ITHICA: Intra-Thread Instruction Checking Approach for Defect-Induced Silent Data Corruptions

Ioanna Vavelidou
Stanford University
vavel@stanford.edu

Subho Sankar Banerjee
Google LLC
subhob@google.com

Eric Xue Liu
Google LLC
ericxliu@google.com

Mike Fuller
Google LLC
mikefuller@google.com

Subhasish Mitra
Stanford University
subh@stanford.edu

Caroline Trippel
Stanford University
trippel@stanford.edu

***Abstract*—Hyperscalers are reporting silent data corruptions (SDCs), presumed to be caused by silicon manufacturing defects, as a threat to datacenter reliability. To support datacenter testing efforts to detect defective CPU servers, this paper presents *ITHICA*, an approach and tool for automatically generating functional tests for defect-induced errors from arbitrary programs by inserting intra-thread, instruction-level error checks, primarily leveraging instruction duplication and output comparison. Our key insight is that the most pernicious defects (those most likely to escape manufacturing testing) cause *inconsistent* errors: two executions of the same instruction given the same inputs within the same thread can produce *different* architectural outputs depending on the execution context in which they run. By exploiting this insight, ITHICA uniquely enables arbitrary programs to serve as tests and localizes affected instructions concurrently with error detection. We use ITHICA to transform industrial hyperscaler tests (our baseline), datacenter programs, and common libraries into functional tests, and evaluate them on over 3,000 CPU servers. ITHICA checks detect 39% more defective servers than baseline industrial checks and yield novel findings on defect behavior that challenge conclusions drawn by prior hyperscaler fleet studies.**

***Index Terms*—reliability, silent data corruption, hardware faults, manufacturing defects, functional testing, datacenter testing.**

I. Introduction

Hyperscalers are reporting *silent data corruptions* (SDCs), presumed to be caused by silicon *manufacturing defects*, resulting in permanent (hard) faults, as a critical threat to datacenter reliability [1]–[7]. Such SDCs occur when a *hardware error* (§II-B) causes a system to output an incorrect result without any indication that the error occurred [8]. Hyperscalers are finding that defect-induced SDCs impact a substantial fraction of deployed hardware, roughly one silicon device per thousand [1]–[4], [6].

To detect defective hardware, hyperscalers have begun subjecting their server fleets to frequent functional testing. This testing has enabled hyperscalers to publish several fleet-wide studies characterizing the SDC problem [1]–[6], [9], but these efforts rely on proprietary, hyperscaler- or vendor-specific tests. While some tests have been open-sourced by hyperscalers and vendors [10]–[12], these public suites are generally smaller in scope, and vendor tests especially tend to be structurally distinct from proprietary variants, making them difficult to build on and improve.

Two bodies of work attempt to overcome the limitations of public functional test suites through automated test generation. The first generates tests that expose defects as captured by a specific fault model on a specific hardware implementation [13], but is limited by the inaccuracy of current fault models for defects [6], [14]–[16] and the scalability challenges of fault-model-driven test generation [17], [18]. The second employs fuzzing to generate tests that thoroughly exercise a specific hardware implementation [19]–[21]. Both approaches generally require below-ISA hardware specifications, which are hardware-vendor-proprietary and unavailable to end-users including hyperscalers, making them inapplicable for independent, in-datacenter fleet evaluation [19]. The one exception is SiliFuzz [19], which requires ISA-level specifications only, but generates short tests that have been deemed less effective than proprietary tests [6], a finding we corroborate in §VIII.

The case for programs as tests. Exercising hardware diversely is essential for triggering defect-induced errors, but without below-ISA specifications end-users cannot control or measure hardware coverage [19]. Instead, this paper proposes heuristically approximating this goal by *deriving functional tests from arbitrary programs*. This strategy is well motivated: real datacenter programs expose defects that synthetic tests miss [6], [22], by exercising complex microarchitectural features needed to trigger defects in ways that are difficult for synthetic tests to imitate [9].

The barrier imposed by existing checks. The key to transforming arbitrary programs into functional tests is instrumenting them with checks for defect-induced errors. Looking to recent work that describes or publishes functional test content [1], [2], [4], [5], [9]–[12], [19]–[21], three methods are used: comparing (i) inputs and outputs of invertible computations (e.g., encryption followed by decryption) [10], [11]; (ii) outputs of

the same computation run on different threads/cores [9]–[11], [19]; and (iii) the output of a computation to a golden value computed on presumed-healthy hardware [1], [2], [4], [5], [9], [11], [12], [19]–[21]. Notably, none of these methods compares the architectural outputs of the same instruction given the same architectural inputs within the same thread.

Unfortunately, these error checking methods severely limit which programs can serve as tests. Few are composed of invertible computations, and most lack well-defined final outputs for golden-value or cross-thread/core checks. Even when such outputs exist, these checks require deterministic execution, making them impractical for programs involving multi-threading, network I/O, or other sources of non-determinism. These challenges are amplified if finer-grained (i.e., more frequent) checking is desired in order to reduce logical error masking or localize errors to instructions. For this reason, prior work performs instruction localization using short tests only [12] or by heuristically attributing errors to instructions that frequently occur across failing tests [4], [5].

This Paper: Enabling programs as tests with ITHICA. Instead, we propose ITHICA (Intra-THread Instruction Checking Approach), an automatic approach and tool that generates functional tests from programs by instrumenting them with *intra-thread*, *instruction-level* checks, primarily leveraging *instruction-duplication and output comparison* [23]–[29]. These checks eliminate the need for final program outputs and significantly reduce determinism requirements, enabling many more programs to serve as tests. Their instruction granularity makes them less susceptible to error masking and enables identification of failing instructions upon detection.

ITHICA's checking approach is enabled by our **key insight**: the most *pernicious* defects—those most likely to escape manufacturing testing into production—cause *inconsistent* errors: two instances of the same instruction given the same architectural inputs within the same thread can *sometimes* produce *different* architectural outputs. Intuitively, such inconsistencies result from extensive microarchitectural and electrical state in modern processors, which creates a highly diverse execution context for each dynamic instruction instance, thereby modulating how defects affect each instance architecturally. In contrast to *consistent* errors—two instances of the same instruction given the same architectural inputs within the same thread *always* produce the *same* wrong architectural output—that can be exposed from *every execution context*, inconsistent errors are sensitive to specific contexts drawn from an overwhelming space of possibilities (§III-A). Notably, inconsistent errors can arise even when a defect affects both instruction instances; for example, on one server we evaluate (§VII-D), two identical instructions that strongly appear to interact with the same faulty hardware component produce *different wrong outputs* (a type of inconsistent error), enabling ITHICA to detect the errors.

Our **first contribution** is ITHICA itself. To our knowledge, it is the first work to validate that defects cause inconsistent errors on real defective hardware and to exploit this phenomenon for defect detection. ITHICA primarily repurposes intra-thread, instruction-level checking techniques from prior work on soft error detection [23]–[25] and post-silicon validation [26]–[29]. Additionally, it introduces a novel technique that proactively diversifies the execution context of memory instructions, by encouraging them to interact with different levels of the memory hierarchy. Implemented as LLVM compiler passes, ITHICA can be applied to arbitrary programs across multiple architectures. By relying neither on proprietary specifications nor fault models, it bridges the research gap between state-of-the-art functional tests deployed in datacenters and what is publicly available to researchers.

Our **second contribution** is a large-scale testing campaign that demonstrates ITHICA's effectiveness. From Google's industrial fleet of multiple millions of CPU servers, over 3,000 suspect- and confirmed-defective servers—spanning at least 10 microarchitectures—were identified and subjected to multiple ITHICA test types: derived from representative open-source hyperscaler test programs [10] (our baseline), fleet-representative workloads [30], and common libraries [31]–[35]. ITHICA detects *100 defective servers*, which we analyze in detail—more than 3× [4] and 5× [12] the number of the two most detailed prior studies. By exploiting inconsistent errors, ITHICA checks detect 39% more defective servers than *native* final output checks within the same binaries, and 69% more than SiliFuzz [19], the only other functional test generation approach operating at or above the ISA level. ITHICA is the first technique to be directly compared against open-source hyperscaler tests on real defective hardware.

Our **third contribution** is a set of novel findings on defect behavior uniquely enabled by ITHICA's instruction-level error checking within a variety of programs and the high number of servers these tests detect. Many of these findings challenge conclusions drawn by recent hyperscaler studies:

1) ITHICA's unique ability to perform instruction localization within arbitrarily long tests enables our discovery that the *sequence-driven execution context* in which a vulnerable instruction executes is the primary predictor of error manifestation (§VII-C). In contrast, *instruction usage stress* (i.e., the dynamic frequency of a failing opcode in a test) [4], [5] and *culprit inputs* are insufficient predictors (§VII-D).
2) ITHICA enables empirically demonstrating the extreme difficulty of reproducing errors with short tests: among the defective servers analyzed, *only one reproduces errors with single-instruction tests*; the rest require sequences ranging from modestly longer to full programs (§VII-D). Prior work characterizing error trends based primarily on those reproducible with short tests [12] may bias error characterization towards the few cases for which short sequences are effective.
3) ITHICA's ability to test a wide variety of instructions in diverse execution contexts enables our finding that the same defect often causes errors *across multiple instruction types at markedly different rates*—up to six orders of magnitude (§VII-E). In one such case, the detecting hyperscaler baseline test is one targeting vector instructions, yet ITHICA also reveals failing non-vector

instructions within the same test program. This finding cautions against broad claims of *hardware localization* (i.e., attributing defects to hardware components) with functional tests in general, especially those that target few instruction types [4], [5], [12].

Our findings suggest effective SDC testing requires instruction-level checking within long, diverse programs. The ideal—exercising every instruction in every possible execution context—remains an open challenge. ITHICA takes a meaningful step towards it: by leveraging the inconsistent nature of defect-induced errors, it opens up a vast space of previously unusable programs as tests, each exposing instructions to execution contexts that prior checking methods cannot leverage.

II. Background

A. Hardware Faults

Hardware faults are physical flaws or malfunctions. They can be permanent or transient.

Permanent Faults. *Permanent (hard) faults* include manufacturing defects, aging-related wear-out, and design bugs.

Manufacturing defects are flaws introduced during silicon chip manufacturing, and typically impact different physical regions of each affected chip in different ways. A special category of defects is *early-life failures* (ELFs), which are characteristic of weak chips that pass pre-deployment testing [6], [36], but induce erroneous behaviors within the first few weeks to months of deployment [37].

Distinct from defects (*extrinsic* flaws) [36], *aging-related faults* are caused by *intrinsic* failure mechanisms (e.g., Bias Temperature Instability (BTI) [38] or Hot Carrier Injection (HCI) [39]), which result from chip wear-out over time. Chip designs incorporate voltage and speed margins to prevent errors from aging faults [40].

Finally, electrical and logic *design bugs* are flaws in a *design*; thus, they affect most or all chips in a manufacturing batch. Electrical bugs manifest under specific operating conditions, namely voltage, frequency, and/or temperature [41]. Logic bugs are flaws in a chip's logical implementation.

Transient Faults. *Transient faults* are malfunctions that occur when energetic particles, such as neutrons from cosmic rays [42] or alpha particles from packaging materials [43], generate electron-hole pairs in semiconductor devices. The resulting charge can accumulate at a transistor's diffusion nodes and flip the state of a logic device, a phenomenon referred to as *transient (i.e., soft)* error. Transient faults are *not* the result of permanent hardware flaws [44].

B. Symptoms of Hardware Faults

A hardware fault can induce a *hardware error*, i.e., an incorrect bit stored in a flip-flop. Whether a permanent fault induces a hardware error may be: *sequence-dependent*, i.e., dependent on the order of circuit stimuli; or *timing-dependent*, i.e., a subset of sequence-dependent that depends on the speed of stimuli [45]. Timing-dependent faults are dependent on a chip's electrical state (i.e., voltages and currents at transistor terminals) and thus influenced by external factors like power supply noise, frequency or temperature variations [46], [47].

A hardware error may or may not manifest as an *architectural error*, i.e., an incorrect architectural state (§III-A). If it does not, the hardware fault is said to be *architecturally masked* [48] or *benign* [49], [50]. Architectural errors can present with various symptoms, including system crashes, system hangs, or silent data corruption (SDC) [6], [22]. An SDC occurs when an architectural error causes the system to output an incorrect result without any indication that the error occurred [8]. Some architectural errors may not affect the system output at all due to *logical masking* [48]–[51], e.g., if the error is multiplied by zero, or a program's output does not depend on it.

C. Testing Silicon Chips for Defects

The SDC phenomenon recently reported by hyperscalers is generally attributed to defects [1]–[5]. Standard and emerging approaches for detecting defective chips are *manufacturing* and *in-datacenter* testing, respectively.

Manufacturing Testing. Manufacturing testing follows chip fabrication and subjects each chip to scan-based *structural tests* [52] and (non-scan) *functional tests* [53]. Scan-based testing uses *design-for-testability* (DFT) features to apply automatically generated test stimuli to logic circuits and inspect their outputs, leveraging fault models and test metrics for systematically generating scan tests [54]–[61]. Functional testing complements scan testing, but lacks scalably computable, high-quality coverage metrics [15], [18]. Manufacturing testing is highly limited in duration due to cost, making exhaustive testing approaches infeasible [15].

In-Datacenter Testing. Historically, in-datacenter testing concluded with software burn-in testing [2] (different from hardware burn-in [62]), assuming hardware faults would be caught by built-in error detection mechanisms. Upon finding that defect-induced SDCs impact a substantial fraction of servers, datacenter operators have begun subjecting them to periodic functional testing (§I) both in- and out-of-production [2], [4], [19].

III. Motivating ITHICA: The Inconsistent Manifestation of Pernicious Permanent Faults

Our key insight in designing ITHICA is two-fold: (1) defects can cause *inconsistent* architectural errors, i.e., errors that cause some instruction, executed from two different *execution contexts* on the same hardware, with identical architectural inputs, to produce *different* architectural outputs; and (2) such defects are the most pernicious. An execution context denotes the entire microarchitectural (including architectural) and electrical state of a chip. We first derive this insight from a recent theoretical result (§III-A). We then build intuition for how permanent faults can cause inconsistent errors with a pedagogical RTL-level fault injection example in simulation (§III-B).

