A MONOTONICITY OF PERTURBATION FUNCTION

Theorem 1. Π *is a monotonically decreasing function.*

Proof. Let $F_1, F_2 \in \wp(\mathcal{P})$ such that $F_1 \subseteq F_2$.

$$\begin{aligned}\Pi(F_1) =&\{\beta' \mid \beta' \in B,\\ &F_1 \subseteq \mathcal{P}_{\beta'}, \mathcal{P}_{\beta'} \setminus F_1 \text{ are obtained from } \mathcal{P} \setminus F_1\}\\ =&\{\beta' \mid \beta' \in B, F_2 \subseteq \mathcal{P}_{\beta'},\\ &\mathcal{P}_{\beta'} \setminus F_2 \text{ are obtained from } \mathcal{P} \setminus F_2\}\\ &\cup \{\beta' \mid \beta' \in B, F_1 \subseteq \mathcal{P}_{\beta'}, F_2 \not\subseteq \mathcal{P}_{\beta'},\\ &\mathcal{P}_{\beta'} \setminus F_1 \text{ are obtained from } \mathcal{P} \setminus F_1\}\\ =&\Pi(F_2) \cup \{\beta' \mid \beta' \in B, F_1 \subseteq \mathcal{P}_{\beta'}, F_2 \not\subseteq \mathcal{P}_{\beta'},\\ &\mathcal{P}_{\beta'} \setminus F_1 \text{ are obtained from } \mathcal{P} \setminus F_1\}\end{aligned}$$

Hence, $\Pi(F_2) \subseteq \Pi(F_1)$ □

Note that in the above proof, features in feature sets such as $\mathcal{P}_{\beta'} \setminus F_1$ are obtained by either retaining or perturbing the features in $\mathcal{P} \setminus F_1$.

A similar proof can be used to prove the monotonicity of $\hat{\Pi}$ as well.

B TYPES OF DATA DEPENDENCIES IN BASIC BLOCKS

While each instruction is processed sequentially by the different components of the CPU, an instruction $inst_j$ can get stalled due to the requirement for a previous instruction $inst_i$ to get completed, creating a *data dependency hazard* (Patterson and Hennessy, 1998). A Read-After-Write (RAW) data-dependency hazard arises when $inst_j$ reads the value in an operand that is written by $inst_i$. $inst_j$ can not get executed until $inst_i$ ends to ensure correct execution. A Write-After-Read (WAR) hazard occurs when $inst_j$ writes to an operand that is read by $inst_i$. A Write-After-Write (WAW) hazard arises when $inst_j$ writes to an operand that is written to by $inst_i$. There can be multiple data dependency hazards, possibly of different kinds, between a given pair of instructions.

C BASIC BLOCK PERTURBATION ALGORITHM

Algorithm 1 presents our stochastic perturbation algorithm Γ to conditionally perturb a given basic block β to β'. The perturbation algorithm creates the graph $\mathcal{G}'$ of β' while preserving a set of instructions/their corresponding vertices $\overline{\mathcal{V}}$, a set of data dependencies/their corresponding edges $\overline{\mathcal{E}}$ and possibly the number of instructions/the number of vertices, denoted by the boolean $preserve_\eta$ which is set to true when the number of instructions η is to be kept constant. If the number of vertices is to be kept constant, then the vertex/instruction deletion operation is forbidden [lines 1-1]. The vertices at the ends of the edges in $\overline{\mathcal{E}}$ are preserved as well [line 1] by adding them to $\overline{\mathcal{V}}$. Then each vertex of $\mathcal{G}$ is perturbed with a probability of $(1 - p_{I,ret})$ if it is not required to be retained [lines 1-1]. If the deletion perturbation operation is in vertexPertOps, then a vertex is deleted or replaced with probabilities of p_{del} and $(1 - p_{del})$ respectively. Otherwise, it is replaced with a valid vertex. The replacement of a vertex/corresponding instruction involves changing its opcode to another opcode that can take the original operands and still constitute valid x86 syntax according to the x86 Instruction Set Architecture. Similarly, each data-dependency edge is perturbed with a probability of $(1 - p_{D,ret})$ if it is not required to be retained [lines 1-1], to form $\mathcal{G}'$ [line 1]. The only perturbation of any data dependency is its deletion, which is conducted by the perturbation of the operands involved in the data dependency.

