Parallel Program Performance Prediction and Collaborative Filtering Techniques

Georgios Chatzopoulos
I&C, EPFL

Abstract—Hardware trends point to architectures with increasing numbers of cores and heterogeneity [1]. However, software developed today can only be tested in significantly smaller platforms, due to cost and availability of such platforms. Accurate performance prediction mechanisms would provide many benefits, including improved platform provisioning, which is a significant problem in cloud computing, and could also help identify the hardware characteristics that impede performance.

This research proposal presents two different approaches to performance predictions: a regression based approach for message-passing applications [2] and a performance modeling toolkit that combines application characteristics and target architecture models to produce cross-architecture performance predictions [3]. Summarizing their advantages and limitations, we conclude that a different approach might be more successful, by using techniques from a different area of computer systems, namely collaborative filtering. Consequently, we present and compare different item-based collaborative filtering recommendation algorithms [4] that can be used to identify similarities given a set of common characteristics and present the opportunities, combining ideas from the aforementioned works, for research in the area of performance predictions using collaborative filtering algorithms.

Index Terms—performance prediction, scalability prediction, parameterized models, collaborative filtering

I. INTRODUCTION

In the past decade we have witnessed a shift in processor design. With Moore’s Law [5] still applicable, and Dennard scaling [6] reaching its limits, processor manufacturers have utilized the advancements in lithography to add more processing cores to chips. Future microprocessors will likely have not only a larger number of processing cores, but also exhibit a significant degree of heterogeneity [1].

Software developers have adapted to this shift and software currently being developed is tailored to utilize parallel platforms. However, application performance is tested on current platforms, and the increase in core numbers, together with the heterogeneity of future multiprocessor systems, renders performance results obtained today less useful in the long term. The problem is also present when deploying applications in the cloud. Customers are unable to predict the behavior of their applications and have to overprovision resources. This causes datacenter facilities to operate at very low utilization [7, 8]. Thus, it is essential that we are able to predict the performance of an application on an arbitrary platform. In order to do that, we have to also better understand the characteristics of an application that have a more profound effect on performance, especially with high numbers of cores.

There are different approaches to modeling and predicting performance of software. These depend on the platforms and parallel APIs used, the accuracy of the predictions necessary and the prediction context. The latter refers to restrictions in execution time, since some of the approaches are time-consuming. An extreme example are cycle-accurate simulations [9, 10], which can provide accurate performance predictions, but with considerable time overhead and effort to create. Even modern, distributed simulators that are designed to have lower overheads, have best-case slowdown of 41x versus native execution [11].

In this research proposal, we present two different approaches to performance modeling and prediction. The first is a high-level, black-box approach for message-passing applications. It explores three different techniques of predicting the performance of such an application, all of which use regression-based models. The first regards the performance of an application as a whole and uses the total execution time to predict the performance in the presence of more resources. The other two techniques distinguish the two contributing factors to the performance of a message-passing application, namely computation and communication. The difference between the two techniques is the way the two different components are
combined: in the first, the longest computation time and its corresponding communication time are considered, while in the second only the computation and communication times on the critical path are considered, in order to remove blocking times from the prediction.

The second approach presented is directed towards parameterized performance modeling of applications. It consists of a toolkit that employs a combination of static and runtime analysis to model application-specific factors in isolation. It models the execution behavior of the application, as well as its memory access patterns, in order to model the behavior for different inputs and execution platforms. The models created are architecture-neutral and can be combined with a platform description, in order to produce cross-platform performance predictions.

Based on these approaches and their limitations, we argue that such approaches are either too general, or too detailed. The former results in them missing behaviors that are not visible in current platforms, while the latter increases the complexity of the prediction process and any simplifications can cause inaccuracies. In this context, we believe that a different area of information systems can give a solution. Collaborative filtering algorithms, commonly used to provide recommendations for movies, music, books and more, have been successfully used in other areas of computer science [12]. Collaborative filtering techniques can be used to quantify the effect of different performance factors on parallel applications and provide a generic and accurate performance prediction mechanism. With this motivation, we also present an overview of item-based collaborative filtering algorithms. Their main function is to identify and quantify the similarity between items, and then use information for existing items to provide predictions for unknown ones. The algorithms presented are well-studied ones, and comprise the basis on which most recommender systems rely.