A. Architectural Errors: Formal Classification

Part one of our insight follows from recent work on formal pre-silicon verification (to detect logic bugs, a type

of permanent fault, §II-A), which proves that a hardware error can manifest as one of three types of architectural errors [63], [64]:[1] (1) an **inconsistent error**, defined above; (2) a **consistent error**, which causes some instruction, executed from every execution context on the same hardware device, to always produce *the same* wrong output for specific architectural inputs; or (3) an **unresponsive error**, which causes some instruction, executed from some context, to fail to produce an output within the expected time frame.

Notably, this result holds regardless of how the hardware error arises, e.g., even if it is caused by a fault that is both timing- and sequence-*independent*. Now, consider such a fault alongside one that is timing- or sequence-*dependent*, where both faults are capable of inducing the same hardware error (§II-B). That is, the first fault induces the hardware error *persistently*, while the second does so only under specific conditions (i.e., *intermittently*). If the error manifests as inconsistent or unresponsive in the first case, it will do so in the second. If the error manifests consistently in the first case, it will newly manifest inconsistently in the second.

Part two of our insight is that faults that induce inconsistent errors are more pernicious (harder to detect), and thus more likely to escape manufacturing testing, than those that induce consistent errors. This is because consistent errors can be exposed by executing instructions with diverse inputs from *any* execution context, while inconsistent errors require executing instructions in *many* distinct execution contexts, drawn from an *overwhelming* space of possibilities.

ITHICA is explicitly designed to detect inconsistent errors and can implicitly detect unresponsive errors, though not consistent ones. This tradeoff pays off empirically: ITHICA detects more defective hardware—that has already undergone manufacturing testing—than state-of-the-art tests that do not exploit this insight (§VII).

B. Pedagogical RTL Fault Injection Example

To build intuition for the theory in §III-A, we present an RTL-level fault injection example featuring a timing- and sequence-independent fault—the case where consistent errors are theoretically more prevalent, and thus where ITHICA is *least* likely to detect an inconsistent error.

Our fault injection targets the open-source RISC-V CVA6 processor [65], a 64-bit, 6-stage, single-, in-order-issue core with speculation. We inject a single (i.e., one-bit) stuck-at fault [14]—a common fault model used in the literature to study permanent faults—into CVA6's SystemVerilog RTL. Specifically, we force `rs1_valid_o` permanently high, causing a persistent hardware error. This signal informs the next instruction to issue that its first operand (`rs1`) is valid. We then run a simple ITHICA test on the "faulty" design in Verilator simulation [66]. The test is an assembly program that features two multiplication instructions (`mul`) using the same architectural inputs, preceded by initialization instructions, as shown next:

```
1 lw a5, %[bval] // Sets value for a5
2 lw a4, %[aval] // Sets value for a4
3 addi t0, a5, 0 // Dummy instruction
4 mul %[s1], a4, a5 // Original mul instruction
5 mul %[s2], a4, a5 // Duplicate (validation) mul instruction
```

We compare the original and duplicate `mul`s' outputs once the simulation has completed. By forcing `rs1_valid_o` permanently high, both `mul`s are forced to perform forwarding of their `rs1` operand value, without explicitly waiting for their producer to complete. Getting the value from the register file is not permitted, due to the presence of the in-flight `lw` producer (line 2) that writes to the same operand. As a result, the first `mul` forwards a reset value (zero) for `rs1`, which was written in the scoreboard when the `lw` instruction issued. The second `mul` forwards the correct value from the `lw` (line 2), which has completed. The same effect is achieved if the *dummy* instruction is placed between the two `mul` instructions instead.

Notably, the microarchitectural simplicity of CVA6 (and the abstractness of RTL) makes such a persistent hardware error **more challenging** for ITHICA to detect: there is *less* opportunity to observe a resulting inconsistent architectural error. Any increase in microarchitectural complexity makes inconsistent errors easier to observe, e.g., a processor with multiple execution units of the same type may dispatch the original and duplicate instructions to different units. However, execution unit redundancy is not a requirement: our real hardware results feature a case of an instruction exhibiting inconsistent errors, detected by ITHICA, despite very likely executing on the same faulty execution unit (§VII-D).

C. Execution Context Diversity

ITHICA exploits our key insight (§III) for defect detection by leveraging **natural diversity**, where the execution context varies passively between the original instruction and its validation (e.g., duplicate in §III-B) instruction (§IV-B), and **proactive diversity** via the novel `MemDiv` transformation (§IV-A), which deliberately perturbs microarchitectural state, and thus execution context, between original and validation instructions. ITHICA also varies input programs (§V-A) and transformations that turn the input programs into tests (§IV-A) to more significantly vary the execution context of both original and validation instructions, thus increasing the likelihood of surfacing defect-induced errors (§III-A).

IV. ITHICA: Intra-THread Instruction Checking Approach for Defect Detection

ITHICA[2] takes as input a program and configuration options (§IV-B) and outputs a functional test by applying one or more of its transformations (§IV-A) to the program (Fig. 1). This section describes test generation with ITHICA and provides details on its design and implementation (§IV-C).

A. ITHICA Program Transformations

Each ITHICA transformation (Table I) inserts additional program instructions to check for errors in the outputs of

[1]Inconsistent, consistent, and unresponsive errors map to *functional consistency*, *single-action correctness*, and *response bound* bugs in prior work [64].

[2]https://github.com/ioannavav/ithica

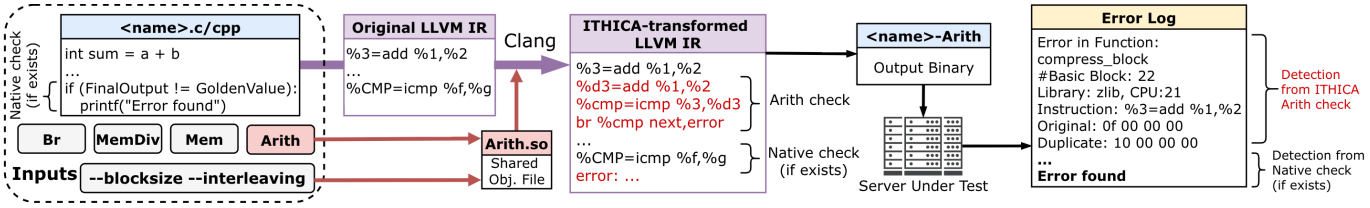


Fig. 1. Given an input program (`<name>.c/cpp`), ITHICA applies one or more transformations—implemented in this paper for LLVM IR—configured with some `blocksize` and `interleaving`, and outputs a functional test `<name>-ITHICA` with intra-thread instruction checks.

specific *original* program instructions, including: (i) *validation* instructions, which perform redundant computations; (ii) *check* instructions, which detect inconsistent errors between original/validation instructions; (iii) *error-reporting branches*, which redirect control flow to an error-reporting basic block when an inconsistent error is detected; and (iv) novel *diversity* instructions, which create proactive diversity in microarchitectural state between the execution of original and validation instructions. The first three transformations below are adapted from prior work on post-silicon validation for design bugs [26]–[29] and soft error detection [23]–[25].

`Arith` instruments arithmetic (i.e., computational) instructions. Validation instructions duplicate the original ones and their results are checked for mismatches against them. Duplicates preserve original data dependencies, allowing error propagation and detection even when comparisons are inserted less frequently (§IV-B).

`Mem` instruments load and store instructions. Loads are duplicated and their outputs are compared against the originals. For stores, validation instructions are loads from the store's address, and their outputs are compared against the original store's value operand.

`Br` instruments conditional branches to verify that they resolve to the correct target by recording the *intended* target address before the branch resolves, and validating it against the *actual* address taken. `Br` is more lightweight than prior control-flow checking work [24], [25], [67], [68]: it exclusively targets conditional branches, and focuses on wrong-target misdirections between two valid targets, which are more likely to manifest silently. Branching to a rogue address is more likely to result in a non-silent architectural error, like a crash (§II-B), which would be detected without functional testing [69].

A limitation of `Mem`, shared by ITHICA and prior work [27], [29], is that validation loads typically retrieve data: via an L1 cache hit (for checking loads), possibly missing errors in load outputs during L1 misses; or a core-local store buffer (for checking stores), possibly missing errors in store outputs at lower memory hierarchy levels. To address this, ITHICA introduces the novel `MemDiv` transformation, representing the first use of *proactive diversity* (§III-C) to check for errors affecting memory instructions.

`MemDiv` extends `Mem` by introducing diversity instructions, namely *mfence* (*memory fence*) and *clflush* (*cache line flush*) (though others are also possible). These instructions flush store buffers and caches by virtual address, respectively, encouraging original and validation memory instructions to interact with and exercise defects within different levels of the memory hierarchy. Specifically, *mfence* encourages retrieval via an L1 cache hit. The subsequent *clflush* forces data retrieval directly from main memory. Note that parity/ECC might not cover all data movement of a complex memory system, nor be able to detect all errors [70].

B. ITHICA Configuration Options

ITHICA supports configuring the location of validation instructions and the insertion frequency of comparison instructions per basic block. Configuration options are available for all transformations except `Br`, which targets at most one instruction per basic block.

Configuring Interleaving. ITHICA's *interleaving* controls the insertion points of validation/diversity instructions relative to originals. An interleaving of n indicates that n original instructions will appear consecutively in program order before their corresponding validation/diversity instructions appear in the same order.

Configuring Block Size. ITHICA's *block size* controls the insertion frequency of comparison instructions. A block size of n indicates that comparisons will be inserted for every n-th original instruction. Validation instructions are inserted for *all* relevant original instructions, ensuring that errors impacting validation instructions can propagate through dependency chains to comparison instructions (unless they get logically masked, §II-B).

C. ITHICA Design and Implementation

Design Choices. Intra-thread instruction checking techniques have been implemented at different levels of the software stack: as source code transformations [26], [71], [72], within compiler IR [68], [73]–[75] or backend [25], [67], [76] passes, and during post-processing of generated assembly [23], [24], [77]. To achieve portability necessary for fleet-scale deployment across diverse microarchitectures, ITHICA operates at the LLVM IR level; IR passes are target-agnostic and can be integrated into existing build systems without compiler modifications.

ITHICA divides checking into four individual passes organized by check type (Table I), and separately implements their combinations (`Arith+Mem`, `Arith+MemDiv`, `Arith+MemDiv+Br`). This modularity allows independent deployment and controlled evaluation of each pass's detection contribution. A design distinction from prior work on soft error detection is check instruction placement (§IV-A). There, logical

TABLE I
ITHICA TRANSFORMATION PASSES AND VALIDATION RULES

Pass	Target Inst	Original Inst	Validation Inst	Check
Arith	Logic, binary, bitwise, vector, unary, aggregate & conversion ops, *GEP*, *icmp*, *fcmp*, *select,* intrinsics & inline asm [78]	r0 = op(s0,s1)	r1 = op(s0,s1)	eq(r0,r1)
Mem	Load (non-atomic, non-volatile)	v0 = load(a0)	v1 = load(a0)	eq(v0,v1)
	Store (non-atomic, non-volatile)	store(v0,a0)	v1 = load(a0)	eq(v0,v1)

MemDiv	Load (non-atomic, non-volatile)	v0 = load(a0)	v1 = load(a0);
			clflush(a0);
			v2 = load(a0)
				eq(v0,v1) &&
				eq(v0,v2)

	Store (non-atomic, non-volatile)	store(v0,a0)	v1 = load(a0);
			mfence();
			v2 = load(a0);
			clflush(a0);
			v3 = load(a0)
				eq(v0,v1) &&
				eq(v0,v2) &&
				eq(v0,v3)

Br	Conditional branch	br(c0,t0,t1)	src: e = c0 ? t0:t1; store(e,tmp) t0: v0 = load(tmp) t1: v1 = load(tmp)	src: None t0: eq(v0,t0) t1: eq(v1,t1)

masking of errors before they become externally visible (i.e., at stores and branches [23], [25]) is desirable, so checks are inserted only at those points. For defects, logical masking prevents detection, so ITHICA supports block sizes down to 1, giving every instruction its own check. Block size of 1 further enables instruction localization, which is useful for characterizing permanent defects (§VII-C–§VII-E), but uninformative for transient soft errors, which do not recur.

Implementation Challenges. The first challenge, unique to ITHICA's LLVM IR implementation, is *redundancy preservation*. Despite applying ITHICA as the final IR-level pass to minimize interference with compiler optimizations, backend optimizations can still eliminate inserted instructions. To prevent this without modifying the LLVM compiler backend, validation loads and stores are marked as volatile; for arithmetic instructions, which cannot be marked volatile directly, their arguments are passed through no-op volatile store-load pairs to achieve the same effect.

The second challenge is ensuring instrumentation *correctness*, which requires two conditions: validation instructions need to be side-effect free (to preserve observable program behavior), and original and validation instructions need to yield the same output in the absence of hardware errors. Most instructions covered by `Arith` always meet these conditions, and are duplicated unconditionally. The exceptions are LLVM intrinsics and inline assembly, which are first checked against LLVM attributes [78] to confirm they are side-effect free, and only then duplicated. `Br` is safe by construction, as it inserts a validation stack store and load with no side-effects. For `Mem` and `MemDiv`, there are multiple original instructions with side-effects for which a validation copy would alter program behavior—including atomic and volatile loads and stores, *atomicrmw*, *cmpxchg*, and *alloca* [78]—which are therefore excluded. Non-atomic/non-volatile loads and stores are safely instrumented under a thread-safety assumption; permitted data races in thread-unsafe multithreaded code can cause false positives. For example, OpenSSL intentionally permits certain data races [79], so ITHICA does not instrument those functions with `Mem` and `MemDiv` in our experiments (§VII).

V. Testing Campaign: ITHICA on Real Hardware

To test our insight (§III) and evaluate ITHICA tests generated from various programs (§V-A), we conduct a large testing campaign (§V-B).

