JPortal: Precise and Efficient Control-Flow Tracing for JVM Programs with Intel Processor Trace

Zhiqiang Zuo*
Nanjing University, China
zqzuo@nju.edu.cn

Kai Ji*
Yifei Wang*
Wei Tao*
Nanjing University, China
{jikai,wef,taowei}@smail.nju.edu.cn

Linzhang Wang*
Nanjing University, China
lzwang@nju.edu.cn

Xuandong Li*
Nanjing University, China
lxd@nju.edu.cn

Guoqing Harry Xu
University of California, Los Angeles
harryxu@cs.ucla.edu

Abstract

Hardware tracing modules such as Intel Processor Trace perform continuous control-flow tracing of an end-to-end program execution with an ultra-low overhead. PT has been used in a variety of contexts to support applications such as testing, debugging, and performance diagnosis. However, these hardware modules have so far been used only to trace native programs, which are directly compiled down to machine code. As high-level languages (HLL) such as Java and Go become increasingly popular, there is a pressing need to extend these benefits to the HLL community. This paper presents JPortal, a JVM-based profiling tool that bridges the gap between HLL applications and low-level hardware traces by using a set of algorithms to precisely recover an HLL program's control flow from PT traces. An evaluation of JPortal with the DaCapo benchmark shows that JPortal achieves an overall 80% accuracy for end-to-end control flow profiling with only a 4-16% runtime overhead.

CCS Concepts: • **Software and its engineering → Compilers**; **Software maintenance tools**; Software performance.

Keywords: control-flow tracing, JVM, Intel PT

ACM Reference Format:
Zhiqiang Zuo, Kai Ji, Yifei Wang, Wei Tao, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. 2021. JPortal: Precise and Efficient Control-Flow Tracing for JVM Programs with Intel Processor Trace. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454096

*Also with State Key Laboratory for Novel Software Technology at Nanjing University. Zhiqiang Zuo is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '21, June 20–25, 2021, Virtual, Canada
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454096

1 Introduction

Modern CPUs are equipped with tracing modules such as Intel processor trace (PT) [19] and ARM embedded trace macrocell (ETM) [18], providing ultra efficiency for control-flow profiling of end-to-end program executions. With these hardware traces, it is possible to reconstruct a program's complete execution flow, enabling a wide spectrum of client applications in testing [5, 28, 47, 61–63], debugging [27, 30, 64, 65], performance analysis [50, 51], *etc.* For example, with a program's control flow, various execution statistics, such as function and statement coverage, path profiles, call tree profiles, *etc.* are all close at hand. Furthermore, hardware traces contain event timestamps, enabling performance analysis such as detection of invocation hot spots. Compared to software-based tracing that is often limited by compiler infrastructures and prohibitively expensive (*i.e.*, slowdowns of dozens to thousands of times), hardware-based tracing has a negligible overhead (2-5% [51, 56]) and does not need any modification to program code, enabling tasks that were previously unimaginable, such as *in-production* profiling across languages, between applications, or even into the OS kernel.

Problems. As of date, hardware-based tracing has been applied only to native programs running on bare metal machines [30]. This is because processors can only profile hardware instructions; for native programs, these instructions can be easily mapped, with the help of compilation metadata, back to the source code they are compiled from. As high-level languages (HLL) such as Java, Go, and Scala are playing an increasingly important role in modern computing, there is a pressing need to extend the efficiency benefit provided by such hardware modules to HLL communities.

However, there are significant challenges in tracing an HLL program with hardware, primarily due to the language runtime on which the HLL program executes. A language

runtime such as the JVM alternates between two execution modes: it starts an execution via interpretation and switches to running the JIT-compiled code when methods become sufficiently hot. As such, for those languages, the instructions seen by CPU are not only structurally different from their HLL source/byte code, but also generated by different code generators under different compilation strategies—the interpreter uses templates to generate code while the JIT compiles code with many optimizations. To make matters worse, the language runtime inserts code to perform various runtime checks (such as barriers and bound checks), creating a huge structural gap between what a developer sees and what CPU executes.

A naïve approach is to selectively instrument a program (*e.g.*, at each important control-flow point), augmenting hardware traces with "delimiters" generated from instrumented code. However, instrumenting even a small number of control-flow points can yield a non-trivial overhead [24, 25, 35], which is often too high to be acceptable in producing settings. In fact, hardware-based tracing is appealing particularly due to its ultra-low overheads; maintaining such overheads precludes use of any instrumentation-based approaches.

JPortal. This work develops JPortal, a tool that can precisely reconstruct the control flow in Java bytecode from traces collected by hardware, and in particular, by Intel Processor Trace (PT). Our idea is to develop a *postmortem* static analysis that analyzes a hardware trace to establish a mapping (or projection) from the trace to a path on the (statically built) interprocedural control flow graph (ICFG) of the program. While this idea appears simple at a high level, doing so *precisely* (*i.e.*, the reconstructed bytecode profiles should faithfully represent what hardware records in the trace) and *efficiently* requires overcoming three major challenges:

First, the interpreter and JIT compiler stand in the way. Mapping low-level instructions back to a high-level CFG is essentially a de-interpretation and de-compilation process. JPortal builds the mapping by leveraging the metadata the JVM maintains in its interpreter and JIT compilers (*i.e.*, C1 and C2) (§3). In particular, the interpreter interprets each Java bytecode instruction by running a piece of code defined by a *template table*. This table can be used to decode the interpreted code. For the JIT compilation, the JIT compilers record the high-to-low-level mapping into debug metadata at each step of compilation (*e.g.*, from bytecode to IR, between different levels of IR, and from IR to machine code). This metadata is originally used for providing debugging information at exceptions; JPortal leverages it for decoding.

Second, hardware traces are often incomplete—a trace collected by PT, especially for a long-running program, can miss information for an arbitrary number of execution periods, each at an arbitrary length. This is because PT can potentially produce hundreds of megabytes of trace data per CPU per second, a rate faster than data can be exported to file, and sometimes faster even than can be recorded to memory. Data loss can occur at any point in a trace, making reconstruction impossible—two consecutive events in the trace may stem from methods far apart; to make matters worse, they can even come from code generated by different generators (*i.e.*, interpreter and JIT). In our experiments, the percentage of missing data ranges from 22.2% to 28.0% under a 128MB per-core trace buffer. Under a smaller buffer size (*e.g.*, 64M), this percentage could go beyond 50%.

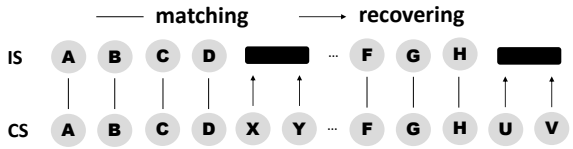


Figure 1. Example of trace recovery.

Missing data leads to trace segmentation—a segment of instructions in the trace can end *unexpectedly*, and the next instruction, as the starting point of a new segment, can be far away from the previous segment. Segmentation creates a significant challenge in *projecting* each segment onto the program's ICFG. Once an unexpected instruction is encountered during decoding, which node/edge on the ICFG does this instruction map to? To tackle this challenge, we formulate control-flow projection as a problem of *disambiguation in NFA-based string matching*. In particular, we treat the static ICFG of a program as a non-deterministic finite state automaton (NFA) and a hardware trace as a segmented string. There may exist multiple paths in the NFA that can match the string (*i.e.*, ambiguity) and our goal is to disambiguate the matching to find one single path that most likely corresponds to the actual execution represented by the string. We develop an *abstraction-guided* projection algorithm that can efficiently find a likely path on the ICFG for a given trace segment (§4).

Third, missing data creates holes between segments. To maximize the profiling accuracy, we must recover the missing data from data that have been collected. To this end, our idea is to recover the incomplete segments (IS) in the trace using the complete segments (CS) whose *contexts* match those of the former (*i.e.*, filling holes with the help of information before each hole). Figure 1 illustrates this idea. However, finding the matching CS is a daunting task because (1) a trace can be extremely long (*i.e.*, efficiency concern) and (2) there can be many matching CSes and we need to narrow the search down to those that *most likely* correspond to the execution of the IS in question (*i.e.*, precision concern).

To prune the search space, we guide matching with contexts, represented as a hierarchy of abstractions (*e.g.*, call/returns, control structures, *etc.*). As such, the matching process can be formulated as a series of *abstraction refinements*—if the context of a complete segment s does not match that of an incomplete segment i at a high abstraction level (*e.g.*, call trace), s can be quickly filtered out. If it does, we *refine the*

abstraction and perform context matching at a lower abstraction level (*e.g.*, control structures). Matching at each level also ranks candidates that advance to the next level based on the matching degree (*e.g.*, how much their suffixes overlap). Finally, JPortal inspects the ranking of the candidates whose contexts match those of *i at all levels* and picks the highest-ranked one as a reference to fill the holes in *i* (§5).

Results. We have implemented JPortal on OpenJDK 12 and evaluated it with the DaCapo benchmark. Our results show that JPortal achieves an overall 80% accuracy in *end-to-end control-flow profiling* with only a 4-16% overhead. JPortal is publicly available: https://github.com/JPortal-system.

2 Background

Intel Processor Tracing (PT). PT is a new hardware feature that exploits a dedicated hardware facility to capture the control flow of instructions with minimal overheads. After properly configured, Intel PT records control-flow information such as branch targets and branch taken indications, and encodes the collected information to produce *trace packets*. There are several types of trace packets generated by PT, including Packet Generation Enable (PGE), Packet Generation Disable (PGD), Taken Not-Taken (TNT), Target Instruction Pointer (TIP), Flow Update Packet (FUP), and Time-Stamp Counter (TSC). PGE and PGD packets record the addresses of the first and last instructions traced, indicating the beginning and the end of tracing, respectively. During execution, the following three kinds of packets record the control flow information. Particularly, TNT packets record whether conditional branches are taken or not. TIP packets log the targets of indirect branches (*e.g.*, `call` and `ret`). FUPs provide the source instruction pointer (IP) for asynchronous events (*e.g.*, interrupts and exceptions). In addition to the control flow information, other information such as timestamps can also be recorded to assist in performance debugging.

