Heterogeneity-Aware Software Performance Characterization via Graph Machine Learning

Ronaldo Canizales
Computer Science Department
Colorado State University
Colorado, USA

Jedidiah McClurg
Computer Science Department
Colorado State University
Colorado, USA

***Abstract*—Today's technology landscape requires hardware platforms with ever-increasing heterogeneity. To obtain efficiency in heterogeneous systems, developers invest considerable time and resources in fine-tuning software programs targeting these platforms. This optimization task becomes more challenging as new hardware designs emerge. Current approaches for estimating software performance require either considerable manual specialized labor, are computationally expensive, or do not generalize well on new hardware architectures.**

We propose a hierarchical learning approach for software performance characterization. This method benefits from a data structure called an e-graph to compactly encode a software of interest, which we obtain from C/C++ programs in 4.35ms on average. We have trained our performance estimator on a 150+ million-instance heterogeneity-aware dataset, and are able to predict execution time across unseen hardware, programs, problem sizes, and workloads, obtaining accuracies up to 99.44%.

***Keywords*-Heterogeneous systems; performance analysis; graph machine learning; scientific applications;**

I. Introduction

Software systems are becoming larger, increasingly complex, and more challenging to analyze and optimize. Manual optimization techniques can be effective in limited scenarios, such as systems running in fixed environments, but the growing diversity of processors, architectures, and accelerators has introduced new challenges [1]. The task of automatic performance optimization has become significantly more difficult due to heterogeneous hardware becoming more prevalent in both research and industry.

Program synthesis techniques, which encompass one approach for automating the generation of optimized code, require accurate and efficient performance estimation models to guide the optimization process [2] [3]. By predicting the performance of a program during its synthesis phase, these models can guide the optimization algorithm toward high-performing solutions without the need for exhaustive trial-and-error execution on various hardware platforms.

Software performance prediction approaches combine a variety of methods. Static analysis focuses on examining code structure to identify potential performance bottlenecks without execution [4]. Another strategy involves early performance testing, where developers assess and optimize performance during the design and coding phases [5] [6]. Additionally, containerization and virtualization are increasingly used to create controlled environments for testing, enhancing performance predictability [7] [8] [9]. As an alternative to these approaches, Machine Learning (ML) models can be trained on historical performance data to predict how software will behave under various conditions [10] [11]. These models utilize features such as code metrics, hardware specifications, and workload characteristics to make predictions.

There is a growing need for efficient and heterogeneity-aware performance estimation methods that can guide the optimization of time-sensitive systems. Although several approaches have been proposed to address these challenges, they often fall short when dealing with complex hardware-software interactions across diverse platforms.

In this context, performance estimation becomes a critical problem: it involves predicting how a program will behave on a given hardware setup. One key metric to predict is execution time, but memory usage and the number of floating point operations (FLOPS) are also useful. Once a reliable and accurate performance prediction model is developed, it eliminates the need to execute the program on actual hardware to guide optimization, drastically improving the efficiency and scalability of the optimization process.

This paper explores how Graph Machine Learning (GraphML) can offer a promising alternative for performance characterization in such heterogeneous environments. Our approach uses a data structure called an e-graph [12] to compactly store a set of target programs. We build predictive models that can efficiently estimate performance across a wide range of hardware architectures, ultimately aiding in the optimization of software performance. Our overall contributions are:

- We introduce an e-graph-based code-to-graph representation based on equality saturation, which encodes a program's control flow and stores global data dependencies only a few hops away. Our characterization is compositional and fine-grained by construction. We are able to encode a given C/C++ program into an e-graph at 4.35ms on average.

- We propose a hierarchical approach to the downstream task of software performance characterization. First, we obtain performance behavior similarity-based clusters; then, we train one classifier and multiple cluster-wise regressors and compose them into a single model.
- We tested our approach in a 150+ million-instance custom dataset to predict software performance across unseen hardware, programs, problem sizes, and workloads. We obtained errors of around 4% on average and smaller than 1% in the best-case scenario.
- We experiment and report results on the tradeoffs that e-graph's embedding length has on accuracy and training time. We also performed an ablation analysis on our clustering and embedding strategies.

A. *Background*

ML models typically perform best when input data is in a structured, tabular format, often referred to as Euclidean data. However, source code is inherently complex and text-based, with intricate variables, data structures, and dependencies that do not naturally fit this format. To address this, it becomes necessary to transform the source code into a more ML-friendly representation, such as vectors. Graph embeddings offer a helpful solution, as source code can be represented as graphs, where nodes correspond to entities like functions, variables, and data structures, while edges represent relationships between them. A variety of graph representation methods exist in the literature, and once the code is represented as a graph, embedding techniques can convert this structure into dense vectors. These vectors capture the code's semantic and structural properties, making them suitable for input into ML models. Figure 1 shows a pictorial summary of this entire process.

The main downside of this approach is that slight modifications to a program can significantly impact its performance. Most common graph representations of source code fail to produce distinct graphs for similar programs, even when performance differs, as they often result in graphs with the same number of nodes and identical node connections. Only a few graph representations capture semantic-level differences, and this variation is typically detectable only by graph embedding methods that consider node content. However, most graph embedding algorithms focus on structural aspects, making them unable to distinguish between performance-differentiated programs.

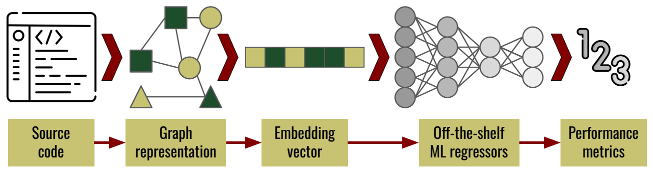


Figure 1. Graph Machine Learning approach overview.