A. Instrumenting Diverse Input Programs

We use ITHICA to instrument three input program types: (i) industry-representative functional test programs comprising Google's **cpu-check (`CC`)** [10]; (ii) industry-representative workloads comprising Google's **Fleetbench (`FB`)** [30], [80]; and (iii) **libraries** [31]–[35] used by `CC` and `FB`. When ITHICA generates a functional test from a program, we append "`-`" and then "`ITHICA`" or the name of the applied transformation (e.g., "`Arith`") to the program name to get the new test name (e.g., `CC-ITHICA` or `CC-Arith`, respectively).

cpu-check. `CC` is a suite of bespoke functional tests with final output checks (§I) developed by Google for in-datacenter testing (§II-C). Test programs include: invertible computations in sequence (e.g., encryption-decryption) with input-output comparisons; checksum and hash computations on random, non-deterministic data with cross-core output comparisons; and AVX computations with vector lane output comparisons. Most of our experiments evaluate ITHICA tests generated from CC, since its native checks provide a comparison baseline, and its C++ implementation allows for microarchitecture-independent LLVM compilation (unlike SiliFuzz tests [19]).

Fleetbench. `FB` consists of C++ microbenchmarks, representing hot functions in Google datacenters, including Proto, Swissmap, Libc, Tcmalloc, Hashing, Compression, and Stl-cord, executed with deterministic data pre-sampled from a production distribution or randomly generated with a fixed seed. We exclude Compression due to its reliance on *OpenMP*, which we could neither statically compile nor access as a system library on our servers.

Libraries. For both `CC` and `FB`, we instrument both top-level and library code. For `CC`, we apply ITHICA to `Libc`, `Libc++`, `Abseil`, `OpenSSL`, and `Zlib`. Due to tight coupling of `Libc` and `Libc++` with GCC, we instead apply ITHICA to LLVM's overlay libraries, `Llvmlibc` and `Llvmlibc++`. For `FB`, we do not explicitly instrument libraries, like we do for `CC`, but rely on `FB`'s build system (Bazel) to rebuild all relevant libraries (e.g., Abseil) with ITHICA.

B. Two-Pool Evaluation Strategy

We conduct experiments on two groups of CPU servers drawn from Google's fleet (consisting of many million CPUs): the **Quarantine Pool (QPool)** and the **Defective Pool (DPool)**.

The QPool consists of more than **3,000 servers** quarantined by Google for out-of-production testing due to suspicious behavior. They span **at least 10 microarchitectures** across two CPU server vendors. It is a *dynamic* pool (i.e., new servers are regularly added and removed) containing both suspect- and

TABLE II
DPOOL CHARACTERISTICS

	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14
μ**arch**	u1	u2	u3	u4	u5	u5	u2	u6	u1	u4	u3	u5	u1	u2
Age (months)	19	6	34	16	45	38	81	19	24	11	11	42	10	64

TABLE III
EVALUATION METRICS – EACH RUN SPANS ONE HOUR

Metric	**Description**
Server Detections	# servers with error detections
Error Detection Rate (EDR)	# runs with error detections / # all runs (%)
Error Frequency (EF)	# error detections / # all runs
Time to Detection (TTD)	time from test start to first error
PC Sensitivity *(per opcode)*	# failing static PCs / # all static PCs (%)
BB Sensitivity *(per opcode)*	# failing basic blocks (BBs) / # all BBs (%)
Input Breadth *(per opcode)*	# unique failing inputs / # all failing inputs (%)

confirmed-defective servers. Servers are offlined to the QPool via three mechanisms: (1) customer complaints about data corruption; (2) flagging by opportunistically-run hyperscaler- and vendor-developed screening tests; and (3) forensics-based analysis of hardware and software exceptions (e.g., fail-stop and fail-slow errors including machine check exceptions, kernel or user crashes, invariant fails in user code).

The DPool consists of 20 confirmed-defective servers, originally part of the larger QPool, that we set aside for dedicated testing with ITHICA. All 20 have failed at least one functional test developed either by Google or by the CPU vendor and exhibit diversity in terms of microarchitecture and age (Table II). Across all our experiments, no functional test (i.e., neither ITHICA nor CC) detects any error on six of them. Thus, we report results for the remaining **14 servers**.

Benefits of Two-Pool Evaluation. The DPool and QPool serve distinct roles. The DPool provides a controlled environment with confirmed-defective servers, enabling us to evaluate our key insight (§III). Its static nature and our exclusive access to it ensure sufficient runtime to evaluate all ITHICA transformations and configurations: we collect **over 2,000 hours of testing time per DPool server**, exceeding the allocated time of previous detailed studies by 1–2 orders of magnitude [4], [5], [12]. In the QPool, ITHICA tests integrate into Google's standard out-of-production testing pipeline alongside other industrial tests, enabling large-scale evaluation in a real-world in-datacenter setting.

VI. EXPERIMENTS OVERVIEW

The experimental evaluation in this paper is partitioned into two subsections: testing on the DPool (§VI-A) and testing on the QPool (§VI-B). Across our experiments, we compare the efficacy of ITHICA tests to CC (our baseline) using four main metrics, defined in the first four rows of Table III. The results are interpreted in detail in §VII.

A. Testing on the DPool

First, to compare CC-ITHICA to CC, we run **eight functional tests** on each DPool server. Seven are CC-ITHICA tests obtained by applying its four main transformations (§IV-A) and three combinations thereof (Arith+Mem, Arith+MemDiv, Arith+MemDiv+Br) on CC, all with block size and interleaving set to 1. The eighth is the original CC (i.e., without ITHICA instrumentation) for comparison. Each test runs 100 times per server (**100 runs**) each lasting **one hour** unless interrupted by a crash. Between runs, the server is rebooted.

Within an ITHICA test, we distinguish between Native checks (checks in the original program, i.e., CC's final output checks) and ITHICA checks (inserted by ITHICA). If either check type detects an error before a crash, the detection counts. CC contains only Native checks; CC-ITHICA contains both. Each run executes hundreds of thousands of rounds of the CC-ITHICA program (and more rounds for CC), each with different randomly-generated inputs, yielding hundreds of thousands of inputs per run.

Fig. 2 shows EDR (Table III) for the seven CC-ITHICA tests and CC across all 14 DPool servers (D1–D14). Parenthetical values show the subset of detections followed by crashes; the *Cr* column shows crashes without any other detection.

To compare different ITHICA interleaving and block size configurations (§IV-B) for the same program, we conduct two other experiments. First, fixing block size to 1, we **sweep interleavings** of 1, 2, 4, 8, and max (basic block length). Second, fixing interleaving to 1, we **sweep block sizes** of 1, 2, 4, 8, and dep, which inserts comparisons at the end of each instruction's dependency chain. We evaluate each configuration with **100 one-hour runs**. Fig. 5 shows the results of these two experiments.

Finally, we evaluate ITHICA test content generated from two types of non-test programs, using Arith with block size and interleaving of 1, since this configuration yielded the most detections on the DPool for CC-ITHICA (Fig. 2, §VII-A): **five libraries** used by CC and **six FB workloads** (§V-A). We evaluate each FB-ITHICA binary for **100 one-hour runs**.

B. Testing on the QPool

Due to limited available execution time in the QPool compared to the DPool, we primarily run **CC-Arith** with block size and interleaving of 1, again due to its success for CC-ITHICA in the DPool. The total execution time accumulated per server varies, ranging from **20 to 100 hours** due to the presence of other tests in the pipeline and the dynamic nature of the QPool.

We secondarily run **CC-Mem, CC-MemDiv, and CC-Br** with block size and interleaving of 1, reporting *only their unique detections* compared to CC-Arith due to their shorter runtime (at least **20 hours**). The results are summarized in Fig. 3, alongside the aggregate DPool results. Our baseline remains Native checks within the same CC-ITHICA binaries. To ensure reliable analysis, these results exclude servers with less than 20 hours per test.

Finally, as in the DPool, we evaluate CC's **five libraries** and the **six FB workloads** transformed with Arith with block size and interleaving of 1 for **20 to 100 hours**.

	D1			D2			D3			D4			D5			D6			D7			D8			D9			D10			D11			D12			D13			D14		
	Ith	Nat	Cr	Ith	Nat	Cr	Ith	Nat	Cr	Ith	Nat	Cr	Ith	Nat	Cr	Ith	Nat	Cr	Ith	Nat	Cr	Ith	Nat	Cr	Ith	Nat	Cr	Ith	Nat	Cr	Ith	Nat	Cr	Ith	Nat	Cr	Ith	Nat	Cr	Ith	Nat	Cr
Arith	100 (9)	0 (0)	0	0 (0)	37 (1)	0	43 (3)	39 (3)	2	89 (4)	4 (4)	11	3 (0)	0 (0)	0	0 (0)	0 (0)	0	97 (91)	70 (66)	3	2 (0)	0 (0)	0	93 (0)	0 (0)	0	1 (0)	1 (0)	1	88 (1)	86 (0)	3	7 (2)	0 (0)	0	0 (0)	20 (2)	12	1 (1)	0 (0)	2
Mem	0 (0)	0 (0)	0	16 (15)	31 (15)	10	50 (0)	61 (0)	1	0 (0)	97 (0)	3	0 (0)	0 (0)	0	0 (0)	0 (0)	0	0 (0)	74 (0)	26	0 (0)	0 (0)	1	0 (0)	0 (0)	3	0 (0)	0 (0)	0	3 (0)	97 (0)	0	0 (0)	0 (0)	0	0 (0)	62 (25)	12	0 (0)	0 (0)	0
MemDiv	0 (0)	0 (0)	0	0 (0)	0 (0)	28	0 (0)	0 (0)	0	0 (0)	100 (0)	0	0 (0)	0 (0)	0	0 (0)	0 (0)	0	0 (0)	100 (0)	0	0 (0)	0 (0)	0	0 (0)	0 (0)	0	0 (0)	0 (0)	0	0 (0)	45 (0)	0	1 (0)	1 (0)	0	3 (2)	1 (0)	1	0 (0)	0 (0)	0
Br	0 (0)	0 (0)	0	0 (0)	23 (0)	0	0 (0)	40 (10)	0	0 (0)	100 (0)	0	0 (0)	0 (0)	0	0 (0)	0 (0)	0	0 (0)	100 (0)	0	0 (0)	0 (0)	3	0 (0)	0 (0)	0	0 (0)	0 (0)	0	0 (0)	99 (0)	0	0 (0)	0 (0)	0	0 (0)	85 (39)	2	0 (0)	0 (0)	0
Arith+ Mem	40 (2)	0 (0)	0	9 (9)	35 (9)	25	42 (8)	42 (8)	0	95 (9)	9 (9)	5	0 (0)	0 (0)	0	0 (0)	0 (0)	0	22 (13)	17 (9)	58	0 (0)	0 (0)	3	85 (1)	0 (0)	5	0 (0)	0 (0)	0	77 (0)	79 (0)	8	4 (3)	0 (0)	0	0 (0)	8 (0)	14	0 (0)	1 (1)	1
Arith+ MemDiv	10 (1)	0 (0)	0	0 (0)	0 (0)	28	70 (3)	10 (3)	0	100 (2)	82 (0)	0	0 (0)	0 (0)	0	0 (0)	0 (0)	0	100 (100)	100 (100)	0	0 (0)	0 (0)	0	100 (0)	0 (0)	0	0 (0)	0 (0)	0	57 (0)	59 (0)	0	3 (0)	0 (0)	0	0 (0)	0 (0)	2	0 (0)	0 (0)	0
Arith+ MemDiv+Br	10 (0)	0 (0)	0	0 (0)	0 (0)	10	60 (0)	10 (0)	0	100 (0)	76 (0)	0	0 (0)	0 (0)	0	0 (0)	0 (0)	0	100 (100)	100 (100)	0	0 (0)	0 (0)	1	99 (0)	0 (1)	0	0 (0)	0 (0)	0	54 (0)	58 (0)	0	0 (0)	0 (0)	0	0 (0)	0 (0)	0	0 (0)	0 (0)	0
CC	—	0 (0)	0	—	73 (6)	8	—	15 (1)	0	—	100 (0)	0	—	0 (0)	0	—	0 (0)	0	—	95 (0)	0	—	0 (0)	2	—	0 (0)	0	—	0 (0)	0	—	98 (0)	2	—	2 (0)	0	—	55 (3)	8	—	1 (0)	1

Fig. 2. EDR of CC-ITHICA (with block size and interleaving of 1) and CC in DPool. Each triplet reports results for ITHICA (*Ith*), Native (*Nat*), and crashes with no detection (*Cr*). The subset of runs that crashed after detection is shown in parentheses. D6 is uniquely detected by Arith for interleaving of 8 (§VII-C).

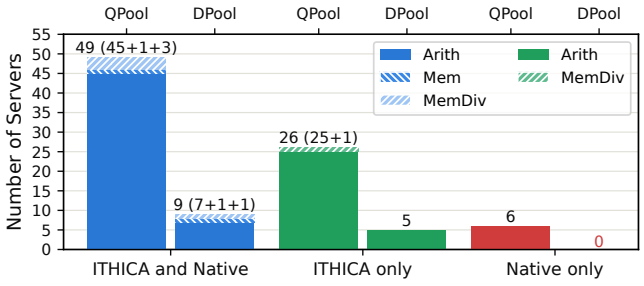


Fig. 3. Server detections across all CC-ITHICA runs in both pools.

VII. Results

We present ten key findings from the experiments in §VI-A and §VI-B, along with follow-up experiments for further analysis. We first establish ITHICA's effectiveness for defect detection (§VII-A, §VII-B). We then leverage ITHICA's instruction-level visibility to characterize defect behavior and the factors governing detection (§VII-C–§VII-E), contrasting our findings with prior work's conclusions (§VII-D, §VII-E).

A. Defects Manifest as Inconsistent Errors

First, consider Fig. 2, produced from running seven CC-ITHICA tests and CC in the DPool (§VI-A). **Obs. 1** The figure shows that CC-ITHICA tests collectively detect all 14 DPool servers. Among the four base transformations, Arith is the most effective, detecting 11 out of 14 servers. This result may reflect survivorship bias (§IX) in the tests used to flag servers for offline testing (§V-B). Br does not detect errors in the DPool but flags non-unique servers in the QPool (§VI-B). **Obs. 2** Four servers (D1, D5, D8, D9) are uniquely detected by ITHICA checks in CC-ITHICA tests, while missed by Native checks in the same binaries as well as by CC. D6 is also uniquely detected by Arith for interleaving of 8 (§VII-C).

