L2 and L3 cache accesses. The matrix multiplication instead has a ramp phase, where the cache accesses rapidly increase in the beginning while saturating at measurement sample 80. SIFT shows stage-alike patterns in cache accesses, where there is first a dormant stage with almost no cache accesses. At sample 30, the cache accesses increase rapidly and remains high until sample 100, where the accesses starts to decrease gradually.

D. Model Comparison

Once the measurement phase finalizes, we create resource usage models using the regression processes listed in table II of each application, using the sample measurements.

Modelling process	Type
Auto Regressive (AR) [1]	Non-param
Auto Regressive Moving Average (ARMA) [28]	Non-param
Auto Regressive Integrated Moving Average (ARIMA) [3]	Non-param
Regressive B-spline [11]	Param
Regressive Natural Spline [5]	Param
Multivariate Adaptive Regressive Spline(MARS) [14]	Non-param
Simple Exponential Smoothing(SES) [4]	Non-param
Double Exponential Smoothing(DES) [16]	Non-param
Triple Exponential Smoothing(TES) [29]	Non-param

TABLE II: Modelling processes evaluated in this work.

Our complete regression model suite generates a total of 54 forecasting models for our three example applications; 18 different models for each application, 9 different models for each cache level. For each of the 54 models, we calculate the RMSPE according to Equation 4. The RMSPE score describes how accurate forecasts made by a model are through calculating the the error size of the predictions. Thus, a lower RMSPE score is preferable over a higher one. Figures 8 and 9 shows the RMSPE scores for the SIFT application. Figures 10 and 11 the same for BUBBLESORT and Figures 12 and 13 the corresponding for MATMULT.

1) SIFT models: The first application we examine executes the SIFT on an 8MB image. The MARS models notably achieve the lowest RMSPE score for both cache levels. Figures 8 and 9 lists the RMSPE value of each different regression process on the left-hand side y-axis. Smaller RMSPE value means the prediction error is lower and is, therefore preferable to a high RMSPE value.

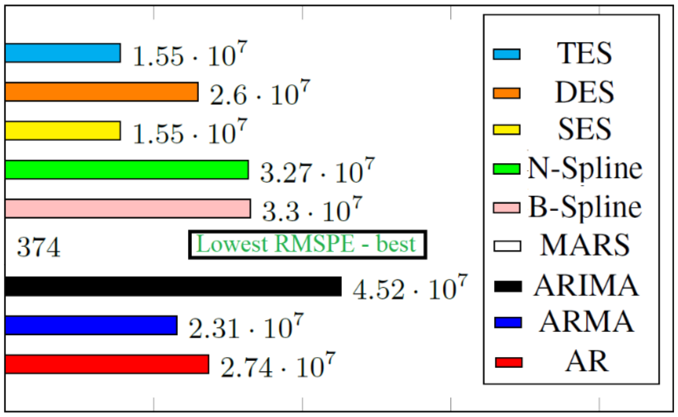

Fig. 8: RMSPE score of L_2-cache usage models from data collected during execution of the SIFT algorithm with a 8MB image.

The SIFT workload identifies edge features in an 8MB image. On the other side of our SIFT RMSPE spectrum, the ARIMA models stand out with the highest RMSPE scores out of the calculated model scores. The remaining modeling processes achieve similar scores within the respective modeling

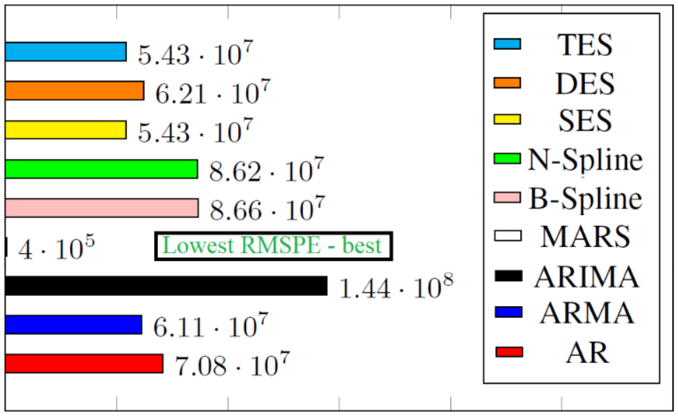

Fig. 9: RMSPE score of L_3-cache usage models from data collected during execution of the SIFT algorithm with a 8MB image.

process family. That is, B- and Natural splines models achieve similar RMSPE scores; the same applies to the Exponential Smoothing family of modeling processes (SES, DES and TES).

2) BUBBLESORT models: Our second application performs a traditional bubble sort on an unsorted integer array 6MB in size with random values. Figures 10 and 11 presents the RMSPE scores of each successfully constructed model.