To minimize the size of packets generated, PT adopts various compression techniques. For example, only one bit is used in TNT to indicate the direction of a conditional. For the unconditional jumps, like `JMP` and `CALL`, no trace packet is generated since the target addresses can be directly inferred from the application assembly. For TIP packets, PT compresses the target address if the upper bytes match the previous address traced. Since the trace data collected is highly compressed, a decoding process is required to retrieve the program control flow. With the trace packets and program's native code as input, a software decoder [9] can reconstruct the control flow of the program executed.

Trace packets are directly written into physical memory by PT, bypassing cache and TLB to reduce performance impact. Multiple discontinuous buffers in the physical address space can be used to maintain the tracing data via a table-like data structure. When a buffer is full, it triggers an interrupt.

Java Virtual Machine (JVM). The JVM is a language runtime that compiles and executes programs (written in many high-level languages such as Java, Scala, Python, *etc.*) that can be compiled to Java bytecode. The JVM takes a bytecode program as input and alternates its execution between interpreted and JIT-compiled code.

In the interpretation mode, a template interpreter generates, for each bytecode instruction, a piece of assembly code implementing the instruction's functionality. During the JVM's initialization, the assembler translates the template assembly code into machine code and loads it into memory. Each bytecode instruction is executed by simply jumping to the starting address of its corresponding machine code.

Once a method becomes sufficiently hot, the JVM switches to the compilation mode. The JIT compiler compiles the method to machine code using two compilers (C1 and C2), each performing various optimizations.

3 Dealing with Compiler Intricacies

We first present how JPortal deals with the intricacies of the JVM's interpreter and JIT.

JPortal consists of two components—*online collection* and *offline decoding*. The online component collects the runtime information needed by decoding, including (1) hardware tracing data (collected by PT) as well as (2) meta-information of machine code that is necessary for decoding. PT records trace packets when machine code is running. These trace packets are periodically dumped to files. In addition to hardware traces that contain only branch information (*e.g.*, the target of a jump), the metadata of the machine code is also needed. For example, given the target address of a jump in a hardware trace, JPortal needs to understand which basic block the address falls in to recover the control flow. JPortal obtains such metadata from the JVM after its initialization.

With a hardware trace and the meta-information of machine code, JPortal's offline component decodes the trace and reconstructs the control flow for the program. Since the bytecode program is executed under different execution modes, both the online collection of machine-code metadata and the offline decoding algorithm must be developed separately for the interpreter and the JIT.

3.1 Flow Reconstruction for Interpreted Code

In the interpretation mode, the machine-code metadata is the *address range* of each instruction template. JPortal collects such information automatically during the JVM's initialization. For certain cases where the machine code for a byte code handler is non-contiguous, multiple sub-ranges could be recorded. Figure 2 shows an example of decoding interpreted code: Figure 2(a) and 2(b) depict the source code and bytecode of a program, respectively.

Each bytecode instruction is interpreted by running the instruction's corresponding machine-code template. Figure 2(c) shows the metadata for a number of bytecode instructions relevant to the example. For instance, when interpreting the bytecode *iconst_0*, the execution jumps to the machine code

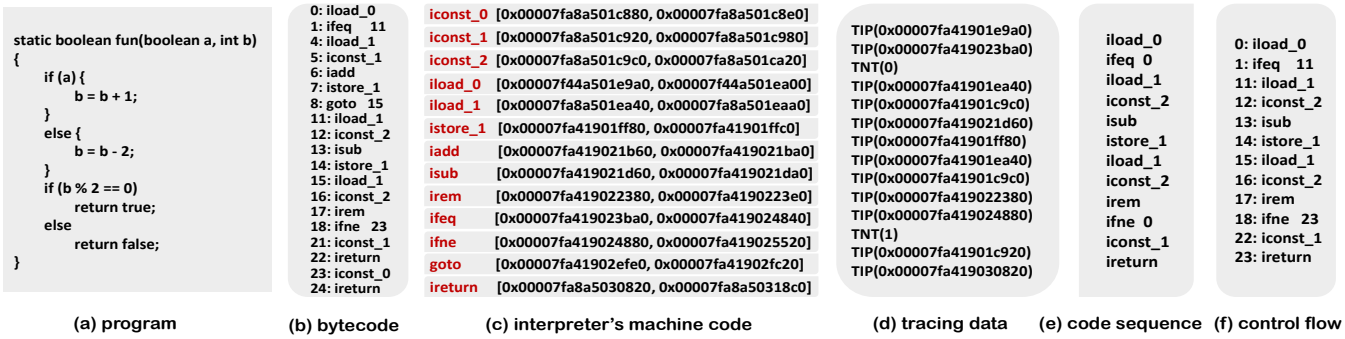


Figure 2. Flow reconstruction for interpreted code: (a) source code; (b) its bytecode; (c) the interpreter's machine-code metadata for relevant bytecode instructions; (d) its hardware trace; (e) the decoded bytecode instruction sequence; and (f) the control flow recovered (numbers show bytecode offset).

that is located in the address range [`0x00007fa8a501c880`, `0x00007fa8a501c8e0`], and runs the code there. The hardware trace recorded by PT is shown in Figure 2(d) where each TIP indicates a jump and the address attached shows the target. For example, the first TIP packet represents a jump to address `0x7fa41901e9a0`, which falls into the address range of *iload_0*'s machine code template.

The offline decoding algorithm consists of two steps. The first step is *decoding a hardware trace* into bytecode instructions. For template-based interpretation, the mapping between the executed machine code and each bytecode instruction is statically defined by the JVM's code template table. As a result, JPortal identifies each bytecode instruction by matching the templates' address ranges against the entry addresses recorded for each bytecode handler. In doing so, we can always precisely determine the bytecode instruction interpreted. Figure 2(e) shows the decoded sequence of bytecode instructions from the hardware trace.

However, with the sequence of bytecode instructions, it is unclear which program path has been executed. As such, the second step is to identify the *flow* between these instructions. To this end, we need the program's ICFG—the executed instructions should correspond to a path in the ICFG. If the hardware trace is complete (*e.g.*, from the first to the last instruction of `main`) and the program runs normally, finding that path is a rather simple task; Figure 2(f) shows the reconstructed control flow for the hardware trace. However, various ambiguities can be introduced in the matching process in a realistic setting (with dynamic control flow and missing data). For example, exception throwing and handling can take the execution to an arbitrary point; interpreted and compiled code can alternate in the execution; to make matters worse, missing data can take the offline analysis from one point to another point that is arbitrarily far, and the path in the middle is completely lost.

3.2 Flow Reconstruction for JITed Code

Under the JIT compilation, a method is compiled directly to machine code. Hence, for each method, its JITed code (and its address range) needs to be collected as machine-code metadata. However, different from the interpreter's code template that is persistent throughout execution, the JITed code is subject to garbage collection (GC) and hence can be removed. As such, JPortal exports (1) the compiled code of a method and (2) its address range (to disk) before it is reclaimed by GC. Such information helps JPortal recognize targets of jumps. To appropriately reconstruct the bytecode-level control flow from the JITed code, we also need the mapping between each bytecode instruction i and the machine-level instructions to which i is compiled. Such a mapping is maintained as debug information in the JIT and accessible to JPortal.

Figure 3. Flow reconstruction for JITed code: (a) compiled code for the program in Figure 2(a); (b) the debug information generated by compilation; (c) its hardware trace; and (d) the machine instructions executed.

For flow reconstruction, JPortal also uses a two-step approach: decoding the hardware trace to control flow of native code and reconstructing the control flow of bytecode. The first step is done by directly decoding the hardware trace on the compiled code via the decoding library [9] provided by Intel, and the second step is done with the help of the JIT's debug information which gives the mapping relation between the compiled code and the respective bytecode.

For example, Figure 3(a) shows the compiled machine code for the program in Figure 2(a). When the compiled code is running, the hardware trace shown as Figure 3(c) is generated. The TIP packet TIP (`0x00007fa4195bfbe0`) indicates the starting of the execution. The TNT packet records the respective branch results in the compiled code. After decoding the trace with the help of the machine code, JPortal acquires the instruction sequence, shown in Figure 3(d). Each line indicates the IP range within which machine code is executed. Next, JPortal reconstructs the control flow of bytecode via the debug information shown as Figure 3(b), which is generated by the JIT compiler during compilation. This step produces the same control flow as shown in Figure 2(f).

JPortal relies on JIT's debug information for decoding. The quality of the debug metadata plays an important role in JPortal's decoding. In OpenJDK 12 where JPortal is implemented, such information is precise enough so that the precision loss in our experiments is relatively small (see Figure 7).

4 Decoding and Reconstruction

This section presents how JPortal decodes a segmented hardware trace and reconstructs its control flow. To disambiguate the use of terms, we use "decode" to refer to the process of translating a hardware trace into a sequence of bytecode instructions (*i.e.*, the first step discussed in §3), "reconstruct" to refer to the projection of the decoded sequence onto the ICFG (*i.e.*, the second step), and "recover" to refer to the process of "hole filling" for missing data. This section focuses solely on decoding and reconstruction, while recovery will be the focus of §5.

Challenges. With the help of the machine-code metadata, the first (decoding) step is straightforward. The second step is significantly more challenging due to data loss, which causes the decoded bytecode sequence to be segmented (into an arbitrary number of subsequences). While this section does not focus on recovery of missing data, the fact that a subsequence can potentially map to many ICFG paths already makes flow reconstruction a difficult problem. Assuming the decoded instruction sequence begins at the very first instruction in `main`, the reconstruction process is trivial until it hits a mismatch—the next bytecode instruction does not match the expected ICFG node, indicating that a new subsequence starts. The question here is: *which ICFG node should be used as the starting point to project this new subsequence?*

Upon a mismatch between an instruction and an ICFG node, we need to find a new ICFG node and resume projection from there.

During decoding, the segmented subsequences can be easily identified and separated. The hardware trace includes a type of meta-events (*i.e.*, perf_record_aux events including the data loss flag and timestamp) indicating the points where data loss happens. JPortal leverages these events to localize data loss and separate subsequences. This section focuses on the following problem: given an arbitrary subsequence ω that does not have any missing data in itself, how to find the subpath on the ICFG that corresponds to ω (*i.e.*, projection)?