Next, consider Fig. 3, produced from running four/seven CC-ITHICA tests in the QPool/DPool (§VI-B/§VI-A). Of the 49 QPool servers detected by both ITHICA and Native checks in CC-ITHICA tests, 45 were flagged by Arith, 1 uniquely by Mem, and 3 uniquely by MemDiv. **Obs. 3** An additional 26 QPool servers were only captured by ITHICA; 25 by Arith and 1 uniquely by MemDiv. Only six servers were detected by Native but not ITHICA. Two likely explanations are gaps in ITHICA's instrumentation and errors that manifest consistently, discussed in §IX. Across both pools, CC-ITHICA tests detect 89 total servers (75/14 in the QPool/DPool)—**39% more than Native checks** in the same binaries. Finally, ITHICA tests derived from FB (§VI-A, §VI-B) detect **11 additional servers** in the QPool (§VII-C), resulting in **100 servers** (89 + 11) detected by ITHICA across both pools.

TABLE IV
Decomposition of Instructions Exhibiting Errors in DPool

Instruction with Error	Original	Validation	Both
% across all DPool runs	31.3%	44.7%	24.0%

Notably, inconsistent errors are not limited to cases where only one of the two instruction instances is affected by a defect: ITHICA can detect errors even when both are. Table IV shows which of the two instructions (original or validation) exhibits an error when ITHICA detects one across DPool runs. **Obs. 4** In 24% of cases, *both* produce *incorrect but different* outputs. These cases are particularly informative because they constrain the possible explanations for the inconsistency: since both instances are incorrect, both were likely affected by the same defective hardware component, yet still produced different outputs. In contrast, when only one instruction is incorrect, we cannot determine whether they interacted with the same component or different ones (i.e., one defective and one not).

Finding 1: Defects, despite being permanent faults, manifest as inconsistent errors in nearly all servers flagged as defective in our testing campaign (Obs. 1–4).

For the Fig. 3 experiment, Table V compares ITHICA and Native checks on several other metrics (Table III). **Obs. 5** Across 58 servers detected by both ITHICA and Native (49/9 in the QPool/DPool), ITHICA achieves, on average, 1.78× EDR improvement, 7.51× EF improvement and 0.68× TTD improvement (1.47× faster).

Finding 2: ITHICA checks outperform final output checks used by hyperscaler tests across all evaluated metrics (Obs. 2, 3, 5).

Fig. 4 shows the distribution of failing instructions and their average EF across all CC-ITHICA tests run in the DPool with

TABLE V
COMPARISON OF ITHICA AND NATIVE DETECTION METRICS (EDR, EF, TTD) FOR 58 SERVERS DETECTED BY BOTH CHECKS

Ratio ITHICA/Native	Geom. Mean	Arith. Mean	Median	Min	Max
Error Detection Rate (EDR)	1.78	4.50	1.17	0.52	100.00
Error Frequency (EF)	7.51	130.15	3.44	0.02	5396.00
Time to Detection (TTD)	0.68	0.77	0.56	0.33	1.72

a block size of 1 and an interleaving of 1, except D6, which is uniquely detected at interleaving of 8 (§VII-C). A "failing instruction" denotes one that exhibits an incorrect output; it does not imply a particular defective hardware unit (§VII-E). Cases where no specific instruction is identified (*no-instr*) can result from an error in an instrumentation instruction itself (e.g., `icmp`), or a crash before logging completes.

Obs. 6 The data reveals a diverse range of affected instructions across servers, spanning arithmetic, floating-point, vector, and memory instructions. Average error frequencies also vary drastically, from less than one per run to over a thousand. Both observations reflect the fact that defects impact different physical regions of each chip in different ways (§II-A).

While ITHICA can detect inconsistent errors regardless of their source, the observed error characteristics in our experiments narrow plausible causes. **Obs. 7** Their frequency and repeatability (Fig. 4) make transient faults less plausible; their device-specific (Fig. 4) yet microarchitecture-agnostic (Table II) nature points away from design bugs; their presence across devices of varying ages (Table II) challenges aging effects as a sole cause. Taken together, these observations are consistent with defects as the likely cause (§II-A).

Finding 3: ITHICA-detected errors in our server pools are most consistent with defects as the underlying cause (Obs. 6, 7).

Implications for Chip-Users: *Intra-thread, instruction-level checking for inconsistent errors is highly effective for defect detection. This finding—not leveraged in any existing functional test for defects [1], [2], [4], [5], [9]–[12], [19]–[21]—enables the use of arbitrary programs as tests, removing bias toward those compatible with existing error checking techniques (§I).*

B. ITHICA Paves the Way for Online Testing

The permanent nature of defects has a direct implication for ITHICA instrumentation. For defect detection (i.e., identifying whether a server is defective), catching *any one* error per server suffices, unlike soft error detection which requires catching *every* corruption that affects an application's output. This distinction enables ITHICA instrumentation to be reduced, trading EDR for lower overhead, while maintaining high defect detection coverage (i.e., number of defective servers detected). Notably, for error characterization, fine-grained instrumentation at every instruction remains necessary (§VII-E).

Two strategies can be used, separately or in combination, for reduced ITHICA instrumentation: *sampling in time* (i.e., choosing the check insertion frequency in checked program regions) and *sampling in space* (i.e., choosing which program regions are checked). In this paper, we investigate a sampling in time strategy: varying *block size* (§IV-B). Fig. 5 shows that a block size of 1 performs best on average, likely due to reduced logical error masking. **Obs. 8** However, larger block sizes demonstrate comparable effectiveness: all DPool servers with reproducible (i.e., more than one) detections at block size 1 (all except for D10 and D14) are also detected at larger block sizes. This indicates that checking frequency can be relaxed without significantly compromising the number of detected servers. The runtime overhead of `Arith` decreases from an average of 2.22× at block size 1 to 1.56× at block size 8 (Table VI).

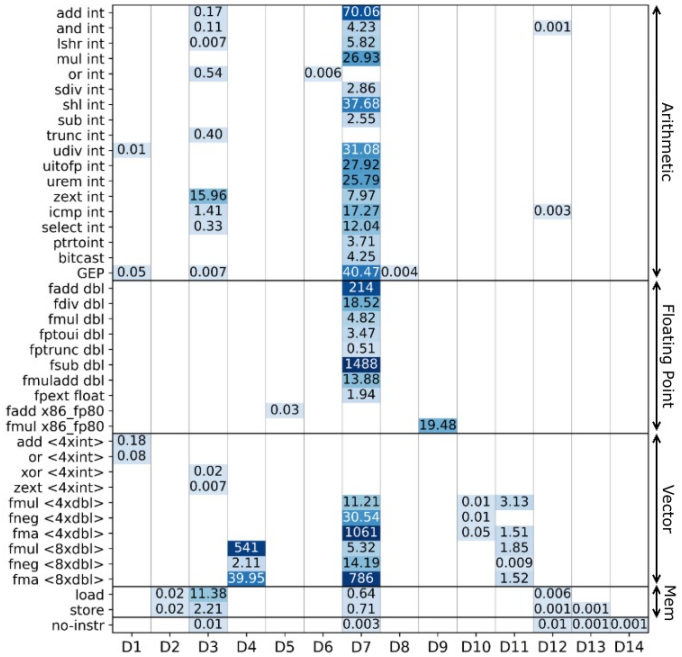


Fig. 4. Distribution of instructions where ITHICA detects errors, with EF calculated across all `CC-ITHICA` runs in DPool.

Finding 4: ITHICA instrumentation and runtime overhead can be reduced at high defect detection coverage (Obs. 8).

Implications for Chip-Users: *The permanent nature of defects can be exploited for relaxed instrumentation. ITHICA's fine-grained, flexible checking, easily applied to new programs and tunable in frequency and scope, can support a wide variety of relaxation strategies. Establishing ITHICA's effectiveness for defects is the first step toward low-overhead online testing—important future work.*

C. Execution Context Drives Defect Detection

ITHICA error detections appear sensitive to two mechanisms for varying execution context. First, the instruction sequence preceding an opcode instance determines whether it executes in a context where it *can* exhibit an inconsistent error. Second, execution context variation via natural and proactive diversity (§III-C) between an original instruction with that opcode and its validation instruction determines whether it *does*.

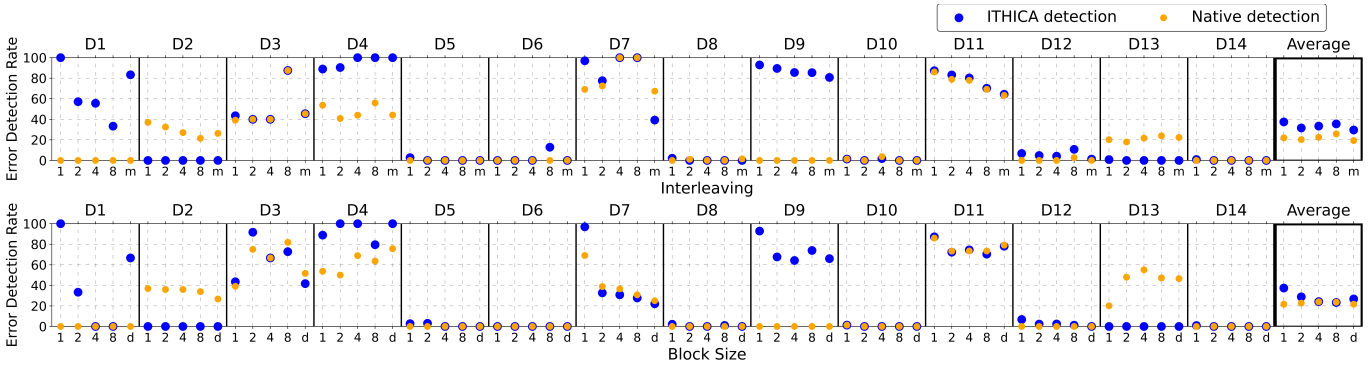


Fig. 5. Impact of interleaving and block size on EDR for CC-Arith in DPool. Top row: effect of varying interleaving (m=*max*, length of the basic block). Bottom row: effect of varying block size (d=*dep*, length of instruction dependency chain). The rightmost panels show average EDR across all servers.

TABLE VI
PERFORMANCE OVERHEAD AND BINARY SIZE INCREASE (IN ×) OF ITHICA COMPARED TO CC, ACROSS DPOOL MICROARCHITECTURES

	ITHICA Pass	Arith (Block Size) (1)	(2)	(4)	(8)	(dep)	Mem	MemDiv	Br
Performance	u1	2.26	2.06	1.88	1.71	1.85	1.13	29.36	1.11
	u2	2.23	1.88	1.66	1.58	1.43	1.10	34.89	1.10
	u3	2.28	1.93	1.75	1.62	1.82	1.23	29.63	1.18
	u4	2.30	1.70	1.56	1.49	1.63	1.11	50.17	1.09
	u5	2.07	1.70	1.53	1.42	1.44	1.09	48.49	1.08
	u6	2.17	1.80	1.61	1.54	1.69	1.07	57.23	1.05
	Avg	2.22	1.85	1.67	1.56	1.64	1.12	41.63	1.10
Binary Size		8.47	5.88	4.59	4.11	6.11	3.06	13.12	1.24

Sensitivity to preceding instruction sequence. First, we examine how preceding instruction sequences influence ITHICA error detections across different instances of the same opcode within the same test. For each opcode-server combination that ITHICA detects an error on, we analyze what fraction of static program counters (PCs) executing the opcode exhibit errors to calculate **PC Sensitivity** (Table III). Opcodes with only one PC or one error are omitted from the table. **Obs. 9** For most failing opcodes, PC Sensitivity is very low (often <1%, Table VII), revealing that the same opcode behaves correctly in the vast majority of its program locations. The same is true for basic block (BB) Sensitivity. A notable exception is vector double instructions, which exhibit high PC and BB Sensitivity. However, for each of these opcodes, all PCs are contained within a *single basic block*, and therefore likely execute under a similar execution context. Similarly, other opcodes with relatively high PC Sensitivity—namely floating-point and vector integer instructions—also exhibit high BB Sensitivity, with failing PCs concentrated within one of only two basic blocks containing them (Table VII).

Next, we show that applying different ITHICA transformations that check at least one common opcode to the same program varies the preceding instruction sequences of those opcodes, similarly affecting detections. **Obs. 10** Per Fig. 2, combined passes (Arith+Mem, Arith+MemDiv, Arith+MemDiv+Br) have the benefit of detecting errors across multiple instruction types (e.g., both arithmetic and memory instructions for D12), but sometimes have lower EDR or entirely miss servers detected by individual passes. This can be attributed to the additional instrumentation instructions changing the instruction sequence compared to individual passes, and thus disrupting the execution context needed for detection.

Finally, instruction sequences preceding checked opcodes vary across input programs, likewise influencing detections. Table VIII shows detected servers across the top-level CC code in CC-ITHICA tests, its library code, and FB. Since CC comprises multiple subtests (§V-A), we separate results per subtest. For each, we report total server detections and unique server detections compared to other tests or subtests.

Among CC subtests, Avx detects the most unique servers. This is unsurprising since it is the only test exercising large (4×, 8×) vector operations. Among the instrumented libraries, Zlib detects the most servers (23 total). **Obs. 11** Of the 31 servers missed by Native in CC-ITHICA (Fig. 3), **eight are detected exclusively by ITHICA within library code**: six in Zlib only, one in OpenSSL only, and one in both Zlib and OpenSSL. This demonstrates that even if Native checks were manually constructed to be more fine-grained in the top-level code, they could still miss errors that manifest in library code and are masked before reaching the library function's output.

Obs. 12 FB tests (§VI-A/§VI-B) detect 24 servers (20/4 in the QPool/DPool), including **11 unique QPool servers** (not in Fig. 3) missed by both top-level CC (both ITHICA and Native) and libraries, for a **total of 100 servers detected by ITHICA**. Importantly, none of the FB programs could be used as tests without ITHICA, as they lack built-in checkable outputs.

