AnICA: Analyzing Inconsistencies in Microarchitectural Code Analyzers

FABIAN RITTER, Saarland University, Germany
SEBASTIAN HACK, Saarland University, Germany

Microarchitectural code analyzers, i.e., tools that estimate the throughput of machine code basic blocks, are important utensils in the tool belt of performance engineers. Recent tools like llvm-mca, uiCA, and Ithemal use a variety of techniques and different models for their throughput predictions. When put to the test, it is common to see these state-of-the-art tools give very different results. These inconsistencies are either errors, or they point to different and rarely documented assumptions made by the tool designers.

In this paper, we present AnICA, a tool taking inspiration from differential testing and abstract interpretation to systematically analyze inconsistencies among these code analyzers. Our evaluation shows that AnICA can summarize thousands of inconsistencies in a few dozen descriptions that directly lead to high-level insights into the different behavior of the tools. In several case studies, we further demonstrate how AnICA automatically finds and characterizes known and unknown bugs in llvm-mca, as well as a quirk in AMD's Zen microarchitectures.

CCS Concepts: • **Software and its engineering** → *Correctness*; *Software verification and validation*; **Software testing and debugging**; • **Theory of computation** → **Abstraction**.

Additional Key Words and Phrases: Throughput Prediction, Basic Blocks, Abstraction, Differential Testing

ACM Reference Format:
Fabian Ritter and Sebastian Hack. 2022. AnICA: Analyzing Inconsistencies in Microarchitectural Code Analyzers. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 125 (October 2022), 29 pages. https://doi.org/10.1145/3563288

1 INTRODUCTION

Making software run faster has always been a major goal for programmers as well as for computer science research. To make software run as fast as possible, we need to have an understanding of how fast some code will execute on a given machine. Recently, research has seen a rise of interest in such an understanding at the lowest level: estimating the throughput of CPU-bound loop-free instruction sequences. This is witnessed by the wide range of microarchitectural code analyzers that give such instruction throughput estimates, e.g., llvm-mca [Di Biagio 2018], uiCA [Abel and Reineke 2022], OSACA [Laukemann et al. 2018], IACA [Intel 2012], CQA [Rubial et al. 2014], Ithemal [Mendis et al. 2019], and DiffTune [Renda et al. 2020]. While these tools vary in the methods they employ – compiler scheduling models, microbenchmarks, or machine learning – they share the goal of estimating the throughput of basic blocks on a given processor.

However, these tools vary significantly in their predictions and deliver inconsistent results. Figure 1 shows the results of an experiment in which we randomly generated 10,000 basic blocks

Authors' addresses: Fabian Ritter, Saarland University, Saarland Informatics Campus, Saarbrücken, Germany, fabian.ritter@cs.uni-saarland.de; Sebastian Hack, Saarland University, Saarland Informatics Campus, Saarbrücken, Germany, hack@cs.uni-saarland.de.

2475-1421/2022/10-ART125
https://doi.org/10.1145/3563288

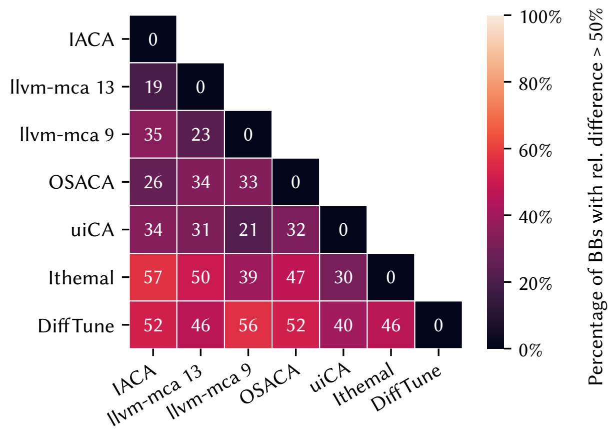


Fig. 1. Heat map showing the percentage of basic blocks with throughput estimates that deviate by more than 50% for each pair of predictors.

consisting of 4 instructions each[1] and let several throughput predictors give their estimate for these blocks assuming the Intel Haswell[2] microarchitecture. For each pair of throughput predictors, the heat map contains an entry indicating the percentage of basic blocks for which the throughput estimates deviated by more than 50% of their average.

Overall, all pairs of predictors exhibit substantial numbers of inconsistencies. As we can see from the inconsistencies in different versions of llvm-mca (23% of the basic blocks in Figure 1 are predicted inconsistently between llvm-mca versions 9 and 13), even closely related implementations are affected. There may be several reasons for these deviations, e.g.:

- The different performance models might fail to capture relevant parts of the execution.
- The tools might be built with different (implicit or explicit) assumptions.
- The learning-based tools might need more training data.
- The implementations might contain bugs.

In any of these cases, finding and characterizing such inconsistencies is valuable. If the cause is unintentional, they can help to improve the tools. If the inconsistencies are the result of deliberate choices of the developers, identifying them helps us to explore the limits of the tools. Therefore, the goal of this work is to automatically discover inconsistencies in the results of instruction throughput predictors and to give insight into their causes.

Our approach, AnICA, applies differential testing [McKeeman 1998] to a pair of basic block throughput predictors. The core idea is to randomly sample inputs and compare the outputs of the tools under investigation. If the tools do not agree, we found an inconsistency. Similar techniques are used in a variety of domains, prominently for compilers [McKeeman 1998], SSL/TLS certificate validators [Brubaker et al. 2014; Chen and Su 2015], but also for software components close to the processor like instruction decoders [Jay and Miller 2018; Paleari et al. 2010; Woodruff et al. 2021].

The existing approaches do however not transfer well to basic block throughput predictors since those provide an unusual setting for differential testing: the above heat map shows that finding inputs that exhibit inconsistencies is not difficult; elaborate methods for searching the input space are therefore not necessary. However, just listing the large amount of inconsistencies we find would

[1]A basic block is generated by individually sampling instructions from the machine-readable x86-64 ISA description from uops.info and instantiating them with valid operands.
[2]Haswell is the only microarchitecture supported by all compared tools.

also not be very helpful to understand and improve the tools under test. The focus of AnICA is therefore to find compact characterizations of large classes of inputs that cause inconsistencies. We apply concepts from abstract interpretation [Cousot and Cousot 1977] to find these compact characterizations and present them together with witnesses for their derivation. These witnesses give insights in two directions:

- They contain examples of represented basic blocks that exhibit inconsistencies and
- they show the boundaries of the problem through similar basic blocks that do not exhibit inconsistencies.

For many of the combinations of tools shown in Figure 1, ten of AnICA's discoveries are sufficient to characterize more than half of the several thousand encountered inconsistencies. We investigate results of AnICA in case studies showing that the results are helpful for improving performance models in several ways: to find modeling bugs and regressions from one tool version to the next, to understand differing modeling assumptions, and to identify underrepresented constructs in the training sets of learned predictors.

In summary, we make the following contributions:

- A novel algorithm based on differential testing and using concepts from abstract interpretation to find compact characterizations of inputs that cause inconsistencies in basic block throughput predictors, presented in Section 3.
- A modular and extensible implementation of this algorithm targeting throughput predictors for the x86-64 instruction set architecture that we evaluate in Section 4.
- Our case studies show how AnICA exposes subtle modeling differences between the tools, identifies a long-standing crash in llvm-mca with a two-instruction test case, and characterizes a series of inaccuracies in llvm-mca's model for the AMD Zen+ microarchitecture. In this process, AnICA even finds an unusual quirk in the Zen+ microarchitecture itself. (Section 5)

2 BACKGROUND: PREDICTING THROUGHPUT

There is a wide range of approaches that estimate the performance of programs. They differ in the kind of input they expect – from short sequences of machine instructions [Intel 2012] to entire programs [Binkert et al. 2011] – as well as the accuracy they strive for – from cycle-accurate [Böhm et al. 2010] to "back-of-the-envelope calculations" [Ofenbeck et al. 2014; Williams et al. 2009].

This work analyzes tools at a specific point of this spectrum: low-level throughput predictors for *basic blocks*, i.e., short sequences of machine instructions without control flow. Throughput here means the sustained rate at which the basic block can be executed infinitely often, either as instructions/iterations per clock cycle or as number of cycles required to execute one instance. These tools typically aim to be close to cycle-accurate, but cannot do an exact simulation of the target hardware. Instead, they use a model of the hardware that is built either from proprietary knowledge about the hardware [Intel 2012], from the scheduling models of a compiler [Di Biagio 2018], through micro-benchmarking [Abel and Reineke 2022], or via machine learning [Mendis et al. 2019; Renda et al. 2020].

Common assumptions for these tools are that all memory accesses hit the L1 cache and that execution of the basic block is in a steady state (it is the body of an innermost loop that is executed indefinitely often). With these properties, the main domain of application for such throughput predictors is in the optimization of short, very hot code regions in programs where performance is crucial.

Even with these common assumptions, the task of predicting the throughput for a given basic block is not easy. Modern processors split instructions into undocumented micro operations ("μops") and reorder them as freely as the data dependencies allow. Numerous undocumented buffers and execution units make the processor fast, but they also impede accurate throughput estimation.

Moreover, there is often not a single well-defined throughput for a given basic block on a microarchitecture [Abel and Reineke 2022]:

- On many microarchitectures, the execution time of some instructions depends on their input values.
- Basic blocks might contain data dependencies if certain input values are pointers to the same memory location.
- Depending on whether the basic block is repeated through a loop or through concatenating many copies, different bottlenecks determine the throughput.

If a tool wants to predict the throughput of arbitrary basic blocks, it needs to assume behaviors for all such points. Very often, basic block throughput predictors do not come with an explicit statement of these assumptions. This makes it difficult to judge which tool is better suited for which task: If two tools are based on different assumptions, we cannot just compare their accuracy with respect to a common ground truth since such a ground truth needs to depend on the chosen assumptions.

In the following section we describe how AnICA compares basic block throughput predictors directly without the need for a ground truth.

3 THE ANICA ALGORITHM

On a high level, AnICA follows the structure of differential testing [McKeeman 1998]: an AnICA *campaign* searches for inconsistencies between a fixed pair of throughput predictors and reports them as *discoveries*. We generate valid input basic blocks, give them to both tools under investigation, and compare their results. The throughput predictors are required to support a common instruction set architecture (ISA), i.e., they need to have compatible input formats. Given a basic block, they should output a real-valued estimate for the number of cycles required for its execution or report an error.

Differential testing is a natural fit to overcome the problems when comparing basic block throughput predictors described in Section 2. No assumptions need to be made about the ground truth. Differences in the assumptions of the tools under investigation are visible as inconsistencies for basic blocks that are affected by these assumptions.

Valuable insight can be gained from comparing the results of a throughput predictor to measurements on the actual hardware rather than other predictors. AnICA naturally supports this, by using a microbenchmarking tool as one of the tools under investigation.[3] However, even if microbenchmarking is very close to the actual hardware, it still makes several assumptions that may not hold when running actual code, for instance by initializing registers and memory with specific values. Therefore, we use the perspective of differential testing for such comparisons with hardware measurements to acknowledge that the involved microbenchmarking tool is also influenced by assumptions and not a definitive ground truth.