B. *Motivating example*

The following simple programs contain doubly nested `for` loops, with a minimal change in the order of loop execution (lines 4-5 are swapped on Listings 1 and 2). This slight modification can significantly impact the performance of a program handling large datasets in memory due to cache locality effects. This happens because accessing memory addresses in a non-contiguous way makes cache misses more frequent, thus increasing the overall execution time. Unfortunately, the most commonly used graph representations for source code fail to produce sufficiently distinct graphs for these two programs. In Figure 2, both representations have the same number of nodes, and the edges connect identical nodes. Here, a basic off-the-shelf approach would produce identical embedding vectors for these programs, preventing performance estimation models from distinguish them, and therefore limiting learning accuracy.

```
1 int main() {
2     int matrix[10][5000] = {{1,2,3, ...}, ...};
3     int i, j;
4     for (i = 0; i < 10; i++)            // i-loop
5         for (j = 0; j < 5000; j++)      // j-lopp
6             matrix[i][j]++;
7     return 0;
8 }
```

Listing 1. Program A: outer i-loop leads to fewer cache misses.

```
1 int main() {
2     int matrix[10][5000] = {{1,2,3, ...}, ...};
3     int i, j;
4     for (j = 0; j < 5000; j++)      // j-loop
5         for (i = 0; i < 10; i++)    // i-lopp
6             matrix[i][j]++;
7     return 0;
8 }
```

Listing 2. Program B: outer j-loop leads to more frequent cache misses.

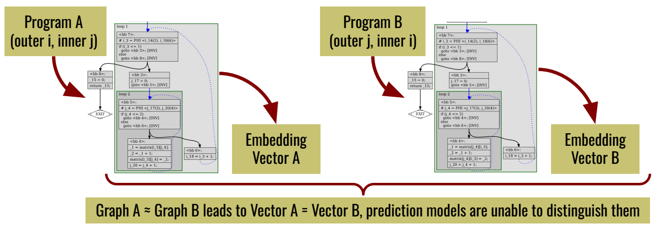
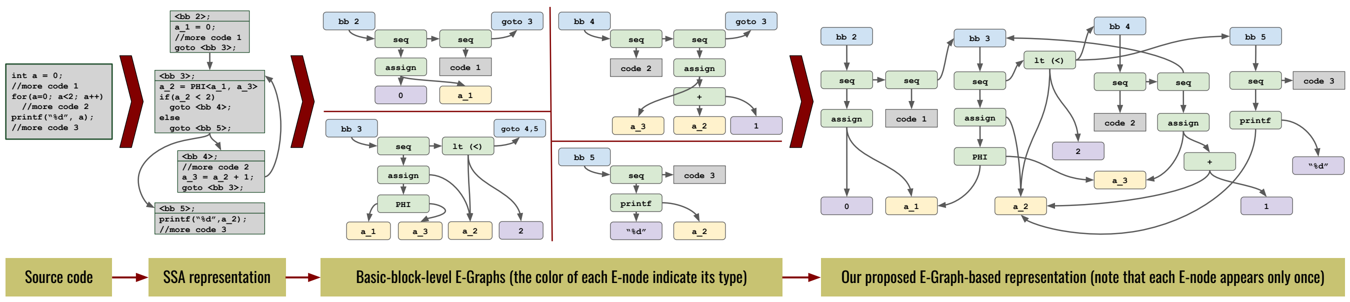

Program A (outer i, inner j)
Embedding Vector A
Program B (outer j, inner i)
Embedding Vector B
Graph A ≈ Graph B leads to Vector A = Vector B, prediction models are unable to distinguish them

Figure 2. Characterization failure: even though there is a significant performance difference, both intermediate representations are identical.

Our key idea is to leverage the ability of the e-graph data structure to compactly store equivalent expressions. For example, see Fig. 4 where two e-graphs corresponding to two programs are merged into a single e-graph. Both programs share a common operation or set of operations. This pattern produces a desired effect in the resulting graph: equivalent operations/variables are stored only once.

This effect is desired in software characterization because it makes it easier to identify (i) global data dependency



```
int a = 0;
//more code 1
for(a=0; a<2; a++)
  //more code 2
printf("%d", a);
//more code 3
```

```
<bb 2>;
a_1 = 0;
//more code 1
goto <bb 3>;
```

```
<bb 3>;
a_2 = PHI<a_1, a_3>
if(a_2 < 2)
  goto <bb 4>;
else
  goto <bb 5>;
```

```
<bb 4>;
//more code 2
a_3 = a_2 + 1;
goto <bb 3>;
```

```
<bb 5>;
printf("%d",a_2);
//more code 3
```

Source code
SSA representation
Basic-block-level E-Graphs (the color of each E-node indicate its type)
Our proposed E-Graph-based representation (note that each E-node appears only once)

Figure 3. Process of obtention and saturation of an e-graph from a simple Static Single Assignment graph representation.

patterns and (ii) similar behavior among different functions or even different programs. Embedding benefits from this compact way of storing equivalent expressions because different programs with common patterns will be connected in the resulting e-graph, thus producing similar vectors.

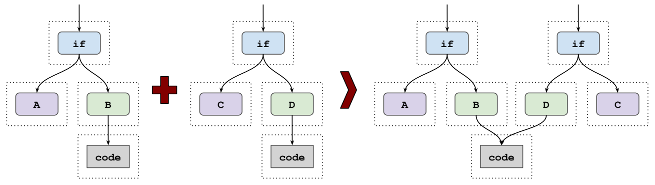


Figure 4. Union of two e-graphs. Dotted boxes represent equivalence classes containing equivalent expressions, operations, variables, or values.

II. Methodology

We propose the following approach to address the downstream task of predicting execution time. The first step is transforming the source code into its Static Single Assignment (SSA) form. SSA is a representation where each variable is assigned exactly once, and every assignment is uniquely named, providing a more structured view of the program's flow. Combining this with e-graphs makes it easier to analyze dependencies and optimize computations, as the value of each expression is defined in a single place.

A. E-graph obtention and saturation

Once the SSA is obtained, we enhance the graph representation by converting it into an e-graph. This process modifies the SSA by separating each element (such as variables and operations) into individual nodes and merging those that share semantic similarities. For instance, even if two operations are distant in the SSA but operate on the same variable or pattern, they would be connected in the resulting graph, which is known as equality saturation [13], [14]. This allows the model to capture semantic relationships between code elements that are not immediately adjacent.

As shown in Figure 3, given a C program, each of the elements of its SSA graph representation is modeled as an individual e-node. Each e-node contains information such as the type, value, and name of the operation or variable. Then, all e-nodes inside each basic block are linked as needed. Any redundancies are removed through saturation, which merges semantically equivalent e-nodes based on rewrite rules. For example, variables of the same name, equivalent integer values, and `go-to` operations among basic blocks.