Algorithm 1: Enumerate and test.

Input: ICFG $\mathcal{G}$, a bytecode sequence ω
Output: A path in $\mathcal{G}$ corresponding to ω

```
foreach Node n in G do
    A ← ConstructNFA(G, n);
    if IsAccepted(A, ω) then
        return the sequence of transitions in A that leads
          to acceptance;
```

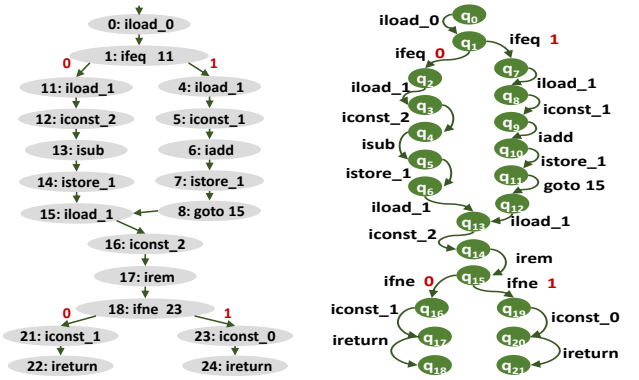
Problem Formulation. First, we formulate control-flow reconstruction as a problem of matching a given string (*i.e.*, subsequence of bytecode instructions) against an automaton-based representation of the program's ICFG. Given an ICFG $\mathcal{G}$ where each node n represents a bytecode instruction s and each edge represents a "potential-next-instruction-to-execute" relation, we define a mapping I that maps each CFG node to the bytecode instruction it represents (*i.e.*, $I(n) = s$). With these notations, we present our formulation below:

Definition 4.1 (ICFG as a Nondeterministic Finite Automaton (NFA)). Given an ICFG $\mathcal{G}$ and the entry node n_0 in the ICFG, we formulate the ICFG as an NFA $\mathcal{A}$ which is a five tuple $(Q, \Sigma, \delta, q_0, F)$ defined as follows:

1. Q is a finite set of states; each state $q \in Q$ corresponds to a node $n \in \mathcal{G}$, representing that node n has been matched during projection; we define a relation N that maps each state q to its corresponding ICFG node (*i.e.*, $N(q) = n$);
2. Σ is a finite set of symbols, each of which represents a bytecode instruction that appears in the trace;
3. $\delta : Q \times \Sigma \to 2^Q$ is a transition function that takes as input a state $q \in Q$ and an instruction $s \in \Sigma$, and returns a subset of states $O \subset Q$, such that each state $o \in O$ corresponds to a successor node of $N(q)$ in $\mathcal{G}$ and s matches the bytecode instruction represented by that node (*i.e.*, $s = I(N(o))$);
4. $S \subseteq Q$ is a set of starting states; since a hardware trace can start at any point, each state in Q could be a starting state;
5. $F \subseteq Q$ is the set of accepting states; since a hardware trace can end at any point, each state in Q could be an accepting state.

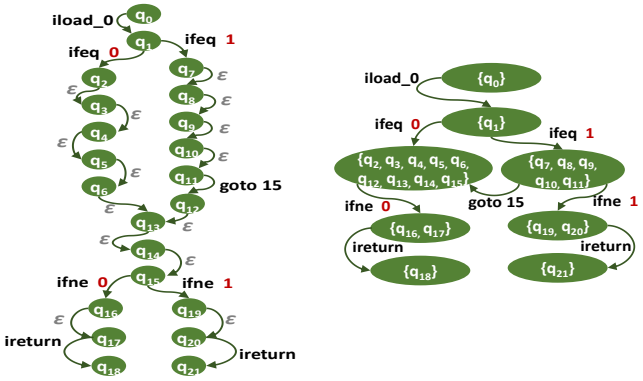
To facilitate reconstruction, we let each method m have an artificially-created starting state $q^m_{init} \in Q$ such that $\delta(q^m_{init}, s^m_0) = q^m_0$ where s^m_0 is the first bytecode instruction in m and q^m_0 is the state corresponding to the entry node n^m_0 of m's CFG, that is, $I(n^m_0) = s^m_0 \wedge N(q^m_0) = n^m_0$.

Figure 4 depicts an example of an (I)CFG for the program in Figure 2(a) (Figure 4a) and its modeling NFA (Figure 4b), assuming that the starting instruction is *iload_0*.



(a) CFG **(b)** NFA

Figure 4. (a) Part of the ICFG of the program in Figure 2(a); (b) its modeling NFA.



(a) Abstract NFA **(b)** DFA

Figure 5. (a) ANFA of the NFA shown in Figure 4b; (b) its simplified DFA after eliminating ϵ-transitions.

Given an NFA $\mathcal{A}$ that models an ICFG and an arbitrary subsequence ω of bytecode instructions decoded, a naïve approach to control flow reconstruction is to use an *enumerate-and-test* algorithm shown in Algorithm 1. In particular, for each state in $\mathcal{A}$, we test if it has an outgoing transition labeled with the first instruction in ω. If so, the state can be used as a starting point for projection. Otherwise, the algorithm continues to try testing the next state. The time complexity of each step is thus $O(t^2)$ where t is the number of states in Q. With each possible starting point, the complexity of matching the subsequence is $O(|\omega|t^2)$. Since there are at most $|Q|$ starting points in $\mathcal{A}$, the complexity of the *enumerate-and-test* algorithm is $O(|Q||\omega|t^2)$. Given that $|Q|$, $|\omega|$ and t are all extremely large, this naïve algorithm can be prohibitively expensive.

Abstraction-Guided Flow Reconstruction. To reduce the search space, we propose an algorithm inspired by the idea of counter-example guided abstraction refinement [29]. In particular, we create a high-level abstraction for both $\mathcal{A}$ and ω. Our basic idea is instead of matching instructions one by one, we can first test if their corresponding abstractions match. If the abstract bytecode sequence can be accepted by the abstract NFA, a *refinement* can be performed to lower the abstraction level and do the matching at the concrete level. Any mismatch at a high level of abstraction would quickly invalidate the starting state and direct the algorithm to try the next state. To formally describe this algorithm, we first define our abstractions:

Definition 4.2 (Abstraction of Bytecode Sequence). Given a decoded bytecode sequence $\omega = \langle b_1, b_2, \ldots, b_n \rangle$, an abstraction of ω, denoted as $\alpha_s(\omega)$, produces an abstract sequence $\widehat{\omega} = \langle \widehat{b_1}, \widehat{b_2}, \ldots, \widehat{b_m} \rangle$ such that the following conditions hold:

- for each $i \in [1, m]$, $\widehat{b_i}$ is a control-flow instruction, *i.e.*, jump, branch, call, or return;
- for any indices $i, j \in [1, n]$ such that $i < j$, and b_i and b_j are control-flow instructions, there exist indices $p, q \in [1, m]$ such that $p < q$, $\widehat{b_p} = b_i$ and $\widehat{b_q} = b_j$.

In fact, the abstraction function turns a bytecode sequence ω into a subsequence $\widehat{\omega}$ of ω which only consists of control-flow instructions in ω.

Similarly, we define the abstraction of an NFA as follows.

Definition 4.3 (Abstract NFA). Given an NFA $\mathcal{A} = (Q, \Sigma, \delta, S, F)$, an abstraction of $\mathcal{A}$ produces an abstract NFA (ANFA), denoted as $\widehat{\mathcal{A}} = (\widehat{Q}, \widehat{\Sigma}, \widehat{\delta}, \widehat{S}, \widehat{F})$ such that:

- $\widehat{Q} = Q$;
- $\widehat{\Sigma} = \Sigma \cup \{\epsilon\} \setminus \bigcup\{inst_i\}$ where $\bigcup\{inst_i\}$ represents all such bytecode instructions that are not control related;
- for each transition $\delta(q_i, inst) = q_j$, there is a corresponding transition $\widehat{\delta}(q_i, \epsilon) = q_j$ if $inst \notin \widehat{\Sigma}$; otherwise, there is a transition $\widehat{\delta}(q_i, inst) = q_j$;
- $\widehat{S} = S$;
- $\widehat{F} = F$.

Theorem 4.4 (Necessary Condition of Acceptance). *Given an NFA $\mathcal{A}$ and a bytecode sequence ω, ω must not be accepted by $\mathcal{A}$ if the abstract bytecode sequence $\widehat{\omega}$ is not accepted by the ANFA $\widehat{\mathcal{A}}$.*

This is straightforward—the ANFA extracts the control flow "skeleton" of the program while $\widehat{\omega}$ extracts the control-flow instructions in the bytecode sequence; if they do not match, it is impossible for $\mathcal{A}$ to accept ω. A proof can be done by contradiction; details are omitted due to space constraints.

By eliminating ϵ-transitions and merging states, we can turn an ANFA into a DFA. Figure 5a and Figure 5b show the ANFA and its DFA (after simplification) for the NFA shown in Figure 4b, respectively. Algorithm 2 shows our algorithm that performs abstraction-guided control flow reconstruction. The two-level matching leads to significantly reduced search space and hence increased efficiency.

Algorithm 2: Abstraction-guided control flow reconstruction.

Input: a control flow graph $\mathcal{G}$, a bytecode sequence ω
Output: the control flow trace corresponding to ω

```
1 ω̂ ← α_s(ω);
2 foreach Node n ∈ G do
3     A ← ConstructNFA(G, n);
4     Â ← α_a(A);
5     if IsAccepted(DFA(Â), ω̂) then
6         if IsAccepted(DFA(A), ω) then
7             return the sequence of transitions in A that
                leads to acceptance;
```

Discussions. Note that Ohmann *et al.* [45] also formulated CFGs as finite state automata so as to efficiently (in polynomial time) answer control-flow queries for incomplete failure reports. By encoding both failure constraints of reports and user queries in the *unreliable trace languages*, a *Possible/Impossible* answer is given by checking intersection-emptiness over context-insensitive traces. Instead of testing for language-level intersection-emptiness, our control-flow reconstruction simply determines whether a dynamic sequence can be accepted by an NFA.