As shown in Section 1, a key challenge for AnICA is that inconsistencies are so common in the random samples that just reporting all basic blocks with inconsistencies leads to an impractical number of reports. Therefore, we center the algorithm around the idea of *abstract basic blocks*, or *abstract blocks* for short: compact representations characterizing sets of basic blocks by common properties. AnICA aims for several goals to make the reported abstract blocks useful:

- The reported basic blocks should be *concise*, i.e., not contain instructions that are irrelevant to the underlying problem. This makes them easy to interpret.

[3]We explore this in a case study in Section 5.3.

```
discoveries ← {};
while termination condition not reached do
    candidate ← sampleBB() ;                                  // Section 3.5
    if candidate is not interesting then                      // Section 3.1
        continue;
    minBB ← minimize(candidate);
    if any d ∈ discoveries subsumes minBB then
        continue ;                                            // Section 3.6
    newDisc ← generalize(minBB) ;                             // Section 3.3
    discoveries ← discovery ∪ {newDisc};
return filterSubsumed(discoveries) ;                          // Section 3.6
```

Alg. 1. Discovering Inconsistencies.

- Each discovery should be *general* by representing as many relevant basic blocks as possible. The more general the discoveries are, the fewer of them need to be inspected.
- The discoveries should be *pertinent*, i.e., not represent basic blocks that do not exhibit inconsistencies in the tools under investigation. Significant numbers of such cases would make the characterization unreliable.

Since the results of AnICA are used to hint at existing problems or show limitations of the tools – rather than, e.g., proving the absence of inconsistencies – none of these goals are strict formal requirements. This fact allows us to employ approximations rather than heavy formal machinery at several points in the following sections.

AnICA's high-level structure, serving as a table of contents for the remainder of this section, is shown in Algorithm 1. We randomly sample a basic block and check whether it is *interesting*, i.e., if the throughput predictors under test exhibit an inconsistency (ll.3-5). We minimize interesting basic blocks (l.6) by greedily removing as many instructions as we can while keeping the block interesting. If the minimized basic block is already represented by a previously discovered abstract block, we do not need to further investigate it (ll.7-8). Otherwise, the basic block is generalized to an abstract block, which is then noted as a new discovery (ll.9-10).

We repeat this process until some termination condition is reached (l.2), e.g., a time budget is expired or a number of subsequent samples did not produce new discoveries. Finally, we check for each discovered abstract block a whether there is a subsequent one whose represented basic blocks include all of a. Such subsumed discoveries provide only redundant information and are therefore filtered from the results (l.11).

In the following subsections, we describe the components of Algorithm 1 in detail as indicated in the comments.

3.1 Interestingness Metric

Not every difference in the output of the tools under investigation is relevant. Since they predict real-valued average execution times based on vastly different models, small deviations are to be expected. Therefore, we use a more refined definition of interestingness:

A basic block is *interesting* if it causes a tool under investigation to crash, or if the relative difference between their predictions $pred_a$ and $pred_b$ exceeds a specified threshold:

$$\frac{|pred_a - pred_b|}{avg(pred_a, pred_b)} = \frac{|pred_a - pred_b| \cdot 2}{pred_a + pred_b} > threshold$$

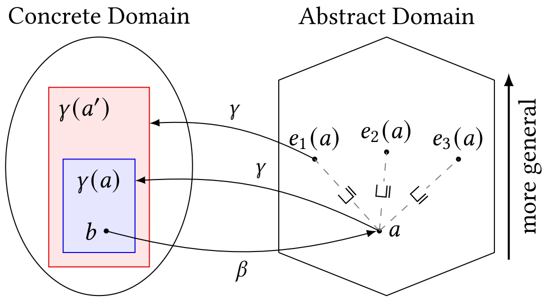


Fig. 2. Relationships between elements of concrete and abstract domain.

As we cannot assume any of the predictions to be the "correct" one, this definition normalizes the absolute difference between the predictions by their arithmetic mean. The interestingness threshold is a parameter of the algorithm that influences what inconsistencies are found.

Other definitions of interestingness are conceivable and may be useful. For example, our AnICA implementation also provides an alternative metric based on the absolute difference:

$$|pred_a - pred_b| > threshold$$

Which metric is the most suitable depends on the inconsistencies we are searching for. The relative difference is effective for focusing on interesting inconsistencies when the predicted numbers of cycles grow larger. With the absolute difference metrics, it is easier to investigate inconsistencies of a few cycles for short-running basic blocks.

3.2 Basic Block Abstraction

We describe our representation of sets of basic blocks as an instance of abstract interpretation [Cousot and Cousot 1977]. Abstract interpretation is a technique commonly used in static program analysis. It provides a theoretical framework to over-approximate the set of possible behaviors of a program. A key insight of the technique is to represent subsets of conceivable program behaviors (denoted as elements of the *concrete domain*) by elements of an *abstract domain*. This is beneficial since the concrete domain of sets of program behaviors is generally too large to work with, whereas the abstract domain can be compact.

In AnICA, we apply this notion of abstraction to a different application domain: instead of program behaviors, we abstract basic blocks. Therefore, our concrete domain contains sets of basic blocks and the abstract domain represents features of these basic blocks such as instruction mnemonics, use of memory, and operand dependencies.

Formally, our concrete domain $\mathbb{C}$ is the power set of the set $\mathbb{B}$ of sequences of instructions in an ISA:

$$\mathbb{C} := \mathcal{P}(\mathbb{B}) \qquad \text{with } \mathbb{B} := \mathsf{Insns}^+$$

An abstract domain $\mathbb{A}$ is a set with a partial order $\sqsubseteq$ that relates domain elements by their generality: if an element a is larger than another b, it represents at least all elements of the concrete domain that b represents. While the abstract domain may be infinite, we do not allow it to have infinite sequences of strictly more general elements.

As usual in abstract interpretation, the AnICA algorithm works independently of the specific abstract domain. The abstract domain only needs to relate to the concrete domain via two functions: a *concretization function* $\gamma : \mathbb{A} \to \mathbb{C}$ and a *representation function* $\beta : \mathbb{B} \to \mathbb{A}$.

The concretization function γ maps each element of the abstract domain to a set containing all basic blocks that it represents. Conversely, the representation function β maps single basic blocks to a representation in the abstract domain.[4] Figure 2 visualizes these functions and the constraints imposed on them to constitute an abstract domain:

$$\forall b \in \mathbb{B}.\ b \in \gamma(\beta(b)) \tag{1}$$

$$\forall a, a' \in \mathbb{A}.\ a \sqsubseteq a' \Rightarrow \gamma(a) \subseteq \gamma(a') \tag{2}$$

These constraints ensure that the functions are consistent with the orders of the domains: Equation (1) ensures that a basic block b is part of the concretization of its representative $\beta(b)$. Equation (2) requires that γ is monotone w.r.t. the domain orders, i.e., that if one abstract block is more general than another, it represents more concrete basic blocks.

While β is often straightforward to implement, γ is unwieldy: if implemented explicitly, it would need to produce very large sets of concrete basic blocks. To avoid this problem, abstract domains in AnICA do not come with an explicit concretization function, but with a *concretization sampler* $\widetilde{\gamma}(a)$ that randomly samples a basic block from $\gamma(a)$.

Our algorithm does not require an explicit generality relation $\sqsubseteq$ either. Instead, we use a set $\mathit{Exps} \subseteq \mathbb{A} \rightharpoonup \mathbb{A}$ of partial *expansion* functions that each map abstract blocks to their immediate successors in the generality relation.

In practice, these expansion functions each describe a way to modify abstract blocks in order to obtain a slightly more general abstract block. Formally, each expansion function $E \in \mathit{Exps}$ needs to be strictly ascending and monotonic:

$$\forall a \in \mathsf{dom}(E).\ a \sqsubseteq E(a) \land a \neq E(a)$$
$$\forall a, a' \in \mathsf{dom}(E).\ a \sqsubseteq a' \Rightarrow E(a) \sqsubseteq E(a')$$

If there is an immediate successor a' to a in the generality order, there should be an expansion function $E \in \mathit{Exps}$ such that $E(a) = a'$. However, we require that for each abstract block a, the number of expansion functions $E \in \mathit{Exps}$ such that $a \in \mathsf{dom}(E)$ is finite.

Example 1. For an informal notion of what an abstract block for the x86-64 instruction set architecture looks like, consider the following example:

Instructions:
(1) mnemonic: `mov`; mem(ory usage): R (= Read)
(2) mnemonic: `add`; category: arithmetic; mem: R+W
Aliasing:
- operand 1 of insn 1 must alias with operand 2 of insn 2

(AB1)

This abstract block represents all basic blocks consisting of two instructions that satisfy constraints on their mnemonics, their category, their use of memory, and the aliasing of their operands.[5] The following is one of the concrete basic blocks represented by the above abstract block:

```
mov rbx, [rdx + 42]
add [r8], rbx
```

[4]We require β instead of the more common *abstraction function* $\alpha : \mathbb{C} \to \mathbb{A}$ since it tends to be easier to define and since our generalization algorithm only ever needs to abstract single concrete basic blocks. With an abstraction function α, β could be defined as $\beta(x) = \alpha(\{x\})$.

[5]We use the term "alias" here for instruction operands that refer to the same data. It is therefore not restricted to memory operands, but also refers to (fully or partially) overlapping register operands.

```
**Input:** basic block $b$
1  $absBB \leftarrow \beta(b)$;
2  if $absBB$ is not interesting then return $b$;
3  $rejected \leftarrow \{\}$;
4  while $True$ do
5      $avail \leftarrow \{E \in Exps \mid absBB \in \mathrm{dom}(E)\} \setminus rejected$;
6      if $avail = \{\}$ then return $absBB$ ;
7      $exp \leftarrow choose(avail)$;
8      $t \leftarrow exp(absBB)$;
9      if $t$ is interesting then $absBB \leftarrow t$ ;
10     else $rejected \leftarrow rejected \cup \{exp\}$ ;
```

Alg. 2. Generalization Algorithm.

The mnemonics fit their constraints, the first instruction only reads memory (at the location `rdx` + 42), whereas the second one reads and writes memory at `r8`. The aliasing constraint is satisfied by using the common `rbx` register.

An expansion function could for example drop the constraint on the mnemonic of the second instruction. The result of this expansion function is the following abstract block:

```
Instructions:
  (1) mnemonic: mov; mem: R
  (2) category: arithmetic; mem: R+W
Aliasing:
  • operand 1 of insn 1 must alias with operand 2 of insn 2
```

It represents all basic blocks represented by the previous one, but is less specific: all arithmetic instructions are now allowed as the second instruction.

Next, we describe how AnICA automatically generalizes interesting basic blocks to concise and pertinent abstract blocks. With this generalization algorithm in mind, we then formalize the details of a modular abstract domain for the x86-64 instruction set architecture in Section 3.4.

3.3 Generalization Algorithm

AnICA generalizes an interesting basic block b as shown in Algorithm 2. The first result candidate is the representative $\beta(b)$, an abstract block that represents the given basic block b as specifically as possible in the abstract domain (l.1). After validating that the initial candidate is interesting (l.2), we choose an expansion to make the candidate more general (ll.7-8). If the expanded abstract block is still interesting, we use it as a new candidate (l.9). Otherwise, we note this expansion as rejected (l.10) and choose a different one. As expansions are monotonic and ascending, a once rejected expansion cannot be useful later in generalization.

