BB-CVXOPT: Basic Block Execution Count Estimation and Extrapolation using Constrained Convex Optimization

Youssef Aly*, Atanu Barai†, Nandakishore Santhi†, Abdel-Hameed A. Badawy*†
*Klipsch School of ECE, New Mexico State University, Las Cruces, NM 80003, USA
†Los Alamos National Laboratory, Los Alamos, NM 87545, USA
*{youssef, badawy}@nmsu.edu, †{abarai, nsanthi}@lanl.gov

***Abstract*—Program execution time often scales with input size, making performance prediction without execution valuable if the model is efficient and scalable. While machine learning has been used for such predictions, we propose using Constrained Convex Optimization at the finer Basic Block (BB) level. Our BB-CVXOPT system models program performance by breaking down a target program into BBs and using a numerical solver to generate polynomial equations for each BB, based on input size. These equations predict BB execution counts, which can then estimate runtime when multiplied by BB execution times for a given system. BB-CVXOPT is architecture-independent, achieving error rates as low as 1.39e-16 and MAPE below 1.0e-07 for the largest 30% of the dataset, and a MAPE of 1.30e-03 for 65% of the dataset.**

I. Introduction

Modern multi-core CPUs have grown increasingly complex, making hardware optimization more challenging. Performance modeling and simulation (ModSim) tools are crucial for understanding the impact of architecture changes on application performance, resource utilization, and power consumption. These tools facilitate hardware-software co-design, help system designers select optimal configurations, and allow developers to fine-tune software without physical hardware. Developing scalable ModSim tools is essential for efficient design space exploration and performance optimization.

A key ModSim workflow is system performance prediction, which evaluates hardware-software interaction. This is valuable in both program development and production use, as it reduces the need for exhaustive program profiling across large input ranges, avoiding computational bottlenecks and long run times.

Recent advancements in CPU architecture analysis have introduced 'basic blocks' (BBs) as fundamental units of performance analysis [1], [2], [3], [4]. BBs are single-entry, single-exit code sections without branching instructions (*e.g.* , $jump$), easily identifiable in assembly code or intermediate representations (IR) like LLVM-IR. Since BBs in LLVM-IR are hardware-independent, they offer an architecture-agnostic approach for performance analysis. BBs provide granular metrics for application performance, typically correlating with input size. Modern ModSim tools [1], [4], [5] and compilers [6] rely on BBs for performance analysis and code optimization.

Consider a simple matrix addition program as shown in Figure 1 with matrices of size $M \times N$ and a time complexity of $O(M \times N)$. Examining the BBs, one might identify tasks such as variable reading, array indexing, addition, and loop condition checking which includes a branching instruction. To estimate the execution count of each BB, quadratic multivariate polynomials in variables m, n such as $a * m * n + b * m + c * n + d$ can represent each BB's execution pattern. After determining coefficients a through d, the BB counts for specific inputs can be predicted. Runtime estimation, which involves summing BB counts multiplied by their execution times, is outside the scope of this paper. This paper introduces BB-CVXOPT, a generalized approach for deriving such equations.

BB-CVXOPT uses Convex Optimization techniques to predict BB execution counts. Our methodology instruments a target program to create a labeled dataset, where input parameters serve as features and BB counts as the output. We generate a set of monomials from the input parameters and the combinations of their multiplications and exponentiations, the previous example would have the set $m * n, m, n, 1$, potentially including monotone functions like $\log_2 x, \sqrt{x}$ for the input parameters. These monomials form polynomials, and a numerical solver minimizes the L2-Norm [7] between polynomial predictions and actual BB counts by solving for the coefficients. Infinitesimally small coefficients are removed, and the model is further refined by the solver.

We demonstrate the feasibility of BB-CVXOPT by evaluating its extrapolation capabilities using a sample BB. The dataset is partitioned, with a polynomial derived from smaller inputs and extrapolated to larger ones. We also study the sensitivity of the system by varying the dataset percentage used to generate the polynomial. Our results indicate the system performs well in extrapolation, though some BBs provide more descriptive information than others.

```
1    for (int i = 0; i < M; i++)
2        for (int j = 0; j < N; j++)
3            a[i][j] = b[i][j] + c[i][j];
```

Fig. 1: Example code snippet.

The contributions of this paper are as follows:

- This work represents the first attempt, to the best of our knowledge, to predict BB counts for CPU applications using Convex Optimization techniques.
- We introduce the BB polynomial model, a mathematical framework that expresses a target program as a suite of equations, with each BB represented as a function of the input parameters. This model enables the extrapolation of BB counts from larger input sets of a program, guided by the BB counts from smaller input sets.
- We demonstrate that the model exhibits competitive accuracy, even for complex applications.

