Chapter 3

Performance Prediction of Multi-core Systems

In the current landscape, high-performance computing demands greater focus on multi-core processing. When designing CPUs, it is essential to account for factors such as processing speed, cache bandwidth, and minimal memory requirements. Data mining tools can significantly aid in selecting the optimal combination of CPU components. In this context, data mining has gained substantial interest in parallel computing to boost multi-core performance. This chapter[1] outlines a parallel strategy akin to traditional parallel programming paradigms to enhance multi-core efficiency using a data mining approach. We explore the Expectation Maximization (EM) method with Gaussian distributions and examine its parallel execution's impact on selected multicore clusters identified through data mining. Additionally, we evaluated our findings in a virtual environment encompassing 32 different CPU families, achieving a speedup of approximately 1.02x. This study also considers data clusters, cache mapping techniques, and ranks various MPI programming techniques.

[1] The work is published by Navin Mani Upadhyay, Ravi Shankar Singh and Shir Prakash Dwivedi, in Displays, titled "Prediction of multicore cpu performance through parallel data mining on public datasets", Volume 71, Pages 102-112, in 2022. [SCI]

3.1 Introduction

The problem addressed in this research pertains to the under utilization of parallel processing capabilities in commonly available desktop devices, specifically within multi-core systems. The core challenges include ensuring symmetry among cores and achieving uniform data distribution, both of which are essential for optimal performance.

Parallelization is crucial in the execution of transactions within a cluster computing environment. Given the time constraints typical of these environments, effective parallelization becomes a complex task. This chapter emphasizes the formulation of reliability in cluster computing through a deadline-based parallel algorithm designed to reduce time complexity in multi-core clustering. This algorithm maintains robustness even in the event of a single node failure.

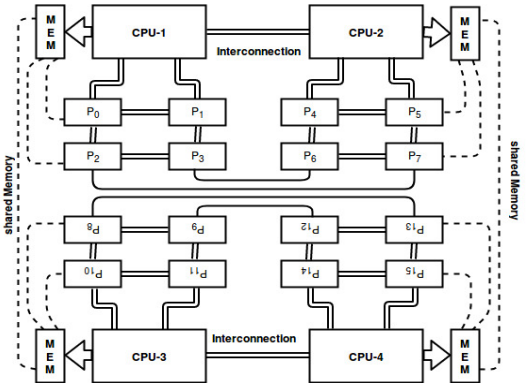

Figure 3.1: Connection-topology of used machines.

Considering a multi-core system $\boldsymbol{P} =< \boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{p_3}, \boldsymbol{p_4}, ..., \boldsymbol{p_n} >$ where each processor will have $\boldsymbol{n}$ identical cores with $\boldsymbol{e_f}$ as a core-efficiency connected in the given topology shown in Figure 3.1. Suppose a Workload of size $\boldsymbol{N}$, created by $\boldsymbol{X}$ number of problems have executed over $\boldsymbol{P}$. Assuming that the processing speed of a multi-core system is $S(P)$. Then, the processing speed is calculated as in Eq. (3.1):

$$No.\ of\ Cores = No.\ of\ Processors \times Cores\ per\ processor$$

$$\begin{aligned} S(P) &= Number\ of\ Cores\ \times\ Individual\ Core\ Processing\ Speed\ \times\ Core\ Efficiency \\ &= Number\ of\ Cores\ \times Speed(Core) \times Avg.(e_f) \quad \forall i = 1 \text{ to } n \end{aligned}$$

Eq. (3.1)

$$C(P) = \begin{cases} C(p_0) + \sum_{p_{i=1}}^{n} C(p_i), & \text{if } p_0 < p_{\text{end}} \\ 0, & \text{otherwise} \end{cases}$$

Where, $\boldsymbol{C(P)}$ be the best cluster.

A workload consisting of $\boldsymbol{X}$ problems, with a total size of $\boldsymbol{N}$, is allocated across $\boldsymbol{P_n}$ processors to minimize computation time through parallel execution. The input parameters include the number of processors $(\boldsymbol{P_n})$, workload size $(\boldsymbol{N})$, and processing speed $(\boldsymbol{S(P)})$, denoted collectively as $(\boldsymbol{P_n, N, S(P)})$. The output is the optimal cluster configuration of the multi-core system, represented as $\boldsymbol{P = C(P)}$. Using a non-linear programming model, the formula is defined as:

$$\begin{gathered} \text{minimize } \mathbf{max}_{i=1}^{X} \tfrac{x_i}{S(P)} \\ size(x_1) + size(x_2) + ... + size(x_n) = N \\ size(x_i) \leq N \qquad i = 1, ..., X \\ size(x_i) > 0 \qquad i = 1, ..., X \\ \forall 1 \leq n \leq P \\ \text{where } p, n, x_i \in Z > 0 \text{ and } s(x) \in R > 0 \\ \text{i.e. minimize } \sum_{i=1}^{n} \Omega(x_i) \end{gathered}$$

This equivalent linear program of the above formula can be re-written as:

$$\begin{gathered} \text{minimize f} \geq \tfrac{x_i}{s(x_i)} \; \forall 1 \leq i \leq n \\ x_i \leq n \qquad i = 1, ..., n \\ x_i \geq 0 \qquad i = 1, ..., n \\ \text{where } p, n, x_i \in Z > 0 \text{ and } s(x) \in R > 0 \\ \text{i.e. minimize } \sum_{i=1}^{n} P_d(x_i) \times \tfrac{x_i}{s(x_i)} \end{gathered}$$

The mathematical models mentioned above for selecting the optimal CPU for parallel processing are designed for specific problem sizes. For each problem size, the average

speedup and efficiency can be evaluated through direct measurement or prediction. Considering time complexity, the concurrent runtime differentiates between scientific and general-purpose applications, as expressed in Eq. (3.2).

$$C(S(x)) = -P\sum_{i=1}^{N} s(x_i)\ln T(x_i) \qquad \text{Eq. (3.2)}$$

Where, $T(x_i) = \sum_{i=1}^{N} g(i)\exp[\frac{-0.5(S(x)-(x))^2}{(P(i))}]$ and $g(i)$ be the time for any decentralized system.

3.2 Proposed Framework

This section outlines the design methodology for selecting multi-cores to create virtual clusters. While many cloud providers, such as Google, Microsoft, IBM, and Amazon, offer highly configured systems, the associated high costs and environmental dependencies pose challenges for typical users [29]. This research leverages conventional PCs, which are more accessible to everyday users. Figure 3.2 presents an overview of the methodology for designing a virtual cluster. In this study, a collection of 131 CPUs and their families were utilized.

Phase-1 outlines a method for data collection (*CPUs available from the years 2014-2018*) using reliable web resources[2] to select the best CPU components. The CPU data encompasses parameters such as machine cycle time in nanoseconds (MYCT), minimum main memory in kilobytes (MMIN), maximum main memory in kilobytes (MMAX), cache memory in kilobytes (CACH), minimum channels in units (CHMIN), and maximum channels in groups (CHMAX). Utilizing these six features, the optimal number of clusters was determined. Traditional partitioning and model-based algorithms necessitate specifying the number of clusters, which is not ideal. An EM clustering algorithm was employed to ascertain the optimal number of clusters by minimizing the mean absolute error (MAE). Subsequently, the EM algorithm was utilized to derive six data clusters, with their mean and standard deviation presented in Table 3.1. This approach is traditional, as

[2] https://www.techpowerup.com/cpudb/

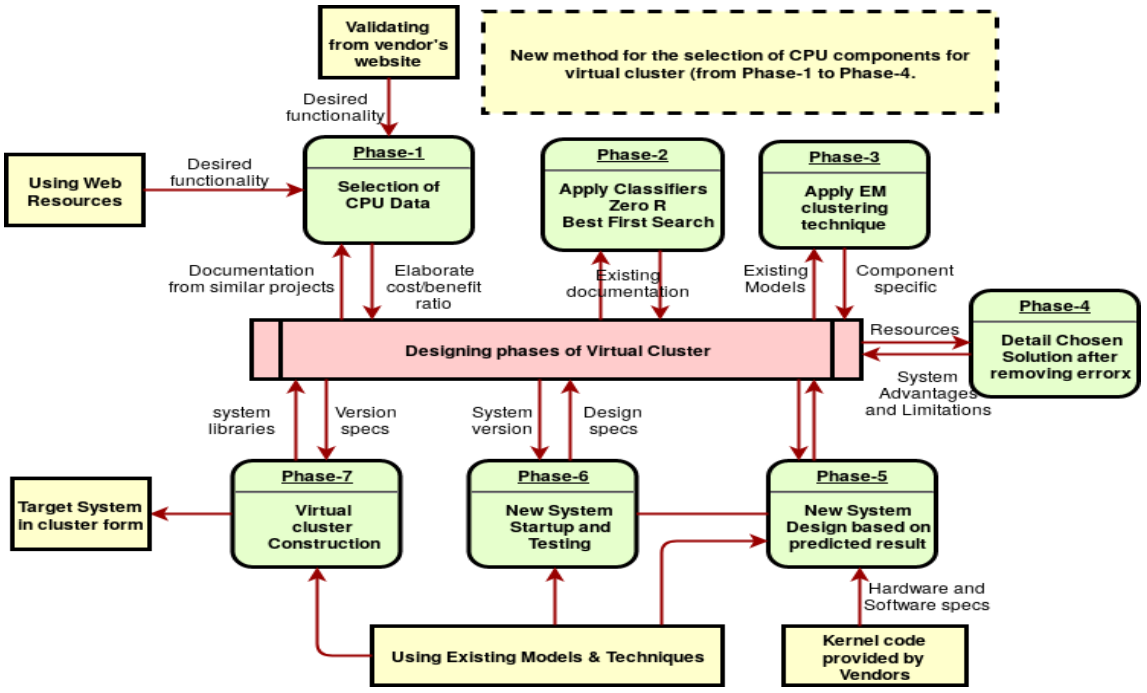

Figure 3.2: Working methodology for designing of virtual cluster.

no single source comprehensively contains information about a specific multi-core CPU. Hence, a meticulous manual search was conducted for each family and code-named computer to gather the pertinent CPU information.