Once no expansion is left, we return the now general and still pertinent candidate (ll.5-6). Termination is guaranteed as the abstract domain has no infinite ascending chains and the set of expansions that apply to an abstract block is finite.

We extend our definition of interestingness (Section 3.1) from basic blocks to abstract blocks for this algorithm. Ideally, an abstract block should be deemed interesting if all represented basic blocks are interesting. As this is prohibitively expensive to check, we approximate this property: we randomly sample represented basic blocks and consider the abstract block interesting if all samples are interesting. The number of samples is a parameter of AnICA.

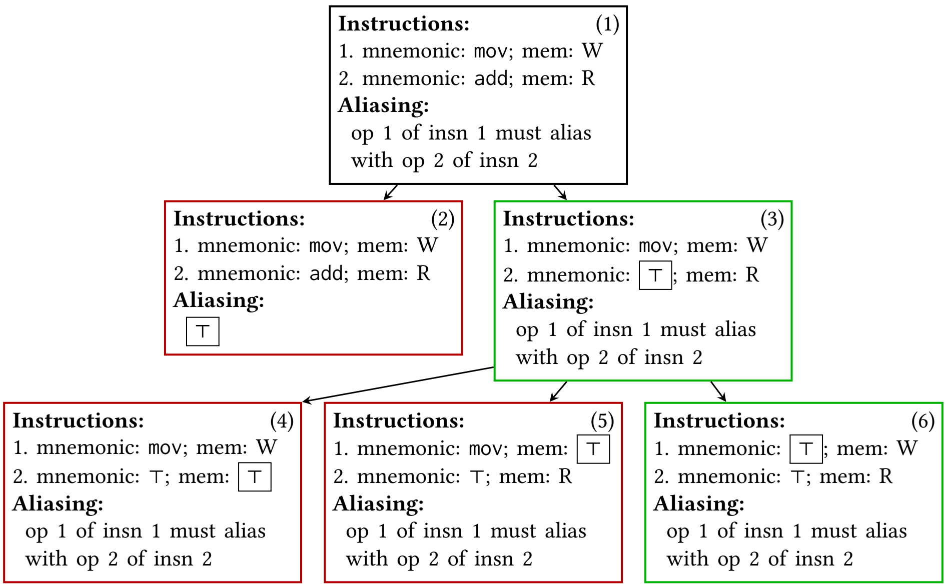


Fig. 3. An example generalization tree.

Example 2. Assume that the predictors under test disagree on the latency of reading a value from memory that was written immediately before. Figure 3 visualizes a run of the generalization algorithm for this problem.

The first hypothesis for an abstract block (1) is the representation of a concrete basic block exhibiting this behavior. In the next step, the algorithm expands the aliasing requirement and reaches abstract block (2). With ⊤, we denote that the component is unconstrained. Since the sampled basic blocks are then no longer restricted to using the same memory location, they are not uniformly interesting, which causes this expansion to be rejected. When we expand the mnemonic of the second instruction (3), the abstract block continues to only cover interesting basic blocks. Allowing any of the instructions to not use memory (4,5) leads to more rejections. Finally, this leaves only the mnemonic of the first instruction to be expanded (6). After that, all components of the abstract blocks are either ⊤ or only affected by rejected expansions. Hence, the algorithm terminates returning abstract block (6).

The order in which expansions are chosen in the generalization algorithm (line 7 in Algorithm 2) affects the result. We approximate an optimal expansion order by generalizing each candidate several times with different random expansion orders. Since we prune subsumed discoveries from the results, we can try arbitrarily many different expansion orders without degrading the quality of our discovery results.

The straightforward nature of the generalization algorithm is helpful when interpreting AnICA's results. As Parnin and Orso [2011] noted, tools that automatically find bugs and present them only abstractly to users do not necessarily help fix the bugs. They formulate the observation that "[p]roviding overviews that cluster results and explanations that include data values [and] test case information [. . .] could make faults easier to identify and tools ultimately more effective." [Parnin and Orso 2011, Section 6.1]

Our generalization algorithm naturally produces such clustered results and explanations in the form of a generalization decision tree like the one in Figure 3. Each decision in this tree is justified by the set of basic blocks that was sampled and evaluated to gauge their interestingness. Our implementation of AnICA therefore includes a graphical interface to inspect the generalization trees of its discoveries to provide users with detailed information and concrete debuggable inputs.

Particularly insightful are the basic blocks that justify the rejection of an expansion. They highlight the limits of an inconsistency's scope in a way that a plain clustering of inconsistent basic blocks could not. We will demonstrate in Section 4 how we can use such results to identify behaviors of throughput predictors that run counter to common expectations.

3.4 Our Abstract Domain

We now define an abstract domain for the x86-64 instruction set architecture as this ISA is supported by most available basic block throughput predictors. The AnICA algorithm is not conceptually limited to this ISA and similar domains can be designed for, e.g., ARM architectures.

Top-Level Abstraction. Our abstract domain $(\mathbb{A}, \sqsubseteq_{\mathbb{A}})$ separates constraints on the individual instructions of the represented basic blocks from constraints on how they interact via their operands. An abstract block thus consists of a sequence of abstract instructions and an abstract alias information:

$$\mathbb{A} := \mathbb{A}_{in}^{+} \times \mathbb{A}_{al}$$

The partial order $\sqsubseteq_{\mathbb{A}}$ only relates abstract blocks with the same number of abstract instructions and is defined through partial orders $\sqsubseteq_{in}$ and $\sqsubseteq_{al}$ among its components:

$$\begin{aligned}(x_{in}, x_{al}) \sqsubseteq_{\mathbb{A}} (y_{in}, y_{al}) :\Leftrightarrow\ & x_{al} \sqsubseteq_{al} y_{al} \wedge |x_{in}| = |y_{in}| \\ & \wedge (\forall 1 \leq k \leq |x_{in}|.\ x_{in}[k] \sqsubseteq_{in} y_{in}[k])\end{aligned}$$

In a similar way, the concretization $\gamma_{\mathbb{A}}$ and representation $\beta_{\mathbb{A}}$ functions are defined based on per-component-functions:

$$\begin{aligned} b \in \gamma_{\mathbb{A}}((a_{in}, a_{al})) &:\Leftrightarrow b \in \gamma_{al}(a_{al}) \wedge |a_{in}| = |b| \wedge (\forall 1 \leq k \leq |a_{in}|.\ b[k] \in \gamma_{in}(a_{in}[k])) \\ \beta_{\mathbb{A}}(b) &:= ([\beta_{in}(i) \mid i \in b], \beta_{al}(b)) \end{aligned}$$

Expansion functions expand one component of the abstract block, i.e., an abstract instruction or the aliasing abstraction, and leave all other components untouched:

$$\begin{aligned} \mathit{Exps}_{\mathbb{A}} :=& \left\{\lambda(a_{in}, a_{al}).\ (a_{in}[k \mapsto E(a_{in}[k])], a_{al}) \text{ if } k \leq |a_{in}| \wedge a_{in}[k] \in \mathsf{dom}(E) \,\middle|\, k \in \mathbb{N}, E \in \mathit{Exps}_{in}\right\} \\ & \cup \left\{\lambda(a_{in}, a_{al}).\ (a_{in}, E(a_{al})) \text{ if } a_{al} \in \mathsf{dom}(E) \,\middle|\, E \in \mathit{Exps}_{al}\right\} \end{aligned}$$

Instruction Abstraction. The set $\mathbb{A}_{in}$ contains abstract instructions that describe sets of *instruction schemes*. An instruction scheme (or instruction variant) is an instruction representation that is parametric in its operands: it specifies the width and kind of the operands, but does not specify an actual register, memory operand, or immediate value. For example, the instruction scheme

```
add ⟨GPR:64⟩, ⟨MEM:64⟩
```

describes 64-bit addition instructions with a register as the first and a memory reference as the second operand.

We extract the instruction schemes for the x86-64 instruction set architecture from uops.info [Abel and Reineke 2019]. Additionally, we collect for each instruction scheme several features such as the mnemonic, the operand types, whether and how it uses memory, and to which instruction category and ISA extension it belongs. Our domain groups instructions through constraints on these features. The form of these constraints for a feature f is determined by its feature abstraction $\mathbb{A}_F[f]$. Table 1 introduces the feature abstractions that we use for our abstract domain.

For simple features like the category and ISA extension to which the instruction belongs or the presence of a `lock` or `rep` prefix, we use the *singleton* abstraction. It expresses that all represented instruction schemes have a specific value for the feature.

For the mnemonic, we use the *edit distances* abstraction. It constrains the represented mnemonics by an upper bound d on the Levenshtein distance from a base string B. If they share their base, abstract values are ordered by their value of d, which is limited by a maximum bound (here: 3). This abstraction allows AnICA to group instructions with similar mnemonics: an abstract value representing only `vaddpd` (an addition for vectors of double-precision floats) can be expanded to also represent `vaddps` (the same operation with single-precision) and the scalar versions `vaddsd` and `vaddss`. The edit distance abstraction is heuristic in nature: neither do all similar instructions have similar mnemonics nor are all instructions with similar mnemonics similar themselves. We nevertheless found this abstraction to be helpful in practice since, for instance, mnemonic suffixes that rarely affect the instruction's performance behavior are common in modern ISAs (e.g., specifying floating-point format and vector width, or the condition for conditional move instructions).

We use the *log sizes* abstraction for the (multi-)set of micro operations required to execute an instruction. AnICA can therefore group instructions by their complexity: an abstract value represents all instruction schemes that are decomposed into less than 2^k μops for a certain k. To avoid infinite ascending chains of abstractions, a maximal value for k is a parameter of this abstraction (here: 5).

Whether and how an instruction accesses memory affects its performance significantly. Our domain therefore uses the fine-grained *subset-or-none* abstraction with subsets of $\{\mathsf{R}, \mathsf{W}, \mathsf{Size} : n\}$ to represent memory usage. This enables AnICA to relax constraints on memory usage step by step. An abstract instruction representing only instructions that Read and Write n bits in memory can be expanded by dropping any of these constraints. The expanded abstract instruction might represent all instructions that at least read n bits from memory. With DefNone, only instructions that do not access memory are represented. We also use this abstraction for the set of operand types that occur in the instruction schemes.

Formally, abstract instructions are tuples of feature abstraction $\mathbb{A}_F[f_i]$ elements for each considered feature f_i:

$$\mathbb{A}_{in} := \mathbb{A}_F[f_1] \times \cdots \times \mathbb{A}_F[f_N]$$

The partial order among abstract instructions relies on the partial orders of the involved feature abstractions:

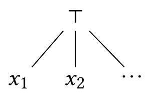
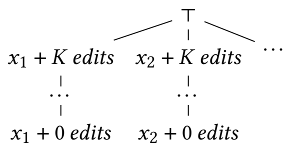
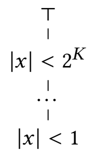
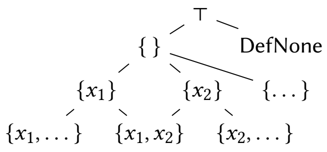
$$(x_1, \cdots, x_N) \sqsubseteq_{in} (y_1, \cdots, y_N) :\Leftrightarrow \bigwedge_{i \in [1,N]} x_i \sqsubseteq y_i$$

The last two columns of Table 1 define the feature concretization and representation functions $\gamma_X^{(f)}$ and $\beta_X^{(f)}$ for each feature abstraction X. They are parameterized by the feature f for which they are used and refer with $f(i)$ to the value of the instruction i for this feature.