The rest of this paper is organized as follows: Section II presents the first, regression-based approach to performance predictions. In Section III we examine the parameterized performance model approach. In Section IV we present the basics of collaborative filtering techniques, in Section V we briefly present our research plan and in Section VI we conclude.

II. A REGRESSION-BASED APPROACH [2]

The first approach to performance predictions we present focuses on message-passing applications and utilizes three different techniques. It does not require any knowledge of the low-level details of the application, but uses several executions with different input sets on a small subset of the processors. It then chooses the most appropriate technique, based on the data gathered during these executions.

A. Prediction Techniques

The prediction methodology employed in this model consists of a regression-based model. The predictor \( \hat{T} \) of the execution time \( T \) has the following form:

\[
\hat{T} = F(x_1, x_2, ..., x_n, q)
\]

where \( x_1, x_2, ..., x_n \) are the input variables of the application that contribute significantly to execution time and \( q \) is the number of processors used for that particular run. We note here that the identification of the input variables to take into consideration can be automated [13]. The function is chosen to minimize the relative error of the predictor for the runs executed

\[
E = \frac{|T - \hat{T}|}{T}
\]

on a logarithmic scale, so as to avoid large values of \( T \) dominating smaller values. For simplicity reasons, a function that is linear to the unknown parameters of the regression is chosen. This has the form

\[
\log_2(T) = \beta_0 + \beta_1 \log_2(x_1) + \beta_2 \log_2(x_2) + \ldots + \beta_n \log_2(x_n) + g(q) + \text{error}
\]

The results of the experimental evaluation suggest that such an approximation is adequate, with \( g(q) \) quantifying the effect of increased resources to the execution time of an application. This effect can be approximated using a function of the form

\[
g(q) = \gamma_0 + \gamma_1 \log_2(q) + [\gamma_2 \log_2(q)]^2
\]

with the last term being optional. A more complex function type could be used, however more detailed functions suffer from overfitting problems. The function kernel presented above is used for all predictions performed as part of the experimental evaluation.

Three models are employed for the predictions performed. The first model regards the application as a whole and approximates the total execution time, using the presented functions. As simple as this approach might seem, it is very effective in some cases, especially when applications are not communication intensive. The other two models consider computation and communication times separately, and use the same linear model to approximate each one individually. The difference between the two consists of which computation and communication times are used for the regression. In the first, the computation and communication times from the processor with the maximum computation time are used. The idea here is that the process with the maximum computation time will have the minimum amount of blocking time. This assumes that the computation load is distributed evenly across processors and load imbalances will have a detrimental effect to the predictions. The second approach that separates computation and communication times considers the critical path of the application. A representation of the last approach can be seen in Figure 1.

To identify the cases where total execution time approximation would yield accurate results, the authors use two heuristics that identify characterize an application as computation intensive: 1) computation comprises more than 90% of execution time for all processor configurations used, and 2) communication time does not increase with the number of processors used. These heuristics are accurate, since the workload is evenly distributed to processors. As a consequence,
computation time across processors should be equal and such heuristics identify applications that do not spend a significant part of their execution time in communication.

B. Accuracy and Limitations

To evaluate the approach, we chose 5 benchmarks from the NAS benchmark suite and 2 from the ASC Purple/Blue suite, and use the proposed heuristics to choose the best method to use. They use up to 512 processors to predict the performance for 1024 processors and present the best and worst predictions, along with the number of processors used for these predictions. They keep the black-box approach by using the PMPI profiling interface to wrap all MPI calls to measure computation and communication times. The results of their experiments show that in 3/7 cases, the simple approach of regressing the execution time of the application is accurate. This is true for applications with small communication times, and provides a fast and straight-forward way of predicting performance.

For the rest of the cases, separately considering the computation and communication times has better accuracy, as communication time in general scales differently to computation time. What is surprising is that considering the critical path does not seem to increase accuracy in general. Another interesting observation from the results of this work is the predictions for CG of the NAS benchmark suite, which shows a weakness of the approach. When considering the execution time of an application, even when separating computation time to communication time, the black-box approach does not provide knowledge of what changed in the execution time. As seen in figure 2, the decrease in execution time observed between 256 and 512 threads is significant, but is due to a specific reason: the predictions are for strong scaling (the workload remains the same across configurations) and for 512 threads the working set fits in the L2 cache. The prediction model however does not have any knowledge of this and expects this behavior to continue for 1024, underestimating the execution time.