Another way to model an ICFG is to use the pushdown automaton (PDA). The difference between the formulations with NFA and PDA is whether the context information (*e.g.*, call-return) is taken into consideration during matching. While a PDA-based representation can filter out infeasible interprocedural paths, it is not necessary for our scenario. Our abstraction-guided approach rules out the possibility of producing an infeasible path as it must not be accepted by our abstraction.

As a statically-built ICFG can be imprecise (including infeasible paths or missing feasible paths), another potential issue is whether the imprecision of the ICFG can lead to reconstruction imprecision? For the infeasible paths in the ICFG, it is not a concern because the instruction sequence decoded from the hardware trace represents the dynamic execution flow, which can only correspond to feasible paths in the ICFG. However, it could lead to performance issues due to unnecessary matching. Another issue here is that certain feasible paths can be lost in the ICFG due to dynamic language features such as reflection. To tackle this problem, if a method invocation is seen in the captured instruction sequence but there does not exist a corresponding call node in the ICFG, JPortal inspects all potential callback methods in the program to find a match.

5 Abstraction-Guided Data Recovery

This section discusses recovery of missing data. In particular, given a hardware trace segmented into n subsequences (ω_1, ω_2, ... ω_n), how can we find a path on the ICFG that can connect the last instruction of ω_i and the first instruction of ω_{i+1} for each $i \in [1, n)$?

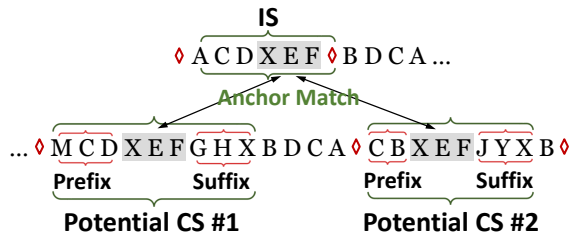


Figure 6. An overview of missing data recovery: the anchor instructions are "XEF".

Problem Formulation. Since data loss occurs frequently, it is necessary to recover the missing parts of the control flow by leveraging information from the completed parts. Our key insight here is that if two segments of instructions share the same (or similar) context (*e.g.*, prefix of a certain length), they are likely to represent the same execution path on the ICFG. If the first segment has missing data, we can recover the data with information extracted from the second segment. Here the terms "incomplete" and "complete" are both relative — a segment a that misses data itself may contain a sequence of instructions (s) that can be used to fill a hole in another segment b; in this case, a and b are complete and incomplete, respectively, with respect to s. We first formulate data recovery as a problem of sequence matching:

Definition 5.1 (Data Recovery). Given an incomplete trace segment $IS = \langle t_{p_m}, \ldots, t_{p_2}, t_{p_1}, \diamond \rangle$ where each t_{p_i} represents a bytecode instruction and $\diamond$ indicates an unknown subsequence, we formulate the problem of recovering $\diamond$ as finding a complete trace segment $CS = \langle t_{p_x}, \ldots, t_{p_2}, t_{p_1}, t'_{s_1}, t'_{s_2}, \ldots, t'_{s_y} \rangle$ such that IS and the part of CS before t'_{s_1} share a common suffix $t_{p_y}, \ldots, t_{p_2}, t_{p_1}$ ($y \leq m \wedge y \leq x$) and there does not exist another sequence CS' such that the length of such a common suffix between CS' and IS is larger than y.

Figure 6 illustrates our main idea. First, we consider each subsequence that starts at an instruction right after a $\diamond$ (*i.e.*, a hole) and ends at the next $\diamond$ as an IS. Our goal is to replace its end $\diamond$ with information learned from a CS. To identify a potential CS for matching, we use the last x instructions before the $\diamond$ in the IS as *anchor instructions*. In Figure 6, the three instructions XEF ($x = 3$) are used as anchor instructions. With these instructions, we quickly locate two potential *CSes*, each of which is divided into two parts: a prefix that contains instructions before the anchor instructions and a suffix that contains instructions after. Finding these potential *CSes* is easy due to the small number of anchor instructions used. Given each potential CS, we compare its prefix p with the IS from their end instructions (*i.e.*, in the reverse order); the CS

Algorithm 3: Data recovery: basic algorithm.

Input: *IS*, all *CSes* that match the anchor instructions of *IS*
Output: One *CS* that matches *IS* with common suffix

```
1 matched ← 0;
2 CS_matched ← null;
3 foreach complete segment CS do
4     matched_local ← ComputeSuffixLength(IS, Prefix(CS));
5     if matched_local > matched then
6         matched ← matched_local;
7         CS_matched ← CS;
8 return CS_matched;
```

whose p shares the longest suffix with the *IS* is used to fill the $\diamond$ in the *IS*.

In Figure 6, the first *CS* wins because its prefix p shares a suffix *CD* with the *IS* while the second *CS* does not have any instruction before the anchors that can match the *IS*. For recovery, we use the suffix of the first *CS* (*i.e.*, *GHX* in the example) to fill the hole in the *IS*. In particular, we start our recovery from reading instructions following the anchor instructions in the winning *CS*. This process finishes until we hit a number y of instructions that match those that follow the $\diamond$ in the *IS* (*i.e.*, *BDCA* in the example). This number y can be specified as a user parameter.

A naïve algorithm, as shown in Algorithm 3 is to enumerate all potential *CSes* (that match the anchor instructions of the *IS*) and find the winning one by comparisons on a per-instruction basis. However, this algorithm is not scalable given the huge number of potential *CSes* and the huge number of instructions in each CS.

Abstraction-Guided Recovery. Similarly to control flow reconstruction, we propose an efficient abstraction-guided algorithm for data recovery. The key idea is that we can leverage the *abstract traces* (similar to those built by the reconstruction algorithms in Section 4) to quickly filter out irrelevant *CSes*. In particular, we extend Definition 4.2 to define a three-tier abstraction hierarchy that we use for data recovery. The first (highest) tier of abstraction is *call structure*, the second (middle) tier is *control structure* (which is exactly Definition 4.2), and the lowest (concrete) tier is *bytecode instruction*. For each trace segment, we construct the two high-level abstractions during the control-flow reconstruction phase discussed in Section 4. In searching for a *CS*, we use an iterative process that compares traces at each abstraction level (from call structures, through control structures, to concrete instructions).

Note that in Section 4, we only use the middle-tier abstraction (*i.e.*, control structure) because the goal there is to project a trace subsequence onto the ICFG where call structures are irrelevant. Since the ICFG is statically built, there is no dynamic call information available on the ICFG.

Definition 5.2 (Tier-l Abstraction of Bytecode Trace). Given a trace segment $\omega = \langle t_1, t_2, \ldots, t_n \rangle$, an abstraction of ω at tier l, denoted as $\alpha_l(\omega)$, produces an abstract segment $\widehat{\omega_l} = \langle \widehat{t_1}, \widehat{t_2}, \ldots, \widehat{t_m} \rangle$ such that (1) for each $i \in [1, n]$, $\widehat{t_i}$ is a tier-l instruction; and (2) for the indices $i, j : 1 \leq i < j \leq n$ such that t_i and t_j are tier-l instructions, there always exist the indices $p, q, 1 \leq p < q \leq m$, such that $\widehat{t_p} = t_i$ and $\widehat{t_q} = t_j$.

In fact, Definition 4.2 is a special case of Definition 5.2. The abstraction function at tier l essentially removes all non-tier-l instructions. In particular, at the call-structure tier (tier 1), we are interested in call/return instructions, while at the control-structure tier (tier 2), we are interested in control-related instructions such as call, return, branch, and jump. The tier-2 instructions include those in tier 1, and the tier-3 (concrete) instructions include those in both tier 1 and tier 2.

Next, we formally define abstraction-guided data recovery.

Lemma 5.3 (Common Suffix Properties). *Let ω_0 be an IS, ω_1 and ω_2 be two potential CSes. Let $\circ$ denote a matching operator—$\omega_0 \circ \omega_1$ generates the longest common suffix between the prefix of ω_1 and ω_0; with our abstractions, we have the following properties:*

- $|\omega_0 \circ \omega_1| \geq |\omega_0 \circ \omega_2| \implies |\alpha_2(\omega_0 \circ \omega_1)| \geq |\alpha_2(\omega_0 \circ \omega_2)|$
- $|\alpha_2(\omega_0 \circ \omega_1)| \geq |\alpha_2(\omega_0 \circ \omega_2)| \Longrightarrow |\alpha_1(\omega_0 \circ \omega_1)| \geq |\alpha_1(\omega_0 \circ \omega_2)|$

Proof. Let $\omega_0 = \langle t_{p_z}, \ldots, t_{p_2}, t_{p_1}, \diamond \rangle$, $\omega_1 = \langle t_{p_x}, \ldots, t_{p_2}, t_{p_1}, \ldots \rangle$, and $\omega_2 = \langle t_{p_y}, \ldots, t_{p_2}, t_{p_1}, \ldots \rangle$. Suppose the common suffix between ω_0 and the prefix of ω_1 is $\omega_0 \circ \omega_1 = \langle t_{p_m}, \ldots, t_{p_2}, t_{p_1} \rangle$; similarly, the common suffix between ω_0 and the prefix of ω_2 is $\omega_0 \circ \omega_2 = \langle t_{p_n}, \ldots, t_{p_2}, t_{p_1} \rangle$. If $|\omega_0 \circ \omega_1| \geq |\omega_0 \circ \omega_2|$, $\omega_0 \circ \omega_2 = \langle t_{p_n}, \ldots, t_{p_2}, t_{p_1} \rangle$ is a suffix of $\omega_0 \circ \omega_1 = \langle t_{p_m}, \ldots, t_{p_2}, t_{p_1} \rangle$. That is, $m \geq n$. This implies that the length of $\alpha_2(\omega_0 \circ \omega_1)$ (*i.e.*, the number of control instructions matched) must be $\geq$ that of $\alpha_2(\omega_0 \circ \omega_2)$, *i.e.*, $|\alpha_2(\omega_0 \circ \omega_1)| \geq |\alpha_2(\omega_0 \circ \omega_2)|$. With the same reasoning, we can deduce $|\alpha_2(\omega_0 \circ \omega_1)| \geq |\alpha_2(\omega_0 \circ \omega_2)|$ implies $|\alpha_1(\omega_0 \circ \omega_1)| \geq |\alpha_1(\omega_0 \circ \omega_2)|$. □

Lemma 5.4 (Matching Relaxed by Abstractions). *Let ω_0 be an IS and ω_1 be a CS. Let $\circ$ denote the same matching operator. With our abstractions, we have the following properties:*

- $|\alpha_2(\omega_0) \circ \alpha_2(\omega_1)| \geq |\alpha_2(\omega_0 \circ \omega_1)|$
- $|\alpha_1(\omega_0) \circ \alpha_1(\omega_1)| \geq |\alpha_1(\omega_0 \circ \omega_1)|$

Lemma 5.4 shows that given an IS ω_0 and a CS ω_1, the common suffix at the tier l (between the abstract segments of ω_0 and ω_1) is always $\geq$ the length of the tier-l abstraction of the common suffix of ω_0 and ω_1. This is easy to see because the lower the tier, the stricter the matching.