One key feature of our e-graph is its efficient creation process, which has linear complexity over the content of a given SSA graph. This means that each SSA feature is visited only once, eliminating the need for backtracking, and ensuring efficient creation performance.

B. Obtaining program-level embedding vectors

Before feeding our non-Euclidean e-graphs to ML algorithms, we must convert them into tabular data. This means that fixed-length vectors will characterize all of our programs. Each possible program would be represented by one point within our high-dimensional embedding space.

While it is necessary to maintain the key individual properties of each program, it is important to ensure that the relative differences between the programs are also reflected in the embedding vectors. Thus, similar programs would be embedded into nearby regions of our high-dimensional space. To accomplish that, we learn the embedding of each graph by treating them as a subgraph of a union of all graphs, an approach that has been useful in other studies [15]. The two key benefits of this collective embedding process are (i) all vectors will share the same latent space, thus ensuring that equivalence of program (sub)expressions results in vector similarity, and (ii) running an embedding algorithm only once is more efficient than doing so for each program, as we show in Figure 13 on the Results section.

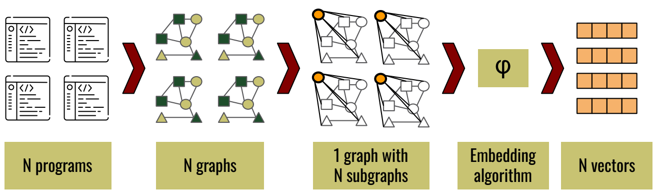
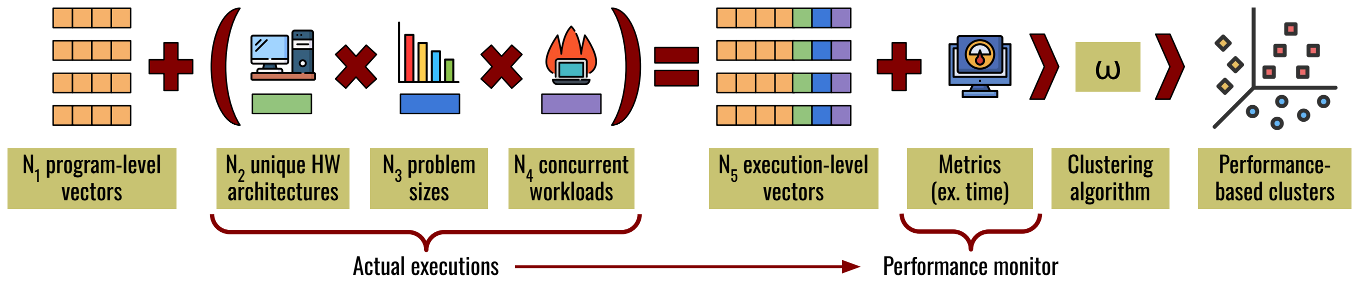


Figure 5. Obtaining program-level embedding vectors.



N_1 program-level vectors
N_2 unique HW architectures
N_3 problem sizes
N_4 concurrent workloads
N_5 execution-level vectors
Metrics (ex. time)
ω
Clustering algorithm
Performance-based clusters
Actual executions
Performance monitor

Figure 6. Obtaining performance-based clusters based on execution-level embedding vectors and performance metrics given by a monitor when executing programs on a heterogeneous set of system's architectures and demands.

We create a sub-graph per program to be embedded. All nodes of each sub-graph are linked to an extra meta-node, shown in orange in Figure 5. The goal of each meta-node is to be the representative of its sub-graph. Next, an embedding algorithm is executed only once on the union of all sub-graphs. Even though all individual nodes will be assigned to a vector, we only care about the embeddings of the meta-nodes. Another approach would be to use a pooling operator over all node-level vectors, e.g., *average* or *max*. We compared this alternative and found that it achieves lower accuracy; more details are given in our Results section.

C. Performance-based clustering

A key part of our proposed approach is to group programs with similar performance behavior together before proceeding to the learning steps; this way, an ensemble of multiple ML models can be trained, each using only similar-behaved programs. We accomplish this by finding clusters based on performance metrics such as execution time, GFLOPS, and memory (RAM + L1, L2, L3 caches) bandwidth.

To find the aforementioned metrics, we use a performance monitor when executing the programs on a heterogeneous set of computers. Each N_1 program is executed several times, one for each combination of N_2 unique hardware architectures, N_3 input problem sizes, and N_4 concurrent workloads measured as the percentage of resources used at the beginning of execution. This leads us to $N_5 = \prod_{i=1}^{4} N_i$ *execution-level* vectors, shown in Figure 6, which are the result of the concatenation of the *program-level* vectors with the information about each specific execution.

D. Learning how to classify programs

Once the performance-based clusters have been obtained, we can label each program-level vector with a tag containing its assigned cluster's ID. This can be shaped as a **multi-class classifying task**, where the embedding vectors are the input, and each cluster ID corresponds to a target class. We can use any off-the-shelf classifier to implement this task.

Once a model is trained and tested, it will be able to classify unseen programs into a cluster of other programs with similar performance behavior. We empirically find and report the optimum number of clusters in the Results section and leave the analytical analysis for it as future work.

E. Learning how to predict performance

We leverage the fact that all programs inside a cluster behave similarly under similar conditions, to train one regressor per cluster. In this way, each regressor can learn in a more precise way than if it were required to learn many types of patterns. This can be shaped as a **cluster-wise regression task**, where the input is an *execution-level* vector, and the output is a performance metric. Once a model is trained and tested for each cluster, its ensemble will be able to predict the performance of an unseen program execution over unseen conditions. A total of $K+1$ models is obtained, where K is the number of clusters (one classifier and K regressors).

We empirically find and report lower Normalized Root Mean Square Errors (NRMSE) than if we train a single regressor over the whole data set of *execution-level* vectors.

F. Inference: our end-to-end solution

Our approach can be used as a black box where the inputs are (a) a given C/C++ program, (b) specifications about the hardware architecture we would like to know how it would behave if executed on it, (c) the problem size, and (d) the concurrent workload of the system at the moment of start the hypothetical execution measured as the percentage of resources already utilized by other programs.