Phase-2 will commence following the validation of the collected data from the vendor's website. To classify and identify the optimal attribute among all CPU components, we applied two algorithms: ZeroR and Best First Search (BFS). The ZeroR method predicted a class value of $\approx \mathbf{41.09}$, with a correlation coefficient of $-\mathbf{0.24}$, a mean absolute error (MAE) of $\approx \mathbf{10.69}$, and a root mean squared error (RMSE) of $\approx \mathbf{15.52}$. These results indicate that the classification is accurate and favorable as compared with the work done by Wang, Yu and Lee [94]. The comparison is as follows:

- *Mean Absolute Error (MAE)*: A lower MAE obtained by Wang, Yu and Lee [94] indicates better performance. The prediction with an MAE of 0.25 is significantly better than ZeroR's MAE of 10.69.
- *Root Mean Squared Error (RMSE)*: RMSE is also an important metric. While not directly compared here, a lower RMSE would typically indicate better performance.

In our case the ZeroR's RMSE is 15.52, the other method is likely to have a much lower RMSE if its MAE is 0.25.

Following this, the BFS algorithm was employed to identify the most suitable attributes for constructing the virtual cluster, with the identified attribute serving as the primary CPU component. The merit of the best subset identified by BFS is $\approx \mathbf{0.87}$, with the selected attribute being the minimum main memory (MMIN), aligning perfectly with the objectives of this research and proving beneficial for further work.

In Phase-3, the Expectation-Maximization (EM) clustering technique is utilized to construct a decision window. This phase builds on the cache partitioning technique employed earlier to manage memory congestion. By leveraging different attributes identified in Phase-2, the EM algorithm effectively categorizes data, optimizing cache usage and enhancing system performance. The EM algorithm's output includes crucial statistical measures such as mean and standard deviation, as presented in Table 3.1. These metrics provide insights into data distribution and help refine cache partitioning strategies, ensuring more efficient memory access and reduced congestion. This interconnected approach through cache partitioning significantly improves overall system efficiency and reliability.

Table 3.1: Result based on the mean and standard deviation of EM algorithm based on six data clusters.

Attributes		Clusters					
		Cluster-1	Cluster-2	Cluster-3	Cluster-4	Cluster-5	Cluster-6
MMIN	mean	**3.74**	**3.52**	**3.26**	**3.64**	**2.74**	**3.11**
	Std. Dev	**0.30**	**0.31**	**0.29**	**0.24**	**0.48**	**0.26**
MYCT	mean	64.00	360.00	75.78	83.98	69.19	79.08
	Std. Dev	0.41	41.57	18.60	28.00	17.48	28.03
MMAX	mean	256.01	**7680**	646.73	477.52	276.84	426.91
	Std. Dev	1.65	**886.81**	303.68	318.64	70.34	273.08
CACH	mean	**5570.91**	8192	**23538.52**	**4493.75**	**4318.87**	**7066.73**
	Std. Dev	**1852.52**	8257.44	**8637.75**	**3568.57**	**2165.84**	**6414.86**
CHMIN	mean	21.28	17.62	25.45	33.95	20.67	27.96
	Std. Dev	3.72	1.55	6.93	1.56	2.82	2.90
CHMAX	mean	36.63	27.65	77.12	37.50	32.60	33.55
	Std. Dev	1.74	3.84	4.31	1.99	3.46	2.57

In this Table 3.1, the MMIN technique demonstrates the smallest deviation, indicating its effectiveness in maintaining consistency across cache partitions. Conversely, the CACHE

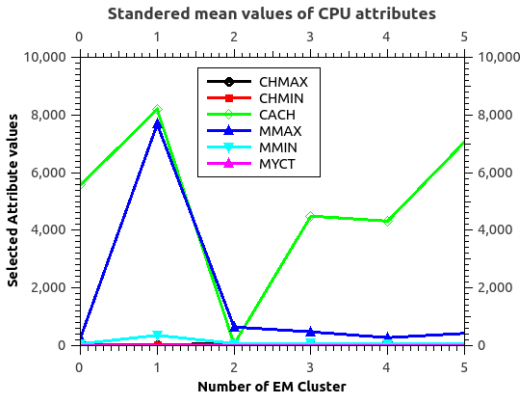

Figure 3.3: Standard mean of EM algorithm based on six data clusters.

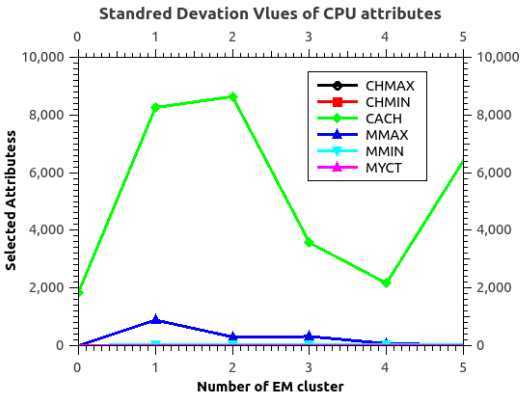

Figure 3.4: Standard deviation of EM algorithm based on six data clusters.

technique exhibits the best mean performance, suggesting its superiority in achieving optimal average performance metrics. Figure 3.3 illustrates the observations graphically, derived from the Expectation-Maximization (EM) clustering based on standard mean values. This visualization highlights the efficacy of cache partitioning techniques in balancing load and optimizing memory access. Additionally, Figure 3.4 presents the results of the standard deviation analysis for the same dataset. This figure underscores the importance of minimizing performance variability, further supporting the case for advanced cache partitioning methodologies to enhance overall system efficiency and reliability.

Phase 4 validates the accuracy of predicted results obtained in Phase 3. Following this validation, a novel virtual system is designed, and clusters are constructed based on Phase 4's outcomes. The steps from Phase 5 to Phase 7 are executed using modified kernel codes[3] provided by CPU vendors. This architecture benefits both cloud-based and cluster-based high-performance systems by leveraging a topology-aware work schedule.

Central to our approach is the use of cache partitioning techniques to enhance system performance and resource management. Cache partitioning allocates specific cache segments to different processes, reducing cache contention and improving overall system efficiency. By integrating this technique, we propose two algorithms with innovative workload scheduling policies tailored for multi-core systems. These algorithms utilize graph mapping methods that align with workload performance objectives defined by system policies [33].

[3] https://elixir.bootlin.com/linux/latest/source/arch/x86/Kconfig.cpu

Our proposed method offers several key advantages. It reduces costs by optimizing the use of existing resources and enhances system utilization through a parallel performance optimization strategy. Additionally, it ensures numerical stability, robust fault coverage, and efficient placement strategies [95]. These benefits underscore the effectiveness of cache partitioning in achieving high performance and reliability in multi-core systems.

3.2.1 Proposed EM-based Clustering Algorithm

As data-mining algorithms play a vital role in the field of information mining. Before applying the Expectation-Maximization (EM) Clustering Algorithm, some well-known data-mining algorithms are used to classify the multi-core systems based on their components. All the algorithms involved in this research have the same parameters and assumptions as described in section 3.1 in terms of $\boldsymbol{P}$ identical processors and $\boldsymbol{X}$ number of problems. Based on the results from these algorithms, the best components of a multi-core system are selected to make the virtual cluster. An EM-based clustering algorithm has been proposed to find the best cluster among all the multi-core clusters, and the likelihood is observed to be lesser than that of the Gaussian-based EM algorithm. The likelihood [66] can be measured by the below formula Eq. (3.3):

$$f_M(x_n) = \sum_{m=1}^{M} \varepsilon_m \cdot \varphi_m(x_n) \qquad \text{Eq. (3.3)}$$

Where M is a maximum degree in the case of the Gaussian model, and in the clustering case, it is the participation of the maximum number of processors to solve a particular problem ($\boldsymbol{X}$). ε_m is the EM parameter and φ_m is the Gaussian Probability Density Function.