The feature concretization functions $\gamma_X^{(f)}$ map *abstract values* from the feature abstraction X to their set of represented instructions i. All feature abstractions have a maximal abstract value $\top$, which represents the absence of any constraint. Therefore its concretization is the same for every domain: the entire set of available instruction schemes. The feature representation functions $\beta_X^{(f)}$ define the value from the feature abstraction that best describes an instruction with the value $f(i)$ for the feature f.

We use these feature concretization and representation functions to define the instruction-wise functions: an abstract instruction imposes the conjunction of the per-feature constraints

Table 1. Feature domains used in AnICA. The domains are shown as Hasse Diagrams, where the partial order is indicated through the lines: if x is connected to y and y is closer to the top, $x \sqsubseteq_F y$ holds.

Domain	Hasse Diagram	Used for	$\gamma_X^{(f)}(av)$ for $av \neq \top$	$\beta_X^{(f)}(i)$
Singletons		Category, ISA-Set, Prefixes	$\{i \mid f(i) = av\}$	$f(i)$
Edit Distances	 $\top$ $x_1 + K$ *edits* $x_2 + K$ *edits* $\cdots$ $x_1 + 0$ *edits* $x_2 + 0$ *edits*	Mnemonic	$\{i \mid dist(f(i), B) \leq d\}$	$(B : f(i), d : 0)$
Log Sizes	 $\top$ $\lvert x\rvert < 2^K$ $\cdots$ $\lvert x\rvert < 1$	Number of μops	$\{i \mid \lvert f(i)\rvert < 2^{av}\}$	$\lfloor \log_2(\lvert f(i)\rvert + 1) \rfloor$
Subset-or-None	 $\top$ $\{\}$ DefNone $\{x_1\}$ $\{x_2\}$ $\{\ldots\}$ $\{x_1, \ldots\}$ $\{x_1, x_2\}$ $\{x_2, \ldots\}$	Memory Usage, Operand Types	if av = DefNone: $\{i \mid f(i) = \emptyset\}$ otherwise: $\{i \mid f(i) \supseteq av\}$	if $f(i) = \emptyset$: DefNone otherwise: $f(i)$

on the represented instructions. Therefore, it concretizes to the intersection of the per-feature concretizations $\gamma_{\mathbb{A}_F[f]}^{(f)}$ applied to the abstract instruction's component $ai[f]$ for each feature f:

$$\gamma_{in}(ai) := \bigcap_{f \in Features} \gamma_{\mathbb{A}_F[f]}^{(f)}(ai[f])$$

To obtain a representative abstract instruction for a concrete one, we apply the representation functions for each feature:

$$\beta_{in}(i) := \left(\beta_{\mathbb{A}_F[f_1]}^{(f_1)}(i), \ldots, \beta_{\mathbb{A}_F[f_N]}^{(f_N)}(i)\right)$$

Analogously, an expansion function for an abstract instruction i takes a non-$\top$ component and replaces it with one of its immediate successors in the generalization order:

$$Exps_{in} := \left\{ \lambda ai.\ ai[f \mapsto y] \text{ if } ai[f] = x \;\middle|\; f \in Features, x \in \mathbb{A}_F[f] \setminus \{\top\}, y \text{ succeeds } x \text{ in } \sqsubseteq_{\mathbb{A}_F[f]} \right\}$$

Aliasing Abstraction. The subcomponent $\mathbb{A}_{al}$ represents aliasing constraints among operands of instructions. We refer to an operand of an instruction via a pair $(idx_i, idx_o) \in Idx := (InsnIdx \times OpIdx)$ of indexes into the sequence of instructions and into the sequence of operands of the instruction. The operand idx_o of instruction idx_i in the basic block b is denoted as $b[(idx_i, idx_o)]$.

An aliasing constraint for a pair of such instruction operand designators can state that they *must* or *must not* alias, or that no constraint applies (denoted as $\top$). The aliasing subcomponent is therefore defined as a mapping as follows:

$$\mathbb{A}_{al} := (Idx \times Idx) \to \{\texttt{must}, \texttt{mustnot}, \top\}$$

An aliasing information g is more general than another h if g imposes the same or a weaker constraint than h for *every* pair of operands:

$$h \sqsubseteq_{al} g :\Leftrightarrow \forall x \in (Idx \times Idx).\ g(x) = \top \vee h(x) = g(x)$$

This component of the abstraction is more intricate than one might expect at first: the instruction abstraction can represent sets of vastly different instruction sequences. One abstract instruction could for example represent a 2-operand integer addition operation and a 3-operand floating point addition. Consequently, the aliasing abstraction needs to handle cases where operands do not match, i.e., cannot possibly alias, or where they are not present at all in some of the represented basic blocks.

We handle these cases with a concretization function that only applies constraints on operands that match and are present in the concrete basic block:

$$b \in \gamma_{al}(h) :\Leftrightarrow \bigwedge_{((i_1,i_2)\mapsto x)\in h} \Big(existAndMatch\big(b[i_1], b[i_2]\big) \\ \Rightarrow \Big(x = \top \vee \big(x = \texttt{must} \wedge doAlias\big(b[i_1], b[i_2]\big)\big) \vee \big(x = \texttt{mustnot} \wedge \neg doAlias\big(b[i_1], b[i_2]\big)\big)\Big)\Big)$$

The representation function β_{al} is defined to capture the must-alias and must-not-alias relations between the matching operands of the concrete basic block:

$$\beta_{al}(b) := \lambda(i_1, i_2). \begin{cases} \texttt{must} & \text{if } b[i_1], b[i_2] \text{ exist, match, and alias} \\ \texttt{mustnot} & \text{if } b[i_1], b[i_2] \text{ exist, match, and do not alias} \\ \top & \text{otherwise} \end{cases}$$

Deciding for a pair of register operands whether they alias is straightforward: they alias if and only if they are the same or if one is a sub-register of the other.[6] Whether memory operands alias depends on the values of registers. We approximate this by considering two memory operands aliasing if they are identical, and not aliasing otherwise. In general, this is not a sound approximation: two memory operands can look entirely different but refer to the same address or vice versa. It is only sound for our use case because the basic block sampling method described in the following section manages memory operands such that they alias if and only if they are syntactically identical.

The expansion functions for the aliasing component of an abstract block each replace a non-$\top$ entry in the aliasing abstraction with $\top$:

$$Exps_{al} := \big\{\lambda h.\ h[x \mapsto \top] \text{ if } h(x) \neq \top \ \big|\ x \in Idx\big\}$$

3.5 Sampling Represented Basic Blocks

Our generalization algorithm relies on a method $\widetilde{\gamma}$ to randomly sample basic blocks that are represented by a given abstract block. For arbitrary elements of our abstract domain, this is a hard problem: sampling a block that fulfills the aliasing constraints essentially corresponds to a graph-coloring register allocation problem [Chaitin et al. 1981] because concrete registers have to be found that comply with the aliasing constraints. Since there are no restrictions on these constraints, arbitrary interference graphs can emerge in general which renders sampling NP-hard in theory.

As the concretization sampler $\widetilde{\gamma}$ is a very common operation in AnICA's generalization algorithm, we do not implement a complete solution to the NP-hard sampling problem. Instead, we proceed greedily as follows:

[6]We do not consider the legacy floating point extensions x87 and MMX; register aliasing is more complicated for the x87 register stack.

(1) For each abstract instruction, choose a represented instruction scheme.
(2) If the schemes have fixed operands[7], select those and set all related must-alias operands accordingly.
(3) Repeatedly: Where an operand is not yet selected, choose one that is not forbidden through must-not-alias constraints. Set all must-alias operands accordingly.

We restrict what registers may be used as register operands to have distinct registers available for the base registers of memory operands that cannot be overwritten. If two memory operands are required to alias, we instantiate them with the same combination of base register and displacement. In case of a no-alias constraint on memory operands, we use different combinations of base register and displacement.

If, at any point in this algorithm, no selection is possible without violating the alias constraints or the requirements of the instruction schemes, the sampling fails and needs to be repeated. For an example, consider the following abstract block for the x86 ISA with two unconstrained abstract instructions and a must-not alias constraint:

Instructions:
(1) $\top$
(2) $\top$
Aliasing:
- operand 2 of insn 1 must not alias with operand 2 of insn 2

If, in the first step of the sampling algorithm, we choose shift instructions with variable shift amount for both instructions, sampling will fail: then both instructions need to have register `c` as second operand (for the shift operand), which would violate the alias constraint.

In practice, sampling rarely fails for the short instruction sequences that we sample: it affected 0.01% of the ca. 4.8×10^6 sampling operations in the campaigns presented in Section 4.3.[8]

3.6 Checking for Subsumption

In Algorithm 1, we check whether concrete or abstract blocks are subsumed by an abstract block to avoid unnecessary generalizations and to prune irrelevant discoveries. The fixed number and positions of instructions in our abstract domain ease sampling basic blocks, but they hinder us here: the concretization $\gamma_{\mathbb{A}}(a)$ of an abstract block a does not contain basic blocks that we would like to consider subsumed by a. The following basic block would not be included in the concretization of abstract block AB1 from Example 1:

```
add [r8], rbx; mov rbx, [rdx + 42]
```

However, the instructions here are the same as in the example represented by AB1, only in a different order that has no impact on the block's sustained throughput: the throughput is determined by the rate at which the basic block can be executed repeatedly for an indefinite number of iterations. What determines the throughput of a basic block bb is therefore the trace of instructions resulting from repeating bb a large number of times. When a basic block bb' results from rotating bb (i.e., removing a sequence of instructions from the beginning and appending it to the end), its trace differs from the one of bb only by short pre- and suffixes whose influence on the execution time vanishes with a growing number of repetitions.

[7]For example, shifts in x86 use the `c` register for their shift amount.
[8]An alternative approach would be to encode the aliasing constraints as SMT formulae and use, e.g., the approach of Dutra et al. [2019] to sample satisfying solutions. In comparison to our approach this would eliminate the chance of sampling errors at the cost of an increased execution time of the sampling steps.

Similarly, if this basic block exhibits an inconsistency in the predictions, it is likely to have the same reason as AB1:

```
mov rbx, [rdx + 42]; nop; add [r8], rbx
```

Yet, it is not included in $\gamma_{\mathbb{A}}(\mathrm{AB1})$ since it contains three instead of two instructions.