Inference consists of four intermediate steps, shown in Figure 7. First, an e-graph is acquired. Second, we embed it

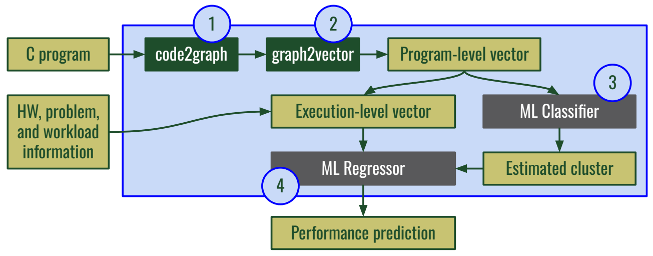


Figure 7. Inference inputs, output, and intermediate steps (numbers 1-4).

into a *program-level* vector, which is concatenated with user-provided parameters to generate an *execution-level* vector. Third, the program is classified into a performance-based cluster. Lastly, a regressor predicts the performance over potentially unseen hardware and other conditions.

III. Experimental setup

Program datasets. We selected three widely-known repositories in the High-Performance Computing and Competitive Programming communities. Rodinia [16] is a benchmark suite for heterogeneous computing; including applications and kernels that target multi-core CPU and GPU platforms experimenting with parallel communication patterns, synchronization techniques, and power consumption. PolyBench [17] is a benchmark suite of numerical computations extracted from operations in various application domains like linear algebra computations, image processing, physics simulation, dynamic programming, statistics, etc. ACM-ICPC[1] is a worldwide annual multi-tiered contest sponsored by IBM, which prompts typically include dynamic programming, sorting, number theory, combinatorics, and greedy, graph, geometric, and network flow algorithms.

Code-to-graph representations. We implemented our e-graph generator in 456 lines of Python code leveraging the Egglog [18] and EasyGraph [19] libraries to handle and convert SSA files obtained through the GCC's preprocessing tool and generate our e-graph representation. We compared our approach vs. HARP [20], an end-to-end graph-based solution to model HPC programs for FPGA-based synthesis built on top of ProgramL [21], and Abstract Syntax Trees (AST) obtained with the help of the PyCParser library [22].

Hardware. We utilized a wide variety of hardware architectures and resources, varying from single-processor machines to the AMD EPYC 74F3 and AMD Ryzen Threadripper 3960X of the HPC cluster at Colorado State University[2]. In terms of CPU, our experimental setup ranged from 8 to 240 total cores. RAM memory capacity ranged from 3GB to 128GB. We also utilized CPU-only systems and others with modern GPUs like Nvidia A100 and GeForce RTX 3090.

Performance Monitoring: We implemented 138 lines of Bash code to automatically execute, collect metrics, and record results in a centralized repository. On every program execution performed on the hardware mentioned above, we collected performance metrics of interest through the Command Line Interface of Intel Advisor [23]. The total amount of computing time expended among all utilized computers sums up to approximately 15,000 hours.

The metrics we obtained were execution time, GFLOPS, and memory (RAM + L1, L2, L3 caches) bandwidths. Our performance-based clustering takes into account all listed metrics. However, our predictive models focus on execution time, and we leave the rest of the metrics to future work.

Predictive generalizability. Our goal is to ensure that our proposed characterization is robust enough to be able to generalize across four dimensions of unseen elements: new hardware architectures, new programs, and resource usage in diverse circumstances, such as problem sizes and workloads.

We address cases where only one variable is unknown, e.g., an existing program on new hardware specifications. Also, we consider combinations of unseen elements, for instance, how a given program would behave in a new system with a higher workload and larger problem input size than the ones considered during training. The distribution of instances in the train, validation, and test subsets must account for combinations of seen and unseen elements.

A random split would not guarantee this property. We use the instances distribution of Table I, which guarantees that both validation and test sets contain unseen programs and hardware while also containing higher and lower problem sizes and workloads than those in the training subset; the same applies to the test subset relative to the validation.

Table I
Distribution of independent variables among subsets

Variable	Total	Train	Val	Test
Programs	60	52	4	4
Hardware	13	9	2	2
Problem Sizes	12	8	2	2
Workloads	8	4	2	2

Learning algorithms. We utilized the Scikit-learn [24] library and leveraged its off-the-shelf algorithms K-Means, Random Forest, and Gradient Boosting for our clustering, classification, and regression tasks, respectively.

Embedding. We leveraged the Python implementation of DeepWalk [25] in the library EasyGraph [19]. We explored the range of vector lengths: $N = [2, 4, 8, 16, 32, 128, 256]$ in all our experiments. We choose powers-of-two inspired by widely used graph embedding algorithms such as [25] [26].

IV. Results

A. Stage-wise e-graph performance

We analyze in Figure 8 how our e-graphs compare to other baselines at every stage of our approach. The three stages of interest are performance-based clustering, multi-class classification of programs, and cluster-wise performance prediction. By comparing how using e-graphs behaves in each stage, we can explore the contribution our graph representation provides to the high-level task of heterogeneity-aware software performance characterization.

E-graphs outperform both baselines at all three stages, detailed as follows. First, as the number of clusters increases, the distance between the centroids and each high-dimensional point decreases. But in the case of e-graphs,

[1]Association for Computing Machinery–International Collegiate Programming Contest. More information at https://icpc.global/.

[2]More details at: https://sna.cs.colostate.edu/hpc/

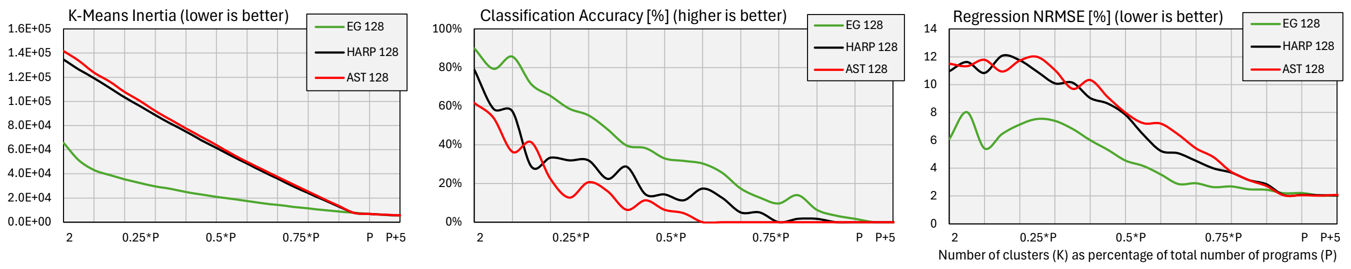


Figure 8. Stage-wise e-graph's results vs. baselines: clustering inertia (left), classification accuracy (center), and regression error (right).