The above formula Eq. (3.3) is remarkable because, in this research, these two functions guide the parallelization strategy and optimization efforts for selecting the best multi-core system cluster. The focus is on achieving efficient parallelization, scalability, and load balancing while minimizing communication overhead for optimal performance [*see section 3.1*]. Further, the mean μ_m and the variance $\boldsymbol{\sigma}_m^2$ play a crucial role in defining the Gaussian probability density functions (PDFs) for the E-step and M-step of the algorithm.

As the Gaussian model (GM) is a part of the basic EM algorithm, the GM parameter is defined by Eq. (3.4).

$$\theta = [\varepsilon_1, ..., \varepsilon_M; \mu_1, ..., \mu_M; \sigma_{21}, ..., \sigma_{2M}]T \quad \text{Eq. (3.4)}$$

If there is no indication of receiving the current data in the selected multi-core system, based on the applied workload. From the above formula Eq. (3.4), if the sample size of X_n in terms of ϵ_i acquire M_i component, then the execution of dataset becomes complete. One can easily find the Speed of selected clusters $C(P)$ by applying the mathematical formula Eq. (3.2).

In this case, Gaussian EM (GEM) parameter [8] will be determined by the below formula Eq. (3.5).

$$Y(k) = \sum_{m=1}^{M} W_m \varphi_m(L(k)) \quad \text{Eq. (3.5)}$$

Where,

$$\varphi_m(\lambda) = exp\left(-0.5(\lambda - \mu_m)^{\frac{2}{\sigma^2}}\right)$$

Furthermore, the maximization of the EM algorithm depends on the following formulas (Eq. (3.6), and Eq. (3.7)):

$$y(k) = \sum_{t=1}^{T} T(t) Pr[T(t) \in C(k)] / \sum_{t=1}^{T} Pr[T(t) \in C(k)] \quad \text{Eq. (3.6)}$$

$$Pr[X(x) \in C(k)] = \exp[\frac{-0.5(\mathrm{S(x)} - (\mathrm{x}))^2}{(\mathrm{P(i)})}] \quad \text{Eq. (3.7)}$$

In our study, we employ the Expectation-Maximization (EM) algorithm for data clustering, leveraging its ability to maximize the likelihood of determining the statistical parameters of the underlying sub-populations within the dataset. A key advantage of EM is its flexibility in handling clusters that are not restricted to spherical shapes, unlike some other clustering

techniques. Additionally, EM allows for the specification of different covariance matrices for each cluster, which enhances our ability to control and refine cluster formation. This capability is crucial for accurately detecting and characterizing sub-populations with distinct features within the data.

To integrate this approach with our cache partitioning technique, we utilize cache partitioning to efficiently manage memory resources during the EM clustering process. By allocating separate cache partitions to different sub-populations, we can reduce cache contention and improve the algorithm's performance. This ensures that each sub-population's data is processed more efficiently, leading to more precise and reliable clustering results. The benefits of this integrated approach are evident in the improved clarity and distinction of the clusters formed, as reflected in Table 3.1. The combination of EM clustering with cache partitioning not only optimizes memory usage but also enhances the detection of varied sub-populations within the dataset.

Algorithm 1 EM using Gaussian

Inputs: $Block\_threat\_size : bts_x, bts_y$
Input matrix: $In\_MatA[m][k], B[k][n]$
Data_Size: $D\_A[R][k], D\_B[k][T]$
Prerequisite: *Load matrix A and B into memory m as* $s_{mA}[mk],\ s_{kT}[kn]$
Conditional distribution of C_p under $P_{y|c}(y|c,\theta)$
Initial Values θ^0
Function: $EM(P_{y,c}(y,c,\theta), P_{y|c}(y|c,\theta), \theta^0)$
Sync_m=0
do:
  for $m_i = 0$ *to* mk **do**
    $q_c^{(t)} \leftarrow P_{c|y}(c|y;\theta^{(t-1)})$
    $shared\_m\_A[mk]$ into $D\_B[o,1,2,...,n]$
    $Result[0,1,2,...Rx[0,1,2,...,R]]$
    M-State:
    $y(k) = \sum_{t=1}^{T} T(t) Pr[T(t) \in C(k)] / \sum_{t=1}^{T} Pr[T(t) \in C(k)]$
    Calculate: $Pr[T(t) \in C(k)]$
  end

return $t, y(t), P_{(y,c)}$

3.2.2 Applying EM-based Clustering Algorithm

In the Algorithm 1, the input parameters are the block thread size denoted by $\boldsymbol{Block\_threat\_size : bts_x, bts_y}$. Later in M-State, it will distribute the data efficiently among cores to ensure load balancing for better parallelization. Further, for measuring the performance concerning scalability, two input matrices are selected from the dataset (Attributes of CPU dataset): $\boldsymbol{A}$ with dimensions $\boldsymbol{m \times k}$ and $\boldsymbol{B}$ with dimensions $\boldsymbol{k \times n}$ where each pair of multiplication are passed with the previously selected bock thread sizes. Further, the data sizes are specified for matrices called $\boldsymbol{D_A}$ and $\boldsymbol{D_B}$ with minor changes in dimensions $\boldsymbol{R \times k}$ and $\boldsymbol{k \times T}$ respectively. After that, based on the above parameters, the prerequisite for the algorithm has turned to reduce the communication overhead with a conditional distribution of $\boldsymbol{C_P}$ under $\boldsymbol{P_{y|c}(y|c,\theta)}$ to achieve better results.

This phase of the algorithm includes the following strategies as shown in Figure 3.1.

- ***Shared Memory Utilization:*** It leverages shared memory for data that needs to be accessed or modified by multiple cores. This reduces the need for explicit communication protocols. So, Ensure that shared memory is efficiently used for storing and accessing common data structures.
- ***Data Partitioning:*** It divides the dataset (called batch size) into partitions, assigning each section to a specific core. Minimize cross-core dependencies to reduce the need for frequent data exchange. Line number 8 of Algorithm 1 minimizes the need for constant synchronization $\boldsymbol{(Sync_m = 0)}$ or data exchange for coordinated parallel processing.
- ***Local Computation:*** The proposed framework maximizes the local computation within each core, which in turn minimizes the necessity for inter-core communication (ICC). This is particularly important for every parallel algorithm, where each core can work independently on a subset of the dataset.

Further, the algorithm sets the initial values for the parameter vector $\boldsymbol{\theta}$ as $\boldsymbol{\theta^0}$ and the EM function takes these parameters as input with a calculated conditional probabilities $\boldsymbol{P_{y,c}(y,c,\theta), P_{y|c}(y|c,\theta)}$. By doing so, the accessing of the data can be optimized, and

the frequency of data retrieval from shared resources will improve. This is the most critical part of parallelization, which includes intelligent caching and pre-fetching strategies.

As the proposed framework shown in Figure 3.1 uses an asynchronous communication to optimize the proposed algorithm further, the probabilities are updated, and parallel computations are performed on shared data. The M-State analysis involves utilizing CPU dataset characteristics for efficient calculations. This will reduce the idle time and improve the core efficiency (e_f).

3.3 Result and Discussion

3.3.1 Prediction Accuracy: Case Study

Several benchmarks and applications reflecting common daily scenarios are thoroughly investigated. These scenarios consist of tasks that mimic typical system usage by individuals. Table 3.2 lists the various scenarios examined in our study. Given that Geekbench[4] versions **2.0** and later are cross-platform, they are utilized in separate case studies (Case-3). This section also includes predictions for the accuracy of mean absolute error measurements. Three case studies are presented, showing that Deep Neural Networks exhibit greater accuracy compared to Logistic Regression (LR). Subsections 3.3.1.1 to 3.3.1.3 provide detailed descriptions of these cases.

To effectively manage and optimize the performance of these scenarios, cache partitioning techniques play a crucial role. By dynamically allocating cache resources based on the workload's demands, cache partitioning ensures efficient utilization and minimizes contention among applications. This leads to improved performance and reduced latency.

By applying cache partitioning, we can allocate separate cache partitions to different categories of workloads such as General, Web Browsing, Multiscreen, and Benchmarks, thereby enhancing overall system performance and responsiveness. This method not only improves the accuracy of performance predictions but also optimizes resource utilization, as demonstrated in our experiments and detailed in the subsequent sections.

[4] https://browser.geekbench.com/processor-benchmarks

Table 3.2: Workloads used in our experiments.

Category	Workloads
General	Clock, Calendar, Email, Messages, Application Download, Application Update, Skype, Google Hangout
Web Browsing	ESPN, News Sites, Wikipedia, Google Images, Facebook
Multiscreen	Skype + Web Browsing, Skype + YouTube, Skype + Gallery, Skype + Facebook, FaceBook + YouTube, Facebook + Gallery
Benchmarks	Geekbench

3.3.1.1 Case 1: Prediction for new SKUs

This case study illustrates the prediction and selection of new SKUs using a cache partitioning technique to enhance performance analysis. Figure 3.5 and Figure 3.6 present a comparative analysis between SPEC and Geekbench, employing Deep Neural Networks (DNN) and Logistic Regression (LR) models. The study involves 352 SKUs for SPEC and 119 for Geekbench. The mean absolute error (MAE) derived from DNN and LR is depicted on the y-axis, ranging from $\mathbf{0.00}$ to $\mathbf{0.30}$ for SPEC and from $\mathbf{0.0}$ to $\mathbf{0.6}$ for Geekbench. We utilized 25 random test sets to determine the average MAE, with the bar heights in the graph representing these averages.