Algorithm 1 Basic Block Perturbation Algorithm

1: **Input:** basic block graph $\mathcal{G}$, vertices to preserve $\overline{\mathcal{V}}$, data-dependency edges to preserve $\overline{\mathcal{E}}$, $preserve_\eta$, $p_{I,ret}$, $p_{D,ret}$, p_{del}
2: **Output:** perturbed basic block graph, $\mathcal{G}'$
3: vertexPertOps = {replacement, deletion}
4: **if** $preserve_\eta$ **then**
5: vertexPertOps.remove({deletion})
6: **end if**
7: $\overline{\mathcal{V}} \leftarrow addVerticesForPreservedDeps(\overline{\mathcal{V}}, \overline{\mathcal{E}})$
8: **for** $v \in GetVertices(\beta)$ **do**
9: **if** $v \notin \overline{\mathcal{V}}$ **and** $rand([0,1]) > p_{I,ret}$ **then**
10: $v \leftarrow PerturbVertex(\mathcal{G}, v, vertexPertOps, p_{del})$
11: **end if**
12: **end for**
13: **for** $\varepsilon \in GetDepEdges(\beta)$ **do**
14: **if** $\varepsilon \notin \overline{\mathcal{E}}$ **and** $rand([0,1]) > p_{D,ret}$ **then**
15: $\varepsilon \leftarrow PerturbEdge(\mathcal{G}, \varepsilon)$
16: **end if**
17: **end for**
18: $\mathcal{G}' \leftarrow \mathcal{G}$

D CASE SPECIFICITY OF PERTURBATION PROBABILITIES

COMET's perturbation algorithm Γ consists of primarily 3 probability terms: $p_{I,ret}$, $p_{D,ret}$, and p_{del} as described in Appendix C. $p_{I,ret}$ and $p_{D,ret}$ are the probabilities of retention of a given instruction and a given data dependency respectively, in the perturbed basic block. p_{del} is the probability of deletion of an instruction when the deletion per-

turbation operation is allowed for instructions. The deletion perturbation operation will not be allowed for instructions when the number of instructions is to be kept constant.

Γ perturbs a basic block β by essentially perturbing every instruction while preserving certain tokens of the instruction from getting perturbed. These preserved tokens correspond to the features that are required to be preserved by Γ and also the features that Γ voluntarily does not attempt to perturb. Γ has voluntary retention of randomly selected basic block features to output perturbed basic blocks β' that are very similar to the original basic block β. Γ attempts to perturb the other tokens of β to obtain β'.

Γ can delete an instruction in case none of its tokens are required to be preserved. Otherwise, Γ replaces a token with another token that can form a basic block with valid x86 syntax alongside the other tokens. Thus, every token has a set of potential replacements. Perturbations to opcode tokens are counted as changes to the instruction features, while perturbations to the operand tokens are considered as changes to any data dependency features. As the perturbation space consists of only valid basic blocks, the overall probabilities of the primitive perturbation operations (instruction deletion, instruction replacement, and data dependency deletion) vary with the target basic block.

Following is an example of this variation. Several tokens of x86 assembly have no possible replacements resulting in no probability of replacement, such as the opcode lea. This is a special opcode that loads the effective memory address of its source operand into the destination register. There is no other x86 opcode that shows similar behavior. Hence, the lea can not be replaced with any other opcode. Such failed attempts at opcode replacement lead to the retention of the instruction, thus leading to an increase in the probability of retention of specific features of the basic block. This change in probabilities is specific to the basic blocks having the lea opcode in its instructions.