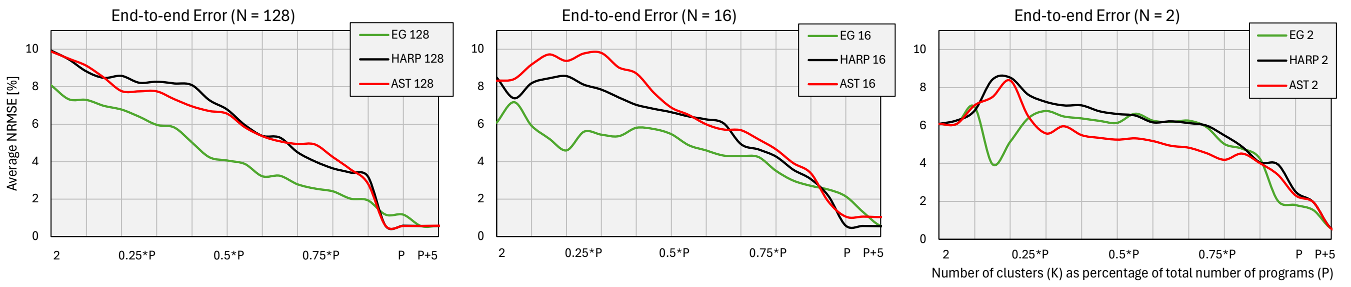


Figure 9. End-to-end e-graph's results vs baselines for different vector sizes N=128 (left), N=16 (center), and N=2 (right). Lower is better in all plots.

it decreases logarithmically instead of linearly like both baselines. By minimizing the cluster's overlap, we decrease the possibility of heterogeneously behaved programs into the same set (in terms of performance metrics).

Second, in any classification task, as the number of classes increases, it is inevitable that the accuracy decreases. However, EG does decrease slower and more consistently than both baselines. EG's accuracy remains above 60% at $K = 0.35P$. Meanwhile, both baselines' accuracy drops below 40% at $K = 0.2P$ and falls to 0% at $K \geq 0.65P$. E-graph's accuracy does not drop to zero until $K \geq P + 2$.

Third, the more heterogeneous a dataset is, the more difficult it becomes for an ML model to learn it. As the number of clusters increases, each one becomes more homogeneous, containing programs that behave similarly between them. While all cluster-wise regressors report a decrease in their prediction error as the number of clusters increments, our experiments show that the usage of e-graphs produces smaller errors compared to the baselines.

B. *End-to-end e-graph performance*

We assess the overall performance of our approach by evaluating the Normalized Root Mean Squared Error (NRMSE), which shows the end-to-end error as a percentage of the possible range of output values, e.g., metrics like program execution time. Figure 9 shows how e-graphs not only outperform both baselines but happens consistently across different embedding sizes $N = \{2, 16, 128\}$.

When e-graphs are embedded with low-dimensional vectors, such as $N \in [2, 16]$, they achieve small errors very soon: around 4% with a fairly small number of clusters, $K \in [5, 6]$. However, its performance is more consistent with higher dimensionalities such as $N = 128$. At $N = 128$ we can see the most straight and predictable NRMSE of all the experiments performed.

Interestingly, cluster-wise regression is robust against misclassifications in the previous stage due to each regressor's generalization capability of extrapolating good enough predictions even for programs outside their cluster.

C. *E-graph's Accuracy-Embedding Length Tradeoffs*

We seek to empirically verify if end-to-end e-graph's accuracy is proportional to *program-level* vector length.

As shown in Figure 10a, it is not always the case that longer vectors achieve smaller NRMEs. However, there is a clear trend that this holds true in the majority of cases, especially when K gets closer to the number of programs.

D. *E-graph's Training Time-Embedding Length Tradeoffs*

Even though we showed that in the majority of the cases, a longer vector length translates to higher accuracy, it is also essential to consider training time. We are interested in analyzing this tradeoff because it may be that a negligible increase in accuracy can only be reached at the cost of a significant increase in training time. Figure 10b shows that vector length is proportional to training time. For instance, the best results are shown by the red and orange lines, $N = 128$ and $N = 256$, respectively. Although both deliver similar accuracy, red is significantly lower than orange in the training time. Thus, the empirically-found optimum vector length is $N = 128$ due to its tradeoff between low NRMSE and reasonably low training time.

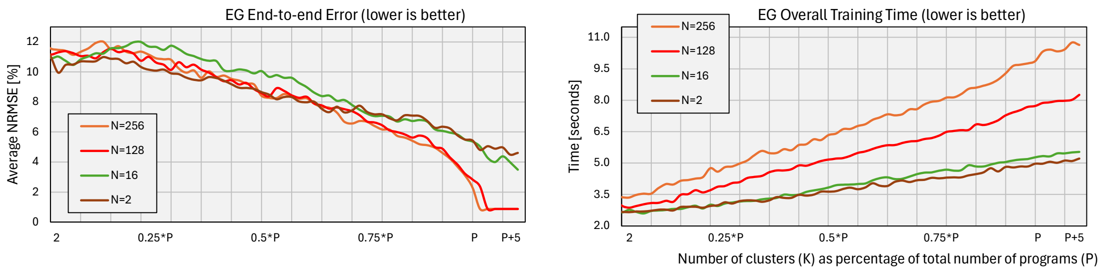


Figure 10. E-graph's Embedding Length Tradeoffs versus Accuracy (left) and Training Time (right). Vector sizes $N \in \{2, 16, 128, 256\}$.