The limitations of the prediction approach presented are mainly due to the very high level modeling of the application. This provides a general, simple model, but some behaviors and characteristics of applications are not considered, resulting in inaccuracies. In order to capture such behaviors, a more detailed approach is necessary, such as the one presented next.

III. A Parameterized Model Approach [3]

The second approach to performance predictions is more detailed one. It models the application and the platform in isolation. Thus, it separates the contribution of application-specific factors from the contribution of architectural characteristics to overall application performance. The toolkit developed operates in a black-box manner, performing static and runtime analysis of the binary file of an application. The application models produced are platform-independent and can be consequently combined with a platform model, producing cross-architecture predictions. The process followed is shown in Figure 3 and presented in the following subsections.

A. Static and Dynamic Analysis

The first part of the analysis is the static analysis of the binary file of the application and the reconstruction of its control flow graph (CFG). This is necessary in order to identify the loops and the instruction mix of basic blocks, used to build the model of the application. The static analysis also includes identification of register and memory dependencies. The former are easier to identify, while the latter cannot be identified from static analysis alone. In this case, the analysis relies on either optimistic or pessimistic assumptions.

With the CFG reconstructed, the toolkit performs binary rewriting to augment the application with code that monitors and logs information regarding memory accesses and basic loop execution frequency, collected during the execution of the application for different inputs. In order to minimize the overhead of the measurements, the minimum number of counters
should be used. Combining the results of CFG theory [14, 15], in order to monitor the execution frequencies of all the edges of a CFG, it is sufficient to monitor the ones that are not part of its spanning tree. By choosing the edges that are part of the spanning tree correctly, we can further minimize the overhead, monitoring the edges least frequently traversed. The choice of edges is enforced by modifying their weights and then constructing a maximum-weight spanning tree. Using the same technique, edges that are hard to instrument can also be avoided. In order to estimate the least frequently traversed edges, a heuristic is used. This takes into consideration the existence of loops, as well as the estimated frequencies of departure nodes, in an iterative fashion.

With the execution frequencies of basic blocks calculated, the last step is to model the memory access behavior of the application. This is done using the access frequency and the reuse distance of memory accesses, monitored during execution. By instrumenting all memory accesses and using a logical clock that each memory instruction increments, the authors create reuse distance histograms, which capture the number and frequency of accesses to each memory block, such as the one shown in Figure 4. Several techniques are used in order to improve the process, such as reference histogram compression and reference group collection. The former consists of coalescing the histogram bins with similar distances to reduce the space required, without reducing the precision of the reuse distance model. The latter is employed to capture the effect of spatial reuse in cache lines. The same thing applies to memory accesses that have the same pattern (i.e. references that are part of the same loop and access the memory with the same pattern). The size of the bins is not fixed, since the target architecture is not known, but multiple hypothetical cache line sizes are used when measuring memory access frequencies and reuse distances, to enable predictions for different architectures.

**B. Parameterized Models and Target Architecture**

After both the static and runtime analysis are finished, the data collected is used to produce models for both the execution frequency and the memory access behavior. The former depends on the number of instructions executed, the mix of instructions and the instruction schedule dependencies. Quadratic programming [16] is used to perform the function approximations of the execution frequency of edge counters, parameterized by the input parameters they depend on. Restrictions to the resulting functions are applied, in order to reduce or remove oscillations of the resulting fit, as well as to ensure that the computed function is of a specific form. The instructions are modeled to a set of generic RISC instruction set, which is independent of the architecture the predictions are made for, as well as the architecture used to collect the data. This approach is more appropriate when the applications have predictable behaviour (i.e. they are not adaptive).

The modeling of the memory behavior is a more challenging problem. That is because reuse distance histograms have to be modeled. On the one hand, using a large number of bins produces a highly accurate model, but with increased complexity and cost for many references that have the same reuse distance. On the other hand, a model that uses a small number of bins produces a compact model which might lack precision. The solution to this problem is to choose a variable size of bins, with the size being a result of collected data analysis. First, the bins that are the result of spatial reuse are modeled. These bins can be identified because they exhibit constant reuse distances across all problem sizes. When referring to bins we in general refer to single memory accesses, but also to memory accesses that have the same pattern (i.e. references that are part of the same loop and access the memory with the same pattern). The rest of the bins are modeled in a recursive way: first, an average distance model is used for all the accesses. Then the accesses are split in two and models are computed for each subset, until the model of a part is the same as the one it came from. This provides more detail where necessary, but also a compact model for groups of accesses that are very similar. The choice of how to split the accesses is done based on a heuristic, which creates subsets that cover equal parts of the remaining range of reuse distances. The subsets could contain a very different fraction of accesses, but accesses with similar reuse distance are kept...
together, increasing convergence speed.