Another example of basic-block-specific probability settings occurs due to data dependencies. The data dependencies in a basic block can be varied with changes in just the opcodes of the corresponding instructions. Thus, while we keep the perturbation probability of a data dependency $(1 - p_{D,ret})$ to be 0.5 in the general case, it can vary with the basic block. A basic block having all the potential replacements for the opcodes involved in a data dependency with similar behavior as the original opcodes will have 0.5 probability of perturbation of the data dependency, while the opcodes for which we have potential replacements show variable behaviors, the data dependency perturbation probability can be more than 0.5. (Opcodes add and sub have similar behavior as they read the value in the source operand and read-write the value in the destination operand. They have different behavior from mov that reads the source operand value and writes to the destination operand. All 3 opcodes could be potential replacements for each other in instructions having certain pairs of operands.)

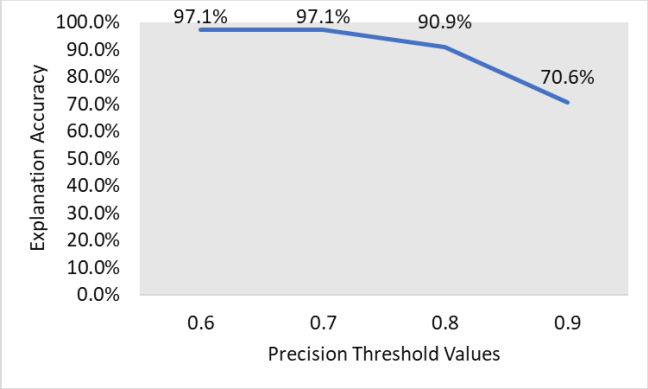

Figure 5. Variation in explanation accuracy with the precision threshold $(1 - \delta)$ setting in COMET

E Ablation and Sensitivity Studies

In this section, we study the variations in our results, with COMET's hyperparameters and design choices. We use our explanation accuracy-based evaluation scheme based on our crude but interpretable cost model that is presented in Section 6, to study the effects of the different hyperparameters and design choices. For this study, we have used the crude cost model for the Haswell microarchitecture. We have randomly selected 100 basic blocks from the BHive dataset (Chen et al., 2019) for which we generate COMET's explanations with different settings. We have dropped the error bars for clarity of the results, as we note from Table 2 that the standard deviations in our results are generally low.

E.1 Precision threshold

In this section, we study the variation in the explanations' accuracy with the precision threshold set in COMET, above which we consider the explanation feature set to be approximately faithful to the cost model's predictions. We want the precision threshold to be high such that the most precise and accurate explanations are given as output. Figure 5 presents the variation in the accuracy of COMET's explanations with various values for the precision threshold $(1-\delta)$ in COMET. We observe that 0.7 is the highest precision threshold that gives the highest accuracy and hence we have set it as the precision threshold in our experiments.

E.2 Perturbation probabilities for instructions

Γ attempts to perturb a given instruction $inst$ in a basic block β only when it is not required to be preserved. Γ retains $inst$ with a probability of $p_{I,ret}$ and perturbs it oth-

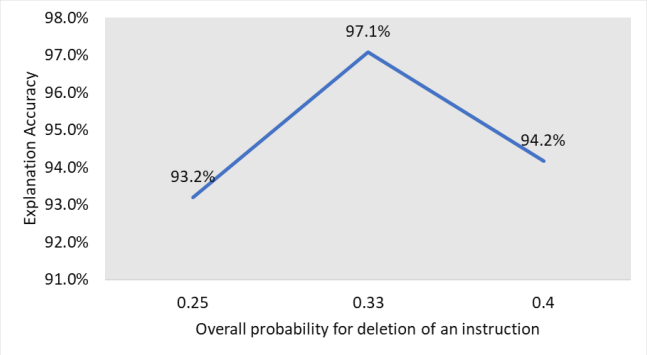

Figure 6. Variation in explanation accuracy with the probability of instruction deletion in Γ