The rest of the paper is structured as follows: Section II provides an overview of relevant prior work. Section III gives the essential background. Section IV details basic block trace extraction and the architecture of the Convex Optimization model. Section V describes our experimental setup. Section VI presents our experimental results. Finally, Section VII encapsulates our conclusions.

II. Related Work

For both CPU [1], [8] and GPU [5] architectures, research has extensively explored Basic Block-based performance analysis and prediction. Various techniques utilize Basic Blocks to assess and predict performance. The LLVM-Machine Code Analyzer [6], inspired by the Intel Architecture Code Analyzer (IACA) tool [9], uses Basic Blocks to statically measure and predict machine code performance throughput on a given CPU, aiding in program diagnostics.

Models predominantly use Neural Networks to learn Basic Block patterns and predict performance. Mendis *et al.* [10] proposed Ithemal, which employs a Long Short-Term Memory (LSTM) recurrent neural network for predicting Basic Block execution times. Chennupati *et al.* [1] developed PPT-AMMP, a regression model focused on scalability, predicts the impact of large unseen inputs on Basic Block invocation counts by incorporating machine learning techniques and Basic Block level data dependency graphs. Aktar *et al.* [11] demonstrated a scalable Basic Block invocation count prediction model for GPUs, achieving accuracies of 97.7% and 98.1% for random predictions, and 93.6% and 92.8% for extrapolation predictions.

To our knowledge, no prior work has demonstrated the scalable modeling of Basic Block invocation counts using Constrained Convex Optimization (CCO). Our proposed technique models programs efficiently while maintaining scalability, using smaller inputs to predict counts for larger inputs with considerable accuracy. This modeling approach is generic and can be easily applied to both CPU and GPU applications, given the availability of Basic Block traces.

III. Background

A. Basic Blocks

Compilers break applications into Basic Blocks (BBs) for analysis and optimization. Basic Blocks are contiguous segments of code without branching instructions such as $jump$. The compiler analyzes these blocks to apply optimizations like loop unrolling. Basic Blocks that are part of loops or are frequently executed are particularly affected by program input.

Figure 3 shows a control flow graph of a sample program with five BBs, separated by branching instructions (*e.g.*, if statements). Each if statement forms a single BB, while loops span multiple Basic Blocks. The program flow starts at the "Begin" node and ends at the "Terminate" node.

B. Convex Optimization

Convex optimization [12] is a method for solving various mathematical problems, one of the most common are minimization problems. These involve a solver program finding the global minimum of a function through iterative adjustments of coefficients until a specific threshold is met. Various mathematical techniques are used to manipulate these coefficients internal to the solver.

Like machine learning (ML) models, convex optimization is versatile and allows interpolation and extrapolation of numerical values based on input features. It also benefits from the ability to apply constraints, such as positivity and non-zero values, which can expedite convergence and improve predictability. Unlike neural networks, convex optimization provides more granular control over the problem through explicitly defined mathematical expressions, often resulting in fewer trainable parameters and reduced training time.

IV. Methodology

This section outlines the methodology for modeling Basic Blocks of a program and predicting their counts. We describe the process of tracing a program to gather BB counts for various input sizes, preparing the counts, preparing the

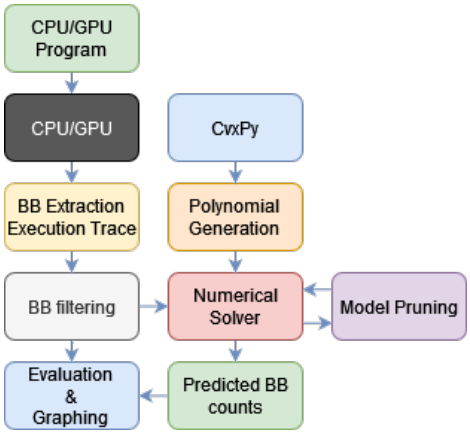

Fig. 2: BB-CVXOPT block diagram

polynomial and formulating the problem. The Basic Block count dataset, polynomial model, and formulated problem are fed into a numerical solver. The output model is then pruned and sent back to the solver for refinement. The final output is evaluated. Figure 2 depicts the BB-CVXOPT system workflow.