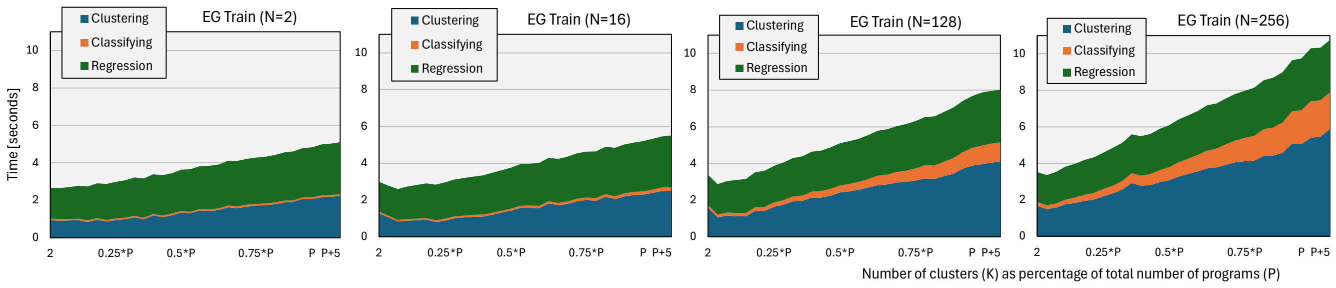


Figure 11. Stage-wise e-graph Training Time analysis, per different embedding sizes, $N \in [2, 16, 128, 256]$, and clusters, $K \in [2, P+5]$.

E. Stage-wise e-graph Training Time Analysis

It is worth analyzing how vector length affects training time at each stage of interest: performance-based clustering, multi-class classification of programs, and cluster-wise performance prediction. Figure 11 shows three highlights. First, clustering represents more than half the overall training time on $N = 2$ but remains constant. Second, training time for the classifier model is by far the quickest stage, but linearly increments until reaching the same time as clustering at $N = 256$. Lastly, regression training time is directly proportional to both vector length and clusters.

F. Training time gains of subgraph-level embedding

We seek to explore the efficiency of the embedding approach we used and explained in Figure 5 compared to a naive process where programs would **not** be treated as subgraphs of a union of all graphs. This means that meta-nodes are not needed, and the embedding task would be individually run over each graph; implying that this task would be executed P times, where P is the number of programs available. This process is detailed in Figure 12.

We performed this experiment using an *average* pooling operator and six additional node-level embedding algorithms: DeepWalk [25], LINE [27], node2vec [26], NOBE and NOBE-GA [28], and SDNE [29]. We leveraged their Python implementation provided by EasyGraph.

Regarding embedding time, Figure 13 points out that running the algorithm only once (but containing all subgraphs plus meta-notes) far outperforms running the algorithm once per each program and then pooling the results. Sub-graph embedding takes 0.485, 0.495, 0.898, and 1.075 seconds for $N = \{2, 16, 128, 256\}$, respectively. The closest competitor is NOBE-GA, which performs well for small vectors but diverges for larger ones. Similar divergence occurs in NOBE and LINE+AST embeddings.

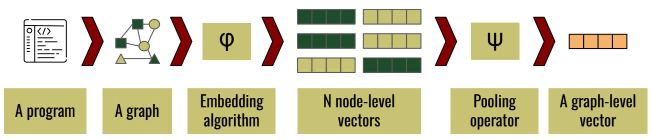


Figure 12. Alternative embedding approach, executed once per graph.

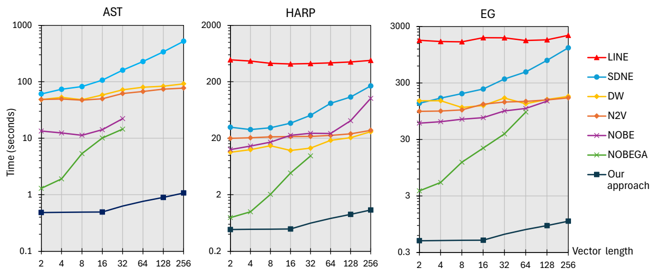


Figure 13. Embedding time gains between our and other approaches. Lower means faster, which is better. This plot is on the log-log scale.

G. Accuracy gains of performance-based clustering

We seek to explore the efficiency of the clustering approach we used and explained in Figure 6 compared to a naive process where programs would **not** be clustered at all. In this scenario, the inputs of the single regression model are all *execution-level* vectors, alongside their corresponding performance metrics as the output. This **coarse-grained** trained model will benefit from being fed over much more data (all executions available) but at the cost of a high level of heterogeneity in such data. Figure 14 shows the inference in this scenario.

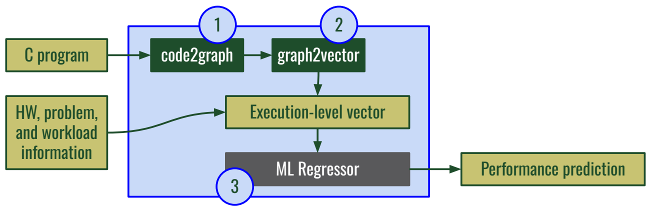


Figure 14. Alternative inference process: no clustering utilized.

We used six regression algorithms to perform this experiment: AdaBoost (ADA), Gradient Boosting (GB), Histogram-based Gradient Boosting (HGB), Multi-Layer Perceptron (MLP), Random Forest (RF), and Support Vector Machines (SVM). The average result is reported. The main results of this paper, obtained by performing inference as stated in Figure 7 and reported in Figure 10, show a substantial accuracy gain over the coarse-grained approach stated in this section. This shows that clustering programs help obtain the lowest NRMSE in the testing subset when characterizing behavior in heterogeneous systems. The best coarse-grained results, approx 4%, are obtained using e-graphs, shown with green in Figure 15. The same Figure shows the accuracy gains of the clustering-based approach over coarse-grained. It is important to note that e-graphs outperform baselines in all cases. We empirically conclude that clustering programs based on performance and training an ensemble of regressor models outperform training a single model. Finally, we explored a variation of our proposed e-graph-based representation, where E-nodes containing values are substituted by placeholders, shown in blue in Figure 15. The number of edges remains the same, but E-nodes do increase. There is a not negligible accuracy gain in doing this, which further analysis we leave for future work.