Finding 5: Whether an instruction exhibits a defect-induced error depends on whether it executes in a vulnerable context shaped by its preceding instruction sequence (Obs. 9–12).

Sensitivity to natural or proactive diversity. **Obs. 13** Fig. 5 shows that natural diversity with an interleaving of 1 (where each validation instruction immediately follows its original) is sufficient to expose inconsistent errors for most servers in the DPool. However, larger interleavings further improve EDR for some servers (e.g., D3, D4). Notably, D6 is exclusively detected at an interleaving of 8.

TABLE VII
PC SENSITIVITY, BB SENSITIVITY AND INPUT BREADTH PER OPCODE AND PER DPOOL SERVER, ACROSS CC-ITHICA RUNS

Category	Failing Opcode	Affected Servers	# Total PCs	# Failing PCs	PC Sensitivity Geom. Mean (Std)	# Total BBs	# Failing BBs	BB Sensitivity Geom. Mean (Std)	# Total Failing Inputs	# Unique Failing Inputs	Input Breadth Geom. Mean (Std)
Integer Arithmetic	add int	D3	1,202	5	0.42% (0.00)	664	4	0.60% (0.00)	22	7	31.82% (0.00)
	and int	D3 / D12	748	3 / 1	0.23% (0.13)	634	3 / 1	0.27% (0.00)	2 / 1	2 / 1	100.00% (0.00)
	lshr int	D3	424	1	0.24% (0.00)	203	1	0.49% (0.00)	1	1	100.00% (0.00)
	or int	D3 / D6	180	7 / 1	1.47% (1.67)	145	7 / 1	1.82% (2.07)	776 / 9	34 / 2	9.87% (8.92)
	trunc int	D3	492	5	1.02% (0.00)	240	5	2.08% (0.00)	6	4	66.67% (0.00)
	udiv int	D1	36	1	2.78% (0.00)	25	1	4.00% (0.00)	1	1	100.00% (0.00)
	zext int	D3	809	46	5.69% (0.00)	489	32	6.54% (0.00)	19.71K	217	1.10% (0.00)
	getelementptr	D1 / D3 / D8	5,568	6 / 1 / 1	0.03% (0.04)	2,412	6 / 1 / 1	0.08% (0.10)	3 / 1 / 4	3 / 1 / 1	63.00% (35.36)
Comparisons	icmp int	D3 / D12	3,968	12 / 1	0.09% (0.14)	3,552	12 / 1	0.10% (0.15)	17 / 1	4 / 1	48.51% (38.24)
	select int	D3	617	7	1.13% (0.00)	419	7	1.67% (0.00)	84	4	4.76% (0.00)
FP Arithm	fadd x86_fp80	D5	4	1	25.00% (0.00)	2	1	50.00% (0.00)	36	6	16.67% (0.00)
	fmul x86_fp80	D9	4	1	25.00% (0.00)	2	1	50.00% (0.00)	17.92M	1.33K	0.01% (0.00)
Vector <4xint>	add <4xint>	D1	11	2	18.18% (0.00)	2	1	50.00% (0.00)	1	1	100.00% (0.00)
	or <4xint>	D1	2	1	50.00% (0.00)	2	1	50.00% (0.00)	16	4	25.00% (0.00)
Vector <4xdbl>	fmul <4xdbl>	D10 / D11	8	3 / 8	61.24% (31.25)	1	1	100.00% (0.00)	6 / 740.92K	4 / 1.57K	3.76% (33.23)
	fneg <4xdbl>	D10	8	5	62.50% (0.00)	1	1	100.00% (0.00)	2	2	100.00% (0.00)
	fma <4xdbl>	D10 / D11	8	7 / 5	73.95% (12.50)	1	1	100.00% (0.00)	77 / 303.20K	19 / 803	2.56% (12.21)
Vector <8xdbl>	fmul <8xdbl>	D4 / D11	8	8 / 8	100.00% (0.00)	1	1	100.00% (0.00)	20.10B / 169.20K	342.02K / 838	0.03% (0.25)
	fneg <8xdbl>	D4 / D11	8	8 / 1	35.36% (43.75)	1	1	100.00% (0.00)	15.55K / 1	280 / 1	13.42% (49.10)
	fma <8xdbl>	D4 / D11	8	8 / 4	70.71% (25.00)	1	1	100.00% (0.00)	313.97M / 261.97K	40.02K / 810	0.06% (0.15)
Memory	load	D2 / D3 / D12	4,481	1 / 66 / 2	0.11% (0.68)	2,754	1 / 37 / 2	0.15% (0.61)	- / 4.06K / 1	- / 29 / 1	8.45% (49.64)
	store	D2 / D3 / D12 / D13	3,015	1 / 19 / 1 / 1	0.07% (0.26)	1,367	1 / 14 / 1 / 1	0.14% (0.41)	1 / 266 / 1 / 1	1 / 8 / 1 / 1	41.64% (42.00)

Beyond natural diversity, proactive diversity may be necessary to expose certain defects. **Obs. 14** As shown in Fig. 2, D13 is uniquely detected by `MemDiv`; all other ITHICA transformations miss it. Errors are localized to the third validation load that checks an original store (and follows a `clflush`) while the previous two loads yield correct results. One additional server in the QPool is similarly detected exclusively by `MemDiv` (Fig. 3). This suggests that the `clflush` and `mfence` instructions inserted by `MemDiv` perturb the server's execution context between original and validation memory instructions sufficiently to expose an inconsistent error.

These observations show that *natural diversity* (§III-C) from modest interleavings is generally sufficient to expose inconsistencies in arithmetic instructions, possibly because their relevant execution context is largely core-side. *Proactive diversity* (§III-C) appears necessary for certain defects affecting memory instructions, possibly due to their relevant execution context spanning core and uncore components.

Finding 6: Detection of defect-induced inconsistent errors depends on (typically modest) execution context variation between original and validation instructions (Obs. 13, 14).

Implications for Chip-Users: *The key to executing instructions in diverse microarchitectural and electrical state contexts with ISA-level control is varying this context* ***implicitly****, by (a) transforming arbitrary programs into tests, (b) varying and combining ITHICA transformations to alter the surrounding instruction mix within a test and (c) varying interleaving to exploit natural diversity within a test; and* ***explicitly****, by using* `MemDiv` *(or similar) for proactive diversity.*

D. Reported Factors are Insufficient for Defect Detection and Reproducibility

As established in §VII-C, variation in execution context for failing instructions across programs explains the program-sensitivity of defect detection—why different programs executing the same failing instruction do not all detect the same defects. Lacking instruction-level visibility across diverse programs, prior work instead proposes *instruction usage stress*—the dynamic frequency of a failing opcode in a test—as a predictor of detection, claiming detecting tests execute the defective opcode more frequently than non-detecting ones [4], [5].

TABLE VIII
TOTAL AND UNIQUE SERVER DETECTIONS AMONG CC, LIBRARIES AND FB FOR DPOOL & QPOOL SERVERS ACROSS ALL BINARIES

	Top-level CC						Libraries					FB					
Test	**Avx**	**Hasher**	**Pattern_Gen**	**Malign_Buff**	**Silkscreen**	**Utils**	**Zlib**	**OpenSSL**	**Abseil**	**Llvmlibc**	**Llvmlibc++**	**Swissmap**	**Proto**	**Libc**	**Tcmalloc**	**Hashing**	**Stl-cord**
Total	19	3	21	17	6	4	23	9	7	0	3	4	8	3	9	11	7
Unique	15	0	7	3	3	0	4	1	0	0	0	1	3	0	3	1	3

To investigate this claim, we focus on servers uniquely detected by a single test and for which at least one failing opcode also appears in at least one other test besides the detecting one. For each such server, if multiple failing opcodes are shared with the non-detecting tests, we select the most frequently failing one. Fig. 6 shows the normalized execution frequency of this opcode across all tests that execute it. **Obs. 15** In 13 of 22 cases (59%), the detecting test is *not* the one with the highest execution frequency of the failing opcode. This supports that execution context, not merely instruction usage stress, impacts error manifestation and therefore detection. Notably, prior work also uses instruction usage stress for instruction localization, heuristically flagging as suspect those instructions that execute most frequently across failing testcases [4], [5] (Table IX); Obs. 15 equally undermines this inference.

Finding 7: The program-sensitivity of defect detection is not explained by the dynamic frequency of a failing opcode (Obs. 15).

Even for the same program running on the same server, not all runs detect errors: as Fig. 2 shows, detecting `CC-ITHICA` tests have `EDR` less than 100% for most servers. Two factors determine run-to-run error reproducibility: *architectural* (input-

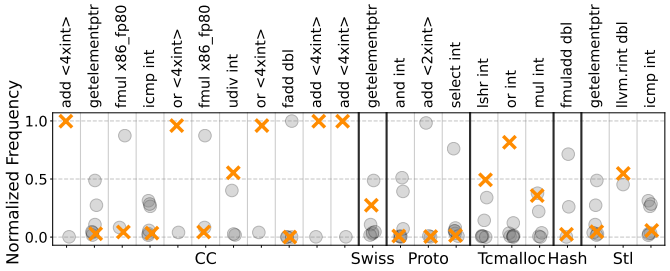
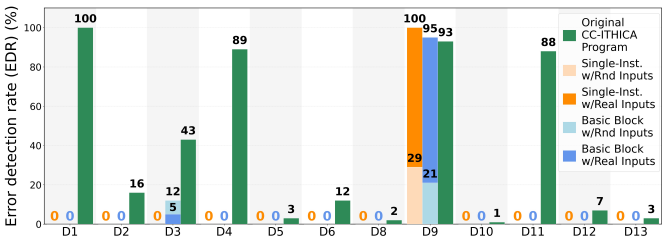


Fig. 6. Normalized execution frequency of failing opcodes for servers (columns) uniquely detected by one program. Orange indicates the detecting program; gray indicates non-detecting programs executing the same opcode.



	D3		D9	
	Instr	BB	Instr	BB
Failing Opcode	None	xor <4xint> / trunc int / zext int	fmul fp80	fmul fp80
Fail BB Length	—	17 / 13.5 / 25.0	—	4.0
Pass BB Length	—	12.0 / 1.0 / 16.2	—	—

Fig. 7. *Top*: EDR of original CC-ITHICA program compared to single-instruction and single-basic-block reproducers (one per failing instruction, averaged). Reproducers are evaluated under both original failing inputs (light bars) and random inputs (dark bars). *Bottom*: For each reproducer, the failing opcodes and lengths of failing vs. non-failing basic blocks.

dependent) and *non-architectural* (non-input-dependent) execution context. In the former case, different runs use different randomly generated inputs, which influence error manifestation (since instructions' inputs are part of their execution context) and detection (e.g., through logical masking), or even control flow (determining which instructions execute at all). In the latter case, microarchitectural and electrical state vary across runs of the same program on the same hardware [6], causing errors to manifest in some runs but not others.

To assess the impact of non-architectural context on the reproducibility of detections across runs, we rerun CC-ITHICA on three servers with high EDR (D3, D4, D7) with fixed inputs across runs, using TTD (Table III) as a proxy for cross-run error reproducibility. Since the program inputs that resulted in detections in earlier experiments are not directly accessible, we use newly-generated random inputs. If the same input, and thus architectural context, repeatedly causes the same error to manifest, TTD should remain constant.

Obs. 16 We find that TTD distributions remain highly variable (e.g., for D3, ranging from seconds to nearly the full hour), demonstrating that non-architectural execution context contributes to error manifestation. Other servers did not exhibit any error detections in this experiment, precluding further analysis.

To assess the length of instruction sequences required for reproducibility of defect-induced errors, we construct two additional types of tests per server: *single-instruction tests* and *basic-block tests*, each derived from original CC-ITHICA programs by isolating a failing instruction or basic block. For each test, we evaluate both real failing inputs (from the original CC-ITHICA runs) and newly-generated random inputs. The basic-block tests range from 1 to 182 instructions in length, with a median of 15 and a mean of 32 instructions. Arith is used to generate tests for all servers except D2/D13, where Mem/MemDiv are used, which uniquely detected them. Each test is run in a loop on each core for **100 hours** (iterating through real or random inputs) and the EDR results are reported in Fig. 7, alongside the EDR of the original CC-ITHICA program from Fig. 2; details for tests with detections are shown at the bottom.

Obs. 17 Typically, neither single-instruction nor basic-block tests detect any errors, even with real failing inputs. There are two exceptions: D3 is not reproduced by single-instruction tests but is by basic-block tests, suggesting that modestly long sequences establish the necessary execution context for its detection. D9, which fails on *fmul x86_fp80*, achieves perfect reproduction with the single-instruction tests with real inputs.

A potential explanation for D9 is reduced microarchitectural execution path non-determinism [81] for its failing LLVM IR instruction. Such non-determinism can arise from multiple sources: compilers may map an LLVM IR instruction to multiple assembly instructions; hardware may map an assembly instruction to multiple micro-ops; and hardware may assign micro-ops to one of multiple functional units. For D9, the first two sources are eliminated and the third is reduced: inspection of the compiled single-instruction LLVM IR reproducer reveals a single assembly instruction, *fmul x86_fp80*; *fmul x86_fp80* maps to a single micro-op per uops.info [82] and the server's microarchitecture documentation (undisclosed); and that micro-op can execute on one of two floating-point units (FPUs). The frequency of *both* original and validation instructions producing incorrect, inconsistent outputs suggests they often interact with the same defective hardware component. Since ITHICA checks detect errors exclusively for *fmul x86_fp80* instructions across the full CC-ITHICA test suite, this component is likely one of the FPUs. However, hardware localization remains inherently opaque at the ISA level (§VII-E). Notably, even with this reduced microarchitectural execution path non-determinism and likely error persistence (§III-A), ITHICA still detects the error, making this case similar to our fault injection example in §III-B.

This experiment, along with the TTD variability observed under fixed program inputs (Obs. 16), suggests that microarchitectural and electrical state, beyond architectural control (§III-C), contribute to defect-induced error manifestation.