The results indicate that Logistic Regression exhibits higher sensitivity for both SPEC and Geekbench, consistently showing a greater mean absolute error in the test set selections. This suggests a non-linear relationship between features and performance, making DNN a more reliable choice for prediction. Specifically, only five SKUs and one desktop were selected for this analysis, with the remaining being servers. The average MAE displayed in the graph indicates a $\mathbf{19.73}\%$ error rate. For SPEC, DNN achieved an error rate of $\mathbf{4.2}\%$ compared to LR's $\mathbf{19.9}\%$, while for Geekbench, DNN and LR reported $\mathbf{10.8}\%$ and $\mathbf{19.9}\%$ respectively. Hence, DNN proves to be more effective for larger datasets, facilitated by the cache partitioning technique which optimizes data handling and improves model accuracy.

Incorporating cache partitioning within this framework allows for better management of memory resources, reducing latency and congestion, and ultimately leading to more precise predictions. This technique ensures that critical data paths are prioritized, thereby enhancing the overall efficiency of DNN models in handling large-scale performance

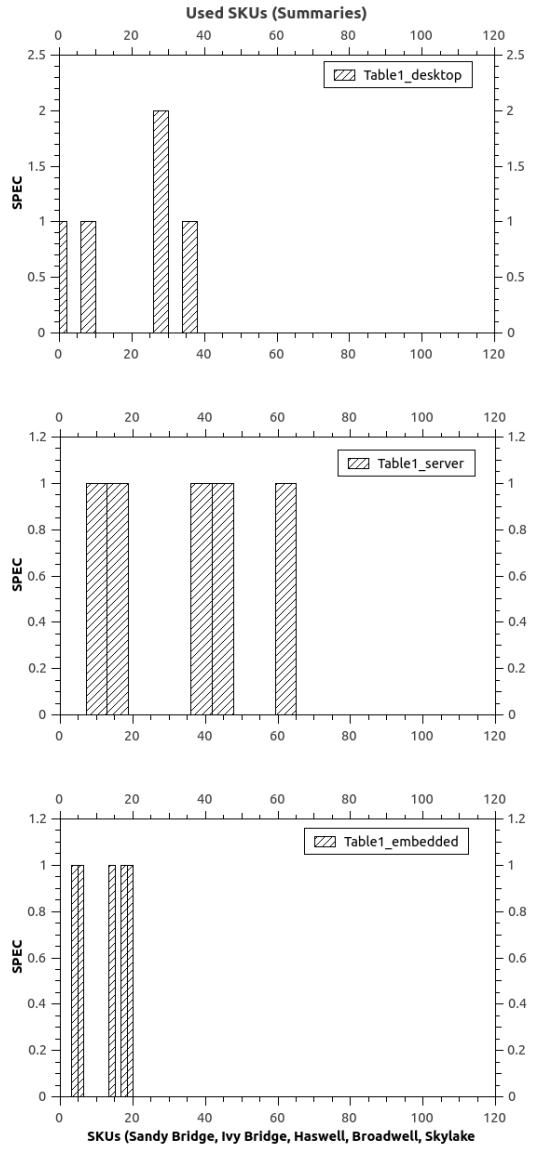

Figure 3.5: Stock Keeping Units (SKUs) breakdown in the Standard Performance Evaluation Corporation (SPECs) dataset

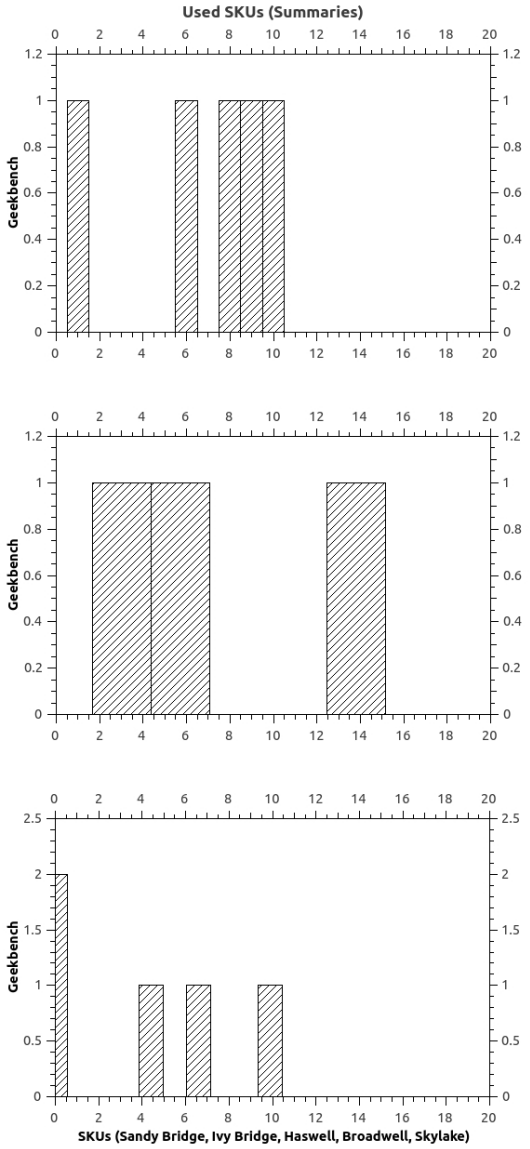

Figure 3.6: Stock Keeping Units (SKUs) breakdown in Geekbenchs datasets.

evaluation tasks.

Based on our experimental study, the mean absolute error (MAE) for Geekbench is higher than SPEC. The accuracy of the simulation-based prediction is about **6**%, which is higher than the method proposed by Liu et al. [96], and Mirghafori et al. [97].

3.3.1.2 Case 2: Self-prediction

This case study illustrates the prediction of new workloads on selected SKUs from case 1, focusing on the application of cache partitioning techniques to enhance performance. This section presents the similarity between predicted workloads and actual outcomes, evaluating the prediction accuracy of the Deep Neural Network (DNN) model employed. By implementing cache partitioning, we aim to optimize memory access patterns and reduce latency, thus improving overall system efficiency. The study also quantifies workload outliers and correlates them with the mean absolute error (MAE) to assess the reliability of the predictions. The integration of cache partitioning not only aids in precise workload prediction but also ensures balanced cache usage, further validating the robustness of the DNN approach.

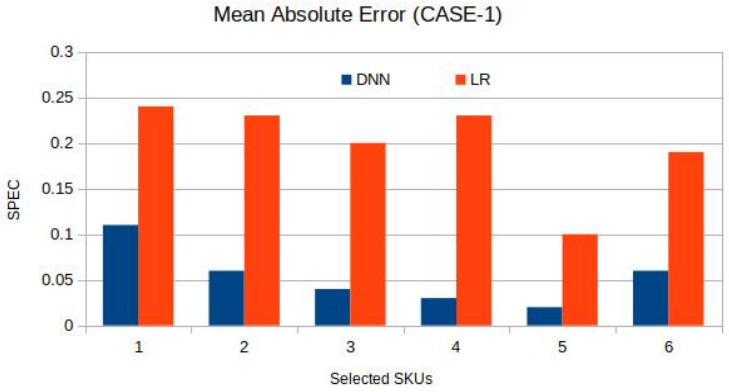

Figure 3.7: The Performance comparison of DNN and LR algorithms in terms of mean absolute error (MAE) for selected SPEC.

Prediction of Results: The results of the Standard Performance Evaluation Corporation (SPEC) benchmarks are illustrated in Figure 3.7 and Figure 3.8, where workloads are organized based on mean absolute error (MAE). To enhance prediction robustness, 50 SKUs were incorporated into the analysis. By examining Figure 3.7 and Figure 3.8, we calculated the standard deviations across all testing sets, revealing a notable difference ranging from $\mathbf{2.26}$ to $\mathbf{3.44}$ in the MYCT attribute. The significance of the standard deviation exceeded $\mathbf{18\%}$ for the CACH attribute.