A. Basic Block Execution Trace Collection and Analysis

Following Chennupati *et al.* [1], the target program is compiled into architecture-independent LLVM-Intermediate Representation (IR) using a compiler like Clang for C or Flang for Fortran. The IR is analyzed by a Basic Block Analyzer (BBA) to identify and instrument BBs across a range of program input settings. BB counts are logged into a CSV file, along with the current input size, creating a dataset.

BBs may have varying execution patterns: some may execute once (*e.g.*, initialization), some may correlate with input sizes (*e.g.*, multiply operations in matrix multiply), and others may not execute at all (*e.g.*, exiting the program if the inner dimensions do not match). Invariant BBs that execute a fixed number of times or occasionally contribute minimally and statically to the total run time and can be excluded temporarily. Their effect can be accounted for when predicting the total run time. These invariant BBs are identified as dataset columns with zero variance. Figure 4 shows the sorted variance of BB counts for 2667 BBs in a sample program where more than 50% of the BBs are static.

BBs that are equally affected by the entire input set are considered duplicates and only need to be modeled once and the predicted counts can be copied respectively, saving time. A correlation matrix can identify duplicates, which will show a correlation factor of 1.0. BBs with non-zero variance but below a certain threshold, or those with a correlation factor less than 1.0 but above a defined threshold, can be eliminated, trading off accuracy to a shorter modeling time.

B. Polynomial Preparation

CvxPy [14] is a Python-embedded modeling language for convex optimization problems. It aids in preparing, formulating, and solving these problems. A polynomial representing

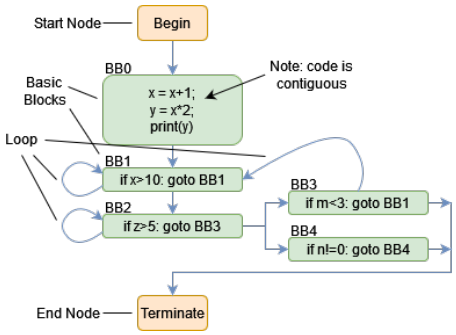

Fig. 3: Basic Blocks (BBs) in a small sample program.

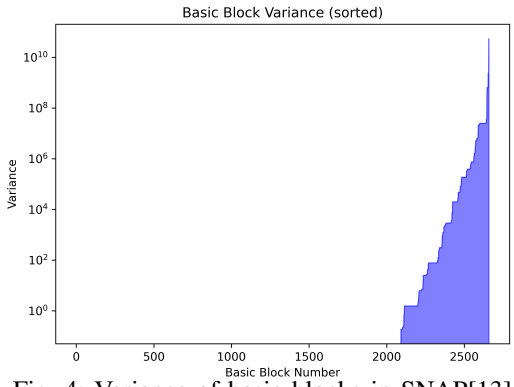

Fig. 4: Variance of basic blocks in SNAP[13]

the count of one BB in a program is given by: $(BBcount_j = a_{j0} * M_{j0} + a_{j1} * M_{j1} + ... + a_{jn} * M_{jn})$, where a_{j0} - a_{jn} are coefficients representing how input size affects the BB count. M_{j0} - M_{jn} are monomials, which are combinations of inputs multiplied and raised to various powers. Monomials are prepared for every target program and vary with the number of inputs. To generalize, mathematical functions like $\log_2 x$ and $\sqrt{x}$ can also be included in the monomials.

Monomials are generated based on the number of inputs and a set maximum order. For instance, with three inputs X, Y, & Z and a maximum order of 4, CvxPy generates terms such as X^0, X^1, X^2, X^3, X^4, and similar terms for Y and Z. All possible multiplicative and exponentiated combinations of these terms are created, provided their total order does not exceed the maximum, 4 here. The generated monomials will be as follows; $1, X^4, X^3, X^3 * Y, X^3 * Z, X^2, X^2 * Y, X^2 * Z, X^2 * Y * Z, X, X * Y^3, X * Y^2, X * Y, X * Z^3, X * Z^2, X * Z, X * Y^2 * Z, X * Y * Z, X * Y * Z^2, Y^4, Y^3, Y^3 * Z, Y^2, Y^2 * Z, Y^2 * Z^2, Y, Y * Z^3, Y * Z^2, Y * Z, Z^4, Z^3, Z^2, Z$

C. Solver

Once the polynomial for $BBcount_j$ is formulated, substituting the input sizes into the equation (X, Y, & Z in the example above) should approximate the corresponding BB count, provided the coefficients (a_0 through a_n) are accurate.

$$e_j = \sqrt{\sum_{\forall k} \left| < \boldsymbol{a}_j, \boldsymbol{M}_{kj} > - y_{kj} \right|^2} \quad (1)$$