We therefore do not rely on the partial order of the abstract domain to implement the subsumption checks in Algorithm 1. Instead, we check for the following definition:

Definition. An abstract block (a^1_{in}, a^1_{al}) *subsumes* another (a^2_{in}, a^2_{al}) if there is a mapping $m : I^1 \longrightarrow I^2$ from the indexes I^1 of the abstract instructions of a^1_{in} to the indexes I^2 of a^2_{in} s.t.

$$\forall i, j \in I^1.\ m(i) \neq m(j) \tag{C1}$$

$$\forall i \in I^1.\ \gamma_{in}(m(i)) \subseteq \gamma_{in}(i) \tag{C2}$$

$$\forall ((i, op_1), (j, op_2) \mapsto x) \in a^1_{al}.\ x \neq \top \Rightarrow a^2_{al}((m(i), op_1), (m(j), op_2)) = x \tag{C3}$$

$$\forall i \in I^1.\ \forall k \in I^2 \text{ between } m(i) \text{ and } m((i+1) \bmod |I^1|).\ \nexists i'.\ m(i') = k \tag{C4}$$

An abstract block a *subsumes* a concrete basic block b if it subsumes $\beta_{\mathbb{A}}(b)$.

In other words, m needs to be injective (C1) and map abstract instructions to at least as specific ones (C2). Furthermore, the aliasing constraints imposed by a^2_{al} on the mapped instructions need to be at least as strong as those imposed by a^1_{al} (C3). Lastly, the order of the mapped instructions $m(i)$ in a^2_{in} needs to be a rotation of the order of the instructions i in a^1_{in} (C4). All instructions of a^1_{in} need to have a counterpart in a^2_{in}, but not vice versa.

We encode these constraints in a boolean formula that is satisfiable if and only if such a mapping exists and use a SAT solver to discharge them. In an AnICA campaign, subsumption checks are not numerous and in our experience, SAT solvers can solve the formulae quickly.

3.7 Ranking Abstract Basic Blocks

When evaluating the usefulness of AnICA discoveries, as well as for guiding developers interested in improving throughput predictors, it is helpful to rank abstract basic blocks by a notion of importance. In the following, we describe three approaches to ranking abstract basic blocks that we found useful when evaluating AnICA and carrying out the case studies presented in Section 5.

3.7.1 Ranking by Interestingness. Every abstract block that results from AnICA's generalization has been checked for interestingness. This means that we sampled a number of represented concrete basic blocks and computed the (relative or absolute) difference between the predictions of the tools under investiation for the basic blocks for each discovery. A natural metric of relevance of the abstract block is therefore the mean prediction difference over the set of sampled basic blocks. The higher it is, the more dramatic is the inconsistency characterized by the abstract block. For inputs that crash a throughput predictor, we set this metric to infinity to indicate maximal interestingness.

3.7.2 Ranking by Generality. Inconsistencies do not need to come with large deviations in the predictions to indicate a significant difference in the tools under investigation. We therefore use *generality* as an alternative metric for ranking abstract blocks. The idea is that we want to find discoveries that affect large classes of concrete basic blocks.

There are several conceivable options to define such a metric, which may differ in the effort required to compute them (e.g., one might sample a large number of basic blocks and check how many of them are subsumed by each discovery) and in how basic blocks are weighted (e.g., should instruction schemes with wide immediate constants be considered more general, because each possible immediate value counts as a different instruction?).

We chose a notion of generality that is inexpensive to compute and operates, like our generalization algorithm, on the granularity of instruction schemes: an abstract block's generality is the minimal number of instruction schemes represented by any of its abstract instructions. While this is a simplification of reality – it ignores aliasing constraints and the number of abstract instructions in the abstract block – this metric was instrumental to find several examples for our case studies.

3.7.3 Maximizing the Number of Subsumed Basic Blocks. If users of AnICA have a concrete set of basic blocks that they consider particularly relevant, e.g., extracted from an important benchmark set, this can be leveraged to a custom-tailored notion of generality. For one, we can rank AnICA's abstract blocks by the number of basic blocks from the set that they subsume.

An extension to this strategy is to solve the following integer linear program (ILP) to obtain a set of k discoveries from *AbsBlocks* that subsume a maximally large portion of the basic block set B:

$$\begin{aligned}
\text{maximize} \quad & \sum_{j \in B} \text{BB.covered}[j] & \\
\text{subject to} \quad & \sum_{i \in AbsBlocks} \text{AB.used}[i] \leq k & \\
& \sum_{i \in AbsBlocks \wedge i \text{ subsumes } j} \text{AB.used}[i] \geq \text{BB.covered}[j] & \text{for all } j \in B \\
& \text{AB.used}[i] \in \{0, 1\} & \text{for all } i \in AbsBlocks \\
& \text{BB.covered}[j] \in \{0, 1\} & \text{for all } j \in B
\end{aligned}$$

The ILP uses two groups of binary variables: an AB.used[i] variable for each abstract block i and a BB.covered[j] variable for each concrete basic block j. If one of the AB.used variables is 1, the corresponding abstract block is chosen as one of the k maximally subsuming discoveries. The first constraint of the ILP ensures that no more than k abstract blocks are selected. If one of the BB.covered variables is 1, the corresponding concrete basic block is subsumed by at least one of the chosen discoveries. We encode this relationship with the second constraint of the ILP: BB.covered[j] cannot be greater than 0 unless an abstract block i that subsumes it is chosen with AB.used[i]. With the ILP's objective term, we require that an optimal solution covers as many concrete basic blocks as possible.

A set of abstract blocks extracted from the values of the AB.used variables in a solution to the ILP is a maximally diverse selection of AnICA discoveries. With an appropriate selection of the parameter k, AnICA's results can thus be summarized as concisely as desired.

4 EVALUATION

The main goal of AnICA is to provide insights into the basic block throughput predictors under investigation. Since this goal is not easily quantified, we evaluate AnICA in two parts: a general investigation of how inconsistencies are generalized (Section 4.3) and a number of detailed case studies to give examples of actual insights gained (Section 5).

4.1 Considered Tools

We compare a broad range of throughput predictors:

IACA [Intel 2012] is a closed-source tool provided by Intel to estimate the performance of basic blocks on their microarchitectures. In April 2019, Intel announced that IACA has reached its end of life. We use the last released version, 3.0.

llvm-mca [Di Biagio 2018], OSACA [Laukemann et al. 2018], and uiCA [Abel and Reineke 2022] are open source basic block throughput predictors. Their processor models are constructed by hand from documentation, contributed by hardware vendors, or inferred from measurements. llvm-mca uses the instruction scheduling models of the LLVM compiler infrastructure [Lattner and Adve 2004]. If not stated otherwise, we use release 13 of llvm-mca, version 0.4.6 of OSACA, and commit `71f2eb6` from uiCA's GitHub repository.[9]

In contrast to this, Ithemal [Mendis et al. 2019] and DiffTune [Renda et al. 2020] infer their models through machine learning. Ithemal predicts throughputs through an LSTM-based neural network that is trained on throughput measurements for a set of basic blocks. We use their provided model that was trained on basic blocks from the BHive [Chen et al. 2019] data set (commits `47a5734` and `87c2468` of the corresponding GitHub repositories[10]). DiffTune is a modified version of llvm-mca where parameters of the processor model are replaced with learned ones. We use the parameters that the authors provide, which were obtained through surrogate learning with an Ithemal-based model (commit `9992f69` in the GitHub repository[11]).

We do not compare the MAQAO Code Quality Analyzer [Rubial et al. 2014] in this evaluation as it requires a loop as input for its throughput prediction, which not all of the other tools support.[12]

4.2 AnICA Parameters

We use the following parameters for AnICA:

- Threshold that the relative difference between to predictions must exceed to be considered interesting: 0.5
- Number of samples to check whether an abstract block is interesting: 100
- Maximal length of sampled basic blocks for discovery: 5 instructions
- Number of randomized generalizations per basic block: 5

In preliminary experiments, we found that variations in the latter three parameters do not affect AnICA's results substantially in terms of the metrics presented in this section. Only if they are selected widely out of range (e.g., only using very few samples to check for interestingness or only investigating very short basic blocks), the performance declines. The threshold of the interestingness metric is of more relevance since it determines what inconsistencies are found. The selected value is relatively large, which causes AnICA to focus on substantial output differences. We found such inconsistencies to be more likely to hint at conceptual differences like the handling of memory dependencies (cf. Section 5.1).

We extract the instruction schemes used for sampling from uops.info [Abel and Reineke 2019]. For the evaluation, we exclude instruction schemes if they satisfy any of the following conditions:

- They are in a SIMD or FP extension other than AVX1&2.
- They are not measured by uops.info.[13]
- They affect control flow.
- They need to be executed in privileged mode.

This leaves us with 2940 instruction schemes. For each campaign, we further exclude all instructions that are not supported by one of the tools under investigation.[14] We configure the throughput

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

[10]https://github.com/ithemal/Ithemal and https://github.com/ithemal/Ithemal-models

[11]https://github.com/ithemal/DiffTune

[12]Our AnICA implementation nevertheless supports tools like MAQAO, with an option to wrap each basic block in a loop when it is given as input to the tools.

[13]This proxy criterion is intended to exclude instructions that are not supported in the microarchitecture, e.g., because they are from outdated ISA extensions.

[14]We consider an instruction supported by a tool if the tool gives a non-zero prediction for a basic block consisting of only the instruction.

Table 2. AnICA campaigns to find 150 inconsistencies, with metrics on how many basic blocks from Figure 1 they explain, ordered by the percentage of interesting basic blocks subsumed.

	IACA, OSACA	llvm-mca 13, 9	IACA, uiCA	OSACA, uiCA	llvm-mca, uiCA	llvm-mca, OSACA	IACA, llvm-mca	DiffTune, OSACA	IACA, Ithemal	Ithemal, OSACA
BBs interesting	26%	23%	34%	32%	31%	34%	19%	52%	57%	47%
int. BBs covered	97%	97%	91%	85%	83%	77%	74%	69%	68%	66%
...by top 10	68%	92%	82%	53%	72%	55%	70%	34%	54%	33%
run time (h:m)	6:34	0:32	1:35	6:59	1:19	5:28	0:38	9:29	4:34	8:13

	DiffTune, llvm-mca	DiffTune, IACA	Ithemal, llvm-mca	DiffTune, uiCA	DiffTune, Ithemal	Ithemal, uiCA
BBs interesting	46%	52%	50%	40%	46%	30%
int. BBs covered	63%	62%	62%	59%	57%	38%
...by top 10	31%	32%	49%	34%	29%	16%
run time (h:m)	5:55	5:54	5:05	6:25	10:02	5:15

predictors to assume the Intel Haswell microarchitecture since it is the only one supported by all considered tools.

The AnICA campaigns ran on a system with an Intel Core i9-10900K processor (10 cores, 20 threads, 3.7 GHz) and 64 GB of RAM. Running the predictors to evaluate the interestingness of basic blocks, which constitutes most of the execution time, is performed with 20 concurrent threads.

4.3 Generalization of Inconsistencies

Figure 1 in Section 1 demonstrates that we can find a large number of inconsistencies among the tools through random testing; enough that investigating them all by hand would be infeasible. This section evaluates how AnICA summarizes these inconsistencies.

The evaluation is based on the same data as Figure 1: a test set of 10,000 randomly sampled basic blocks consisting of 4 instructions each. We sample these as described in Section 3.5 from an abstract block with 4 instructions and no constraints. We ran AnICA for each pair of tools until around 150 discoveries were found. Table 2 contains a column for each AnICA campaign.[15] The first line repeats the data from Figure 1: the percentage of basic blocks in the test set that are interesting, i.e., for which the relative difference of the predictions exceeds 50% of their average.