Based on Lemma 5.3 and Lemma 5.4, we obtain the following necessary condition for pruning.

Theorem 5.5 (Necessary Condition of Pruning). *Let ω be an IS, ω_1 and ω_2 be two potential CSes. We have the following conditions:*

- $|\omega_0 \circ \omega_1| \geq |\omega_0 \circ \omega_2| \Longrightarrow |\alpha_2(\omega_0) \circ \alpha_2(\omega_1)| \geq |\alpha_2(\omega_0 \circ \omega_2)|$
- $|\alpha_2(\omega_0) \circ \alpha_2(\omega_1)| \geq |\alpha_2(\omega_0 \circ \omega_2)| \Longrightarrow |\alpha_1(\omega_0) \circ \alpha_1(\omega_1)| \geq |\alpha_1(\omega_0 \circ \omega_2)|$

Proof. The proof can be done by applying a transitivity rule over Lemma 5.3 and Lemma 5.4. First, according to Lemma 5.3, $|\omega_0 \circ \omega_1| \geq |\omega_0 \circ \omega_2|$ implies $|\alpha_2(\omega_0 \circ \omega_1)| \geq |\alpha_2(\omega_0 \circ \omega_2)|$. According to Lemma 5.4, $|\alpha_2(\omega_0) \circ \alpha_2(\omega_1)| \geq |\alpha_2(\omega_0 \circ \omega_1)|$. Transitively, we have $|\alpha_2(\omega_0) \circ \alpha_2(\omega_1)| \geq |\alpha_2(\omega_0 \circ \omega_2)|$. Similarly, we can prove $|\alpha_2(\omega_0) \circ \alpha_2(\omega_1)| \geq |\alpha_2(\omega_0 \circ \omega_2)| \implies |\alpha_1(\omega_0) \circ \alpha_1(\omega_1)| \geq |\alpha_1(\omega_0 \circ \omega_2)|$ holds. □

Theorem 5.5 provides a way for JPortal to quickly test, given ω_2, a *CS* that has been found to match the *IS* to some degree, whether a new CS ω_1 can match the *IS* better. Specifically, if the condition $|\alpha_2(\omega_0) \circ \alpha_2(\omega_1)| < |\alpha_2(\omega_0 \circ \omega_2)|$ holds, we must have $|\omega_0 \circ \omega_1| < |\omega_0 \circ \omega_2|$ (*i.e.*, the negation of the first rule in Theorem 5.5), which means there is no way for ω_1 to beat ω_2 in matching ω_0. Since testing $|\alpha_2(\omega_0) \circ \alpha_2(\omega_1)| < |\alpha_2(\omega_0 \circ \omega_2)|$ only needs the abstract segments of ω_0 and ω_1 (*i.e.*, $|\alpha_2(\omega_0 \circ \omega_2)|$ can be remembered when ω_2 was processed earlier), this test can be done much more efficiently than a concrete instruction-level test.

Based on this idea, we propose Algorithm 4 for efficient recovery. Given an *IS*, and all potential *CSes* that match the anchor instructions of the *IS*, the algorithm first retrieves their tier-1 and tier-2 abstract segments ($\widehat{IS}_1$ and $\widehat{IS}_2$, and $\widehat{CS}_1$ and $\widehat{CS}_2$), respectively. For each *CS*, we use a function ComputeSuffixLength (shown as Lines 13-21 in Algorithm 4) to compute the common suffixes at all three tiers between the *IS* and the *CS*. The function returns a tuple $\langle ml_1, ml_2, ml_3 \rangle$, which indicates the length of the tier-1, tier-2, and tier-3 common suffix, respectively. As shown by Line 9 of Algorithm 4, if the tier-3 (concrete) common suffix (ml_3) between *IS* and the current *CS* is longer than the maximum length found so far, we update the maximum length and let a pointer $CS_{matched}$ point to *CS*. Finally, Algorithm 4 returns $\mathcal{CS}_{matched}$ that will be used for data recovery in *IS*.

The procedure ComputePrefixLength takes as input *IS*, $\widehat{IS}_2$, $\widehat{IS}_1$, *CS*, $\widehat{CS}_2$, $\widehat{CS}_1$, and the maximum lengths of the common suffixes found for *IS* so far at the three levels $\langle m_1, m_2, m_3 \rangle$. It returns the updated maximum lengths after checking *CS*. The procedure computes the length of the common suffix at each tier (from 1 to 3) between *IS* and *CS*. If the tier-l's common suffix length ml_l is less than the recorded maximum common suffix length at the same tier m_l (*i.e.*, $|\alpha_l(\omega_0) \circ \alpha_l(\omega_1)| < |\alpha_l(\omega_0 \circ \omega_2)|$), the procedure returns directly rather than exploring lower tiers. According to Lemma 5.5, this is safe—we are guaranteed to not miss a candidate *CS* that can better match the *IS*.

Recovery. As an extension to Algorithm 4, JPortal actually finds, for each *IS*, the top *N CSes* that share the longest common suffix with the *IS*. JPortal follows this ranking (from high to low) to use an *CS* for recovering the ⋄ in the *IS*. The recovery uses instructions from the suffix of the *CS*. However, this does not guarantee a perfect match. For example, following these instructions in the suffix of the *CS* may not be able to connect to the post-⋄ instructions in the *IS* (*e.g.*, *BDCA* in Figure 6). To quickly identify problems, we use timestamps from the hardware trace. For example, we calculate a time range d for the missing data in each ⋄ (*i.e.*, by diffing the timestamps attached with the packet before and after the ⋄). During recovery, if after reading d's worth of instructions from the *CS*, we still cannot reach the post-⋄ instructions in the *IS*, we terminate this process and try the next *CS* on the list. If no *CS* can fill the ⋄, JPortal walks the ICFG and returns a random path to connect the pre- and post-⋄ instructions.

Algorithm 4: Abstraction-guided *CS* search.

Input: an *IS*, all potential *CSes*
Output: a *CS* matching *IS* with the longest common suffix

```
1  IS1^ ← α1(IS) ; // Obtain the abstraction of IS at tier 1
2  IS2^ ← α2(IS) ; // Obtain the abstraction of IS at tier 2
3  ⟨mg1, mg2, mg3⟩ ← 0;
4  CS_matched ← null;
5  foreach potential CS do
6      CS1^ ← α1(CS);
7      CS2^ ← α2(CS);
8      ⟨ml1, ml2, ml3⟩ ← ComputeSuffixLength(IS, IS2^, IS1^,
         CS, CS2^, CS1^, ⟨mg1, mg2, mg3⟩);
9      if ml3 > mg3 then
10         ⟨mg1, mg2, mg3⟩ ← ⟨ml1, ml2, ml3⟩;
11         CS_matched ← CS;
12 return CS_matched;
13 Function ComputeSuffixLength
       Input: IS = ⟨t_{p_m}, ..., t_{p_2}, t_{p_1}, ⋄⟩, its tier-2 segment IS2^,
              its tier-1 segment IS1^; a CS, it tier-2 abstract
              CS2^, the tier-1 abstract CS1^, and the maximum
              common suffix length found so far ⟨m1, m2, m3⟩
       Output: the common suffix length at three tiers
               ⟨m1, m2, m3⟩
14     ml1 ← length of common suffix between IS1^ and CS1^ ;
15     if ml1 < m1 then continue;
16     ml2 ← length of common suffix between IS2^ and CS2^ ;
17     if ml2 < m2 then continue;
18     ml3 ← the length of common suffix between IS and CS;
19     if ml3 ≥ m3 then
20         ⟨m1, m2, m3⟩ ← ⟨|α1(IS ∘ CS)|, |α2(IS ∘ CS)|, ml3⟩
21     return ⟨m1, m2, m3⟩;
```

6 System Implementation

Collecting Hardware Traces. JPortal uses `perf_events` to configure PT tracing and data exportation, and in particular,

the following system call `perf_event_open`:

int perf_event_open(*struct perf_event_attr* ∗ *attr*, *pid_t pid*, *int cpu*, *int group_fd*, *unsigned long flags*);

When the user specifies PT-related parameters, the system call `perf_event_open` establishes a channel to monitor PT events and returns a file descriptor. In addition to the event monitored, one can also specify which threads and CPU cores to monitor via the `pid` and `cpu` parameters. Here an event is invoked on each core by setting pid to the ID of the Java main thread. Meanwhile, via the attribute *inherit*, we can enable PT to monitor all the threads forked by the Java main thread. With the file descriptor returned by `perf_event_open`, we can use `ioctl`, which takes the descriptor as input, and enables and disables the tracing. To collect the tracing data, we create a memory buffer and export the data to disks periodically. In our experiments, the buffer size is set to 128MB per CPU core. This is done via the system call *mmap* with the same file descriptor as input.