Further details are provided in Figure 3.9 and Figure 3.10, which present the mean absolute error (MAE) and total execution time (in milliseconds) for self-prediction following the

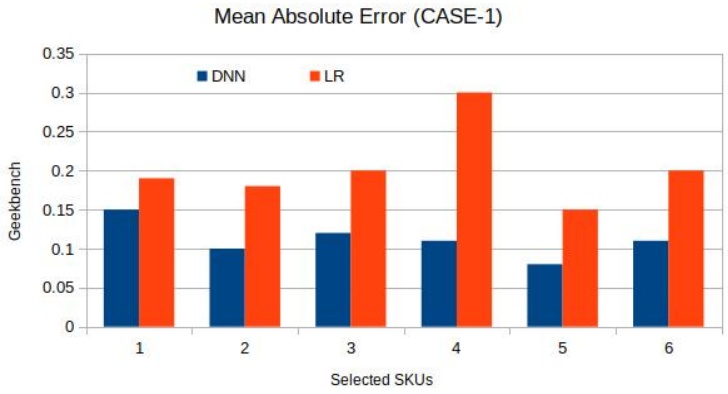

Figure 3.8: The Performance comparison of DNN and LR algorithms in terms of mean absolute error (MAE) for selected Geekbench

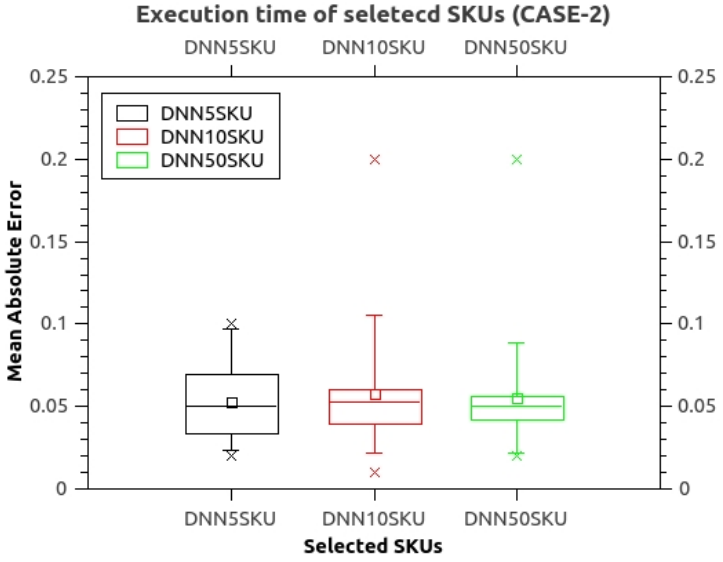

Figure 3.9: The Mean Absolute Error of selected SKU based on DNN for Geekbench (Self prediction after adding 50 SKUs).

addition of 50 SKUs. The MAE sorts the workloads, with outliers positioned on the right, and average MAEs displayed as bars on the right side. Although the line representing the addition of 5 SKUs is omitted, the average bar for this case (case-1) is included. Across all testing sets, standard deviations are generally greater than $\mathbf{18\%}$.

Moreover, the MAE values for 5, 10, and 50 SKUs are $\mathbf{3.8\%}$, $\mathbf{3.3\%}$, and $\mathbf{3.1\%}$, respectively. A noticeable improvement in MAE is observed when increasing from 10 to 50 SKUs, though adding more SKUs introduces greater complexity compared to increasing from 1 to

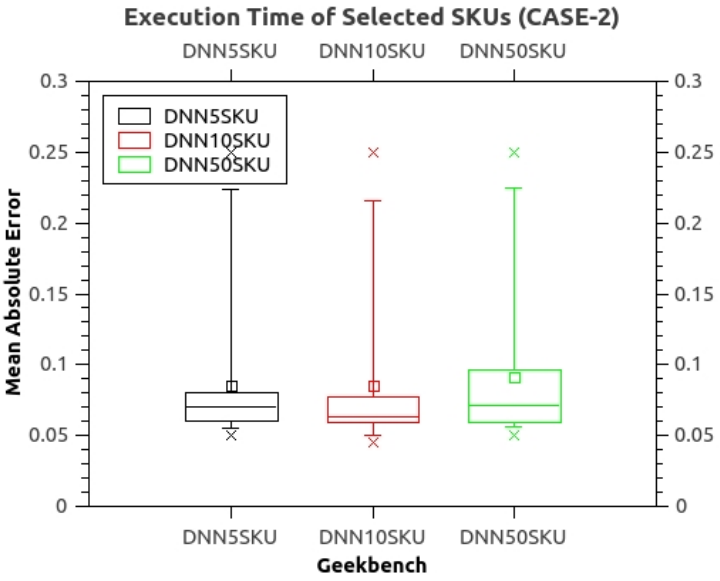

Figure 3.10: The Mean Absolute Error of selected SKU based on DNN for SKUs (Self prediction after adding 50 SKUs for case-2).

5 SKUs. Consequently, the observed improvement in MAE for predicting new workloads justifies the addition of more SKUs.

These predictions are underpinned by a cache partitioning technique, which allocates specific cache segments to different workloads, thereby reducing interference and improving prediction accuracy. This technique not only optimizes cache utilization but also enhances the overall performance and reliability of the system by minimizing cache contention among competing processes.

Benchmarks and Outliers: We have applied an EM-based algorithm to study the outliers of the workloads. In this algorithm, the first task is splitting the workloads into clusters (cluster 1 and cluster 2). The Figure 3.11 considers a scaling on 639 configurations of 352 SKUs. The green stars are centroids of both of the clusters identified by k-means. From this Figure 3.11 it is observed that the SPEC FP workloads are more spread out than the SPEC Int. Two different types of SPECs have been used to experiment. The first type is 462.libquantum shown by the red dots at the bottom left in the figure, whereas the second type is 481.wrf which also forms a cluster with SPEC FP benchmarks and extracts the outliers near the top.

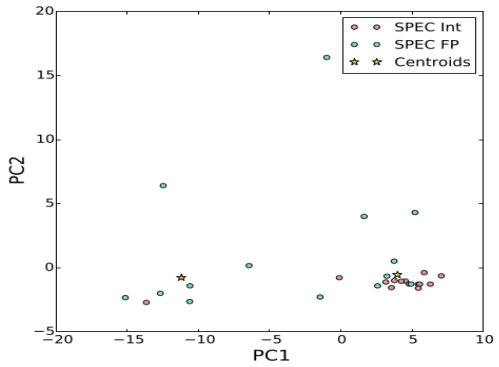

Figure 3.11: Principal Component Analysis (PCA) of Standard Performance Evaluation Corporation (SPECs) workload performance based on SPEC FP and SPEC Int.

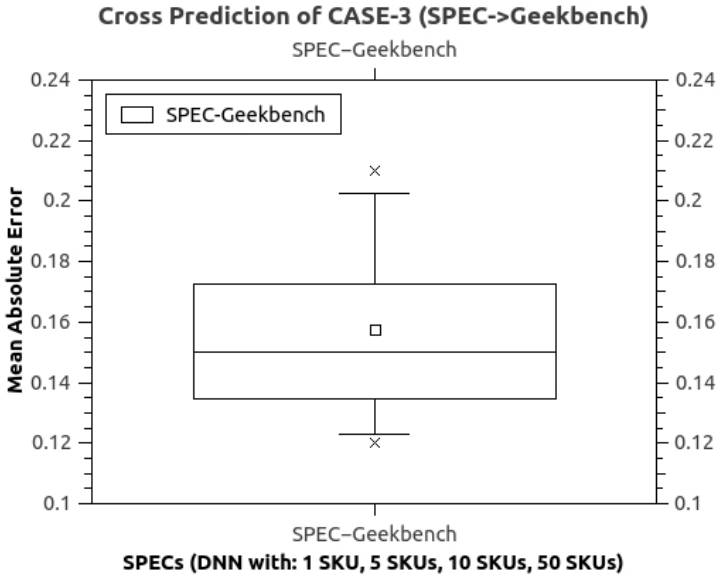

Figure 3.12: Cross-prediction of SPEC workloads, predicted with the Geekbench dataset

Suppose the plotted centroids are represented as $\boldsymbol{C_1}$ and $\boldsymbol{C_2}$ and the distance between both of the clusters are calculated by the below formula Eq. (3.8).

$$distance = I(x \in cluster\ 1)||x - C_1||_2 + I(x \in cluster\ 2)||x - C_2||_2 \qquad \text{Eq. (3.8)}$$

Where x is the location of the workload, and I contains two arguments, true and false, where for true, the integer representation is $\mathbf{1}$ and for false, the integer representation is $\mathbf{0}$.

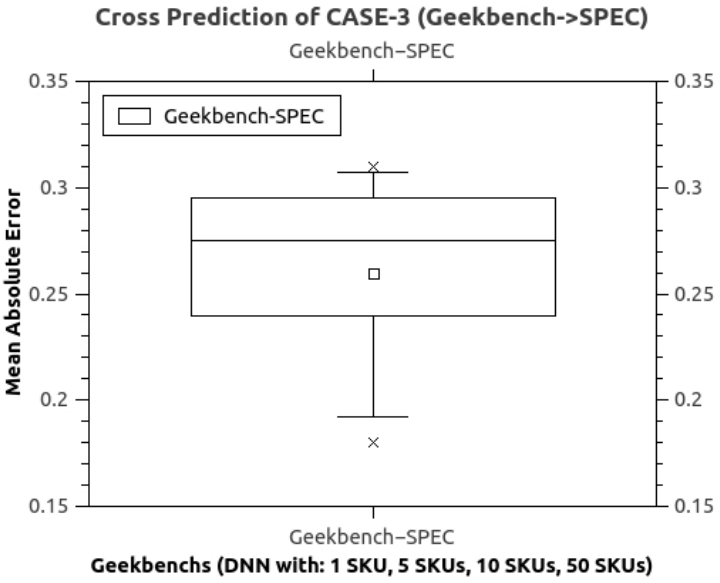

Figure 3.13: Cross-prediction of Geek-bench workloads, predicted with the SPEC dataset.