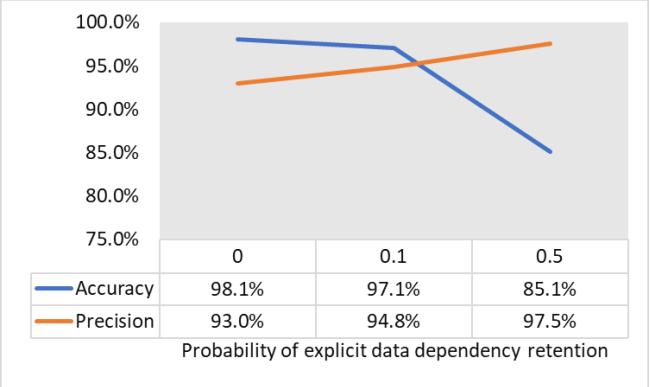

Figure 7. Variation in explanation accuracy and precision with the probability of explicit data dependency retention

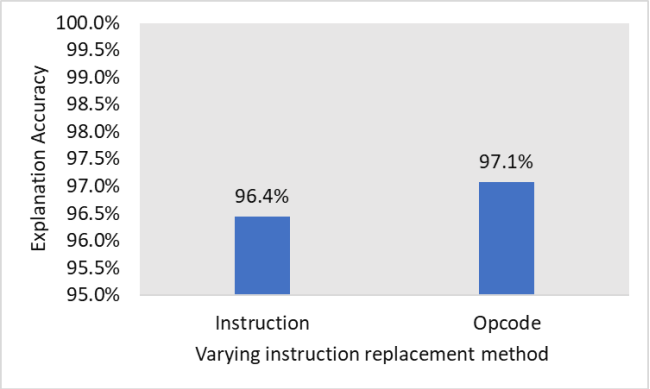

Figure 8. Variation in explanation accuracy with just opcode and whole instruction replacement schemes.

erwise. There are 2 potential operations for perturbing $inst$: Deletion and Replacement (with valid x86 instruction), each probabilities p_{del} and $(1 - p_{del})$ respectively. We have set $p_{del} = 0.33$ based on a sensitivity study that we conducted with respect to this hyperparameter, for all of our experiments. Figure 6 presents our findings. We find that our choice of $p_{del} = 0.33$ leads to the maximum accuracy among other candidates.

E.3 Perturbation probabilities for data dependencies

Similar to the case for instructions, Γ attempts to perturb a given data dependency δ in a basic block β with probability $(1 - p_{D,ret})$. As discussed in Section D, the exact probabilities of the retention/deletion of data dependencies are basic-block-specific. However, we vary these probabilities by varying the probability of explicit retention of a data dependency, i.e. the probability by which a data dependency will be retained for sure. This probability is a lower bound for $p_{D,ret}$ and higher values of this lower bound imply higher values for $p_{D,ret}$ for any given basic block. Figure 7 shows our findings. We have shown the variation in explanation precision as well, as we observe precision to have a trend different from explanation accuracy in this case. We find that a value of 0.1 for this probability parameter leads to optimum values for both explanation accuracy and precision. Thus, we have selected the explicit data dependency retention probability to be 0.1 in COMET.

E.4 Replacement of instructions

Γ considers only the changes to an instruction's opcode as changes to the feature corresponding to the instruction. However, another possibility could be to consider operand changes (such that their types and sizes are preserved) as well as changes to the instruction feature. We analyze the effects of the two instruction changing/replacement schemes in Figure 8. We observe that the accuracy of the explanations is higher with just the opcode replacement method, justifying our choice of this instruction replacement scheme.