Collecting Machine-Code Metadata. For decoding, JPortal collects all the metadata of application-related machine code including that used by both the interpreter and the JIT. For interpretation mode, JPortal obtains the machine code metadata from the template loaded during the JVM's initialization. To obtain the JITed code, we modified the JVM to allocate a shared memory region of 2MB immediately after the JIT's code cache as a buffer. Our JVM maintains two pointers `tail` and `head` that specify the boundary of this buffer. Once a method is compiled and its code is stored in the code cache, our JVM copies the code into the buffer. Meanwhile, the exporting process periodically checks the `head` and `tail` pointers of the buffer to export code.

Filtering Out Irrelevant Data. By default, PT records the information of all the machine code executed, including system calls, JVM code, as well as other application code running concurrently on the same core. Hence, a large amount of data is irrelevant to the application of interest, which can cause not only performance issues but also loss of useful data. To tackle this problem, we need to filter out packages corresponding to irrelevant programs. Intel PT offers the functionality of instruction pointer filtering. By configuring certain registers, PT enables the generation of trace packets only when the processor executes code within certain IP ranges. If an IP is outside of these ranges, generation of some packets is blocked [19]. JPortal leverages this functionality to collect only the interpreted and JITed code from the JVM application being traced. Since these two types of machine code are both maintained in the code cache, JPortal can use the code cache boundary (which does not change throughout the execution) as the IP range to filter out irrelevant data.

Dealing with Inlined Code. When reconstructing the control flow, JPortal leverages the debug information to determine the mapping between bytecode and machine code. However, when inlining happens, such mapping cannot be used. For each machine instruction executed, JPortal verifies if it is compiled from code from an inlined method. This can be done by checking the index associated with the machine instruction in the debug information. If it is, we further retrieve the inlined method's signature with the help of the debug metadata. With the signature of the inlined method, we can easily find the bytecode instruction in that method that corresponds to the machine instruction executed.

Multi-Cores and Multi-Threads. At the hardware level, PT records data from each physical core separately. As a thread can be scheduled on different cores, its trace is distributed across cores. For a single-threaded program, we first collect the tracing data from all cores and then piece together fragments corresponding to the program based on the process information embedded in the trace. For a multi-threaded program, we obtain, at each core, the thread switching information, which records the timestamps at which each thread begins to run. This information helps us segregate the tracing data from each core based on threads. Next, we use the treatment for a single-threaded program to piece together information across cores that corresponds to each thread.

7 Evaluation

Hardware & Software Environments. JPortal was built on top of OpenJDK 12 – the latest version of Oracle's HotSpot JVM. All of our experiments were conducted on a machine with one Intel i7-6700 CPU (3.40GHz with 8MB cache, 8 physical cores and 16 logical cores in total), 16GB memory and 256G SSD storage, running Ubuntu 20.04.

Subjects & Client Applications. We used DaCapo-9.12 [26] as our benchmarks. We ran them with default workloads. Five programs including xalan, eclipse, tradebeans, tradesoap, and tomcat use libraries that are not compatible with OpenJDK 12 [54], and hence always crash. Note that this is *not* due to any issue in our implementation. Therefore, we excluded these programs from the experiments. Table 1 lists the characteristics of the remaining nine programs, including their version information, lines of code, numbers of methods, numbers of classes, and whether they have multiple threads.

Table 1. Characteristics of subject programs.

Subject	Version	#LoC	#Methods	#Classes	Threaded
avrora	1.7.110	70,117	9,501	1,828	single
batik	1.7	195,232	2,430	15,211	single
fop	0.95	105,889	1,314	9,968	single
h2	1.2.121	119,693	471	7,026	multiple
jython	2.5.1	209,016	3,288	31,201	single
luindex	2.4.1	39,864	560	4,365	single
lusearch	2.4.1	40,194	563	4,371	multiple
pmd	4.2.5	60,472	727	5,055	multiple
sunflow	0.07.2	21,962	255	1,762	single

Table 2. Slowdown in times; columns shown are statement coverage (SC), path frequency (PF), control flow (CF), hot methods (HM) for instrumentation-based profiling, as well as xprof (xp) and JProfiler (JP) for sampling-based profiling.

		Instrumentation-based				Sampling-based	
Subject	JPortal	SC[24]	PF[25]	CF[24]	HM	xp[16]	JP[8]
avrora	1.154	29.940	43.777	3555.073	11.038	1.059	1.512
batik	1.084	1.603	1.776	46.322	2.322	1.262	1.331
fop	1.044	2.182	1.947	41.631	1.969	1.309	1.221
h2	1.128	10.114	13.507	1266.685	50.840	1.056	1.140
jython	1.165	3.600	7.113	502.163	14.657	1.052	1.519
luindex	1.041	2.027	2.403	80.776	3.817	1.115	1.272
lusearch	1.162	13.979	24.093	1706.262	8.203	1.168	1.509
pmd	1.086	1.140	1.258	5.320	2.040	1.063	1.822
sunflow	1.156	6.343	10.767	887.897	14.564	1.151	1.464

Our goal is to evaluate JPortal in terms of overhead and profiling accuracy. JPortal can collect rich control flow information for bytecode programs, enabling reconstruction of various types of profiling information. To understand the effectiveness of JPortal, we compared JPortal with both instrumentation-based and sampling-based profiling techniques. In particular, we compared JPortal with four existing profiling techniques—statement coverage profiling, path profiling, control flow profiling, and hot method profiling. The first three are instrumentation-based while the last one is sampling-based. For the first three techniques, we reimplemented Ball and Larus' approaches for statement coverage profiling [24], path profiling [25] and control flow tracing [24] with the help of the ASM framework [1]. As for sampling-based profiling, we directly compared JPortal with two existing hot-method profilers: xprof [16], which is HotSpot's built-in profiler, as well as JProfiler [8], which is a widely-used Java profiler.

Our evaluation answers the following research questions:

- Q1. How much overhead does JPortal incur? How does it compare to the overheads of state-of-the-art profiling techniques? (§7.1)
- Q2. How accurate is JPortal? How does it compare to the accuracy of the aforementioned techniques? How effective is our recovery mechanism? (§7.2)
- Q3. How fast is JPortal in decoding and recovery? (§7.3)

7.1 Runtime Overhead

To answer the first question, we compared the runtime overheads between JPortal and the existing techniques. For the sampling-based tools xprof and JProfiler, we used one sample per 10ms, which is the default setting in xprof. In particular, we measured the slowdowns (*i.e.*, running time with profiling enabled divided by the original running time). Each program is executed three times and the average running times are used in our experiments. Table 2 reports the slowdowns in terms of the statement coverage profiling (SC) [24], path frequency profiling (PF) [25], control flow profiling (CF) [24], and hot methods profiling (HM). JPortal, which performs end-to-end execution profiling, has an overhead between 4.1% and 16.5%, which is much lower than that of instrumentation-based profiling techniques (which can be as high as 3555× for jython). Sampling-based profiling tools, xprof (xp) and JProfiler (JP) enjoy a relatively low overhead of 6%–82%, although it is still higher than that of JPortal, which does hardware-based profiling.

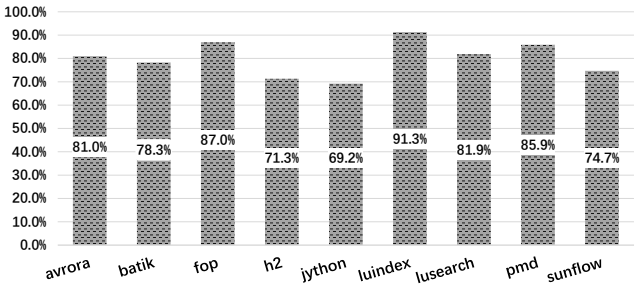


Figure 7. JPortal's overall accuracy, using profiles collected by the instrumentation-based approach as ground truth.

Note that JPortal's overhead is higher than the overhead previously reported (2-5% [51, 56]) for applying PT on native applications. There are two major reasons. First, to interpret one bytecode instruction, the interpreter often produces many more instructions. Second, JPortal has an additional overhead of collecting machine-code metadata, which does not exist when native programs are profiled.

7.2 Profiling Accuracy

Another important metric is profiling accuracy, which is the focus of this subsection. As the control flow profile includes the information of statement coverage and path frequency, we measured JPortal's accuracy of decoding and recovery using the profile generated by Ball and Larus' instrumentation-based control flow profiling technique [24] as ground truth. Figure 7 shows JPortal's profiling accuracy, obtained by measuring the degree of matching between each JPortal-reconstructed control flow path and its corresponding path collected by the baseline approach—the higher, the better. JPortal achieves an overall accuracy of 80%.

Recall that JPortal's analysis has two major components: (1) trace decoding and call flow reconstruction for each trace segment, and (2) recovering missing data between these segments. To understand the source of inaccuracies and JPortal's effectiveness in each of these components, we broke each reconstructed control flow path into (1) a part that is directly reconstructed from the hardware trace, and (2) a second part that is recovered for missing data. Table 3 shows the detailed breakdown between these parts for three programs that have more than 10% of their data missing under three different buffer settings (*i.e.*, 256M, 128M, and 64M). In particular, PMD and PDC report, respectively, the percentages of data lost due to buffer overflow and data captured successfully. PR and PD report, respectively, the percentages of the final control

Table 3. A breakdown of data captured and lost, as well as JPortal's reconstruction accuracies under different buffer sizes.

	batik			h2			sunflow		
	256M	128M	64M	256M	128M	64M	256M	128M	64M
Percent of missing data (PMD)	0%	22.23%	39.75%	19.30%	28.03%	54.28%	10.40%	22.67%	45.04%
Percent of recovered (PR)	-	11.79%	16.44%	10.88%	16.95%	29.14%	5.05%	9.26%	15.13%
Recovery accuracy (RA)	-	53.05%	41.36%	56.35%	60.48%	53.69%	48.52%	40.86%	33.59%
Percent of data captured (PDC)	100%	77.77%	60.25%	80.70%	71.97%	45.72%	89.60%	77.33%	54.96%
Percent of decoded (PD)	85.40%	66.53%	51.42%	61.18%	54.36%	34.38%	74.94%	65.43%	45.74%
Decoding accuracy (DA)	85.40%	85.55%	85.34%	75.81%	75.53%	75.20%	83.64%	84.61%	83.22%

flow profiles that are recovered using the algorithm in §5 as well as reconstructed from the captured data (§4). Relatedly, RA and DA report the accuracies of JPortal's recovery and decoding/reconstruction, respectively.