Equation 1 describes the convex optimization problem given to the numerical solver, where $\boldsymbol{a}_j$ represents the vector of unknown coefficients for BB j, $\boldsymbol{M}_{kj}$ denotes the vector of monomials for BB j and k-th parameter setting, and y_{kj} denotes the measured count for BB j for the k-th parameter setting. The solver aims to minimize the gap between the polynomial prediction and actual counts across all given input settings. Here, e_j is called the "L2-norm" of the fitting error - as is common across a number of domains and quite popular in machine learning literature[7]. Constrains such as positivity can be applied to the vector of

coefficients $\boldsymbol{a}_j = [a_{j1}, a_{j2}, \ldots, a_{ji}, \ldots, a_{j|M_{kj}|}]$. These can assist convergence but may affect accuracy.

D. Evaluation

The accuracy of the polynomial models is evaluated using the Mean Absolute Percentage Error (MAPE) between predicted and measured BB counts. Equation 2 describes the MAPE for BB j, x_k denotes the predicted count for the k-th parameter setting and y_k denotes the measured count. BB execution counts will vary greatly across different target programs, thus MAPE is chosen so the performance of the BB-CVXOPT system can be compared.

$$MAPE_j = \frac{1}{n} \sum_{\forall k} \left| \frac{x_k - y_k}{y_k} \right| \tag{2}$$

Due to the nature of equation 2, counts y_k of zero in the dataset lead to infinite MAPE values, which are dismissed for clarity. The predicted and measured counts are plotted to visually assess correlation, and the modeling time is recorded.

To demonstrate extrapolation, the dataset is sorted by BB count and split into two parts. The solver uses the lower count portion to find coefficients, and the higher portion tests the polynomial's extrapolation ability for larger inputs, evaluated using MAPE calculations.

V. Experimental Setup

In this section, we prove the viability of the BB-CVXOPT by conducting a case study on SNAP [13]. We go through how all the components come together to predict BB counts and then we evaluate the accuracy as well as extrapolation power for large unseen inputs. All experiments were run on an AMD EPYC 7702P processor machine with 64 cores running at 2 Ghz clock with 16MB L3 cache.

A. Target Program

SNAP, short for SN (Discrete Ordinates) Application proxy, is a modern discrete ordinates neutral particle transport application. A sophisticated computational approach aimed at solving intricate problems related to the transport of sub-atomic particles, specifically neutrons and gamma particles. This method plays a crucial role in understanding the distribution of these particles across various dimensions, including space, angle, energy, and time. In SNAP, three-dimensional space is modeled using a structured Cartesian mesh. This mesh serves as a fundamental framework for solving the transport equation, which is at the core of the particle movement analysis. The methodology hinges on a concept known as "discrete ordinates", which entails computing solutions for a finite set of specific directions or angles. Each angle carries a corresponding weight, and these solutions are computed for each individual angle. A total of seven dimensions come into play, encompassing three dimensions in space, two dimensions in angle, one dimension in energy, and one dimension in time. The governing transport equation, particularly in the space-angle dimensions, embodies a hyperbolic nature, allowing information to propagate from source points to downstream destinations.

As described in Section IV-A, SNAP was compiled using Clang v3.9.0 into LLVM-IR with O3 optimization. It was then analyzed and instrumented to get the BB counts for various input sizes.

SNAP's seven inputs correspond to the seven dimensions (N_x, N_y, N_z, I_{chunk}, N_{mom}, N_{ang}, N_g). It has 2667 BBs. A set of 1296 different combinations of the inputs were used and the output BB counts were put into a CSV file. BB number 330 appears to be the most prominent with 3,932,160 executions at input 16, 16, 16, 1, 4, 8, 4.

A Python script was used to process the dataset, invoke the numerical solver, and analyze the results. The dataset was loaded into a Pandas DataFrame for manipulation. Removing zero-variance columns resulted in 568 BBs. Figure 4 shows SNAP BB variance. Furthermore, removing duplicate columns resulted in 93 unique BBs. These reductions are discussed in Section IV-A.

B. Polynomial Preparation, Formulation & Evaluation

The maximum order of monomials was set to 7 to match the number of inputs in SNAP, allowing all inputs to contribute through multiplicative combinations. CvxPy takes about one hour to generate the monomial set for 7 inputs, a maximum order of 7 and 1296 dataset samples. This time grows exponentially with the maximum order and number of inputs. The monomials are saved to disk to be ready for future modeling sessions.