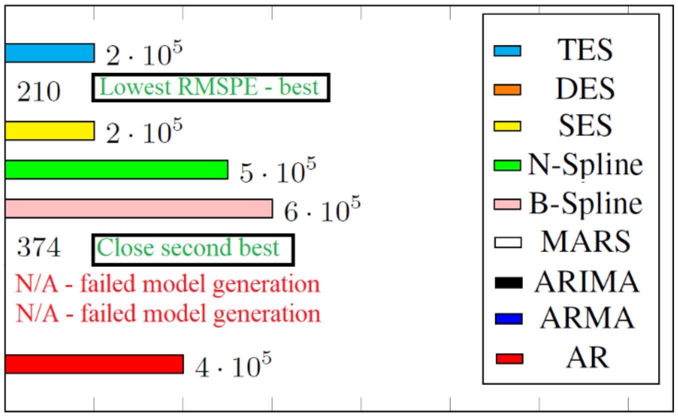

Fig. 10: RMSPE score of L_2-cache usage models from data collected during execution of the BUBBLESORT algorithm with a 6MB array.

The BUBBLESORT RMSPE show MARS, SES, and TES outperform the other regression processes in prediction error. Furthermore, ARMA and ARIMA models failed to construct models from the dataset, due to lack of invertibility in the Moving Average(MA) component. We were, thus, unable to calculate RMSPE values for these modeling processes.

3) MATMULT models: Our third and final application performs a matrix multiplication between two square matrices with a working set size of 1MB for each matrix. Amongst the L_2-cache models, the SES and TES models achieve, nearly identical, the lowest RMSPE scores, as seen in Figure 12. The B-spline L_2-cache model achieves the highest RMSPE score, and the ARIMA model fails to construct due to lacking data invertibility for the MA component. The RMSPE scores calculated for the L_3-cache usage models show a different out-

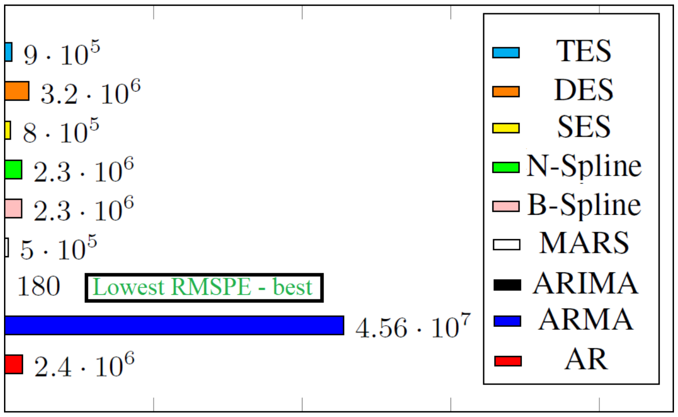

Fig. 11: RMSPE score of L_3-cache usage models from data collected during execution of the BUBBLESORT algorithm with a 6MB array.

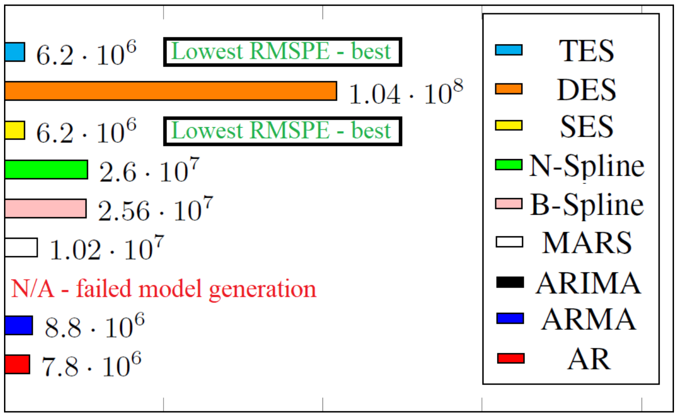

Fig. 13: RMSPE score of L_3-cache usage models from data collected during execution of the MATMULT algorithm with a 1MB working set.

come. Double Exponential Smoothing achieves a significantly higher RMSPE score, as visualized in Figure 13, while SES and TES models provide the smallest RMSPE scores.

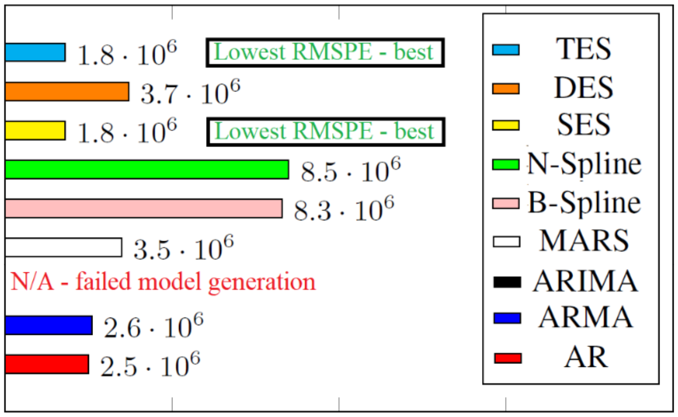

Fig. 12: RMSPE score of L_2-cache usage models from data collected during execution of the MATMULT algorithm with a 1MB working set.