The percentage of data loss is dependent on multiple factors such as CPU frequency, the application's instruction characteristics, I/O latency, and buffer sizes, etc. As shown, for the same program, the larger the buffer size, the less the data loss. It is clear that most of the accuracy loss stems from the data loss. Our recovery algorithm in §5 achieves an accuracy of 51.4% across these programs under the 128M buffer size. The recovery algorithm is less effective when more data is lost. This is expected since more data loss leaves a lower chance for a *CS* to exist. For sunflow, a moderate-sized program, its data loss even reaches 40% under a 64M buffer. A careful inspection found that sunflow has a much higher trace generation rate than other programs.

Table 4. Accuracy in hot method detection.

Subject	xprof [16]	JProfiler [8]	JPortal
avrora	2	4	7
batik	0	5	6
fop	1	6	8
h2	0	4	6
jython	1	1	6
luindex	1	2	7
lusearch	4	4	6
pmd	4	5	7
sunflow	1	4	6

JPortal's decoding/reconstruction accuracy is much higher (*i.e.*, 82%). However, we are not able to precisely reconstruct 100% of control flow for the captured trace data. There are a few reasons. First, for multi-threaded programs, JPortal first separates the tracing data collected at each core based on threads. To do this, JPortal records the timestamps of thread switching points, and use these timestamps to separate the trace. However, the timestamps embedded in a hardware trace can be inconsistent with those recorded at thread switching points, resulting in occasional mistakes in data separations. Furthermore, due to JIT optimizations such as loop transformation and method inlining, the debug information used for decoding is often imprecise as well.

We also compared the accuracy in hot method detection between JPortal and the two performance profilers – xprof and JProfiler, which are based on sampling. We identified the 10 hottest methods based on the control flow profile collected from Ball and Larus' instrumentation-based technique and used these methods as the ground truth. Next, we obtained the top 10 methods from JPortal's profiles as well as those of xprof [16] and JProfiler [8]. Finally, we compared these reports with the ground truth. Table 4 reports the number of hot methods in the intersection between the ground truth and the report from each profiler. For each sampling-based profiler, we ran it three times and reported the best result in Table 4. Clearly, JPortal's report is much closer to the ground true than the sampling-based profilers (and simultaneously has a much lower overhead).

Table 5. Trace size and time for decoding/reconstruction and recovery; baseline is the instrumentation-based control flow profiling technique [24]; reported for each technique are data sizes (**TS**) and decoding times (**DT**); for JPortal, we additionally report the recovery time (**RT**); '-' indicates there is no data loss.

	Baseline [24]		JPortal		
Subject	TS (MB)	DT (min)	TS (MB)	DT (min)	RT (min)
avrora	8301.4	113.2	773.4	20.4	-
batik	176.4	4.2	1197.6	4.8	1.0
fop	109.1	1.7	520.7	3.5	-
h2	14946.7	198.9	3067.7	33.1	16.7
jython	1735.0	19.7	829.8	12.5	-
luindex	81.4	1.7	192.7	1.6	-
lusearch	1174.8	20.1	1067.2	6.1	-
pmd	3.2	3.2 secs	174.9	1.1	-
sunflow	1808.6	33.5	1052.3	10.9	6.6

7.3 Performance of Decoding and Recovery

To understand the performance of JPortal's offline performance of reconstruction and recovery, we measured the trace size as well as the running time for each benchmark under the instrumentation-based approach and JPortal. Table 5 reports our results. For most programs, JPortal finishes decoding within 10 minutes. The recovery time is also reasonably short (*i.e.*, around 8 minutes across the three programs with missing data). In contrast, the instrumentation-based approach [24] records more tracing data for most programs. The time spent on data decoding varies across programs.

8 Related Work

8.1 Program Profiling

Instrumentation-based Profiling. In the JVM community, multiple instrumentation frameworks have been developed, such as Soot [14], ASM [1], BIT [37], RoadRunner [31], and DiSL [2, 41]. They provide high-level APIs that hide low-level details of bytecode or intermediate representation (IR), making instrumentation much easier to do.

In particular, ASM [1] is a Java bytecode instrumentation framework, which provides a way to dynamically manipulate Java classes. Soot [14] is a Java optimization infrastructure. It provides intermediate representations (IRs) at different abstraction levels for analyzing and transforming Java bytecode. DiSL [2, 41] is an aspect-oriented language developed for Java bytecode instrumentation. It leverages high-level language constructs for concise instrumentation and high performance of the instrumented code. Similar to DiSL, BISM [52] is a lightweight low-level instrumentation framework suitable for runtime verification.

A large body of binary instrumentation frameworks, such as Pin [12, 40], DynamoRIO [3], Valgrind [15], Dyninst [4] have been developed to assist in implementing various dynamic analyses on binaries. IR-level instrumentation tools such as LLVM [10] and CSI [46] have also been extensively used. Recently, Lehmann and Pradel developed Wasabi [38], which is a binary instrumentation framework for analyzing WebAssembly. Sen *et al.*, developed Jalangi [48], which provides supports for dynamic analyses of JavaScript programs.

There also exists a body of JVM-based profiling techniques [21, 36, 43, 49, 57–59] that are used primarily to detect performance problems.

Sampling-based Profiling. To reduce the runtime overhead, sampling-based profiling tools such as xprof [16], hprof [6], JProfiler [8], and YourKit [17], have been developed and extensively used in practice. Instead of capturing the complete program behavior, sampling-based approaches only collect data at a certain frequency; with the collected data, they can statistically estimate the overall behavior of the program. As a result, their overheads can be controlled by tuning the sampling rate. However, finding the right sampling rate is often challenging, and hence the accuracy of profiling data is usually not guaranteed [42, 60].

Hardware-based Profiling. Another category of profiling leverages hardware-provided profiling/tracing functionalities. For instance, the Performance Monitoring Unit (PMU), which is extensively used by profiling tools (such as Linux perf [11] and Intel Vtune [7]), exploits PMU counters to record events of interest. Although it has been shown to be effective in many use scenarios [20, 22, 23], its applicability is limited by the few PMU registers available.

In the past decade, dedicated hardware modules have been developed to support powerful instruction and data tracing. Such modules include Intel processor trace (PT) [19] and ARM embedded trace macrocell (ETM) [18, 53]. These modules record minimal tracing information at runtime. Control flow can be reconstructed offline with libraries such as libipt [9] and ptm2human [13] (for Intel PT and ARM ETM, respectively). Hardware tracing avoids any modification to executables. More importantly, it incurs negligible runtime overheads (around 2-5% in general [51, 56]) to the execution of the software being traced.

8.2 Applications of Hardware Tracing

Hardware tracing has already enabled a variety of client tasks, including testing/fuzzing [5, 28, 47, 63], debugging [30, 33, 34, 44], security enforcement [32, 39, 55], performance tuning [50, 51], *etc.* Multiple fuzzing systems, such as Honggfuzz, kAFL, PTFuzz, and PTrix, collect code coverage information via PT so as to efficiently guide the fuzzing process. In security enforcement, Griffin [32] and FlowGuard [39] both exploit Intel PT to enforce control-flow integrity (CFI). Cui *et al.* [30] proposed REPT that combines online hardware tracing with offline binary analysis to achieve efficient reverse debugging for field failures. PT has also been utilized to debug concurrency and kernel bugs [33, 34]. Moreover, Sharma and Dagenais [50, 51] have employed hardware tracing for latency profiling and performance analysis.

Despite these many advantages provided by hardware tracing, it has so far been used only to trace native programs running on bare metal machines. High-level languages such as Java, Go, and Scala cannot benefit from it due to the runtime-induced gap between application code and machine code executed. JPortal closes this gap so that these benefits can be easily extended to HLL applications.

9 Conclusion

This paper presented JPortal, a JVM-based profiling tool that bridges the gap between the low-level hardware (machine-code) traces and the high-level control flow of JVM programs. By leveraging a postmortem static analysis that analyzes the low-level traces collected by hardware, JPortal is able to precisely and efficiently track the dynamic control flows of Java bytecode with ultra-low runtime overheads.

Acknowledgment

We thank the anonymous reviewers for their thorough and insightful comments. We are especially grateful to our shepherd Ben Liblit for his feedback. This work was partially supported by the National Natural Science Foundation of China (No. 61932021 and 61802168), the Natural Science Foundation of Jiangsu Province (No. BK20191247), the Fundamental Research Funds for the Central Universities (No. 14380065), the US National Science Foundation under grants CNS-1613023, CNS-1703598, CNS-1763172, CNS-2006437, and CNS-2007737, and by the US Office of Naval Research under grants N00014-16-1-2913 and N00014-18-1-2037.

References

[1] [n.d.]. ASM: an open-source Java bytecode manipulation and analysis framework. https://asm.ow2.io/. Accessed: 2020-10-31.

[2] [n.d.]. DiSL: an open-source Java bytecode instrumentation framework. https://disl.ow2.org/view/Main/. Accessed: 2020-10-31.

[3] [n.d.]. DynamoRIO: a runtime code manipulation system. https://dynamorio.org/. Accessed: 2020-10-31.

[4] [n.d.]. Dyninst: tools for binary instrumentation, analysis, and modification. https://dyninst.org/. Accessed: 2020-10-31.

[5] [n.d.]. Honggfuzz: a feedback-driven security oriented software fuzzer. http://honggfuzz.com. Accessed: 2020-10-31.

[6] [n.d.]. hprof: an open source java profiler. http://java.sun.com/developer/technicalArticles/Programming/HPROF.html. Accessed: 2020-10-31.

[7] [n.d.]. Intel® VTune™ Profiler: a commercial application for software performance analysis. https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html. Accessed: 2020-10-31.

[8] [n.d.]. JProfiler: a commercial java profiler. https://www.ej-technologies.com/products/jprofiler/overview.html. Accessed: 2020.