The second line shows the percentage of the set of interesting basic blocks from the test set that are subsumed (cf. Section 3.6) by an AnICA discovery. At 74% to 97%, these ratios are very high for comparisons of IACA, llvm-mca, uiCA, and OSACA. This indicates that AnICA inferred general descriptions of the differences between these tools.

The third line further demonstrates that AnICA effectively condenses the inconsistent basic blocks for manual inspection: it gives the ratio of interesting basic blocks in the test set that are subsumed by a subset of only ten discoveries of the AnICA campaign. In eight of the campaigns, these numbers were higher than 50%, meaning that in these cases only ten of AnICA's discoveries are sufficient to plausibly explain more than half of the inconsistently predicted basic blocks. In every case except for the Ithemal/uiCA campaign, ten of the AnICA discoveries subsume more than 1000 inconsistent basic blocks from the dataset, ranging up to 3060 subsumed inconsistencies

[15]For brevity, we only include a comparison of llvm-mca version 9 and the current llvm-mca version 13.

in the IACA/Ithemal campaign. The discovery subsets for this metric were computed with the strategy to maximize the number of subsumed basic blocks presented in Section 3.7.3, applied to the interesting basic blocks in the test set.

The time required to find these discoveries, as displayed in the last line, mainly depends on how fast the tools produce their predictions. While not the focus of this work, we observe that in this setup, IACA, llvm-mca, and uiCA were considerably faster than OSACA, Ithemal, and DiffTune.

The campaigns that include Ithemal and DiffTune still cover a substantial number of inconsistencies, but AnICA finds less potential for generalization here than in the other campaigns. We can identify reasons for this observation from the results for these campaigns: AnICA's generalizations terminate early in several instances where these tools produce results that run counter to common expectations.

For example, Ithemal produces different results for basic blocks that only differ in the specific register that they use, as can be seen in its predictions for basic blocks consisting of a single "rotate left" operation:

Basic Block	`rol r12, cl`	`rol r10, cl`
Predicted Cycles	0.35	1.01

All other tools predict equal throughputs for these blocks.

AnICA groups instructions by instruction schemes, i.e., a form that abstracts from the specific operands of the instruction. It therefore does not generalize inconsistencies that are not independent of the concrete registers used. Most throughput predictors share this notion and do not change their prediction if, e.g., operand registers in the basic block are replaced (while preserving dependencies). This assumption is evidently not enforced in Ithemal's neural network.

AnICA's results demonstrate this issue and therefore justify the insight that Ithemal might, e.g., benefit from training data where basic blocks are included multiple times with different but semantically equivalent register allocations. Since the measured throughputs for these would be the same, the neural network might learn to abstract from the specific register used.

DiffTune learns parameters for llvm-mca and can therefore not produce different predictions based on the specific operands of the instructions as Ithemal does. We can however observe that instructions that are very similar are predicted differently by DiffTune. For example, AnICA finds that the abstract block in Figure 4e represents an inconsistency between DiffTune and IACA. This abstract block covers arithmetic right shift instructions, which, as the witnessing experiments in AnICA's generalization decision tree show, DiffTune predicts slower than IACA if they use memory and faster if they do not use memory. However, this discovery also indicates that instructions with a mnemonic that is only slightly different, like the logical right shift operations `shr`, are not predicted inconsistently. From the experiments that reject the expansion to a mnemonic edit distance of 1, we can see that DiffTune gives different predictions for `shr` and `sar` instructions, in contrast to most other tools. This different treatment of similar instructions invites for a closer inspection, but it restricts AnICA's generalizations.

In summary, we observe that AnICA's generalization is very effective for the majority of considered tools. Where generalization is not as effective, the results are nevertheless insightful and point to concrete problems.

5 CASE STUDIES

The previous section shows that AnICA is able to summarize thousands of inconsistencies between throughput predictors by a small number of abstract blocks. For DiffTune and Ithemal, it also presents first lessons learned from AnICA's results. We further investigated AnICA's discoveries and found several kinds of insights, for which we present examples in the following:

Instructions:
(1) cat: logical; memory: R+W

(a) llvm-mca 13 & 9, DiffTune

Instructions:
(1) memory: R+W;
requires less than 8 μops

(b) IACA, uiCA

Instructions:
(1) memory: R+W; cat: binary
(2) memory: W
Aliasing:
- op 1 of insn 1 must alias with op 1 of insn 2

(c) uiCA, Ithemal

Instructions:
(1) mnemonic: `vpsubq` + 1 edit;
memory: R
(2) mnemonic: `fxrstor[64]`

(d) llvm-mca 12

Instructions:
(1) mnemonic: `sar` + 0 edits

(e) IACA, DiffTune

Fig. 4. Abstract blocks causing inconsistent behavior found by AnICA. Feature abstractions are summarized for brevity.

Table 3. Predictions for the cycles required to execute basic blocks that differ in their memory dependencies.

Basic Block	uiCA	OSACA	IACA	llvm-mca 13	llvm-mca 13 alias	llvm-mca 9	DiffTune	Ithemal
`add [rcx+16],rbx; add [`**`rcx+16`**`],rbx`	12.0	12.0	2.0	2.1	14.0	14.0	14.1	5.9
`add [rcx+16],rbx; add [`**`rcx+128`**`],rbx`	6.0	6.0	2.0	2.1	14.0	14.0	14.1	5.9
`add [rcx+16],rbx; add [`**`rdx+16`**`],rbx`	6.0	6.0	2.0	2.1	14.0	14.0	14.1	6.0

- They uncover different assumptions in the tools that can lead to dramatically different predictions. (Section 5.1)
- They find newly introduced regression bugs in subsequent versions of the same tool (Section 5.1) as well as long-existing bugs (Section 5.2).
- AnICA can characterize a variety of inaccuracies in llvm-mca's model for the AMD Zen+ microarchitecture, as well as an unusual quirk in the microarchitecture itself. (Section 5.3)

5.1 Memory Dependencies

Data dependencies through memory operands are a challenge for basic block throughput predictors. If subsequent writes to and reads from memory refer to the same location, the write needs to be completed before the read.[16] If they access disjoint memory locations, they can execute independently. However, which of the cases applies may not be obvious or depend on the inputs. AnICA's results show that the tools handle these cases quite differently.

Table 3 shows how the throughput predictors handle memory dependencies on three example basic blocks: one with a guaranteed memory dependency (first line), one with independent instructions (second line), and one where the instructions may be independent, given suitable register values (third line). We see three plausible throughput prediction results for executing such basic blocks in a loop:

- Two cycles, if there are no dependencies through memory and each instruction uses the processor's store unit for one cycle.
- Six cycles, if there is no memory dependency between the two instructions, but each instruction depends on its own result from the previous iteration. They then form two dependency chains with a latency of 6 cycles, which can be executed in parallel.

[16]More specifically, the written value needs at least to be computed and put into a store buffer, from which it can be forwarded to subsequent reads.

- Around 12 cycles, if all memory accesses depend on each other, forming a single large dependency chain.

llvm-mca in its default setting [LLVM 2021b] and IACA[17] assume the first case. AnICA shows that for llvm-mca in the outdated version 9, this was not the case: it discovers, e.g., that the abstract block in Figure 4a represents an inconsistency between llvm-mca version 9 and 13. The older version was affected by a bug that led to predictions as if all memory accesses aliased. While this bug has been manually discovered and fixed in the past, AnICA automatically finds this regression with a minimal example for reproducing the bug.

We also find that DiffTune's learned parameters for llvm-mca attempt to bypass this assumption: AnICA finds the same abstract block in Figure 4a in the campaign for llvm-mca 13 and DiffTune. The example basic blocks for this discovery show that DiffTune also predicts throughputs as if all accesses were dependent on each other. Regarding this, Renda et al. [2020] remark in their evaluation that DiffTune learned a "degenerately high" latency for instructions that read and write memory from the same location. llvm-mca also provides an override switch to assume that all memory accesses alias, which leads to results similar to DiffTune's.

AnICA discoveries like the one in Figure 4b indicate that uiCA, OSACA, and Ithemal do not share IACA's assumption that no memory operations alias. These three tools recognize the data dependency formed by a single instruction that reads *and* writes with itself, therefore AnICA reports no discovery like those in Figure 4a and 4b between them. However, the abstract block displayed in Figure 4c allows us to identify the difference in line 1 of Table 3: uiCA and OSACA rightly assume that the two memory locations alias if they are identical. AnICA provides the basic block treated in line 1 of the table for this abstract block.

For a user of these tools, this discrepancy can significantly affect the outcome: if the memory operands in the application alias in a non-obvious way, the observed cycles would exceed the results of uiCA and OSACA by a factor of two, and those of IACA and the default setting of llvm-mca by a factor of 6.

Since this inconsistency affects a large number instruction schemes, corresponding discoveries were easy to find in AnICA's report with discoveries ranked by their generality (cf. Section 3.7.2).

5.2 FXRSTOR Crash in llvm-mca

To uncover crashes, AnICA can compare a single tool with itself. Figure 4d shows an abstract block we found when investigating llvm-mca. This abstract block crashes the tool with an assertion, which AnICA always counts as interesting. `FXRSTOR` instructions, which restore the state of floating point control registers from memory, require the processor to execute a large number of micro operations. If one of the resources that it accesses is also used by a different instruction nearby, e.g., a vector subtraction with a memory operand, a bug in llvm-mca is triggered.

For LLVM release 13, this bug has recently been reported and fixed independently of our research.[18] The report includes a large input with more than 300 instructions to trigger the bug. AnICA automatically discovered the issue and provides a minimal input of just two instructions.

This discovery appears prominently in AnICA's results when the discoveries are ranked by their interestingness (cf. Section 3.7.1) since crashes in a tool under investigation are reported as maximally interesting.

Table 4. Abstract blocks capturing inconsistent behavior found by AnICA in llvm-mca/nanoBench campaigns on the AMD Zen+ microarchitecture. The descriptions are not generated by AnICA. The "Resulting Cycles" column displays the number of cycles predicted by llvm-mca (mca) and the cycles measured by nanoBench (nb) for the basic block in the preceding column.

	Abstract Block	Example Basic Block	Resulting Cycles	Simplified Description
(A)	Instructions: (1) opschemes: {`R:flag_df`}	`cmpsq`	mca: 100 nb: 3.0	llvm-mca models complex instructions, certain shifts, and horizontal vector operations very pessimistically.
(B)	Instructions: (1) opschemes: {`R:cl`} isa-set: `I386`	`shld r11, rdx, cl`	mca: 100 nb: 3.0	
(C)	Instructions: (1) mnemonic: haddpd + 3 edits; category: SSE3	`hsubpd xmm15, xmm12`	mca: 100 nb: 6.5	
(D)	Instructions: (1) mnemonic: `bsf` + 1 edit opschemes: {`W:GPR:64`} Aliasing: • op 1 of insn 1 must not alias with op 2 of insn 1	`bsr rcx, r11`	mca: 0.3 nb: 4.0	LLVM's model for bit-scan instructions implies a wrong throughput.

(E)	Instructions:
	(1) mnemonic: add + 2 edits
	opschemes: {`RW:GPR:64`}
	memory: DefNone
	(2) mnemonic: and + 3 edits
	opschemes: {`RW:GPR:64`}
	memory: R
	Aliasing:
	• op 1 of insn 1 must alias with op 1 of insn 2
		`add r8, 0x2a`
		`adc r8, qword ptr [r14]`
			mca: 5.03
			nb: 2.0
				llvm-mca misses that memory loads can start before other operands are available.