V. Dicussion of applicable methods

We examine several regressive modeling processes with the purpose of forecasting L_2-cache, and L_3-cache usage, expressed as the number accesses done per unit of time performed to a given cache level.

We examine both parametric and non-parametric modeling approaches. We state that parametric processes are too inflexible to model cache usage, as shown by the relatively high RMSPE scores of B- and Natural Spline models. Parametric models require set parameters before construction, requiring users to learn the resource usage pattern before setting optimal parameters. Setting parameters before-hand means additional testing time for finding the optimal parameters rather than actual testing, which contrasts our goal of reducing testing costs. As such, non-parametric models are preferable since they are more flexible compared to parametric alternatives.

Flexibility is a desired trait as cache usage patterns are not uniform across all applications. However, despite a higher degree of flexibility, non-parametric models are not fault-free since there is difficulty finding good fits when the measurement values are highly fluctuating [19]. Our work models three applications which do not have fluctuating resource usage characteristics to that extreme extent, and as such, we did not encounter the issue. A method of combating irregular usage patterns is adjusting the frequency to minimize the difference between each measurement. This solution is not without its problems, as some applications might have mixed and periodical resource boundness. Thus, measuring too frequently becomes an issue, as some applications might not finish a viable amount of work between each measurement.

In four out of six cases, MARS achieves the lowest RMSPE values of the non-parametric regression processes evaluated in this paper. TES and SES present a considerably lower RMSPE value, modeling the matrix multiplication application than MARS. These lower RMSPE values are visible in figures 12 and 13. The difference in low RMSPE values, depending on the different applications we use, suggests the best regression processes is dependent on the sample characteristics. Since MARS provides the best overall RMSPE value, it will be the best when forecasting resource usage of an application with completely unknown execution characteristics.

There is an inability to construct some of the regressive models with a moving average component, including ARMA and ARIMA, which happens as a consequence of wrongful tuning of the moving average component input parameter. Fine-tuning the input parameters of our models, however, means we need to add additional parameters into the model, which goes outside of the limitation of using only first-order models. As a final remark, referring back to this paper's original question: Can we predict a given application's resource usage using regression models? We argue that this is doable, as indicated by our results.

Since most CoTS hardware typically implements a broad set of performance counters, there are opportunities for complete resource usage forecasts. Our approach using frequency-based measurements on individual applications makes resource usage forecasts possible for any application as long as the hardware implements the performance counters, which are of interest.

VI. Summary

In this paper, we evaluate methods for forecasting the resource usage of 3 different applications. We evaluate auto-regressive, spline regressive, and exponential smoothing as approaches for modeling applications and their usage of CPU L_2-cache L_3-cache. We compare the modeling fitness using RMSPE against each other to single out an ample modeling process. Our evaluation shows that MARS shows the most promise for forecasting application resource usage among the nine different evaluated regression processes.

A. Future work

We want to extend the MARS-work presented in this paper with resource scheduling forecasting using MARS processes making it possible to schedule processes in a cache-aware fashion. The processes do not interfere with each other, thus decreasing the risk for cache contention.

The three workloads we use in this paper are traditional Bubble Sort, Matrix Multiplication and Scale-invariant feature transform(SIFT). These commonly appear in larger applications and are thus representative for small chunks of code within a system. Future work includes examining our approach in conjunction with large-scale system solutions which include more complex applications with different resource usage profiles. Future work also includes conducting the resource forecasts in a scheduling environment where we test our hypothesis on resource contention. Ideally, two applications executing their most cache heavy phases should not run simultaneously since it builds a perfect environment for resource contention. Since we build resource usage profiles, the final goal is to create a new scheduling technique to mitigate cache contention through analysis of the cache access patterns.

In this work, we only use one hardware platform and one compiler flag. The resource usage profiles will be different using different hardware and compiler flags since the . Our approach is however agnostic since it only uses events produced from the performance counters. We however want to verify the RMPSE calculations against other hardware with different cache configurations to compare RMSPE differences for the models.

References

[1] H. Akaike. Fitting autoregressive models for prediction. *Annals of the institute of Statistical Mathematics*, 21(1):243–247, 1969.