[9] [n.d.]. libipt: an Intel(R) Processor Trace decoder library. https://github.com/intel/libipt. Accessed: 2020-10-31.

[10] [n.d.]. LLVM: a compiler infrastructure. https://llvm.org/. 2021.

[11] [n.d.]. perf: a performance analyzing tool in Linux. https://github.com/torvalds/linux/tree/master/tools/perf. Accessed: 2020-10-31.

[12] [n.d.]. Pin: a dynamic binary instrumentation framework. http://www.intel.com/software/pintool. Accessed: 2020-10-31.

[13] [n.d.]. ptm2human: a decoder for trace data outputted by Embedded Trace Macrocell (ETMv4). https://github.com/hwangcc23/ptm2human. Accessed: 2020-10-31.

[14] [n.d.]. Soot: an open-source Java compiler infrastructure. https://github.com/soot-oss/soot. Accessed: 2020-10-31.

[15] [n.d.]. Valgrind: an instrumentation framework for building dynamic analysis tools. https://valgrind.org/. Accessed: 2020-10-31.

[16] [n.d.]. xprof: the internal profiler for hotspot. http://java.sun.com/docs/books/performance/1stedition/html/JPAppHotspot.fm.html. Accessed: 2020-10-31.

[17] [n.d.]. YourKit: a commercial java profiler. https://www.yourkit.com/. Accessed: 2020-10-31.

[18] 2019. Arm® Embedded Trace Macrocell Architecture Specification ETMv4.0 to ETMv4.5. (2019).

[19] 2019. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide.* Chapter 35: Intel Processor Trace.

[20] Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation* (Las Vegas, Nevada, USA) *(PLDI '97)*. 85–96. https://doi.org/10.1145/258915.258924

[21] Matthew Arnold and Barbara G. Ryder. 2001. A Framework for Reducing the Cost of Instrumented Code. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. 168–179. https://doi.org/10.1145/378795.378832

[22] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. 2013. Production-Run Software Failure Diagnosis via Hardware Performance Counters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. 101–112. https://doi.org/10.1145/2451116.2451128

[23] Joy Arulraj, Guoliang Jin, and Shan Lu. 2014. Leveraging the Short-Term Memory of Hardware to Diagnose Production-Run Software Failures. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. 207–222. https://doi.org/10.1145/2541940.2541973

[24] Thomas Ball and James R. Larus. 1994. Optimally Profiling and Tracing Programs. *ACM Trans. Program. Lang. Syst.* 16, 4 (July 1994), 1319–1360. https://doi.org/10.1145/183432.183527

[25] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture* (Paris, France) *(MICRO 29)*. 46–57.

[26] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). 169–190. https://doi.org/10.1145/1167473.1167488

[27] Cheng Cai, Qirun Zhang, Zhiqiang Zuo, Khanh Nguyen, Guoqing Xu, and Zhendong Su. 2018. Calling-to-Reference Context Translation via Constraint-Guided CFL-Reachability. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. 196–210. https://doi.org/10.1145/3192366.3192378

[28] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. 2019. PTrix: Efficient Hardware-Assisted Fuzzing for COTS Binary. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security* (Auckland, New Zealand) *(Asia CCS '19)*. 633–645. https://doi.org/10.1145/3321705.3329828

[29] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. *J. ACM* 50, 5 (Sept. 2003), 752–794. https://doi.org/10.1145/876638.876643

[30] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. 17–32.

[31] Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Toronto, Ontario, Canada) *(PASTE '10)*. 1–8. https://doi.org/10.1145/1806672.1806674

[32] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. 585–598. https://doi.org/10.1145/3037697.3037716

[33] Xinyang Ge, Ben Niu, and Weidong Cui. 2020. Reverse Debugging of Kernel Failures in Deployed Systems. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). 281–292. https://www.usenix.org/conference/atc20/presentation/ge

[34] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. 582–598. https://doi.org/10.1145/3132747.3132767

[35] James R. Larus. 1999. Whole Program Paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '99)*. 259–269. https://doi.org/10.1145/301618.301678

[36] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. 2006. Online Performance Auditing: Using Hot Optimizations without Getting Burned. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. 239–251. https://doi.org/10.1145/1133981.1134010

[37] Han Bok Lee and Benjamin G. Zorn. 1997. BIT: A Tool for Instrumenting Java Bytecodes. In *USENIX Symposium on Internet Technologies and Systems (USITS 97)*. https://www.usenix.org/conference/usits-97/bit-tool-instrumenting-java-bytecodes

[38] Daniel Lehmann and Michael Pradel. 2019. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. 1045–1058. https://doi.org/10.1145/3297858.3304068

[39] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan. 2017. Transparent and Efficient CFI Enforcement with Intel Processor Trace. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 529–540. https://doi.org/10.1109/HPCA.2017.18

[40] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. 190–200. https://doi.org/10.1145/1065010.1065034

[41] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: A Domain-Specific Language for Bytecode Instrumentation. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development* (Potsdam, Germany) *(AOSD '12)*. 239–250. https://doi.org/10.1145/2162049.2162077

[42] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2010. Evaluating the Accuracy of Java Profilers. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. 187–197. https://doi.org/10.1145/1806596.1806618

[43] Khanh Nguyen and Guoqing Xu. 2013. Cachetor: Detecting Cacheable Data to Remove Bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. 268–278. https://doi.org/10.1145/2491411.2491416

[44] Z. Ning and F. Zhang. 2019. Hardware-Assisted Transparent Tracing and Debugging on ARM. *IEEE Transactions on Information Forensics and Security* 14, 6 (2019), 1595–1609. https://doi.org/10.1109/TIFS.2018.2883027

[45] Peter Ohmann, Alexander Brooks, Loris D'Antoni, and Ben Liblit. 2017. Control-flow recovery from partial failure reports. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). 390–405. https://doi.org/10.1145/3062341.3062368

[46] Tao B. Schardl, Tyler Denniston, Damon Doucet, Bradley C. Kuszmaul, I-Ting Angelina Lee, and Charles E. Leiserson. 2017. The CSI Framework for Compiler-Inserted Program Instrumentation. *Proc. ACM Meas. Anal. Comput. Syst.* 1, 2, Article 43 (Dec. 2017), 25 pages. https://doi.org/10.1145/3154502

[47] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*. 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[48] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. 488–498. https://doi.org/10.1145/2491411.2491447

[49] Ajeet Shankar, Matthew Arnold, and Rastislav Bodik. 2008. Jolt: Lightweight Dynamic Analysis and Removal of Object Churn. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (Nashville, TN, USA) *(OOPSLA '08)*. 127–142. https://doi.org/10.1145/1449764.1449775

[50] Suchakra Sharma. 2015. Hardware Tracing for Fast and Precise Performance Analysis. *The New Stack* (10 2015).

[51] S. D. Sharma and M. Dagenais. 2016. Hardware-assisted instruction profiling and latency detection. *The Journal of Engineering* 2016, 10 (2016), 367–376. https://doi.org/10.1049/joe.2016.0127

[52] Chukri Soueidi, Ali Kassem, and Yliès Falcone. 2020. BISM: Bytecode-Level Instrumentation for Software Monitoring. In *Runtime Verification*, Jyotirmoy Deshmukh and Dejan Ničković (Eds.). 323–335.

[53] Neal Stollon. 2011. *ARM ETM*. Springer US, Boston, MA, 213–218. https://doi.org/10.1007/978-1-4419-7563-8_13

[54] The same crashes encountered as reported by the following issues when running certain subjects on OpenJDK 12. 2020. https://github.com/dacapobench/dacapobench/issues/175;https://github.com/dacapobench/dacapobench/issues/184.

[55] Xiayang Wang, Fuqian Huang, and Haibo Chen. 2019. DTrace: fine-grained and efficient data integrity checking with hardware instruction tracing. *Cybersecur.* 2, 1 (2019), 1. https://doi.org/10.1186/s42400-018-0018-3

[56] Mario Wolczko and Cansu Kaynak. 2017. Processor Tracing for Virtual Machines. In *The Workshop on Modern Language Runtimes, Ecosystems, and VMs (MoreVMs)*.

[57] Guoqing Xu. 2012. Finding Reusable Data Structures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. 1017–1034. https://doi.org/10.1145/2384616.2384690

[58] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the Flow: Profiling Copies to Find Runtime Bloat. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. 419–430. https://doi.org/10.1145/1542476.1542523

[59] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding Low-Utility Data Structures. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. 174–186. https://doi.org/10.1145/1806596.1806617

[60] Hao Xu, Qingsen Wang, Shuang Song, Lizy Kurian John, and Xu Liu. 2019. Can We Trust Profiling Results? Understanding and Fixing the Inaccuracy in Modern Profilers. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. 284–295. https://doi.org/10.1145/3330345.3330371

[61] Yibiao Yang, Yanyan Jiang, Zhiqiang Zuo, Yang Wang, Hao Sun, Hongmin Lu, Yuming Zhou, and Baowen Xu. 2019. Automatic Self-Validation for Code Coverage Profilers. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. 79–90. https://doi.org/10.1109/ASE.2019.00018

[62] Yibiao Yang, Yuming Zhou, Hao Sun, Zhendong Su, Zhiqiang Zuo, Lei Xu, and Baowen Xu. 2019. Hunting for Bugs in Code Coverage Tools via Randomized Differential Testing. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. 488–499. https://doi.org/10.1109/ICSE.2019.00061

[63] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min. 2018. PTfuzz: Guided Fuzzing With Processor Trace Feedback. *IEEE Access* 6 (2018), 37302–37313. https://doi.org/10.1109/ACCESS.2018.2851237

[64] Zhiqiang Zuo, Lu Fang, Siau-Cheng Khoo, Guoqing Xu, and Shan Lu. 2016. Low-Overhead and Fully Automated Statistical Debugging with Abstraction Refinement. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. 881–896. https://doi.org/10.1145/2983990.2984005

[65] Zhiqiang Zuo, Siau-Cheng Khoo, and Chengnian Sun. 2014. Efficient Predicated Bug Signature Mining via Hierarchical Instrumentation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. 215–224. https://doi.org/10.1145/2610384.2610400