3.3.1.3 Case 3: Cross-prediction

To compare cross-prediction and self-prediction, the workloads for both suites are interchanged. Specifically, SPEC workloads are used to predict Geekbench performance, and Geekbench workloads are used to predict SPEC performance. The results of these comparisons are illustrated in Figure 3.12 and Figure 3.13. These figures reveal that cross-prediction has a significantly higher mean absolute error as the number of added SKUs increases (from 5 to 50). The average mean absolute error is $\mathbf{13.4}\%$ for cross-prediction, while for self-prediction, it is only $\mathbf{3.1}\%$.

The consistently higher error in cross-prediction, as observed in Figure 3.12 and Figure 3.13, can be attributed to the fundamental differences between SPEC and Geekbench workloads. This disparity underscores the importance of effective cache partitioning techniques in multi-core systems. By employing cache partitioning, we can better manage and isolate different types of workloads, thereby reducing interference and improving the accuracy of performance predictions. Cache partitioning not only helps in balancing the cache allocation but also enhances the overall system performance by minimizing cross-workload interference. This technique ensures that each core's workload is optimally managed, leading to more reliable and efficient performance outcomes.

3.3.2 Ranking of SKUs based on selected Cases

This section presents an analysis of performance prediction techniques applied to Deep Neural Networks (DNN) and Logistic Regression (LR). Specifically, we explore the impact of cache partitioning techniques on the predictive capabilities of these models. The simulated SKUs are evaluated based on their respective weights, which are then ranked according to performance metrics. By comparing these results against baseline metrics, our methodology for SKU selection demonstrates enhanced accuracy and reliability.

3.3.2.1 Case 2: Self-prediction

Our approach demonstrates superior performance in selecting new SKUs using benchmarks such as SPEC and Geekbench. The corresponding outcomes are illustrated in Figure 3.14 and Figure 3.15. To determine optimal configurations, we examined $\mathbf{100}$ combinations of weights, transitioning between prioritizing frequency and cache size. Specifically, we combined (frequency (GHz) $\times\, \boldsymbol{w}$) with (cache (MB) $\times\, (\mathbf{1} - \boldsymbol{w})$), incrementing $\boldsymbol{w}$ from $\mathbf{0}$ to $\mathbf{1}$ in steps of $\mathbf{0.01}$. In Figure 3.14 and Figure 3.15, we normalized the ranges of frequency and cache values using a linear combination ($\mathbf{0.9} \times$ frequency $+ \mathbf{0.1} \times$ cache). This approach leverages cache partitioning techniques to ensure balanced consideration of both frequency and cache attributes in SKU selection strategies.

Through experimentation across three distinct scenarios (frequency variations, varying cache sizes, and their combined effect), we employed the top-3 accuracy methodology to evaluate performance metrics. Our findings indicate that frequency emerges as the most influential metric. As depicted in Figures 3.14 and Figure 3.15, Geekbench frequency measurements exhibit comparable accuracy with respect to the DNN model. Analysis of the box plots reveals Geekbench's heightened sensitivity to frequency compared to SPEC benchmarks. Therefore, for optimal SPEC selection, it is advisable to consider combinations of both frequency and cache sizes, underscoring the importance of cache partitioning techniques in enhancing system performance.

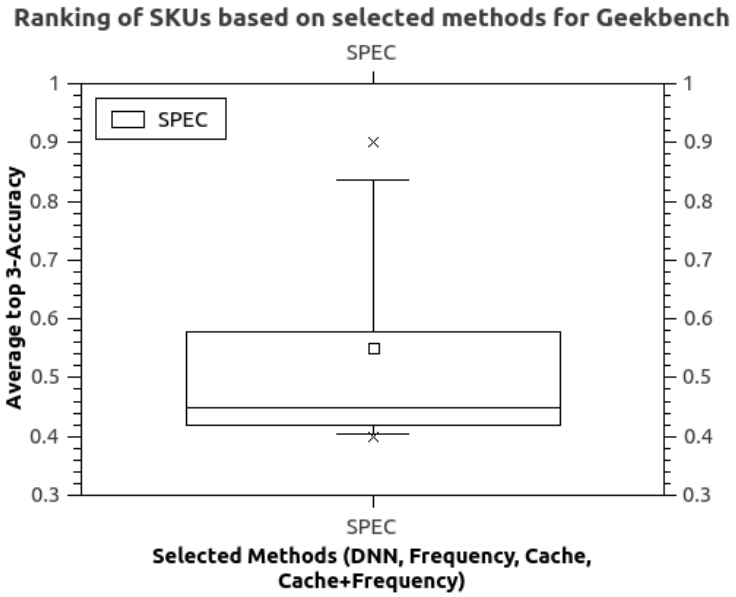

Figure 3.14: SKUs ranking results of self prediction (case-1) for Standard Performance Evaluation Corporation (SPEC)

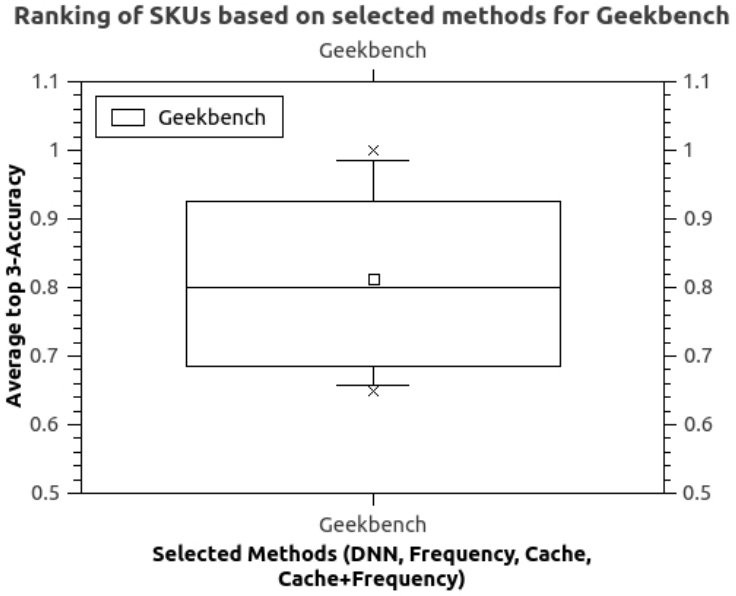

Figure 3.15: SKUs ranking results of self prediction (case-1) for Geekbench.

3.3.2.2 Case 3: Cross-prediction

This section conducts a comparative analysis of top-3 accuracies across three scenarios involving variations in frequency, cache sizes, and their combination. Specifically, we

have chosen two distinct benchmarks: 462.libquantum and 481.wrf. Figure 3.16, and Figure 3.17 illustrate the comparison results for these benchmarks. Each scenario includes ten additional SKUs to forecast performance under new workloads, emphasizing the impact of cache partitioning techniques on predictive accuracy and performance stability.

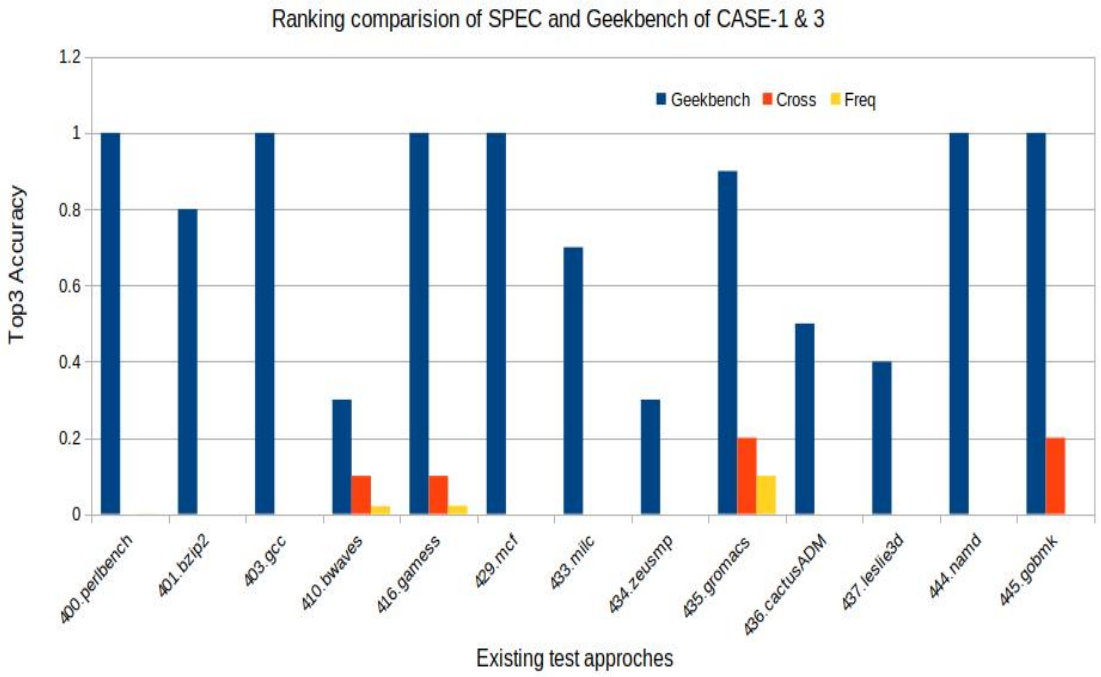

Figure 3.16: Ranking comparison based on existing test approaches over Standard Performance Evaluation Corporation (SPEC) for self (case-1) and cross prediction (case-3.)