After the bins have been calculated, they are modeled by two polynomial functions, one that models how the number of accesses changes with the problem size and one that models how the average reuse distance of accesses in the bin changes with the problem size. The parameterized model for each access can then be computed by combining the two polynomials, giving a model like the one presented in Figure 5. The figure presents the model for one of the most frequently executed memory accesses in the Sweep3D benchmark. In this plot, the x axis represents the problem size, the y axis represents the normalized frequency of accesses and the z axis represents the reuse distance of the access. By intersecting the surface with the corresponding orthogonal planes, we can get hit and miss rates for each access, for arbitrary sizes of L1 and L2 cache sizes, as shown in Figure 6.

The last step of the prediction is to map the execution and memory access models to the target architecture. To this end, a configurable scheduler that takes as input an architecture description is used. The description includes the type and number of execution units, the mapping of instructions to units. Finally, it quantifies the latencies of instructions and memory accesses for different levels of the cache hierarchy. Using this description and the models created, the scheduler can then estimate the hit/miss rates and the memory access latencies, in order to predict the execution time of the application on the target architecture.

### C. Accuracy and Limitations

To evaluate their approach, the authors use six benchmarks, 5 from the NAS benchmark suite and the ASCI Sweep3D benchmark. For memory access latencies, the authors use empirical values derived from microbenchmarks. The applications are modeled on a RISC system and predictions are produced for a different RISC platform. The results of their predictions for cache and TLB miss rates are accurate within 10% of the measured values for all predictions. The reasons behind these errors are some simplifications of their memory model. The use of reuse distance to predict hit and miss rates assumes a fully associative cache, based on the results of Beyls and D’Hollander [17] that show that reuse distance predicts the number of cache misses accurately even for caches with a low associativity level or direct mapped caches. This difference has two results. First, conflict misses are not present in a fully associative cache, and as such not modeled. This results in lower miss rates than the actual ones. The second result is the difference in behaviour when the working set size exceeds the L2 cache size. In a fully associative cache, the miss rate increases abruptly, while set associative caches exhibit a smoother increase. Both results can be seen in Figure 7, when observing the L2 miss count for Sweep3D.

The predictions for the execution time of the applications are less accurate. Errors for execution time predictions are as high as 20%, with an example of high prediction errors presented in Figure 8. The main problem with execution time predictions is the translation of information on cache miss rates to visible memory access latencies, something that the authors have not resolved yet. Modern out-of-order superscalar architectures execute instructions in a different order, hiding some of the cache miss penalties. Moreover, compiler optimizations, such as instruction reordering can change the visible latencies of memory accesses, affecting the quality of the predictions. Finally, cache misses do not always incur the same penalties. An example is the eviction of a dirty cache line, which has a much higher latency than a cache miss on an empty cache line. The values used for the experimental evaluation of the models were derived from microbenchmarks, but also adjusted according to empirical knowledge, and account for part the errors in execution time predictions.

### D. Summary

So far, we have seen two different approaches to performance predictions. They are quite different in nature, the first considering applications at a high level, with no knowl-
The main challenges that user-based collaborative filtering algorithms face are: recommendation quality in sparse matrices and scalability. The first refers to situations where users have rated only a small portion of the items. In these cases, a nearest neighbor algorithm might be unable to make any recommendations. The second challenge is that for nearest neighbor algorithms, the computation grows with both the number of users and the number of items. The authors propose item-based collaborative filtering algorithms as a solution to these two challenges of user-based algorithms. Item-based algorithms avoid the computation of the neighbors which is the bottleneck in conventional recommender algorithms. Moreover, item relationships are in general static and precomputations can improve the performance of the process. Finally, discovery of similar users is not necessary, and item-based algorithms can produce better results in sparse matrices.