[2] R. Azimi, David K. Tam, L. Soares, and M. Stumm. Enhancing operating system support for multicore processors by using hardware performance monitoring. *SIGOPS Oper. Syst. Rev.*, 43(2):56–65, April 2009. ISSN 0163-5980.

[3] G. EP. Box and G. M. Jenkins. Time series analysis: Forecasting and control san francisco. *Calif: Holden-Day*, 1976.

[4] R. G. Brown. Exponential smoothing for predicting demand. In *Operations Research*, volume 5, pages 145–145, 1957.

[5] J. Cao, M. Valois, and M. S. Goldberg. An s-plus function to calculate relative risks and adjusted means for regression models using natural splines. *Computer methods and programs in biomedicine*, 84(1):58–62, 2006.

[6] X. Chen, Q. Quan, Y. Jia, and K. Cai. A threshold autoregressive model for software aging. In *2006 Second IEEE International Symposium on Service-Oriented System Engineering (SOSE'06)*, pages 34–40, 2006.

[7] Y. Cho, Y. Kim, S. Park, and N. Chang. System-level power estimation using an on-chip bus performance monitoring unit. In *2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 149–154, Nov 2008. doi: 10.1109/ICCAD.2008.4681566.

[8] M. Courtois and M. Woodside. Using regression splines for software performance analysis. In *Proceedings of the 2nd International Workshop on Software and Performance*, WOSP '00, page 105–114, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 158113195X. doi: 10.1145/350391.350416.

[9] J. Danielsson, Jägemar Marcus, Tiberiu Seceleanu, Moris Behnam, and Mikael Sjödin. Run-time cache-partition controller for multi-core systems. In *In 45th Annual Conference of the IEEE Industrial Electronics Society (IECON), 2019*, 2019.

[10] Jakob Danielsson, Tiberiu Seceleanu, Marcus Jägemar, Moris Behnam, and Mikael Sjödin. Testing performance-isolation in multi-core systems. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 604–609. IEEE, 2019.

[11] C. De Boor. On calculating with b-splines. *Journal of Approximation theory*, 6(1):50–62, 1972.

[12] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using papi for hardware performance monitoring on linux systems. 08 2009.

[13] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Cache pirating: Measuring the curse of the shared cache. In *2011 International Conference on Parallel Processing*, pages 165–175. IEEE, 2011.

[14] J. H. Friedman. Multivariate adaptive regression splines. *The Annals of Statistics*, 19(1):1–67, 1991. ISSN 00905364.

[15] M. Hatzimihail, M. Psarakis, D. Gizopoulos, and A. Paschalis. A methodology for detecting performance faults in microprocessors via performance monitoring hardware. In *2007 IEEE International Test Conference*, pages 1–10, Oct 2007. doi: 10.1109/TEST.2007.4437646.

[16] C. C. Holt. Forecasting seasonals and trends by exponentially weighted moving averages. *International journal of forecasting*, 20(1):5–10, 2004.

[17] G Lowe. Sift-the scale invariant feature transform. *Int. J*, 2: 91–110, 2004.

[18] 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*, volume 710, 1999.

[19] omitted for blind review. Master's thesis.

[20] Robertwgh. Ezsift. URL https://github.com/robertwgh/ezSIFT. accessed: 2020-10-12.

[21] F. T. Schneider, M. Payer, and T. R. Gross. Online optimizations driven by hardware performance monitoring. *SIGPLAN Not.*, 42 (6):373–382, June 2007. ISSN 0362-1340.

[22] S. Seabold and J. Perktold. statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*, 2010.

[23] Andreas Sembrant, David Eklov, and Erik Hagersten. Efficient software-based online phase classification. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 104–115. IEEE, 2011.

[24] R. Sharma, A. Nori, and A. Aiken. Bias-variance tradeoffs in program analysis. volume 49, pages 127–137, 01 2014.

[25] S. Shimizu, R. Rangaswami, H. A. Duran-Limon, and M. Corona-Perez. Platform-independent modeling and prediction of application resource usage characteristics. *Journal of Systems and Software*, 82(12):2117 – 2127, 2009. ISSN 0164-1212.

[26] L. Torvalds. Perf tools. URL https://github.com/torvalds/linux/tree/master/tools/perf. accessed: 2020-07-07.

[27] S. Vogl and C. Eckert. Using hardware performance events for instruction-level monitoring on the x86 architecture. 01 2020.

[28] P. Whittle. *Hypothesis testing in time series analysis*, volume 4. Almqvist & Wiksells boktr., 1951.

[29] P. R. Winters. Forecasting sales by exponentially weighted moving averages. *Management science*, 6(3):324–342, 1960.

[30] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. Coloris: a dynamic cache partitioning system using page coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 381–392. IEEE, 2014.