CvxPy is used to define the expression for the problem as described in Section IV-C, minimizing for equation 1. The modelling has three stages, first solver run, pruning and second solver run. Five mutually exclusive constraint settings were implemented, "Positive Coefficients", "Positive Coefficients only after pruning", "Positive Polynomial Value" and "Positive Polynomial Value only after pruning" as well as "No Constraint". Positive coefficients forces the solver to find coefficients that are only positive. "Only after pruning" constrains the coefficients to be positive only in the second run. "Positive Polynomial Value" forces the evaluation figure of the entire polynomial to be positive irrespective of the coefficients, corresponding to the natural count nature of BBs, never negative.

Two solvers were considered: Splitting Conic Solver (SCS)[15] and GUROBI Optimizer[16]. Both solvers aim to minimize the "L2-norm" between the polynomial evaluation value and the BB count from the execution trace. Due to the large number of monomials, pruning is applied by sorting the coefficients by their absolute value, selecting a user-defined maximum number of terms to represent each BB. The pruned monomials are then passed back to the solver for further refinement.

The resulting polynomial for each BB is evaluated using MAPE, as described in equation 2. The solver runs five times per BB, each with a different constraint setting, and the run with the lowest MAPE is recorded along with its corresponding constraint. The MAPE is then plotted. The dataset and predicted BB counts are also plotted on opposite axes. The time taken to model and evaluate a BB and the mean MAPE for each BB are also noted.

The polynomial predicts how often a BB will execute based on input size. The BB counts can be used with modeling tools like PPT [17], [4], [18].

VI. Results

A. Validation

After testing several Basic Blocks, some had a lower MAPE with the SCS solver, while others performed better with GUROBI. For a few non-intersecting BBs, neither solver could converge. As a result, a combination of both solvers was used in all experiments, selecting the one that achieved the lowest MAPE or converged when the other did not. The maximum number of solver iterations was set to 10,000, with the stopping threshold (eta) at the default 1e-8. Coefficients smaller than 1e-8 were discarded to avoid numerical errors, and the top 100 coefficients after pruning were kept, a number chosen arbitrarily.

$$e_k = |x_k - y_k| \tag{3}$$

Figure 5 shows a strong correlation between input and prediction with normalized values. Figure 6 displays the absolute error, ranging from $8.89e-11$ to $3.33e-7$, defined by equation 3, where x_k is the predicted count and y_k is the dataset count. The average MAPE across all inputs was $3.95e-13$, with some data points omitted as in Section IV-D. This took 32.1 seconds using all 1296 inputs, with the solver itself running for 7.85 seconds. The dataset maintained the original factorial order from program instrumentation with no sorting. The resulting polynomial expression is,

$$BB_{330}\ Count = (2.0 * (N_{mom}^2) - 2.0 * (I_{chunk}^2)) * (N_{ang} * N_g * N_x * N_y * N_z)$$

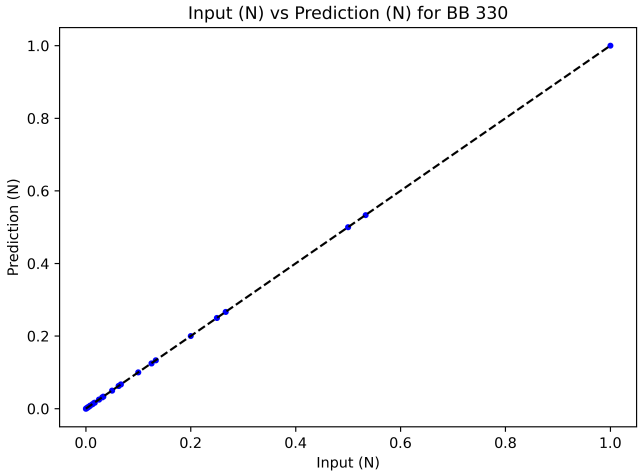

Fig. 5: Normalized dataset measured BB counts (X-axis) vs. normalized solver approximation (Y-axis) for BB 330 across all input combinations