(F)	Instructions:
	(1) mnemonic: `shl` + 2 edits
	opschemes: {`0x0`}
	isa-set: I186
	(2) opschemes: {`R:flag_of`}
		`shl r9w, 0x0`
		`setno r11b`
			mca: 1.04
			nb: 25.7
				Reading flags after a shift by 0 incurs a penalty on Zen+.

5.3 Comparing llvm-mca to Measurements

AnICA has little requirements on the tools under investigation: they only need to produce a throughput estimate for a given basic block. Consequently, we can also use AnICA to compare a throughput predictor to a tool that runs input basic blocks as microbenchmarks on the actual hardware.

In this case study, we apply AnICA to compare the predictions of llvm-mca's model for the AMD Zen/Zen+ microarchitecture to microbenchmarks performed with nanoBench [Abel and Reineke 2020] on an AMD Ryzen 5 2600X processor. For these discoveries, we configured AnICA to consider an abstract block interesting if the absolute difference between measured and predicted throughput of all of 50 sampled basic blocks is at least 2 cycles. This ensures that we can identify the more

[17] as previously noted by Abel and Reineke [2019]
[18] http://bugs.llvm.org/PR50725

subtle inconsistencies in the results. We further restrict the instructions considered by AnICA for sampling and generalization: we exclude instruction schemes that read *and* write memory to avoid discovering more variants of the problem described in Section 5.1.

It is important to note that the nanoBench measurements are just another tool under investigation, not a ground truth. We configure nanoBench to group 10 instances of the measured instructions in a loop body that is executed 100 times while the passing processor cycles are measured with a hardware performance counter after 10 warm-up iterations. We use defaults for the remaining settings of nanoBench, which entails among other things that most registers are initialized with arbitrary values (except for those used as memory addresses). These assumptions affect the measured cycles, rendering the measurements unsuitable for use as definitive ground truth. The strength of AnICA's differential testing perspective is that neither tool needs to be assumed as "correct" to obtain interesting insights.

Table 4 shows the selected AnICA discoveries that we discuss in the following. We refer to the discoveries by the identifier in the first column. The second column contains the abstract block reported as discovery by AnICA. With the generalization decision tree and the corresponding evaluated basic blocks, AnICA provides more additional information than we can present here. We therefore instead only show one example basic block sampled from the abstract block and the results of nanoBench (nb) and llvm-mca (mca) for it in the third and fourth columns. In the last column, we annotate a short summary of the problem characterized by the discovery.

Microcoded Instructions. When ranking AnICA discoveries by their interestingness (cf. Section 3.7.1), the ones that stand out the most are those concerning instructions that llvm-mca predicts to require 100 cycles to execute. This mainly affects microcoded instructions, e.g., the string operations, which commonly read the direction flag register `df`, summarized by discovery (A). When the processor's instruction decoder encounters such instructions, it produces a (potentially large and/or varying) number of μops that need to be executed by the processor's functional units. LLVM's Zen+ scheduling model (and consequently llvm-mca) handles most such instructions in a coarse way that just assigns them a latency of 100 cycles.

However, this strategy is also used for more unexpected instructions like certain bit shifts (discovery (B)) and horizontal vector operations (discovery (C)). While these modeling decisions are not per se bugs, they can make the Zen+ model of llvm-mca effectively unusable for any task that uses such instructions. LLVM's issue tracker contains a report for this behavior that has been submitted independently of our work.[19]

Bit-Scan Instructions. Discovery (D) shows an apparent bug in LLVM's Zen+ scheduling model for bit-scan instructions.[20] The measurements with nanoBench, as well as the instruction latency table provided by AMD [2017], show that a `BSR` instruction has a latency of 4 cycles and requires 4 cycles to be executed in a steady state (a throughput of 0.25 instructions per cycle).

The aliasing component of AnICA's abstract block (D) shows that llvm-mca predicts the latency consistently with measurements: the must-not-alias constraint could not be dropped during generalization, which means that no inconsistency is found if the instruction forms a dependency chain with itself. Without aliasing, llvm-mca underestimates the required execution time. The scheduling model of LLVM [2021, l. 235] provides an explanation: the affected instructions are modeled with a plausible latency, but they only use one of the architecture's four arithmetical/logical units. Therefore, llvm-mca assumes that up to four independent instances of bit-scan instructions can be

[19] https://github.com/llvm/llvm-project/issues/53242
[20] The bit-scan instructions `BSF`/`BSR` determine the index of the least/most significant bit set in their second operand and write it to the first operand.

executed per cycle. We reported this and four other results from similar AnICA discoveries to the LLVM developers. These reports show errors in LLVM's Zen+ scheduling model for a total of 72 of our instruction schemes. The bugs were confirmed and fixed by the developers.

Inaccuracies in Load Operand Usage. With discovery (E), we learn that llvm-mca over-estimates the time required to execute instructions that depend on the result of a preceding instruction and load from memory. The hardware is evidently able to issue a new instruction in every cycle for the corresponding example basic block; the load latency (4 cycles for L1 cache hits on Zen+) completely overlaps with the remaining computation. llvm-mca's model does not account for this behavior: here, the loading instruction always starts executing with the instruction it depends on, causing the load latency to be visible. This problem has also been independently reported in the LLVM issue tracker.[21]

A Microarchitectural Quirk. AnICA's results not only highlight oddities in prediction tools, they can also show unusual behavior in the processor under test. The discovery (F) shows how AnICA automatically found a microarchitectural quirk of the Zen architectures that has been previously described by Abel [2020]: Bit shifts by zero (which invoke a special case where the flag registers are not updated[22]) cause a severe execution time penalty if they are followed by instructions that read the flag registers. llvm-mca's model omits this unexpected corner case of AMD's Zen architectures.

In all of the above cases, AnICA automatically discovered an unexpected inconsistency and provided helpful insight with its generalization. Such insights could otherwise only be gained through tedious manual effort. The fact that we, additionally to finding new bugs, automatically rediscovered several previously reported problems in the llvm-mca predictions indicates that AnICA finds problems that are relevant to the users of llvm-mca.

6 RELATED WORK

To the best of our knowledge, AnICA is the first work to apply differential testing to microarchitectural code analyzers. This section describes other approaches to evaluate such tools and contrasts AnICA to previous work in differential testing.

6.1 Testing Throughput Predictors

Most of the available basic block throughput predictors come with an evaluation of their prediction accuracy. A common approach to evaluating throughput predictors is to measure the relative error from and the correlation with execution time measurements on a chosen set of basic blocks. This is done for OSACA [Laukemann et al. 2018], Ithemal [Mendis et al. 2019], DiffTune [Renda et al. 2020], and uiCA [Abel and Reineke 2022]. They all use basic blocks that were extracted from the binaries of common benchmarks and open source programs whose throughput was measured using different methodologies. Of particular note is BHive [Chen et al. 2019], which is used in the evaluations of DiffTune and uiCA. It is an openly available set of such basic blocks with annotated measured throughputs for several Intel microarchitectures. The evaluation of uiCA identifies cases where assumptions made for the ground truth measurements affect which tool is "more accurate" than another, motivating our differential approach.

Evaluating the prediction accuracy on basic blocks from compiled programs is helpful when the expected use of the tools is on similar basic blocks. However, such basic blocks are lacking when systematically exploring inconsistencies of the tools: of the 2940 instruction schemes that we use in our evaluation, 2002 (i.e., 68%) do not occur in any basic block of the BHive data set and 525

[21] https://github.com/llvm/llvm-project/issues/50899
[22] see, e.g., https://www.felixcloutier.com/x86/sal:sar:shl:shr

(i.e., 18%) of the instruction schemes are enough to represent 99% of the BHive basic blocks. The BHive benchmarks therefore leave a gap in the input space when testing throughput predictors that AnICA intends to close.

BHive also includes an approach to help developers identify problems with their throughput predictors. They cluster basic blocks from the data set based on their use of execution units in the processor (e.g., "vectorized code" and "code with mainly memory operations"). If a tool performs particularly poor on a cluster of basic blocks, the developers can focus on improving support for the associated category. This direction is however considerably less specific than the inconsistencies that AnICA reports to the user.

A concurrently published work by Abel [2022] investigates the prediction accuracy of DiffTune, providing a very simple set of parameters for llvm-mca that outperform the learned DiffTune parameters on the BHive data set in terms of prediction accuracy. These findings are consistent with the unexpected predictions we encountered in our DiffTune campaigns (Section 4.3).

EXEgesis [Chatelet 2018; Google 2018; LLVM 2021a] is a project to validate LLVM's performance models and, consequently, llvm-mca. For a given instruction scheme, EXEgesis executes a microbenchmark on the target machine and measures its performance characteristics. EXEgesis can compare the measured performance to the corresponding information in LLVM's scheduling model. In contrast to AnICA, EXEgesis does not generate experiments with multiple instructions to test their interactions and it is closely integrated with LLVM. Comparisons with other predictors are therefore not supported.

Approaches that infer models for throughput predictors are also evaluated against existing ones on a measured ground truth: PMEvo [Ritter and Hack 2020] and Palmed [Derumigny et al. 2022] both use basic blocks without data dependencies, whose throughput is bound by the processor's functional units. For Palmed, the basic blocks mirror basic blocks observed in the binaries of benchmark suites (without the dependencies). The basic blocks used for PMEvo are more similar to the ones we use here: they are randomly sampled in a way that avoids data dependencies. The evaluation of uops.info [Abel and Reineke 2019] points out some inconsistencies in IACA, but focuses on the usage of resources for single instructions.

6.2 Differential Testing

Differential testing [McKeeman 1998] is commonly used to find bugs in tools where no ground truth is available. There are general frameworks for differential testing tools like Nezha [Petsios et al. 2017], but they mainly focus on effectively exploring a sparse space of inconsistencies. As the space of inconsistencies among basic block throughput predictors is not sparse, there is little benefit in using these frameworks.

AnICA's use of minimization and abstraction can be seen as a form of triage in the usual nomenclature [Manès et al. 2019]. The concepts and notations borrowed from abstract interpretation give us a way to systematically implement a generalized deduplication of inputs.

Previous research already used differential testing for other tools operating on machine code. Several differential fuzzing approaches [Jay and Miller 2018; Paleari et al. 2010; Woodruff et al. 2021] focus on instruction decoders. However, these works differ from our setting in their goal and, consequently, in the inputs that they generate. They generate bit sequences that are (or are close to) machine instructions, for which they check the results of a group of instruction decoders. Since we aim to find inconsistencies in the throughput predictions, we only produce valid instruction sequences. Woodruff et al. [2021] note that they encounter large numbers of nearly identical discoveries that are difficult to deduplicate, making human analysis essential. This mirrors our motivation to use abstraction to reduce the manual investigation effort for analyzing the discoveries.

Revizor [Oleksenko et al. 2021] is a differential testing approach that also generates random instruction sequences. They compare a CPU's behavior with that of a simulation that does not leak information to find side channel attacks. In contrast to AnICA, their instruction sequences include control flow. They define a number of patterns on the dependencies between consecutive instructions that are similar to the constraints represented by our aliasing abstraction. Revizor however uses these patterns only as a metric to control the size of the instruction sequences that they sample. Since abstraction is central to AnICA, we designed the basic block abstraction to cover more complex alias constraints as well as constraints on the involved instructions, which are beyond the scope of Revizor's patterns.