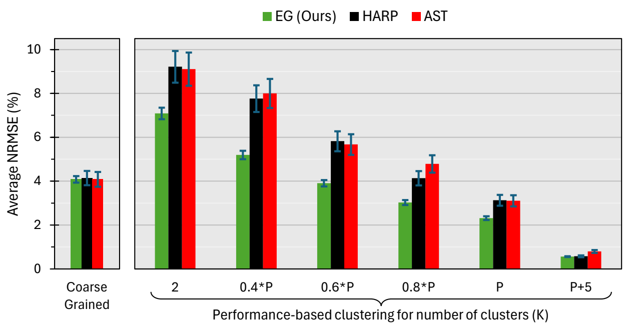


Figure 15. Average results of *coarse-grained* trained models ($K = 1$). E-graphs show accuracy gains over baselines. However, the best overall results are obtained by utilizing clustering, as detailed in Section II and IV.

V. Discussion

In this work, we show that the information related to a program's control flow, data dependencies, and interaction with heterogeneous hardware architectures benefits from being characterized by an e-graph data structure, which compactly stores equivalent expressions. The graph representation we propose can be obtained in 4.35ms on average for computationally-demanding benchmark programs designed for execution in heterogeneous systems.

We report comprehensive results showing that obtaining performance-based clusters allows our approach to adjust the trade-off between accuracy and training time by varying the number of clusters as a proportion of the total number of programs available in the training set. We empirically found an optimal tradeoff between vector length and model accuracy; $N = 128$ appears to be the best candidate for embedding source codes into e-graphs due to its high accuracy and low training time. Although we recognize the need for an analytical characterization study, we leave it for future work. We also report empirical evidence of the advantages of running an embedding algorithm over the union of all e-graphs of interest and obtaining vectors through subgraph-level meta-nodes instead of the traditional pooling of node-level results. We report increased performance both in terms of increased accuracy and shorter training time. While we believe our approach to be an important step toward the high-level goal of heterogeneous-aware software performance characterization, especially concerning hardware systems not available at hand or which do not yet have a physical implementation, there are engineering and research challenges we plan to address in future work.

A. Limitations

Although we used a relatively limited set of hardware architectures for the experiments, and accuracy may be further improved by including more diverse systems, our experimental setup is reflective of a wide variety of commonly used platforms, ranging from limited-resources CPU-only machines to a High-Performance Computing cluster with high-end processing units. The same limitation applies to our selected program datasets, which currently contain only C and C++ source code. Future work will include more programming languages and will consider additional workloads such as ML training/inference, data wrangling operations, and multimedia processing.

VI. Related Work

Search Techniques for Optimization in Various Architectures. Kernel scheduling, an optimization challenge in various computing architectures, benefits from advanced search techniques that explore big and non-smooth optimization spaces. OpenCase [30] enhances performance by incorporating user-provided prior knowledge into search algorithms, achieving improvements in execution speed.

Similarly, Droplet Search [31] accelerates the search process through a greedy heuristic approach, offering faster results in kernel optimization with a smaller set of transformations. However, while offering improved search speeds, both techniques still face challenges in achieving optimal solutions across diverse architectures and in more complex kernel transformation spaces, suggesting room for further refinement in handling more diverse optimization scenarios.

Graph-based Program Representation in ML. Adoption of graph-based representations for program analysis has been a significant trend in the integration of GraphML with programming languages. PrograML [21], one of the early initiatives, introduces a low-level, language-agnostic graph structure for characterizing programs, facilitating their analysis and optimization through ML models. By utilizing message-passing GNNs, approaches like PrograML and HARP [20] support tasks like control flow reachability and data dependencies, replacing traditional heuristic-based methods that are fragile and expensive. However, their focus on intermediate representations like LLVM and XLA presents some limitations to usability across diverse programming languages.

Graph Neural Networks (GNNs) for Automated Accelerator Optimization. The application of GNNs in hardware optimization, particularly in FPGA accelerator designs, presents a promising avenue for automating the optimization process. Methods like GNN-DSE [32] and HARP [20] integrate GNNs into high-level synthesis for FPGAs optimization, offering design space exploration with high accuracy and fast feedback on design configurations. These methods outperform traditional heuristic-driven approaches in scalability. However, its success heavily relies on accurately representing the design space. Inadequate or overly simplistic graph representations can compromise the model's ability to generalize and predict optimal configurations when dealing with complex system architectures.

Parallelism and Performance-Oriented Modeling. Parallelism discovery and characterization are possible via graph design heuristics involving aggregated data types, such as PerfoGraph [33]. This helps capture the complexities of performance in program execution, essential for optimizing compilers and hardware accelerators. However, the challenge persists in capturing performance information within the embedded representation, especially in heterogeneous systems.

VII. Conclusion and Future work

First, we introduced an e-graph-based code-to-graph representation based on equality saturation, which encodes a program's control flow and stores global data dependencies only a few hops away. Our characterization is compositional and fine-grained by construction. On average, we can encode a given C/C++ program into an e-graph in 4.35ms. Second, we propose a hierarchical approach to the downstream task of software performance characterization. Initially, we obtain performance behavior similarity-based clusters; then, we train one classifier and multiple cluster-wise regressors and compose them into a single model. Third, we tested our approach in a 150+ million instances custom dataset, obtaining errors of around 4% on average and smaller than 1% in the best-case scenario while predicting software performance across unseen hardware, programs, problem sizes, and workloads. Lastly, we experiment and report results on the tradeoffs that e-graph's embedding length has on accuracy and training time.

As a next step, we are interested in supporting languages such as Python and Fortran, often used in scientific computing [34]. This should be possible via a translation-based approach [35], or by extending our e-graph converter to other languages, which should be relatively straightforward, provided there is an efficient way of obtaining its SSA form. We are also interested in providing a deeper characterization of the optimum number of clusters, which minimizes both prediction NRMSE and training time; we will investigate using formal methods to obtain provable boundaries for optimum values of N and K. Another direction is to explore vector combinations and concatenations, i.e., combining more than one approach into a single vector. From a hardware perspective, we want to explore more diverse architectures, including FPGAs, TPUs, custom accelerators, and embedded systems [36]. We also plan to leverage our performance-based clusters to identify performance bottlenecks and automatically synthesize actionable insights for software developers [37]–[39]. Lastly, we think dynamically optimized code analysis can benefit from our framework, in particular when comparing different optimization levels in a compiler or when auto-generating programs targeting heterogeneous architectures, e.g., TVM [40].

Acknowledgments

We thank Ravi Mangal for helpful feedback on an earlier version of this work. We are grateful to the anonymous reviewers for their detailed feedback on this paper. This work is partially supported by the US National Science Foundation under grant No. 2318970.