An important hyperparameter that we have set according to our intuitive understanding is the ϵ error, which marks the radius of the ball of acceptable cost predictions around the prediction of cost model $\mathcal{M}$ for basic block β ($\mathcal{M}(\beta)$). For our crude cost model $\mathcal{C}$, we have kept ϵ to be a quarter of one unit of its cost prediction, as the least change in its cost prediction can be a quarter unit ($\frac{\Delta n}{4} = 0.25$). For the practical cost models such as Ithemal and uiCA, we have set ϵ as 0.5 cycles of throughput prediction, as that is the least, significant change in practically-useful throughput values.

F Perturbation function output sizes

The perturbation function, $\Pi_\beta : \wp(\mathcal{P}_\beta) \rightarrow \wp(\mathcal{B})$ maps a given set of basic block features $\mathcal{F}$ to the set of basic blocks $\mathcal{B}_\mathcal{F}$ that have $\mathcal{F}$ and where the other features are obtained from perturbations to the features in $\mathcal{P}_\beta \setminus \mathcal{F}$. In this section, we provide estimates of cardinalities of $\mathcal{B}_\mathcal{F}$ for some basic

blocks β and feature sets $\mathcal{F}$. With this analysis, we allude to the practical intractability of generating ideal black-box explanations for cost models.

Note that, as $\mathcal{P}_\beta$ is the set of all features (all basic features and all of their functions) of β, it can be an infinite set itself. $\hat{\mathcal{P}}_\beta \subset \mathcal{P}_\beta$, hence for $\mathcal{F} \subseteq \hat{\mathcal{P}}_\beta$, $\hat{\Pi}_\beta(\mathcal{F}) \subseteq \Pi_\beta(\mathcal{F})$. Hence, $|\hat{\Pi}_\beta(\mathcal{F})| \leq |\Pi_\beta(\mathcal{F})|$. Thus, we provide estimates for $|\Pi_\beta(\mathcal{F})|$ by reporting the rough values for $|\hat{\Pi}_\beta(\mathcal{F})|$.

First, consider the basic block β_1 in Listing 4, for $\mathcal{F} = \emptyset$. $|\hat{\Pi}_{\beta_1}(\emptyset)| \approx 1.94 \times 10^{38}$. As we add more elements to $\mathcal{F}$, the size of $|\hat{\Pi}_{\beta_1}(\mathcal{F})|$ will reduce due to the constraints introduced to the perturbations.

```
1  vdivss xmm0, xmm0, xmm6
2  vmulss xmm7, xmm0, xmm0
3  vxorps xmm0, xmm0, xmm5
4  vaddss xmm7, xmm7, xmm3
5  vmulss xmm6, xmm6, xmm7
6  vdivss xmm6, xmm3, xmm6
7  vmulss xmm0, xmm6, xmm0
```

Listing 4. Basic block β_1 for perturbation function size estimation

Next, for $\mathcal{F} = \{inst_1\}$ i.e. with no perturbations to instruction 1 in β_1, $|\hat{\Pi}_{\beta_1}(\mathcal{F})| \approx 6.58 \times 10^{29}$.

Similarly, consider the basic block β_2 in Listing 5, for $\mathcal{F} = \emptyset$. $|\hat{\Pi}_{\beta_2}(\emptyset)| \approx 1.63 \times 10^{32}$. For $\mathcal{F} = \{inst_2\}$ i.e. with no perturbations to instruction 2 in β_2, $|\hat{\Pi}_{\beta_2}(\mathcal{F})| \approx 2.77 \times 10^{28}$.

```
1  shl eax, 3
2  imul rax, r15
3  xor edx, edx
4  add rax, 7
5  shr rax, 3
6  lea rax, [rbp + rax - 1]
7  div rbp
8  imul rax, rbp
9  mov rbp, qword ptr [rsp + 8]
10 sub rbp, rax
```

Listing 5. Basic block β_2 for perturbation function size estimation

Thus, we find that the perturbation function's output set can have very high cardinality, posing a challenge for generating desirable explanations.