Finding 8: Reproduction of defect-induced errors generally cannot be done deterministically with ISA-level control (Obs. 16, 17).

Having just established that supplying the right inputs to the right opcodes is insufficient for reproduction, we further characterize the distribution of instruction-level input values that result in error detections for each opcode-server combination for our full ITHICA programs, using **Input**

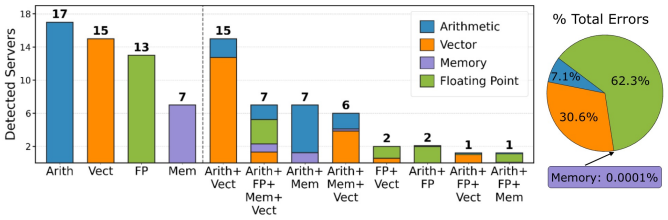


Fig. 8. Combinations of failing instruction types across ITHICA tests. Color segments reflect average per-server error breakdown, along with overall error breakdown (pie chart).

Breadth (Table III). **Obs. 18** Table VII shows that, for most opcodes and servers, the number of unique failing inputs and Input Breadth are high, indicating that errors are largely not confined to a few *culprit* input values.

Finding 9: For vulnerable instructions, selecting the "perfect" input appears to be neither necessary nor sufficient to induce a defect-induced error (Obs. 16–18).

Implications for Chip-Users: *Prior hyperscaler studies use instruction usage stress as a predictor of defect detection and as a basis for instruction localization [4], [5]; others rely on short tests for instruction localization [12]. Our findings contradict both: instruction execution frequency does not reliably predict which tests detect a defective server, and therefore cannot reliably localize failing instructions, and far fewer errors are reproducible with short tests (quantified for our experiments above) than with long ones (Table IX, Conclusion 1).*

E. Hardware Localization from the ISA Level is Opaque

Fig. 4 shows that some DPool servers exhibit errors concentrated in a single instruction type (e.g., D10), while others show errors across types (e.g., D1, D12). Fig. 8 extends this analysis by examining the distribution of errors across instruction types for 93 of all 100 ITHICA-detected servers (89 by `CC-ITHICA` and 11 unique to `FB-ITHICA`); seven are excluded from this analysis due to a crash that prevented the logging of the failing instruction from completing (*no-instr*, §VII-A). For each combination of instruction types, the figure shows the number of servers exhibiting errors for that combination and the average per-server breakdown of number of errors by instruction type. The pie chart aggregates this breakdown across all 93 servers.

Obs. 19 Among all 93 servers analyzed, 44% exhibit errors spanning multiple instruction types, rather than a single type, consistent with prior work observing multi-family test failures [4], [5], [9]; we newly observe this phenomenon at the instruction level and quantify the rate disparity between types. **Obs. 20** Despite accounting for a large share of detected servers, arithmetic and memory instructions contribute only a small fraction of errors: 92.9% originate from floating-point and vector instructions. In particular, floating-point instructions exhibit errors six orders of magnitude more frequently than memory instructions, on average. These observations suggest that floating-point and vector instructions may be easier to reproduce or, equivalently, have reduced microarchitectural execution path non-determinism (§VII-D).

TABLE IX
COMPARISON WITH RELATED HYPERSCALER FLEET STUDIES

		Alibaba [4], [5]	**SEVI** [12]	**ITHICA** *(this work)*
Evaluation	Suspect-Defective Pool Size	Unclear	>2,500	>3,000
	Servers Detected & Analyzed	27–30	18	**100**
Testing Method	Error Assumption	Consistent (implicit)	Consistent (explicit)	Inconsistent (validated on real HW, §VII-A)
	Inst. Types Checked	— (not inst.-level)	Vector only	All (Arith, FP, Vec, Mem, CF)
	Error Check Technique	Final output w/ golden value	Inst.-level w/ golden value	Intra-thread inst.-level
	Instruction Localization	Post-detection via inst. usage stress	Concurrent w/ detection	Concurrent w/ detection
Conclusions	1. Factors Determining Detection	Inst. usage stress, Temperature	Opcode, Input, Temperature	Sequence-driven execution context (§VII-C); Inst. usage stress & opcodes/inputs alone are insufficient (§VII-D)
	2. Vulnerable HW Unit	ALU, FPU, Vec.Unit, Cache, TrxMem	Vector multiplier	Shows prior work conclusions are unreliable (§VII-E)
	3. Solution Proposed	Can focus on vulnerable features	ABFT for matmul kernels	Diverse programs as tests (§VII-C); Inst. localization concurrently with detection (§VII-D)

ITHICA's ability to detect errors across multiple instruction types within the same test illustrates why drawing hardware localization conclusions from tests that check only a narrow set of instructions is problematic. **Obs. 21** For example, the only baseline test that detects D1 is the `Avx` subtest of CC, which is a vector-specialized test. From this test alone, one may blame a vector-specific hardware unit. Yet ITHICA's instruction-level checks within the same test reveal non-vector instructions are also affected, suggesting a different cause. Overall, in our experiments, vector instructions exhibit errors on 46 servers, 31 of which feature errors in other instruction types as well. This challenges prior work's hardware localization conclusions that attribute errors occurring in vector instructions to vector units, using vector-specific tests alone [12].

Finding 10: The same defect often causes errors across instruction types, at markedly different rates (Obs. 19–21).

More generally, while ITHICA's instruction-level visibility enables *scrutinizing* hardware localization conclusions, it cannot definitively *establish* them. This is a limitation of all ISA-level testing approaches [4], [5], [9]–[12], [19], [20], including ITHICA. A defect may reside *anywhere* along an instruction's microarchitectural execution path or in physically adjacent transistors serving unrelated hardware components, none of which is visible at the ISA level. Moreover, the compiler's mapping of LLVM IR to assembly, and the hardware's mapping of assembly to micro-ops and micro-ops to functional units, introduce microarchitectural execution path non-determinism inherent to ISA-level approaches (§VII-D).

Implications for Chip-Users: *Hardware localization is inherently opaque from ISA-level observations. Although ITHICA's instruction-level checking across diverse instructions and programs does not overcome this opacity, it surfaces the full range of LLVM instructions affected on each server, from which prior hardware localization conclusions [4], [5], [12] (Table IX, Conclusion 2) can be scrutinized and, in some cases,*

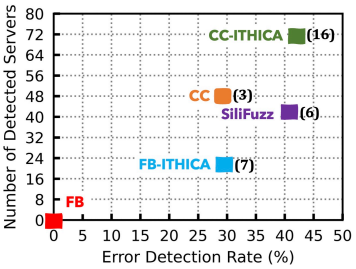


Fig. 9. Comparison of SiliFuzz, CC and ITHICA tests across all commonly tested servers. Unique server detections for each test (missed by all other tests) are shown in parentheses.

discredited. Incorrect hardware localization conclusions risk guiding future software and hardware mitigations toward the wrong or insufficient components (Table IX, Conclusion 3).

VIII. Comparison with SiliFuzz

Fig. 9 compares `CC`, `CC-ITHICA`, and `FB-ITHICA` with SiliFuzz [19], [83], the only fuzzing tool for defects that does not rely on below-ISA specifications (§X). The tests are compared by number of total and unique detected servers and average `EDR`. `TTD` and `EF` statistics are not available for SiliFuzz and are therefore omitted. Similar to §VI-B, we exclude servers with less than 20 execution hours per test. For each remaining server, we use an equal number of one-hour executions per test, set to the minimum completed by any single test on that server. SiliFuzz detects significantly fewer servers than `CC-ITHICA` (42 vs. 71). For the servers it detects, its average `EDR` is smaller than but comparable to `CC-ITHICA` (40.8% vs. 42.1%). Notably, `FB-ITHICA` detects seven servers missed by all other tests.

IX. Limitations

This section discusses limitations of our fleet study in general and ITHICA tests in particular.

Server Pool Bias. The servers evaluated in our study represent a biased sample of Google's overall fleet. Our results in §VII–§VIII are derived from the QPool and DPool, which comprise suspect- and confirmed-defective servers, respectively (§V-B). Because these out-of-production pools are populated using non-ITHICA screening mechanisms, they inherently over-represent servers with defects that those existing tests can flag. This selection bias likely explains why ITHICA's `Arith` checks dominate our experimental results (§VII-A), given that many standard screening tests explicitly stress arithmetic operations [10], [19]. We note, however, that this bias is standard across the literature: prior hyperscaler fleet studies [2], [4], [5], [9], [12] evaluate similarly-populated pools, suggesting our study examines a comparable server population.

Missing Defect-Induced Errors. ITHICA tests can miss defect-induced errors—e.g., those on the six servers in Fig. 3 that `ITHICA` checks miss but `Native` checks detect—because of instrumentation gaps or because they manifest consistently.

Instrumentation Gaps: Even at maximum instrumentation (block size of one), ITHICA does not instrument, and thus does not check, every LLVM IR instruction executed in the program it transforms into a test. For example: (1) instructions with side-effects (i.e., that are not safe to duplicate), including atomic and volatile memory instructions (§IV-C), (2) memory instructions in thread-unsafe code (§IV-C), (3) uninstrumented parts of `Libc/Libc++` (§V-A), and (4) kernel code executed as part of system calls. Gaps (1)–(3) apply only to `ITHICA` checks, while (4) is shared between `ITHICA` and `Native` checks. Despite (2), ITHICA still instruments most (non-memory) opcodes in thread-unsafe programs, which are difficult to check with prior error checking techniques (§I).

Consistent Errors: By design, `ITHICA` checks detect inconsistent errors only, not consistent ones; the `Native` checks in `CC/CC-ITHICA` can in principle detect both. ITHICA's focus stems from our insight that consistent errors are theoretically easier to detect, since they manifest from every execution context, whereas inconsistent errors manifest from only select ones (§III-A). We show that it pays off empirically: `ITHICA` checks detect 39% more defective servers than `Native` checks within the same test programs, and `ITHICA` checks in non-test workloads detect additional unique defective servers.

Heuristic Coverage. Finally, ITHICA tests can fail to induce an error on a defective server due to ITHICA neither measuring nor controlling *hardware coverage*—the fraction of distinct microarchitectural and electrical state contexts (§III) and thus potential fault sites. As a result, its tests may never exercise a server's defect site in a context where the defect manifests. Today, no metric is known to accurately measure hardware coverage of functional tests for defects: existing metrics—structural coverage [54]–[61], ACE-based [20], [51], or delay-fault [47] vulnerability—all score tests against fault models that have not been shown to accurately represent defects [6], [14]–[16].

Our evaluation therefore reports detection outcomes (Table III) instead, as do all prior fleet studies [1], [2], [4], [5], [9], [12]. Only two prior techniques for defects define and optimize a coverage metric, but neither is shown to predict detection outcomes on real defective hardware. SiliFuzz [19] maximizes control-flow coverage of a software CPU emulator as a proxy for hardware coverage, and acknowledges the two do not necessarily correspond; empirically, ITHICA detects 69% more servers than SiliFuzz (§VIII). Harpocrates [20] defines its own coverage metric for defects, but evaluates its correlation with detection outcomes (i.e., errors detected) using stuck-at fault injection rather than real defective hardware. Therefore, like other functional tests, ITHICA tests cannot *systematically* improve coverage of defect fault sites; instead, they *heuristically* improve it through the test-diversity levers of §VII-C, validated on real defective hardware.

X. Related Work

Prior works most relevant to ITHICA fall into four categories: soft error detection, post-silicon validation, fuzzing, and hyperscaler fleet studies of the CPU SDC problem. Table X compares ITHICA to existing functional test suites [10], [11] and test generation techniques [19], [20] for defects.

Soft Error Detection. EDDI [23] is the first to propose intra-thread instruction checks via instruction duplication and output comparison. CFCSS [24] is the first to introduce intra-thread checks for branches; it does so by validating runtime control-flow against compile-time control-flow graphs. SWIFT [25]

reduces EDDI's performance overhead by omitting memory instruction protection (relying on ECC [84]) and by reusing idle processor resources for duplicate computation, and improves CFCSS's control-flow coverage. Several variations of the above techniques propose further performance or coverage improvements, either purely in software [67], [72]–[77], [85] or with architectural support [86], [87]. Analogous software-only [88]–[90] and architecture-supported [89], [91] techniques have been proposed for GPUs.

Unlike prior soft error detection work, whose goal is application protection during in-production execution, ITHICA's goal is *eventual* defect detection during out-of-production testing. This distinction affects how error checks are inserted—for which instructions (sampling in space) and how frequently (sampling in time)—and the relevance of non-check program-diversity levers (§VII-C), as described below.

Error Check Insertion: Sampling in time is applicable to ITHICA but not soft error detection. The former requires eventual defect detection, so it can skip checks (e.g., by increasing block size, §VII-B); the latter must check every instruction that can affect a program's output to guarantee fault tolerance. Sampling in space is applicable to both, but relevant mechanisms for achieving it differ. For example, soft error detection can exploit logical masking to prune error checks for instructions deemed incapable—either by static program analysis [76], [85] or by characterization on known inputs [74], [75]—of causing an error at a program's output. Such pruning is inapplicable to ITHICA, since logical masking harms its defect detection capabilities (§IV-C, §VII-B).

Diversity Levers: ITHICA's non-check levers (§VII-C) diversify how a program exercises the hardware, heuristically improving defect detection. These levers are irrelevant to soft error protection, which addresses transient rather than permanent faults and seeks to prevent such errors altogether.

Post-Silicon Validation. QED [26]–[29] adapts its intra-thread instruction checks from soft error detection to detect logic and electrical design bugs during post-silicon validation. Thus, QED is intended to be run on relatively few (hundreds [2]) silicon samples before mass production. Per §IV-A, ITHICA further adapts these checks for defect detection and demonstrates their effectiveness in this new setting *for the first time*. In contrast to QED, ITHICA is intended to be used on millions of deployed servers during in-datacenter testing. These differences in fault scopes and deployment scenarios between QED and ITHICA lead them to adopt different program transformations and make different implementation choices.