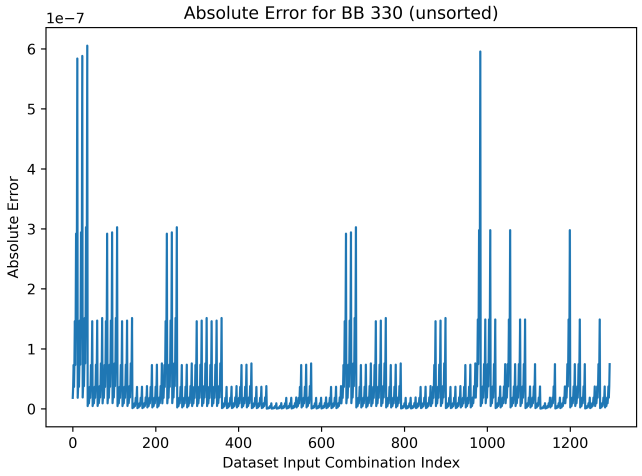

Fig. 6: Shows the Absolute Error between the BB column and the solver approximation for BB 330 across all input combinations (X-axis)

B. Extrapolation

To extrapolate, the data is sorted in ascending order, with the lower portion used to calculate the coefficients, and the solver evaluated on the unseen higher portion. The most prominent BB count is used for sorting, and the dataset is split 70:30.

Figure 7 shows a strong correlation, with blue points representing smaller inputs used to solve the polynomial and red points showing larger extrapolations.

Figure 8 displays the absolute error, with the red dotted line marking the split point. Across all the inputs, the average MAPE is $4.18e-16$ for the modeling portion and $1.39e-16$ for novel inputs. Modeling time was 368.2 seconds using 907 input combinations (70% of inputs). The two largest terms of the resulting polynomial are:

$$BB_{330}\ Count = (2.0*(N_{mom}^2) - 0.67*(1 + I_{chunk} + I_{chunk}^2)) * (N_{ang} * N_g * N_x * N_y * N_z)$$

C. Program Wide Approximations

The extrapolation power is further investigated across the remaining BBs of SNAP [13]. The dataset is sorted by the BB count of the most prominent BB (330).

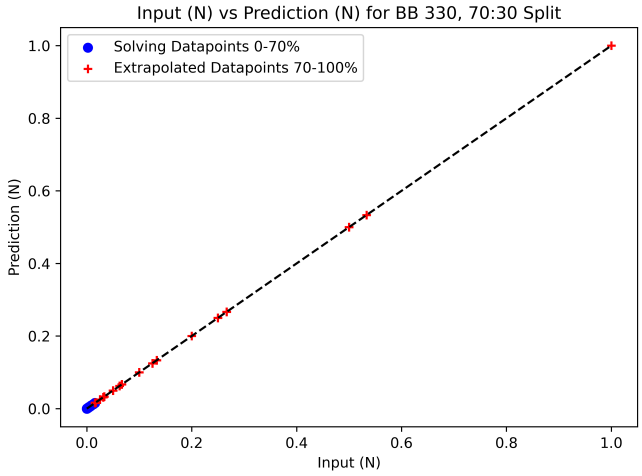

Fig. 7: Normalized dataset BB counts (X-axis) vs. normalized solver approximation (Y-axis) for BB 330 across all input combinations for 70:30 split

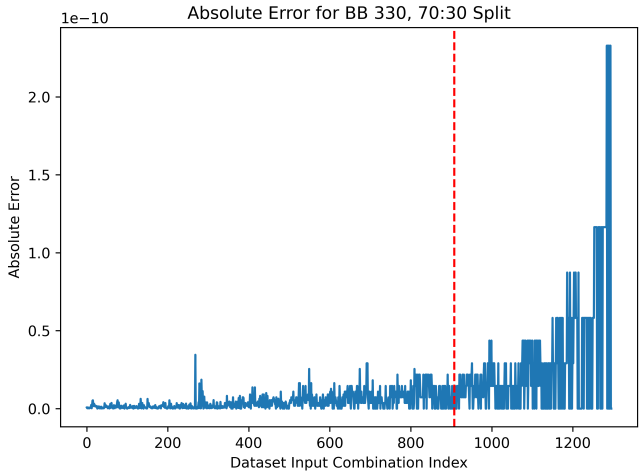

Fig. 8: Shows the Absolute Error between the BB column and the solver approximation for BB 330 across all input combinations (X-axis) for 70:30 split

Figure 9 shows the MAPE for all 93 Basic Blocks, sorted in ascending order. Blue represents the portion used to obtain the polynomial and red represents the extrapolated portion. Each data point reflects the MAPE of one BB across all input combinations for that portion. MAPE is on the Y-axis, and the BB index is on the X-axis. The solving MAPE ranged from $1.85e-16$ at BB 785 to $9.52e-9$ at BB 2129, with a mean of $1.32e-10$. The extrapolating MAPE ranged from $1.39e-16$ at BB 330 to $1.27e-8$ at BB 2129, with a mean of $1.71e-10$.