7 POSSIBLE EXTENSIONS

A strength of AnICA is that the throughput predictors under investigation are treated as black boxes. The resulting flexibility opens a range of further use cases for AnICA with no or minor adjustments to the implementation:

Comparing Different Benchmarking Assumptions. When benchmarking the execution time of basic blocks, tools like nanoBench [Abel and Reineke 2020] have to make assumptions on how the blocks should be executed. For instance, they need to initialize registers and memory regions with specific values and choose whether basic blocks should be wrapped in a loop or concatenated sufficiently often. If these choices are configurable (as with nanoBench), AnICA can investigate the effect of different configuration decisions on the measurements. From discovery (F) in our llvm-mca case study (Section 5.3), we would, e.g., expect to find inconsistencies depending on whether the registers are initialized with 0 or not.

Comparing Measurements on Different Microarchitectures. We have presented results for comparing pairs of throughput predictor tools (Section 4.3) as well as for comparing a throughput predictor to measurements on the modeled hardware (Section 5.3). A natural next step would be to compare measurements on two different hardware implementations of an instruction set architecture to each other. This would allow us to investigate performance differences of subsequent generations of CPUs by the same manufacturer, or different trade-offs made by two manufacturers in competing CPU models.

Comparing Port Usage Models. The AnICA algorithm can also be applied to subcomponents of performance models that affect only individual aspects of basic block throughput prediction. For instance, approaches like uops.info [Abel and Reineke 2019], PMEvo [Ritter and Hack 2020], and Palmed [Derumigny et al. 2022] build models for how individual instructions use a CPU's execution resources. These models are able to predict the throughput of basic blocks without data dependencies.

AnICA could therefore investigate differences between the models produced by the individual approaches, as well as deviations between a model and measurements on the actual hardware. For this application domain, the presented basic block abstraction should be adjusted such that only basic blocks with as few data dependencies as possible are sampled. Consequently, the aliasing component of the basic block abstraction then does not capture meaningful information anymore and may be dropped.

Since these approaches infer their models from microbenchmarks, the results of AnICA may be helpful to improve the models by characterizing classes of benchmarks that are missing.

8 CONCLUSION

State-of-the-art tools for basic block throughput prediction often do not agree in their results, for a variety of reasons. To understand and improve them, we proposed AnICA, a tool to differentially test basic block throughput predictors. AnICA uses notions from abstract interpretation to generalize inconsistencies in a systematic way. Our evaluation shows that AnICA can summarize thousands of inconsistencies in a few dozen descriptions that directly lead to high-level insights into the different behavior of the tools.

AnICA further provides interesting points for future research: the core algorithms are independent of the application to throughput predictors and might therefore be of benefit in other domains of differential testing.

9 DATA AVAILABILITY STATEMENT

This article is accompanied by an artifact [Ritter and Hack 2022]. The artifact provides the implementation of the AnICA algorithms used for our evaluation and case studies. It also includes the results of the described AnICA campaigns with a graphical user interface for inspection as well as means for reproducing them. A development version of the AnICA implementation is also available on Github at https://github.com/cdl-saarland/AnICA.

REFERENCES

Andreas Abel. 2020. *Automatic Generation of Models of Microarchitectures.* Ph. D. Dissertation. Universität des Saarlandes. https://d-nb.info/1212853466/34

Andreas Abel. 2022. DiffTune Revisited: A Simple Baseline for Evaluating Learned llvm-mca Parameters. In *Machine Learning for Computer Architecture and Systems 2022.* https://openreview.net/forum?id=dw4evoj6AE

Andreas Abel and Jan Reineke. 2019. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19).* ACM, New York, NY, USA, 673–686. https://doi.org/10.1145/3297858.3304062

Andreas Abel and Jan Reineke. 2020. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* IEEE, 34–46. https://doi.org/10.1109/ISPASS48437.2020.00014

Andreas Abel and Jan Reineke. 2022. uiCA: Accurate Throughput Prediction of Basic Blocks on Recent Intel Microarchitectures. In *ICS '22: 2022 International Conference on Supercomputing, Virtual Event, June 28 - 30, 2022*, Lawrence Rauchwerger, Kirk W. Cameron, Dimitrios S. Nikolopoulos, and Dionisios N. Pnevmatikatos (Eds.). ACM, 33:1–33:14. https://doi.org/10.1145/3524059.3532396

AMD. 2017. *Software Optimization Guide for AMD Family 17h Processors.* AMD.

Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (2011), 1–7. https://doi.org/10.1145/2024716.2024718

Igor Böhm, Björn Franke, and Nigel P. Topham. 2010. Cycle-Accurate Performance Modelling in an Ultra-Fast Just-in-Time Dynamic Binary Translation Instruction Set Simulator. In *Proceedings of the 2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2010), Samos, Greece, July 19-22, 2010*, Fadi J. Kurdahi and Jarmo Takala (Eds.). IEEE, 1–10. https://doi.org/10.1109/ICSAMOS.2010.5642102

Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014.* IEEE Computer Society, 114–129. https://doi.org/10.1109/SP.2014.15

Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register Allocation Via Coloring. *Comput. Lang.* 6, 1 (1981), 47–57. https://doi.org/10.1016/0096-0551(81)90048-5

Guillaume Chatelet. 2018. llvm-exegesis: Automatic Measurement of Instruction Latency/Uops. https://lists.llvm.org/pipermail/llvm-dev/2018-March/121814.html Accessed: 2021-07-22.

Yishen Chen, Ajay Brahmakshatriya, Charith Mendis, Alex Renda, Eric Atkinson, Ondrej Sỳkora, Saman Amarasinghe, and Michael Carbin. 2019. BHive: A Benchmark Suite and Measurement Framework for Validating x86-64 Basic Block Performance Models. In *2019 IEEE international symposium on workload characterization (IISWC). IEEE.* https:

//doi.org/10.1109/IISWC47752.2019.9042166

Yuting Chen and Zhendong Su. 2015. Guided Differential Testing of Certificate Validation in SSL/TLS implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 793–804. https://doi.org/10.1145/2786805.2786835

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

Nicolas Derumigny, Théophile Bastian, Fabian Gruber, Guillaume Iooss, Christophe Guillon, Louis-Noël Pouchet, and Fabrice Rastello. 2022. PALMED: Throughput Characterization for Superscalar Architectures. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Korea, Republic of, April 2-6, 2022*, Jae W. Lee, Sebastian Hack, and Tatiana Shpeisman (Eds.). IEEE, 106–117. https://doi.org/10.1109/CGO53902.2022.9741289

Andrea Di Biagio. 2018. llvm-mca: A Static Performance Analysis Tool. http://lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html

Rafael Dutra, Jonathan Bachrach, and Koushik Sen. 2019. GUIDEDSAMPLER: Coverage-guided Sampling of SMT Solutions. In *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, Clark W. Barrett and Jin Yang (Eds.). IEEE, 203–211. https://doi.org/10.23919/FMCAD.2019.8894251

Google. 2018. EXEgesis: Automatic Measurement of Instruction Latency/Uops. https://github.com/google/EXEgesis Accessed: 2021-07-22.

Intel. 2012. Intel Architecture Code Analyzer. https://software.intel.com/en-us/articles/intel-architecture-code-analyzer

Nathan Jay and Barton P. Miller. 2018. Structured Random Differential Testing of Instruction Decoders. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 84–94. https://doi.org/10.1109/SANER.2018.8330199

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California. https://doi.org/10.1109/CGO.2004.1281665

Jan Laukemann, Julian Hammer, Johannes Hofmann, Georg Hager, and Gerhard Wellein. 2018. Automated Instruction Stream Throughput Prediction for Intel and AMD Microarchitectures. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 121–131. https://doi.org/10.1109/PMBS.2018.8641578

LLVM. 2021. LLVM 13 Scheduling Model for AMD Zen/Zen+ CPUs. https://github.com/llvm/llvm-project/blob/release/13.x/llvm/lib/Target/X86/X86ScheduleZnver1.td. Accessed: 2022-04-07.

LLVM. 2021a. llvm-exegesis - LLVM Machine Instruction Benchmark. https://llvm.org/docs/CommandGuide/llvm-exegesis.html Accessed: 2021-07-22.

LLVM. 2021b. llvm-mca - LLVM Machine Code Analyzer. https://llvm.org/docs/CommandGuide/llvm-mca.html Accessed: 2021-11-15.

Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2946563

William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107. http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, Long Beach, California, USA, 4505–4515. http://proceedings.mlr.press/v97/mendis19a.html

Georg Ofenbeck, Ruedi Steinmann, Victoria Caparrós Cabezas, Daniele G. Spampinato, and Markus Püschel. 2014. Applying the Roofline Model. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*. IEEE Computer Society, 76–85. https://doi.org/10.1109/ISPASS.2014.6844463

Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. 2021. Revizor: Fuzzing for Leaks in Black-box CPUs. *CoRR* abs/2105.06872 (2021). arXiv:2105.06872 https://arxiv.org/abs/2105.06872

Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. N-Version Disassembly: Differential Testing of X86 Disassemblers. In *Proceedings of the 19th International Symposium on Software Testing and Analysis* (Trento, Italy) *(ISSTA '10)*. Association for Computing Machinery, New York, NY, USA, 265–274. https://doi.org/10.1145/1831708.1831741

Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, Matthew B. Dwyer and Frank Tip (Eds.). ACM, 199–209. https://doi.org/10.1145/2001420.2001445

Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. 2017. NEZHA: Efficient Domain-Independent Differential Testing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 615–632. https://doi.org/10.1109/SP.2017.27

Alex Renda, Yishen Chen, Charith Mendis, and Michael Carbin. 2020. DiffTune: Optimizing CPU Simulator Parameters with Learned Differentiable Surrogates. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 442–455. https://doi.org/10.1109/MICRO50266.2020.00045

Fabian Ritter and Sebastian Hack. 2020. PMEvo: Portable Inference of Port Mappings for Out-of-Order Processors by Evolutionary Optimization. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 608–622. https://doi.org/10.1145/3385412.3385995

Fabian Ritter and Sebastian Hack. 2022. *AnICA: Analyzing Inconsistencies in Microarchitectural Code Analyzers (Artifact)*. https://doi.org/10.5281/zenodo.6818171

Andres Charif Rubial, Emmanuel Oseret, Jose Noudohouenou, William Jalby, and Ghislain Lartigue. 2014. CQA: A Code Quality Analyzer Tool at Binary Level. In *21st International Conference on High Performance Computing, HiPC 2014, Goa, India, December 17-20, 2014*. IEEE Computer Society, 1–10. https://doi.org/10.1109/HiPC.2014.7116904

Samuel Williams, Andrew Waterman, and David A. Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (2009), 65–76. https://doi.org/10.1145/1498765.1498785

William Woodruff, Niki Carroll, and Sebastiaan Peters. 2021. *Differential Analysis of x86-64 Instruction Decoders*. Technical Report. 152–161 pages. https://doi.org/10.1109/SPW53761.2021.00029