Program transformations: Driven by our insight that execution context diversity is important for defect detection (§III-C), ITHICA uses several levers to increase it, including the novel `MemDiv` transformation, which detects unique servers in our experiments (§VII-C). It is possible for `MemDiv` to benefit post-silicon validation, though it has not been evaluated for that purpose. Additionally, QED implementations of control-flow checking transformations can leverage known test inputs—a reasonable assumption for post-silicon validation. ITHICA's `Br` cannot, and does not, as it operates on arbitrary programs.

TABLE X
TESTS AND TEST GENERATION TECHNIQUES FOR DEFECTS

Technique	Check Type				Evaluation		Spec
	Golden Value	Cross-Core	Invert. Comp.	Intra-thr. Instruct.	Fault Inject.	Real HW	
cpu-check [10]	✗	✓	✓	✗	–	–	C/C++
OpenDCDiag [11]	✓	✓	✓	✗	✗	✓	C/C++
SiliFuzz [19]	✓	✓	✗	✗	–	–	ASM
Harpocrates [20]	✓	✓	✗	✗	✓	✗	Uarch
ITHICA *(this work)*	✗	✗	✗	✓	✗	✓	LLVM

Implementation choices: QED leaves library code uninstrumented, due to its source-code-level implementation. ITHICA's LLVM IR passes, by comparison, can instrument any code compiled through LLVM, including libraries—where ITHICA detects unique servers in our experiments (§VII-C).

Fuzzing. A large number of fuzzing works exist, targeting defects, design bugs, or security bugs [19], [20], [92]–[96]; several of them rely on below-ISA specifications [20], [92]–[94]. Closest in motivation to ITHICA is PathFuzz [92], which uses real-program checkpoints as fuzzer seeds for pre-silicon validation, but, unlike ITHICA, requires deterministic seeds, restricting usable programs. More broadly, ITHICA checks could be applied to fuzzer-generated sequences to replace the golden outputs they require [19], [20], [92]–[94].

Hyperscaler Fleet Studies. Like ITHICA, the Alibaba studies [4], [5] and SEVI [12] evaluate their techniques using pools of suspect-defective servers. Table IX compares their testing methods and findings with ITHICA's. Other works quantify the prevalence and report detection trends of hardware errors across large CPU fleets [1]–[3], [6], [9], [22] or study SDC vulnerability through fault injection [97]–[100]. Among them, PinDrop [9] advocates for continuous testing with a variety of complex tests at scale—a philosophy that aligns with ITHICA's goal of enabling arbitrary programs to be used as tests. Mitra et al. [6] finds that a significant fraction of defects are detected by production workloads in the field, motivating the instrumentation of such workloads with ITHICA.

XI. CONCLUSION

We present ITHICA, an approach for automatically generating functional tests from arbitrary programs to detect defective CPUs. ITHICA exploits the insight that the most pernicious defects manifest as inconsistent errors—validated here for the first time on real hardware—to build intra-thread, instruction-level checks that outperform baseline industry checks and enable novel findings on defect behavior that challenge conclusions of prior hyperscaler studies.

ACKNOWLEDGMENT

This work was supported in part by the U.S. National Science Foundation under award 2321489 and a Sloan Research Fellowship. We also gratefully acknowledge gifts from Google, Meta, and the Open Compute Project. Finally, we thank Peter Hochschild, David Culler, Parthasarathy Ranganathan, Liqun Cheng, Martin Dixon, Konstantin Shtoyk, and Rama Govindaraju for their continued support.

REFERENCES

[1] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, "Silent data corruptions at scale," *CoRR*, vol. abs/2102.11245, 2021, https://arxiv.org/abs/2102.11245.

[2] H. D. Dixit, L. Boyle, G. Vunnam, S. Pendharkar, M. Beadon, and S. Sankar, "Detecting silent data corruptions in the wild," 2022. [Online]. Available: https://arxiv.org/abs/2203.08989

[3] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021.

[4] S. Wang, G. Zhang, J. Wei, Y. Wang, J. Wu, and Q. Luo, "Understanding silent data corruptions in a large production cpu population," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

[5] ——, "Understanding silent data corruption in processors for mitigating its effects," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 4, Nov. 2024. [Online]. Available: https://doi.org/10.1145/3690825

[6] S. Mitra, S. Banerjee, M. Dixon, R. Govindaraju, P. Hochschild, E. X. Liu, B. Parthasarathy, and P. Ranganathan, "Silent data corruption by 10x test escapes threatens reliable computing," 2025. [Online]. Available: https://arxiv.org/abs/2508.01786

[7] N. George, S. Gurumurthi, V. Sridharan, H. D. Dixit, E. Goksu, B. Parthasarathy, A. Huffman, T. Macieira, A. Sinha, D. Liberty, L. Minwell, and R. S. Chappell, "Silent data corruption in artificial intelligence: A growing challenge for large-scale machine learning," *IEEE Micro*, vol. 46, no. 1, pp. 66–72, 2026.

[8] C. Constantinescu, I. Parulkar, R. Harper, and S. Michalak, "Silent data corruption — myth or reality?" in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008, pp. 108–109.

[9] P. W. Deutsch, H. D. Dixit, G. Vunnam, C. Moran, E. Ozer, and S. Sankar, "Pindrop: Breaking the silence on sdcs in a large-scale fleet," in *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2026, pp. 1–14.

[10] Google, "Google cpu-check torture test," https://github.com/google/cpu-check, 2020.

[11] Intel, "Opendcdiag," 2021, https://github.com/opendcdiag.

[12] Y. Mei, S. Varshini, H. Dixit, S. Sankar, and K. V. Rashmi, "Sevi: Silent data corruption of vector instructions in hyper-scale datacenters," in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '26. New York, NY, USA: Association for Computing Machinery, 2026, p. 1711–1726. [Online]. Available: https://doi.org/10.1145/3779212.3790217

[13] J. Ma, M. Ganaiem, M. Burbage, T. Gregersen, R. McAmis, F. Gabbay, and B. Kasikci, "Proactive runtime detection of aging-related silent data corruptions: A bottom-up approach," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2025, p. 220–235. [Online]. Available: https://doi.org/10.1145/3622781.3674182

[14] E. McCluskey and C.-W. Tseng, "Stuck-fault tests vs. actual defects," in *Proceedings International Test Conference 2000 (IEEE Cat. No.00CH37159)*, 2000.

[15] W. Li, C. Nigh, D. Duvalsaint, S. Mitra, and R. D. Blanton, "Pepr: Pseudo-exhaustive physically-aware region testing," in *International Test Conference*, 2022.

[16] D. Lerner, E. Hansen, S. Sakthivelu, C. Carrizo, S. Tzu-Lin, T. Macieira, and B. Kelly, "Screening for manufacturing defects that manifest as silent data errors in data center processors," in *2025 IEEE International Electron Devices Meeting (IEDM)*, 2025, pp. 1–4.

[17] M. Prabhu and J. A. Abraham, "Functional test generation for hard to detect stuck-at faults using rtl model checking," in *2012 17th IEEE European Test Symposium (ETS)*, 2012.

[18] S. Kundu, S. Sengupta, and R. Galivanche, "Test challenges in nanometer technologies," in *Proceedings IEEE European Test Workshop*, 2000.

[19] K. Serebryany, M. Lifantsev, K. Shtoyk, D. Kwan, and P. Hochschild, "Silifuzz: Fuzzing cpus by proxy," 2021. [Online]. Available: https://arxiv.org/abs/2110.11519

[20] N. Karystinos, O. Chatzopoulos, G.-M. Fragkoulis, G. Papadimitriou, D. Gizopoulos, and S. Gurumurthi, "Harpocrates: Breaking the silence of cpu faults through hardware-in-the-loop program generation," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 516–531.

[21] N. Karystinos, G.-M. Fragkoulis, O. Chatzopoulos, D. Gizopoulos, and S. Gurumurthi, "Harpocrates++: Automated functional program generation against cpu faults and silent data corruptions," *IEEE Micro*, vol. 46, no. 1, pp. 19–28, 2026.

[22] R. Dutta, H. D. Dixit, R. Van Riel, G. Vunnam, and S. Sankar, "Hardware sentinel: Protecting software applications from hardware silent data corruptions," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 482–497. [Online]. Available: https://doi.org/10.1145/3676641.3716258

[23] N. Oh, P. Shirvani, and E. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[24] ——, "Control-flow checking by software signatures," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

[25] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "Swift: software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, 2005, pp. 243–254.

[26] T. Hong, Y. Li, S.-B. Park, D. Mui, D. Lin, Z. A. Kaleq, N. Hakim, H. Naeimi, D. S. Gardner, and S. Mitra, "Qed: Quick error detection tests for effective post-silicon validation," in *2010 IEEE International Test Conference*, 2010.

[27] D. Lin, T. Hong, Y. Li, F. Fallah, D. S. Gardner, N. Hakim, and S. Mitra, "Overcoming post-silicon validation challenges through quick error detection (qed)," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013, pp. 320–325.

[28] E. Singh, C. Barrett, and S. Mitra, "E-qed: electrical bug localization during post-silicon validation enabled by quick error detection and formal methods," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 104–125.

[29] D. Lin, T. Hong, Y. Li, E. S, S. Kumar, F. Fallah, N. Hakim, D. S. Gardner, and S. Mitra, "Effective post-silicon validation of system-on-chips using quick error detection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 10, pp. 1573–1590, 2014.

[30] Google, "Fleetbench," 2022, https://github.com/google/fleetbench.

[31] "Zlib," 2024, https://zlib.net/.

[32] "Openssl," 2024, https://github.com/openssl/openssl.

[33] "Abseil," 2024, https://github.com/abseil/abseil-cpp.

[34] "The llvm c library," 2024, https://libc.llvm.org/.

[35] "Llvm libc++," 2024, https://github.com/llvm/llvm-project/blob/main/libcxx.

[36] M. Sauer, Y. M. Kim, J. Seomun, H.-O. Kim, K.-T. Do, J. Y. Choi, K. S. Kim, S. Mitra, and B. Becker, "Early-life-failure detection using sat-based atpg," in *2013 IEEE International Test Conference (ITC)*, 2013, pp. 1–10.

[37] T. W. Chen, K. Kim, Y. M. Kim, and S. Mitra, "Gate-oxide early life failure prediction," in *26th IEEE VLSI Test Symposium (vts 2008)*, 2008, pp. 111–118.

[38] B. C. Paul, K. Kang, H. Kufluoglu, M. A. Alam, and K. Roy, "Negative bias temperature instability: Estimation and design for improved reliability of nanoscale circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 4, pp. 743–751, 2007.

[39] E. Takeda and N. Suzuki, "An empirical model for device degradation due to hot-carrier injection," *IEEE Electron Device Letters*, vol. 4, no. 4, pp. 111–113, 1983.

[40] M. Agarwal, B. C. Paul, M. Zhang, and S. Mitra, "Circuit failure prediction and its application to transistor aging," in *25th IEEE VLSI Test Symposium (VTS'07)*, 2007, pp. 277–286.

[41] P. Patra, "On the cusp of a validation wall," *IEEE Design & Test of Computers*, vol. 24, no. 2, pp. 193–196, 2007.

[42] J. F. Ziegler and W. A. Lanford, "Effect of cosmic rays on computer memories," *Science*, vol. 206, no. 4420, pp. 776–788, 1979.

[43] T. C. May and M. H. Woods, "A new physical mechanism for soft errors in dynamic memories," in *16th International Reliability Physics Symposium*, 1978, pp. 33–40.

[44] S. Mukherjee, J. Emer, and S. Reinhardt, "The soft error problem: an architectural perspective," in *11th International Symposium on High-Performance Computer Architecture*, 2005.

[45] J.-M. Li and E. McCluskey, "Diagnosis of sequence-dependent chips," in *Proceedings 20th IEEE VLSI Test Symposium (VTS 2002)*, 2002, pp. 187–192.

[46] J. T.-Y. Chang, C.-W. Tseng, C.-M. J. Li, M. Purtell, and E. J. McCluskey, "Analysis of pattern-dependent and timing-dependent failures in an experimental test chip," *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*, pp. 184–193, 1998. [Online]. Available: https://api.semanticscholar.org/CorpusID:16286356

[47] P. W. Deutsch, V. Q. Ulitzsch, S. Gurumurthi, V. Sridharan, J. S. Emer, and M. Yan, "Delayavf: Calculating architectural vulnerability factors for delay faults," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 231–245.

[48] D. Sorin, *Fault tolerant computer architecture*. Morgan & Claypool Publishers, 2009.

[49] S. Mukherjee, *Architecture design for soft errors*. Morgan Kaufmann, 2011.

[50] G. Papadimitriou and D. Gizopoulos, "Avgi: Microarchitecture-driven, fast and accurate vulnerability assessment," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 935–948.

[51] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003, pp. 29–40.

[52] E. B. Eichelberger and T. W. Williams, "A logic design structure for lsi testability," in *Papers on Twenty-Five Years of Electronic Design Automation*, ser. 25 years of DAC. New York, NY, USA: Association for Computing Machinery, 1988, p. 358–364. [Online]. Available: https://doi.org/10.1145/62882.62924

[53] J. Shen and J. A. Abraham, "Native mode functional test generation for processors with applications to self test and design validation," *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*, pp. 990–999, 1998. [Online]. Available: https://api.semanticscholar.org/CorpusID:14132281

[54] R. L. Wadsack, "Fault modeling and logic simulation of cmos and mos integrated circuits," *The Bell System Technical Journal*, vol. 57, no. 5, pp. 1449–1474, 1978.

[55] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar, "Transition fault simulation," *IEEE Design & Test of Computers*, vol. 4, no. 2, pp. 32–38, 1987.

[56] G. L. Smith, "Model for delay faults based upon paths," in *International Test Conference*, 1985.

[57] K. Heragu, J. Patel, and V. Agrawal, "Segment delay faults: a new fault model," in *Proceedings of 14th VLSI Test Symposium*, 1996.

[58] T. Storey and W. Maly, "Cmos bridging fault detection," in *Proceedings. International Test Conference 1990*, 1990.

[59] S. Ma, P. Franco, and E. McCluskey, "An experimental chip to evaluate test techniques experiment results," in *Proceedings of 1995 IEEE International Test Conference (ITC)*, 1995.