G Crude interpretable cost model details

We define $cost_{inst}(inst)$ as the throughput of the instruction $inst$ on actual hardware. We obtain the throughputs of instructions over actual hardware from https://www.uops.info/table.html. We define $cost_{dep}(\delta_{ij})$ as in (10). Our intuition behind keeping the costs of WAR and WAW type of dependencies to be 0 is that these dependencies are not true dependencies and can be generally resolved by the compiler by register renaming (Patterson and Hennessy, 1998). The RAW data dependency, on the other hand, is a true dependency. As the two instructions forming a RAW dependency will be executed sequentially on hardware, the addition of their individual costs would be a good proxy for the actual throughput cost brought in by the data dependency.

$$cost_{dep}(\delta_{ij}) = \begin{cases} 0, & \delta_{ij} = \text{WAR/WAW} \\ cost_{inst}(inst_i) + cost_{inst}(inst_j), & \delta_{ij} = \text{RAW} \end{cases} \tag{10}$$

We define the $cost_\eta(n) = \eta/4$ as the cost for having n number of instructions (denoted by η) in a given basic block β. We derive the expression for the cost of number of instructions from the simple baseline model presented in (Abel and Reineke, 2022).

Our choice of $\mathcal{C}$ is microarchitecture-specific as the costs of individual instructions vary across microarchitectures. We have developed $\mathcal{C}$ models for the Haswell and Skylake microarchitectures, only for the purposes of evaluating COMET's explanations.

H Studied dataset and cost models

H.1 BHive dataset

BHive dataset[1] (Chen et al., 2019) is a benchmark suite of x86 basic blocks. It contains roughly 300,000 basic blocks annotated with their average throughput over multiple executions on actual hardware for 3 microarchitectures: Haswell, Skylake, and Ivy Bridge. We have generated explanations for basic blocks in this dataset.

The dataset can be partitioned by 2 criteria: by *source* and by *category* of its basic blocks. Partition by source annotates each block with the real-world code base from which it has been derived. Examples of BHive sources are Clang and OpenBLAS. Partition by category annotates each basic block by its type, characterized by the semantics of the instructions in the block. There are 6 types of blocks: Scalar, Vector, Scalar/Vector, Load, Store, and Load/Store.

H.2 Ithemal

Ithemal[2] (Mendis et al., 2019a) is an ML-based cost model, which predicts the throughput of input x86 basic blocks for

[1] https://github.com/ithemal/bhive
[2] https://github.com/ithemal/Ithemal

a given microarchitecture. It is open-source and is currently trained for the Haswell, Skylake, and Ivy Bridge microarchitectures on the BHive dataset. A separate instance of Ithemal needs to be trained for every microarchitecture, due to the difference in the actual throughput values obtained over different hardware. Ithemal's throughput prediction is a floating point number, as it is trained on the BHive dataset.

Ithemal consists of a hierarchical multiscale RNN structure. The first RNN layer takes embeddings of tokens of the input basic block and combines them to create embeddings for the instructions in the basic block. The second RNN layer takes the instruction embeddings as input and combines them to create an embedding for the basic block. The basic block embedding is passed through a linear regressor layer to compute the throughput prediction for the basic block.

Ithemal exhibits roughly 9% Mean Absolute Percentage Error for the Haswell microarchitecture on the BHive dataset. As Ithemal outputs only its throughput prediction and no insights into why the prediction was made, it can not be reliably deployed in mainstream compiler optimizations.

H.3 uiCA

uiCA[3] (Abel and Reineke, 2022) is an analytical simulation-based cost model for several latest microarchitectures released by Intel over the last decade. uiCA's simulation model is hand-engineered to accurately match the model of each Intel microarchitecture and must be manually tuned to reflect new microarchitectures. It can output detailed insights into its process of computing its throughput prediction of input x86 basic blocks, such as where in the CPU's pipeline its simulator identified a bottleneck for the execution of the basic block.

[3] https://github.com/andreas-abel/uiCA