The 93 BBs were distributed across CPU cores to parallelize the modeling. Some BBs may not accurately represent

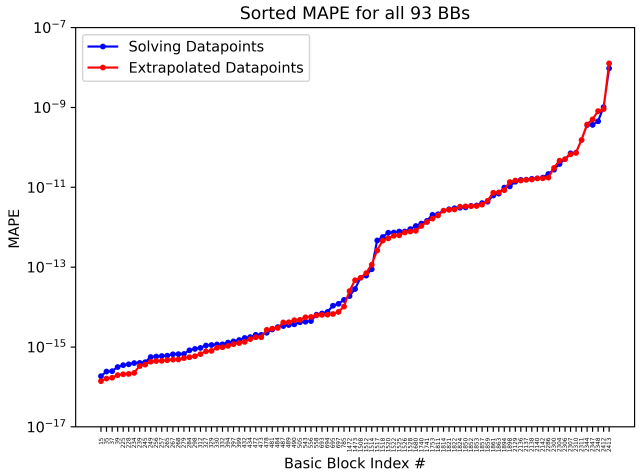

Fig. 9: MAPE of all 93 BB columns, X-axis is the index of all BBs. The upper input 30% is unseen data for extrapolation. Blue represents the solving portion, red represents the same BBs when extrapolating.

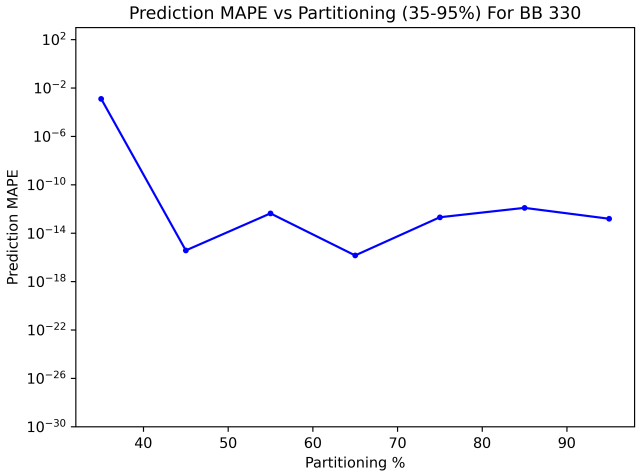

Fig. 10: MAPE vs testing data percentage for BB 330

the target program's behavior with respect to input size, potentially leading to higher MAPE; see Sections IV-A and V-A.

D. Sensitivity Study

The partitioning is varied to study how far the model can extrapolate for our target program. Given the sparsity of the lower 35% of the dataset, partitioning is varied from 65:35 old:novel to 95:5 old:novel with 10% increments, and the most prominent BB 330 used for this experiment.

Figure 10 shows the MAPE of BB 330 on the vertical axis at different partitioning levels (percentage old data) on the horizontal axis. The MAPE ranged from $1.45e-16$ to $1.30e-03$, highlighting the strong extrapolation power of BB-CVXOPT with only 35% of the dataset.

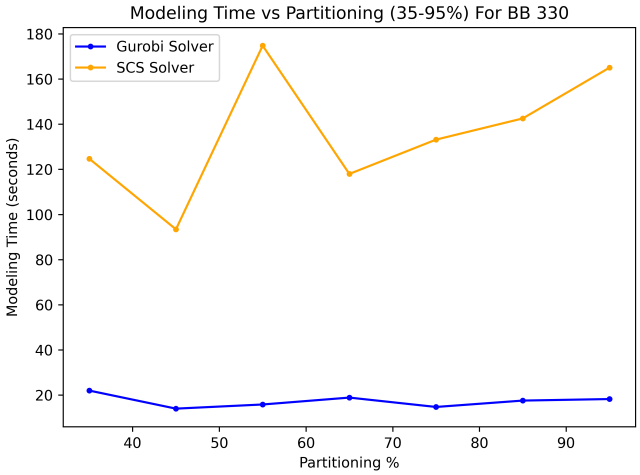

Fig. 11: Time (seconds) taken by both solvers to approximate BB 330 given only 35% to 95% of the data

Figure 11 shows the solving time for BB 330 on the vertical axis at different partitioning levels on the horizontal axis. The mean modeling time was 17 seconds for Gurobi and 135 for SCS. As more data is fed into the solver, outliers average out as the polynomial generalizes better hence the higher variance in modelling time given less data.