References

[1] H. Andrade and I. Crnkovic, "A review on software architectures for heterogeneous platforms," in *APSEC*. IEEE, 2018.

[2] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *TPDS*, 2020.

[3] Y.-H. Lai, E. Ustun, S. Xiang, Z. Fang, H. Rong, and Z. Zhang, "Programming and synthesis for software-defined fpga acceleration: status and future prospects," *TRETS*, 2021.

[4] S. Tan, Q. Jiang, Z. Cao, X. Hao, J. Chen, and H. An, "Uncovering the performance bottleneck of modern hpc processor with static code analyzer: a case study on kunpeng 920," *CCF Transactions on High Performance Computing*, 2024.

[5] S. Pargaonkar, "A comprehensive review of performance testing methodologies and best practices: software quality engineering," *IJSR*, 2023.

[6] G. Denaro, A. Polini, and W. Emmerich, "Early performance testing of distributed software applications," in *WOSP*, 2004.

[7] R. Mancini, T. Cucinotta, L. Abeni *et al.*, "Performance modeling in predictable cloud computing," in *CLOSER*, 2020.

[8] C. Li, Í. Goiri, A. Bhattacharjee, R. Bianchini, and T. D. Nguyen, "Quantifying and improving i/o predictability in virtualized systems," in *IEEE/ACM IWQoS*. IEEE, 2013.

[9] S. Kundu, R. Rangaswami, K. Dutta, and M. Zhao, "Application performance modeling in a virtualized environment," in *HPCA*. IEEE, 2010.

[10] H. Ha and H. Zhang, "DeepPerf: Performance prediction for configurable software with sparse DNN," in *ICSE*, 2019.

[11] P. Malakar, P. Balaprakash, V. Vishwanath, V. Morozov, and K. Kumaran, "Benchmarking machine learning methods for performance modeling of scientific applications," in *PMBS*. IEEE/ACM, 2018.

[12] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "egg: Fast and extensible equality saturation," *ACM on Programming Languages*, POPL 2021.

[13] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," *Logical Methods in Computer Science*, 2011.

[14] J. McClurg, M. Claver, J. Garner, J. Vossen, J. Schmerge, and M. E. Belviranli, "Optimizing regular expressions via rewrite-guided synthesis," in *PACT*. ACM, 2022.

[15] B. Adhikari, Y. Zhang, N. Ramakrishnan, and B. A. Prakash, "Sub2vec: Feature learning for subgraphs," in *Advances in Knowledge Discovery and Data Mining*, Cham, 2018.

[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE IISWC*, 2009.

[17] T. Yuki, "Understanding polybench/c 3.2 kernels," in *IMPACT Workshop*, 2014.

[18] Y. Zhang, Y. R. Wang, O. Flatt, D. Cao, P. Zucker, E. Rosenthal, and Z. Tatlock, "Better together: Unifying datalog and equality saturation," *PLDI*, 2023.

[19] M. Gao, Z. Li, R. Li, C. Cui, X. Chen, B. Ye, Y. Li, W. Gu, Q. Gong, X. Wang, and Y. Chen, "Easygraph: A multifunctional, cross-platform, and effective library for interdisciplinary network analysis," *Patterns*, 2023.

[20] A. Sohrabizadeh, Y. Bai, Y. Sun, and J. Cong, "Robust gnn-based representation learning for hls," in *ICCAD*, 2023.

[21] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather, "Programl: Graph-based deep learning for program optimization and analysis," 2020.

[22] E. Bendersky, "Github–eliben/pycparser: Complete c99 parser in pure python," 2019.

[23] O. Klaus-Dieter, "Intel advisor: Vectorization and roofline analysis," *SATG*, 2022.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, and D. Cournapeau, "Scikit-learn: Machine learning in Python," *JMLR*, 2011.

[25] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: online learning of social representations," in *SIGKDD*, 2014.

[26] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," *SIGKDD*, 2016.

[27] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *24th International Conference on World Wide Web*, 2015.

[28] F. Jiang, L. He, Y. Zheng, E. Zhu, J. Xu, and P. S. Yu, "On spectral graph embedding: A non-backtracking perspective and graph approximation," 2018.

[29] D. Wang, P. Cui, and W. Zhu, "Structural deep network embedding," *22nd ACM SIGKDD*, 2016.

[30] Y. Kim, P. Černý, and J. Dennis, "Performance search engine driven by prior knowledge of optimization," in *ARRAY*, 2015.

[31] M. Canesche, V. Rosário, E. Borin, and F. Quintão Pereira, "The droplet search algorithm for kernel scheduling," *ACM Trans. Archit. Code Optim.*, 2024.

[32] A. Sohrabizadeh, Y. Bai, and Y. Sun, "Enabling automated fpga accelerator optimization using graph neural networks," *ICCAD*, 2021.

[33] A. TehraniJamsaz, Q. I. Mahmud, L. Chen, N. K. Ahmed, and A. Jannesari, "Perfograph: A numerical aware program graph representation for performance optimization and program analysis," *NeurIPS*, 2023.

[34] R. Canizales, L. Mixco, and J. McClurg, "Parallelizing accelerographic records processing," in *ParSocial*, 2024.

[35] D. Hardin, J. Davis, D. Greve, and J. McClurg, "Development of a translator from LLVM to ACL2," in *ACL2*, 2014.

[36] I. Dagli, A. Cieslewicz, J. McClurg, and M. E. Belviranli, "Axonn: energy-aware execution of neural network inference on multi-accelerator heterogeneous socs," in *DAC*, 2022.

[37] J. McClurg, L. Z. Baker, R. Canizales, and D. Karki, "Towards synthesis of application-specific forward error correction (FEC) codes," in *HotNets*. ACM, 2024.

[38] J. McClurg, "Program Synthesis for Software-Defined Networking," PhD thesis, CU Boulder, Jan 2018.

[39] J. Protzenko, S. Burckhardt, M. Moskal, and J. McClurg, "Implementing real-time collaboration in touchdevelop using AST merges," in *MobileDeLi*. ACM, 2015.

[40] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, and C. Guestrin, "TVM: end-to-end optimization stack for deep learning," *CoRR*, 2018.