IV. ITEM-BASED COLLABORATIVE FILTERING [4]

Collaborative filtering algorithms are mainly used for recommendations. For example, they are used in e-commerce sites to recommend books, music, movies and other types of products. In general, they operate on a matrix like the one presented in Figure 9. The rows of this matrix are the users of the system and the columns are the items that are rated. Each cell of the matrix contains the rating that the corresponding user has given to the item.

Given a matrix of ratings, where rows represent users and columns represent items, the goal is to recommend to user \( U \) items that he has not rated yet, but is likely to rate favorably. The recommendation process consists of two main steps. The first is the prediction of the ratings \( P_{u,j} \) that the user would give to items not yet rated. After the ratings are predicted, the next step is to recommend the N top items, that the user will like the most. The process can be seen in Figure 10.

Collaborative filtering recommender systems face a trade-off between performance and quality of predictions. The former is due to the computations necessary, in particular the search for neighbors in user-based systems, which depends on the size of the matrix. This performance problem is more profound in the presence of users with many ratings, or "long user rows". The latter is the accuracy of the recommendations, which is the main reason for using collaborative filtering systems. The performance of a recommender system can in general be improved by relaxing our expected recommendation accuracy, and recommendation accuracy can be increased with a cost of extra computations.

Two main categories can be identified: user-based collaborative filtering algorithms and item-based ones. User-based algorithms recommend items to a user \( U \), based on the preferences of users similar to user \( U \). They discover neighbors, other users who have historically had similar taste to user \( U \) and then recommend items that the neighbors have liked, but user \( U \) has not yet discovered. Item-based collaborative filtering algorithms have a different approach, considering only items. They identify similarities between items and predict the ratings, based on ratings of user \( U \) for similar items. Then they recommend items based on these predicted ratings.

The first step of item-based collaborative filtering algorithms is the computation of the similarities between any two items. This computation takes into consideration only users that have rated both items and uses a number of different ways to compute the similarity between the items. In all the following formulas, \( i \) denotes the item in question and \( j \) each item whose similarity with \( i \) we are calculating. \( R_{u,i} \) denotes the rating of user \( u \) for item \( i \) and \( R_{u,j} \) the rating of user \( u \) for item \( j \). Finally, \( U \) denotes the users that have rated both \( i \) and \( j \), \( \bar{R}_u \) denotes the average rating of user \( u \) and \( \bar{R}_i \) the average rating of item \( i \).

The first method is cosine-based similarity. It calculates the similarity between two items \( i \) and \( j \) as the cosine of the angle between the two vectors of the items. The similarity is given by the formula

\[
sim(i, j) = \cos(\bar{R}_i, \bar{R}_j) = \frac{\bar{R}_i \cdot \bar{R}_j}{\|\bar{R}_i\|_2 \times \|\bar{R}_j\|_2}
\]  

(5)

The cosine-based similarity is the most basic computation method. However, it does not take into consideration the differences in rating scale between different users. This problem is addressed in the adjusted cosine similarity, which subtracts the corresponding users average rating from the rating of items \( i \) and \( j \). The formula for the computation of the similarity is

\[
sim(i, j) = \frac{\sum_{u \in U}(R_{u,i} - \bar{R}_u)(R_{u,j} - \bar{R}_u)}{\sqrt{\sum_{u \in U}(R_{u,i} - \bar{R}_u)^2 \times \sum_{u \in U}(R_{u,j} - \bar{R}_u)^2}}
\]  

(6)
The last similarity computation method tries to solve the problem of ratings varying between users, by using the Pearson-r correlation of the two items $i$ and $j$ in order to identify a statistical relationship between the items, given by the formula:

$$\text{sim}(i, j) = \frac{\sum_{u \in U}(R_{u,i} - \overline{R_i})(R_{u,j} - \overline{R_j})}{\sqrt{\sum_{u \in U}(R_{u,i} - \overline{R_i})^2} \sqrt{\sum_{u \in U}(R_{u,j} - \overline{R_j})^2}}$$  \hspace{1cm} (7)

**B. Predictions**

After the similarities between items have been computed, the rating for item $i$ is predicted, using the ratings of similar items. For this computation, there exist several different techniques, two of which are presented here.