VII. Conclusion

Handling larger input configurations can present computational complexities and scalability challenges. This work introduces BB-CVXOPT, a tool designed for predicting the execution counts of basic blocks in CPU applications. BB-CVXOPT employs a mathematical framework to generate polynomials for extrapolating basic block counts and validates its effectiveness through a case study of a complex program. The results indicate that BB-CVXOPT can successfully extrapolate basic block counts even for larger input values of the application, demonstrating a high degree of accuracy across various extrapolation levels. Given only 35% of the dataset, it was able to achieve $1.30e-03$ MAPE, showcasing its immense extrapolation power. Furthermore, we will demonstrate the practical utility of BB-CVXOPT when used in conjunction with a PPT and integrate it into a modeling and simulation framework, showcasing its versatility.

Acknowledgment

We would like to thank the anonymous reviewers for their feedback. Triad National Security LLC partially supports this work, under subcontracts # 581326 and # C4975. Any opinions, findings, or conclusions expressed in this paper do not necessarily represent the views of the DOE or the US Government. This paper has been approved for public release under LAUR # LA-UR-24-27278.

References

[1] G. Chennupati, N. Santhi, P. Romero, and S. Eidenbenz, "Machine learning–enabled scalable performance prediction of scientific codes," *ACM Trans. Model. Comput. Simul.*, vol. 31, no. 2, apr 2021.

[2] A. Abel and J. Reineke, "Accurate throughput prediction of basic blocks on recent intel microarchitectures," 2021.

[3] W. Zhang, M. Hao, and M. Snir, "Predicting hpc parallel program performance based on llvm compiler," *Cluster Computing*, vol. 20, 06 2017.

[4] A. Barai, Y. Arafa, A.-H. Badawy, G. Chennupati, N. Santhi, and S. Eidenbenz, "Ppt-multicore: performance prediction of openmp applications using reuse profiles and analytical modeling," *The Journal of Supercomputing*, pp. 1–32, 2022.

[5] A. Betts and A. Donaldson, "Estimating the wcet of gpu-accelerated applications using hybrid analysis," in *2013 25th Euromicro Conference on Real-Time Systems*, 2013, pp. 193–202.

[6] LLVM. (2023) Llvm machine code analyzer. [Online]. Available: https://llvm.org/docs/CommandGuide/llvm-mca.html

[7] X. Luo, X. Chang, and X. Ban, "Regression and classification using extreme learning machine based on l1-norm and l2-norm," *Neurocomputing*, vol. 174, pp. 179–186, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S092523121501139X

[8] A. Barai, N. Santhi, A. Razzak, S. Eidenbenz, and A.-H. A. Badawy, "Llvm static analysis for program characterization and memory reuse profile estimation," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '23. New York, NY, USA: Association for Computing Machinery, 2024.

[9] Intel, "Intel architecture code analyzer," 2019. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/tool/architecture-code-analyzer.html

[10] C. Mendis, A. Renda, D. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 4505–4515. [Online]. Available: https://proceedings.mlr.press/v97/mendis19a.html

[11] S. Aktar, H. Abdelkhalik, N. H. Turja, Y. Arafa, A. Barai, N. Panda, G. Chennupati, N. Santhi, S. Eidenbenz, and A.-H. Badawy, "Bb-ml: Basic block performance prediction using machine learning techniques," 2022.

[12] S. P. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[13] B. R. Zerr Joe. (2015) Snap: Sn (discrete ordinates) application proxy. los alamos national laboratory (lanl). [Online]. Available: https://github.com/lanl/SNAP

[14] S. Diamond and S. Boyd, "CVXPY: A Python-embedded modeling language for convex optimization," *Journal of Machine Learning Research*, vol. 17, no. 83, pp. 1–5, 2016. [Online]. Available: https://www.cvxpy.org/

[15] B. O'Donoghue. (2021) Splitting conic solver. [Online]. Available: https://www.cvxgrp.org/scs/

[16] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2023. [Online]. Available: https://www.gurobi.com

[17] A. Barai, G. Chennupati, N. Santhi, A.-H. Badawy, Y. Arafa, and S. Eidenbenz, "PPT-SASMM: Scalable Analytical Shared Memory Model: Predicting the Performance of Multicore Caches from a Single-Threaded Execution Trace," in *The International Symposium on Memory Systems*, ser. MEMSYS 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 341–351.

[18] Y. Arafa, A.-H. Badawy, A. ElWazir, A. Barai, A. Eker, G. Chennupati, N. Santhi, and S. Eidenbenz, "Hybrid, scalable, trace-driven performance modeling of gpgpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.