From Figure 3.16 and Figure 3.17, it is evident that self-prediction yields higher accuracy compared to cross-prediction. This enhanced prediction accuracy simplifies rank prediction significantly. The results indicate that both self-prediction and cross-prediction outperform frequency-based ranking, highlighting that frequency alone is suboptimal for SKU selection.

The analysis of Figure 3.16 and Figure 3.17 further reveals that many SPECs and Geekbenchs lack cross-platform support, particularly in multi-core environments reliant solely on frequency (case 1), resulting in lower participation. Therefore, the adoption of cache partitioning techniques becomes imperative for optimizing multi-core system selections within clusters.

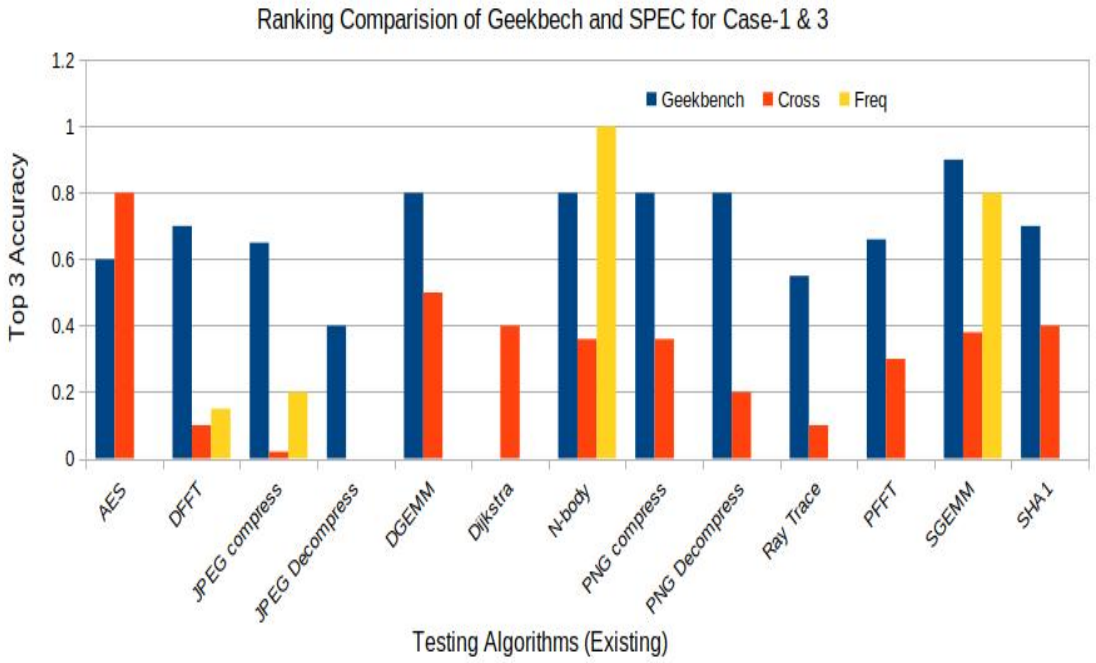

Figure 3.17: Ranking comparison based on existing test algorithms over Geekbenchs for self (case-1) and cross prediction (case-3.)

3.3.3 Similarity measurement and Validation

This section emphasizes the similarity measures between all the benchmarks used in our proposed method. The measurement is based on micro-architecture and performance matrices.

3.3.3.1 Micro-architecture

The micro-architecture of benchmarks is independent of its performance matrices, which allows for a comparison between two different programs. We have covered a wide range of program characteristics to measure the performance of micro-architectures of used benchmarks. The performance matrices are as follows:

- ***Mix-Instructions based:*** We have measured a relative performance based on frequencies of different operations like DMA and instruction streams. The

mix-instruction-based micro-architecture is useful for calculating and understanding the memory access bound. The memory access bound is categorized into two categories: computation and memory access. In our case, the microprocessor executes 1024 instructions per second. From these, 613 instructions are computation-bound (arithmetic operations), and 411 are memory access-bound to load or on-store operations. Therefore, the system would be categorized as memory access-bound because a significant portion of its functions is constrained by memory access.

- ***Memory Block Size:*** The Memory Block size is responsible for storing the part of code having an entry and exit instruction. This will also help in instruction-level parallelism. The main work of this memory block size is to measure the essential block size between the instruction stream and their consecutive units.

- ***Branch Dependency Distance:*** The branch dependency is defined as the number of derived instructions between different branches, i.e. forward branch, backward branch, upward branch and downward branch. Since we have used instruction-level parallelism, the branch dependency distances are measured by their inherited properties. Based on their distances, the resultant dependencies are categorized into seven different classes, i.e. 1, 2, 4, 8, 16, 32 and above 32. The maximum distance shows a more significant percentage of parallelism.

- ***Data Locality:*** The Locality of data can be classified into two categories, i.e. spatial and temporal. Apart from data, another locality is available for benchmarks based on instruction level locality. The instruction level locality is also classified into two classes, as mentioned above (spatial and temporal). Both of these classes are computationally intensive towards memory. The computation includes performance metrics and average dependency distancing measurements. The cache block size is important in calculating the average dependency distance. It uses four different sizes of cache block windows of $\mathbf{16}$, $\mathbf{64}$, $\mathbf{256}$ and $\mathbf{4096}$ sizes. A user may select anyone from these different window sizes according to their workload.

3.3.3.2 Statistical Analysis of Benchmarks

We utilized a set of 13 matrices, as illustrated in Figure 3.16 and Figure 3.17, to conduct an extensive analysis of SPEC and Geekbench benchmarks. This dataset encompasses 34 variables, each representing distinct micro-architectural characteristics. Our analysis focused on 14 cases; however, we narrowed our investigation to three critical cases: baseline (frequency), self-prediction, and cross-prediction. Due to the inherent complexity of the data, deriving meaningful insights directly from it proved challenging. To facilitate our analysis, we employed principal component analysis (PCA), a multivariate analysis method. PCA enabled us to assess and compare program distributions across each benchmark. Further details on SPEC CPU benchmarks can be found in Table 3.3.

Table 3.3: Standard list of used SPEC CPUs

SPEC	Program	No. of Instructions (billion)	Kernel Name	SPEC	Program	No. of Instructions (billion)	Kernel Name
	espresso	0.5	FP		espresso	0.5	FP
	li	7	CG		li	6.8	CG
	eqntott	-	CG		eqntott	-	CG
	gcc	-	CG		gcc	0.1	CG
	spice2g6	-	CG		spice2g6	-	CG
	doduc	1.03	FP		doduc	-	FP
	fpppp	1.17	FP		fpppp	-	FP
	nasa7	6.2	FP		nasa7	1.03	FP
	tomcatv	1	LU		tomcatv	2.55	LU
	go	0.5	FP		compress	3.05	FP
	li	6.8	CG		sc	3.53	FP
	m88ksim	6.8	CG		mdljdp2	44	FP
	compress	6.8	CG		wave5	10.2	FP
	ijpeg	6.8	CG		hydro2d	4.69	FP
	gcc	6.8	CG		Swm256	4.72	FP
	perl	6.8	CG		alvinn	-	FP
	vortex	6.8	CG		ora	6.23	FP
	wave5	6.8	CG		ear	-	FP
	hydro2d	6.8	CG		su2cor	0.9	FP
	swim	6.8	CG		fpppp	-	FP

- ***Principal Components Analysis:*** Principal Component Analysis (PCA) is a fundamental method within classical multivariate statistical analysis, aimed at reducing dataset dimensionality. By computing principal components through linear combinations of existing variables, PCA facilitates the extraction of essential features while minimizing information loss. This technique is particularly valuable in scenarios where datasets exhibit high dimensionality, allowing for efficient data representation and interpretation. Moreover, PCA plays a crucial role in various fields, including pattern recognition, image processing, and exploratory data analysis, contributing significantly to enhancing understanding and decision-making processes in complex datasets. It also transforms the existing variables $P_1, P_2, ..., P_n$ into principal components $P_{p1}, P_{p2}, ..., P_{pn}$ by using below formula Eq. (3.9).

$$P_p i = \sum_{j=0}^{p} pos_{ij} P_j \qquad \text{Eq. (3.9)}$$

Where, $[P_{p1}] > Var[P_{p2}] > ... > Var[P_{pn}]$.

It is observed from this relationship that the $[P_{p1}]$ contains the most information and $[P_p n]$ contains the least information about the datasets. Based on this property, we can use any of the principal component variables as per our requirement. At least $\mathbf{75\%}$ of the total information from the dataset has been used in this work.

3.3.3.3 Benchmark subsets Validation

In this section, Benchmark subsets are validated as it is essential to know which subsets are representative of the Standard Performance Evaluation Corporation (SPEC), which has designed some CPUs called SPEC-CPUs benchmarks. We have used Inter-Process Communication (IPC) and layer-1 (L1)-cache memory-based Miss-ratio technique for all the benchmarks shown in Table 3.3. Finally, we have compared it with other existing related papers [98, 70, 5, 99, 71, 16].