The first method uses the **Weighted Sum**. This method uses the average of the ratings given by user $u$ to items similar to $i$, weighted by their similarity to $i$, which causes more similar items to affect the average more. The formula of this computation is the following:

$$P_{u,i} = \frac{\sum_{N \in \text{SimilarItems}}(S_{i,N} \ast R_{u,N})}{\sum_{N \in \text{SimilarItems}}(S_{i,N})}$$  \hspace{1cm} (8)

The second method takes into consideration the fact that two items might be similar, yet their rating vectors might be distant (in a Euclidean sense), and as such it regresses the ratings for item $N$ based on the ratings for item $i$, using the function:

$$\overline{R_N} = \alpha \overline{R_i} + \beta + \text{error}$$  \hspace{1cm} (9)

**C. Performance and Quality Evaluation**

For the evaluation of item-based collaborative filtering algorithms, the authors use data from the MovieLens recommender system they have developed. They divide the available data to a training and working set, according to a variable $x$, which denotes the percentage of data used as training and use the mean absolute error of predicted ratings as their accuracy metric. Apart from the different item-based algorithms, a user-based collaborative filtering algorithm which considers all users and employs the Pearson nearest neighbor algorithm is used for comparison purposes.

The evaluation first focuses on the different similarity algorithms and the sensitivity of the training/test ratio $x$ to the quality of the results. These experiments indicate that the adjusted cosine similarity algorithm outperforms the other two and that $x = 0.8$ provides the best results, and as such is used in the rest of the experiments.

The concept of precomputed data is then utilized for the item-based algorithms. By using precomputed similarities, the item-based algorithms can speed up their predictions significantly. Additionally, prediction quality can be traded-off for performance, by considering only the top $l$ similar items, referred to as model size hereinafter. The item-based algorithms are first compared to the user-based algorithm for different sparsity and neighborhood sizes. The quality of the predictions is consistently better, although the improvement is not significantly large. The experiments then focus on the sensitivity and performance of item-based algorithms, with regard to the model size and the sparsity of the rating matrix. The regression-based algorithm provides better quality of recommendations for sparse rating matrices than the weighted-average algorithm. However, in the presence of more ratings, it suffers from data overfitting problems, which causes its accuracy to decline. Finally, an important observation is that for the different sparsity values ($x$) considered, the model size choice can give performance benefits, without sacrificing quality significantly, as shown in Figure 11. Notice that for model sizes between 100 and 175 the differences in quality of prediction are small, while the throughput of the systems improves significantly for smaller model sizes.

**V. CURRENT WORK AND RESEARCH PLAN**

Performance predictions are a challenging area, with the complexity of approaches growing with different programming APIs, platforms and synchronization techniques. This explains the many different approaches to performance modeling and performance predictions, as well as the limited scope of many of them. A general-purpose approach that works for different classes of applications and has low overhead, both in time and effort on the predictions, is hard to develop and existing approaches usually solve part of the problem.
Our current work in the area explores the idea of keeping the black-box approach, but gathering as much information as possible, using performance counters. It collects performance counter measurements regarding stalled cycles due to resource unavailability and then uses analytical functions from predefined function kernels to approximate the values. The modeled performance counters are combined into a performance metric that accurately models the performance of an application and can predict unexpected behaviors of an application successfully. As part of this work, we have found that there are certain cases in which the behavior of an application changes drastically for an increased number of cores and information from executions with smaller numbers of cores do not provide information for these changes. There are, however, similarities in the behavior of applications, as certain characteristics of an application cause similar behaviors under different cases.

We believe that more detailed modeling of applications that captures these characteristics is possible. Based on the successful application of collaborative filtering techniques on other areas [12], we believe that it is possible to identify the factors of applications that affect performance, as well as the ones that model aspects such as synchronization. By identifying the key contributors to the performance of an application, we can apply collaborative filtering techniques to identify similarities between applications, in both performance and behavior. Given that we can identify similarities in application behavior, we can then use a minimum set of benchmarks, in order to predict the performance of any application on any platform, for which measurements of the benchmarks are available. We believe this approach to be promising for generic performance predictions, independent of platforms, APIs used and synchronization techniques, that would provide a practical solution to this problem.

VI. CONCLUSION

In this paper we presented current trends and opportunities for future work in the area of performance predictions. We presented two very different approaches, with different goals, which show that completely different approaches can yield equally good predictions. However, they both have limitations due to their modeling of the behavior of an application. Based on these observations and our current work, we proposed possible future work in the area, by combining ideas from a different area of computer systems, namely collaborative filtering systems, which we presented. Finally, we concluded with a plan for future research, towards a practical, generic and accurate performance prediction mechanism.

REFERENCES