- ***Computing IPC:*** To evaluate the average Inter-Process Communication (IPC) across diverse micro-architecture settings, we analyzed program subsets from the SPEC CPU2000 benchmarks, specifically focusing on configurations like 462.libquantum and 481.wrf. We computed IPC values for issue widths of 8 and 16 nodes. The results, illustrated in Figure 3.18, depict the IPC averages across the entire benchmark suite. This analysis builds on Huang et al.'s methodology [100], which comprehensively assesses IPC performance under varying cache partitioning strategies.

- ***Configuration for IPC:*** The configurations of selected benchmark CPUs for improved inter-process communications are shown in Table 3.4.

  The details about branch predictors and different cycle penalties can be found in [100]. From Table 3.3, we observe that each cluster has a different number of programs. Hence, the weight assigned to each representative program should depend on the number of programs that it represents.

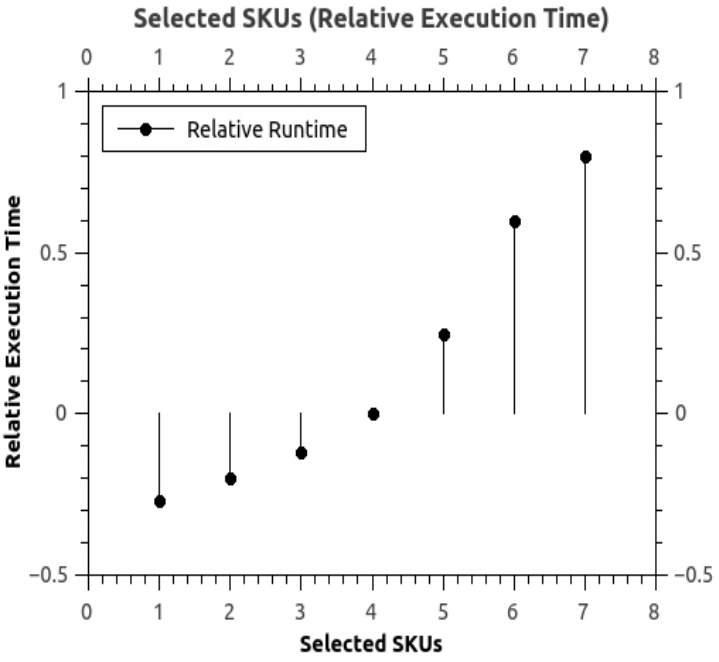

Figure 3.18: Relative execution time of selected SKUs.

Table 3.4: CPU configuration specifications

Component	8-way Machine Configuration	16-way Machine Configuration
Cache Configuration		
L1 Instruction/Data Cache	32KB 2-way set-associative	64KB 2-way set-associative
L2 Cache	1MB 4-way set-associative	2MB 8-way set-associative
Functional Units		
Instruction Arithmetic Logic Units (I-ALU)	4 units	16 units
Instruction Multiply/Divide Units (I-MUL/DIV)	2 units	8 units
Floating-Point Arithmetic Logic Units (FP-ALU)	2 units	8 units
Floating-Point Multiply/Divide Units (FP-MUL/DIV)	1 unit	4 units

3.3.4 Statistical Test and Comparison

This section introduces a statistical analysis aimed at evaluating the effectiveness of cache partitioning techniques compared to existing methods [21]. Specifically, the study examines differences between traditional clusters and newly established virtual clusters. Initially, the Friedman test [84]] assesses significant disparities in average impact. Results indicate substantial differences, ranging from $\mathbf{1.6}$ to $\mathbf{13.44}$ for MEI, a benchmark in cluster computing. For pMEI, speedup improvements range from $\mathbf{0.02}$ to $\mathbf{2.19}$, while Cluster B demonstrates enhancements from $\mathbf{0.04}$ to $\mathbf{1.8}$, as detailed in Table 3.5.

Subsequently, the Holm-Bonferroni procedure [84] serves as a post-hoc test to determine

Table 3.5: The Friedman test on Mining erasable item-set MEI and MEIs based on multi-core processors pMEI based method for running time

Cluster Name	Iteration	Execution time-value based on cluster ((MEI), (pMEI)				Test value	State Result
		MEI	$pMEI_A$	*Proposed_Algo_*Cl_A	*Proposed_Algo_*Cl_B	F_f ($d_f = 3$)	Is $F_f > x^2$?
Cluster A	1	5.10997	0.28429	0.43802	0.11229	20.85	Null Hypothesis Rejected
	2	10.44236	0.56932	0.92024	0.22400		
	3	13.44840	1.11182	1.87415	0.44454		
	4	9.59246	1.11644	1.87891	0.44937		
	5	7.86855	0.03612	0.04748	0.01455		
	6	1.65049	0.07156	0.09660	0.02905		
	7	1.61954	0.14217	0.19861	0.05719		
	8	4.89208	2.19081	0.63744	2.17903		
Cluster B	1	4.85260	0.62234	0.62234	0.62234	19.471	Null Hypothesis Rejected
	2	4.94620	0.60050	0.60050	0.60050		
	3	4.85474	0.59273	0.59273	0.59273		
	4	4.86798	0.59418	0.59418	0.59418		
	5	4.10985	0.60091	0.60091	0.60091		
	6	4.79927	0.61573	0.61573	0.61573		
	7	4.90513	0.62120	0.62120	0.62120		
	8	4.91244	0.60606	77.57683	0.60606		

Table 3.6: The estimation of p-value based on Holm procedure (unadjusted) for post-hoc analysis

cluster	p-values								
	Control method=$FFTCl_A$			Control method=$FFTCl_B$			Control method=$FFTCl_C$		
	MIE	$MIECl_B$	$ProposedCl_C$	*pMIE*	$pMIECl_A$	$ProposedCl_C$	*Proposed*	$ProposedCl_A$	$ProposedCl_B$
A	9.20E-07	0.01927	0.00114	0.00015	0.01927	6.30E-06	7.40E-10	0.00114	6.30E-06
B	7.00E-08	0.48	1	8.10E-07	0.48	0.48	7.00E-08	1	0.48

hypothesis rejection levels for each scenario, detailed in Table 3.6. Here, a confidence level $\boldsymbol{\alpha_c = 0.05}$ and degrees of freedom $\boldsymbol{D_f = 3}$ are maintained across both clusters.

The Friedman test reveals significant discrepancies in average impact spread across iterations, documented in Table 3.5. Rejecting the null hypothesis $(\boldsymbol{H_0})$, which posits equivalence between compared methods without significant differences, the test statistic $\boldsymbol{F_f}$ exceeds the critical $\boldsymbol{X^2(\alpha_3, D_f)}$ value ($\boldsymbol{F_f > 11.0705}$). Thus, Table 3.5 illustrates the Friedman test results for average impact spread, consistently rejecting the null hypothesis across iterations. Similarly, the post-hoc procedure corroborates these findings by rejecting the null hypothesis in each instance.

Conclusively, the application of the Holm-Bonferroni procedure underscores discernible variations among proposed methods, affirming their competitive edge over base algorithms [21] and other existing approaches in terms of runtime efficiency.

3.4 Summary

This research presents reasonable findings about multi-core performance based on the experiments conducted using data mining techniques. The experimental setup for the research includes a virtual cluster environment composed of seven phases, from selecting CPU data to the final virtual cluster construction and considering the dataset that illustrates the components of multi-core systems. The experimental study uses ZeroR and Best First Search algorithms on the earlier mentioned experimental setup and the dataset where ZeroR algorithm resulted in a prediction of class accuracy of ≈ 41.09, correlation coefficient of -0.24, mean absolute error ≈ 10.69 and Root Mean Square Error (RMSE) is ≈ 15.52. Similarly, BFS algorithm resulted ≈ 0.87 [69]. This research also improves the speed-up of the CPUs on large production problems with eight cores and above using the EM algorithm. The selection of the EM algorithm is made with almost consideration of all the requirements by carefully analyzing the existing mining algorithms, and it is observed that the CACHE performance obtained is also better. To evaluate the prediction accuracy of the results obtained thus far, three cases (case-1, case-2, and case-3) based on frequency, cache and both frequency and cache, respectively, are considered and applied to DNN and LR using the CPU dataset. It is observed that for SKUs of 5, 10, and 50, accuracy of $\mathbf{3.8}\%$, $\mathbf{3.3}\%$, and $\mathbf{3.1}\%$ is obtained respectively by using DNN and LR. Based on the results obtained, the ranking of SKUs is done accordingly. Further for the Geekbench the accuracies are $\mathbf{19.73}\%$, $\mathbf{4.2}\%$, $\mathbf{19.9}\%$. This research has successfully brought up an efficient parallel multi-core system with high performance and significant speed compared with other existing systems. As a future scope, this work can be extended to MPI programming paradigm for multi-core cluster systems, and performance variations can be evaluated. As the algorithms used in this research yield the highest likelihood value, a similar technique can be applied for the competition of speed up and efficiency of multi-core systems in the near future.