[60] F. Hapke, R. Krenz-Baath, A. Glowatz, J. Schloeffel, H. Hashempour, S. Eichenberger, C. Hora, and D. Adolfsson, "Defect-oriented cell-aware atpg and fault simulation for industrial cell libraries and designs," in *2009 International Test Conference*, 2009.

[61] E. McCluskey, "Quality and single-stuck faults," in *Proceedings of IEEE International Test Conference - (ITC)*, 1993.

[62] A. Vassighi, R. Kacprowicz, C. Carranza, and W. Riordan, "Characterizing infant mortality in high volume manufacturing," in *2008 IEEE International Reliability Physics Symposium*, 2008, pp. 717–718.

[63] F. Lonsing, S. Mitra, and C. W. Barrett, "A theoretical framework for symbolic quick error detection," in *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*. IEEE, 2020, pp. 1–10. [Online]. Available: https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_9

[64] S. Chattopadhyay, K. Devarajegowda, B. Zhao, F. Lonsing, B. A. D'Agostino, I. Vavelidou, V. D. Bhatt, S. Prebeck, W. Ecker, C. Trippel, C. Barrett, and S. Mitra, "G-qed: Generalized qed pre-silicon verification beyond non-interfering hardware accelerators," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.

[65] OpenHW Group, "CVA6 RISC-V CPU," 2019, https://github.com/openhwgroup/cva6.

[66] W. Snyder, P. Wasson, D. Galbi, G. Lore *et al.*, "Verilator: Open-source SystemVerilog simulator and lint system," 2003–2026. [Online]. Available: https://verilator.org

[67] M. Didehban and A. Shrivastava, "nzdc: A compiler technique for near zero silent data corruption," in *2016 53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[68] Z. He, Y. Huang, H. Xu, D. Tao, and G. Li, "Demystifying and mitigating cross-layer deficiencies of soft error protection in instruction duplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607078

[69] O. Chatzopoulos, G. Papadimitriou, D. Gizopoulos, H. D. Dixit, and S. Sankar, "From gates to sdcs: Understanding fault propagation through the compute stack," in *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2025, pp. 1–7.

[70] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 415–426.

[71] D. Lin, T. Hong, F. Fallah, N. Hakim, and S. Mitra, "Quick detection of difficult bugs for effective post-silicon validation," in *DAC Design Automation Conference 2012*, 2012, pp. 561–566.

[72] R. Venkatasubramanian, J. Hayes, and B. Murray, "Low-cost on-line fault detection using control flow assertions," in *9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003.*, 2003, pp. 137–143.

[73] Y. Huang, S. Guo, S. Di, G. Li, and F. Cappello, "Mitigating silent data corruptions in hpc applications across multiple program inputs," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022, pp. 1–14.

[74] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson, "Ipas: intelligent protection against silent output corruption in scientific applications," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 227–238. [Online]. Available: https://doi.org/10.1145/2854038.2854059

[75] Q. Lu, K. Pattabiraman, M. S. Gupta, and J. A. Rivers, "Sdctune: A model for predicting the sdc proneness of an application for configurable protection," in *Proceedings of the 2014 international conference on compilers, architecture and synthesis for embedded systems*, 2014, pp. 1–10.

[76] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," *ACM SIGPLAN Notices*, vol. 45, p. 385, 03 2010.

[77] Z. He, H. Xu, and G. Li, "A fast low-level error detection technique," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2024, pp. 90–98.

[78] "Llvm language reference manual," https://llvm.org/docs/LangRef.html, 2022, accessed: 2022-10-19.

[79] "OpenSSL Debian Manpages," https://manpages.debian.org/testing/libssl-doc/OPENSSL_LH_doall_arg.3ssl.en.html.

[80] A. Abel, Y. Li, R. O'Grady, C. Kennelly, and D. Gove, "A profiling-based benchmark suite for warehouse-scale computers," in *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2024, pp. 325–327.

[81] Y. Hsiao, N. Nikoleris, A. Khyzha, D. P. Mulligan, G. Petri, C. W. Fletcher, and C. Trippel, "Rtl2mμpath: Multi-μpath synthesis with applications to hardware security verification," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 507–524.

[82] A. Abel and J. Reineke, "uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures," in *ASPLOS*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 673–686. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304062

[83] Google, "Silifuzz," 2021, https://github.com/google/silifuzz.

[84] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[85] J. Yu, M. J. Garzaran, and M. Snir, "Esoftcheck: Removal of non-vital checks for fault tolerance," in *2009 International Symposium on Code Generation and Optimization*. IEEE, 2009, pp. 35–46.

[86] R. Rangan, S. S. Mukherjee, J. Chang, D. I. August, N. Vachharajani, and G. A. Reis, "Design and Evaluation of Hybrid Fault-Detection Systems," in *Proceedings. 32nd International Symposium on Computer Architecture*. Los Alamitos, CA, USA: IEEE Computer Society, Jun. 2005, pp. 148–159. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ISCA.2005.21

[87] J. Ohlsson and M. Rimen, "Implicit signature checking," in *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, 1995, pp. 218–227.

[88] C. Kalra, F. Previlon, N. Rubin, and D. Kaeli, "Armorall: Compiler-based resilience targeting gpu applications," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 2, May 2020. [Online]. Available: https://doi.org/10.1145/3382132

[89] A. Mahmoud, S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing software-directed instruction replication for gpu error detection," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 842–854.

[90] X. Wei, N. Jiang, H. Yue, X. Wang, J. Zhao, G. Li, and M. Qiu, "Approxdup: Developing an approximate instruction duplication mechanism for efficient sdc detection in gpgpus," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 4, pp. 1051–1064, 2024.

[91] M. B. Sullivan, S. K. S. Hari, B. Zimmer, T. Tsai, and S. W. Keckler, "Swapcodes: Error codes for hardware-software cooperative gpu pipeline error detection," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 762–774.

[92] Y. Xu, S. Wang, D. Tang, N. Sun, and Y. Bao, "Pathfuzz: Broadening fuzzing horizons with footprint memory for cpus," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3649329.3655911

[93] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "Difuzzrtl: Differential fuzz testing to find cpu bugs," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1286–1303.

[94] R. Kande, A. Crump, G. Persyn, P. Jauernig, A.-R. Sadeghi, A. Tyagi, and J. Rajendran, "TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3219–3236. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/kande

[95] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3237–3254. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/trippel

[96] A. Konovalov, "Coverage-guided usb fuzzing with syzkaller," 2019, https://www.offensivecon.org/speakers/2019/andrey-konovalov.html.

[97] O. Chatzopoulos, N. Karystinos, G. Papadimitriou, D. Gizopoulos, H. D. Dixit, and S. Sankar, "Veritas - demystifying silent data corruptions: uarch-level modeling and fleet data of modern x86 cpus," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2025, pp. 1–14.

[98] G. Papadimitriou and D. Gizopoulos, "Silent data corruptions: Microarchitectural perspectives," *IEEE Transactions on Computers*, vol. 72, no. 11, pp. 3072–3085, 2023.

[99] D. Gizopoulos, G. Papadimitriou, O. Chatzopoulos, N. Karystinos, H. D. Dixit, and S. Sankar, "Silent data corruptions in computing systems: Early predictions and large-scale measurements," in *2024 IEEE European Test Symposium (ETS)*, 2024, pp. 1–10.

[100] G. Papadimitriou, D. Gizopoulos, H. D. Dixit, and S. Sankar, "Silent data corruptions: The stealthy saboteurs of digital integrity," *2023 IEEE 29th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pp. 1–7, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:261315246

APPENDIX

A. Abstract

This artifact includes the ITHICA LLVM IR passes (`Arith`, `Mem`, `MemDiv`, `Br` and their combinations) used to transform real programs into tests for hardware defects (first contribution in §I). It demonstrates that ITHICA passes successfully instrument the two programs used in the paper: cpu-check (`CC`) and Fleetbench (`FB`). For ease of use, the Zenodo artifact provides a pinned Docker build environment per program (kept separate due to disjoint build requirements), containing all dependencies needed to compile both programs (with and without ITHICA) from source. The artifact reproduces the runtime performance overhead and binary size increase that ITHICA transformations impose relative to the uninstrumented `CC` baseline, equivalent to Table VI. Pre-built binaries for all configurations of Table VI are provided to reproduce the measurements directly, even without recompilation. The remaining results (§VII–§VIII) require access to proprietary defective hardware owned by Google, and therefore cannot be reproduced.

B. Artifact check-list (meta-information)

- **Transformations:** Code for ITHICA's LLVM IR compiler passes, `Arith`, `Mem`, `MemDiv`, and `Br`, and their combinations `Arith+Mem`, `Arith+MemDiv` and `Arith+MemDiv+Br`.
- **Program:** The input programs compiled with ITHICA: cpu-check and Fleetbench. Both are open-source and shipped with the artifact.
- **Compilation:** No dependencies beyond Docker. Clang/LLVM/lld 14.0.6 and Bazel 7.3.1 are used within the two Docker images.
- **Binary:** Pre-built, statically linked x86-64 binaries available.
- **Hardware:** Required: x86-64 Linux machine with at least 16 GB of RAM and 40 GB of free disk space on the Docker filesystem. Preferred: 32 cores or more and hyperthreading enabled. (Reference host: Intel Xeon Gold 6226R; 2×16 cores, 2 threads/core, 500 GB RAM).
- **Runtime Environment:** Docker and Bash. Network access.
- **Metrics:** Runtime performance overhead and binary size increase relative to the baseline.
- **Experiments:** Two experiments (§F): E1 reproduces the results of Table VI—it measures per-ITHICA-pass overhead and binary-size increase on `CC`, including parameter sweeps for `Arith`; E2 measures the same metrics on `FB` with and without `Arith` (not in the main paper). Separately, this artifact allows rebuilding both programs from source.
- **How much disk space required (approximately)?:** 40 GB.
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes for Docker environment setup. To recompile all binaries from source: additional 8 hours.
- **How much time is needed to complete experiments (approximately)?:** 65 minutes total to execute all binaries.
- **Publicly available?:** https://github.com/ioannavav/ithica
- **Code licenses (if publicly available)?:** MIT License.
- **Workflow automation framework used?:** Docker.
- **Archived (DOI)?:** https://doi.org/10.5281/zenodo.21554153

C. Description

1) How to access: The artifact is archived on https://doi.org/10.5281/zenodo.21554153 as a complete, ready-to-run copy (pre-built binaries and toolchain image included). The source code is also on https://github.com/ioannavav/ithica.

The artifact contains two directories, `cpu-check-ae/` and `fleetbench-ae/`, one per program. Each holds the ITHICA pass sources, the pinned program source, the pinned toolchain, pre-built binaries (to avoid heavy recompilation of some programs if needed), and three driver scripts: `setup.sh` provisions the toolchain, `build.sh` compiles the program with a chosen ITHICA pass, and `measure.sh` runs the (resulting or pre-built) binaries and reports the metrics.

2) Hardware dependencies: x86-64 Linux machine with 40 GB of free disk space and 16 GB RAM. To reproduce the reported results closely, a host with at least 32 physical / 64 logical CPUs is recommended. Reference host: Intel Xeon Gold 6226R: 2 sockets × 16 cores *per socket* = 32 physical cores, and 2 threads per core = 64 logical CPUs; 500 GB RAM. The results obtained are expected to vary depending on the underlying hardware (§H).

3) Software dependencies: Only Docker (the cpu-check toolchain is exported from a pinned GNU Guix environment via `guix pack` and the Fleetbench image is built from a Dockerfile). Network access is required while building either image. On hosts where Docker's bridge network cannot reach GitHub, add `--network=host` to `docker build`.

D. Installation

Use the Zenodo copy for the installation steps below and results reproduction using pre-built binaries. The build steps in §F for all programs use these images:

- *For cpu-check:*

```
cd cpu-check-ae
scripts/setup.sh   # docker load (shipped image)
```

- *For Fleetbench:*

```
cd fleetbench-ae
scripts/setup.sh   # docker build (Dockerfile)
```

E. Experiment workflow

The evaluation first compiles the two programs with ITHICA; `build.sh` builds the target pass and compiles the program in one step; `build_pass.sh` builds a pass separately. Two script-driven experiments then measure the resulting binaries: for cpu-check, E1 measures the per-pass overhead and binary size increase compared to the uninstrumented baseline (main result, reproducing Table VI), and E2 measures the same two metrics for Fleetbench with and without `Arith` (with block size and interleaving of 1). A toy program (`toy_overhead/toy_program.cpp`) ships in the cpu-check image so users can quickly instrument and run it (`cpu-check-ae/scripts/toy_overhead.sh`).

F. Evaluation and expected results

The central claim reproduced by this artifact is the per-pass **runtime performance overhead** and **binary size increase** of the ITHICA transformations relative to the uninstrumented `CC` baseline, i.e., the results of Table VI and Obs. 8 in the paper. The first goal of the evaluation is to successfully use the ITHICA passes to compile the two programs; the measurement experiments then run the resulting binaries. Expected values on the reference host are given in Tables XI and XII; absolute values vary with the underlying hardware (§C2).

TABLE XI
RESULTS EQUIVALENT TO TABLE VI, HERE MEASURED ON A NON-INDUSTRIAL (NON-PROPRIETARY) SERVER: PERFORMANCE OVERHEAD AND BINARY SIZE INCREASE (IN ×) OF ITHICA COMPARED TO CC

ITHICA Pass	Arith (Block Size) (1)	(2)	(4)	(8)	(dep)	Mem	MemDiv	Br
Performance	2.18	1.79	1.64	1.52	1.75	1.17	53.67	1.17
Binary Size	8.47	5.88	4.59	4.11	6.11	3.06	13.12	1.24

Server: Intel Xeon Gold 6226R, 2.9GHz, 2×16 cores, 2 threads/core, hyperthreading on, L1: 1MB D+I, L2: 32MB, L3: 44MB, Mem: 500 GB

TABLE XII
PERFORMANCE OVERHEAD AND BINARY SIZE INCREASE (IN ×) OF FB-ARITH COMPARED TO THE UNINSTRUMENTED FB BASELINE MEASURED ON THE REFERENCE HOST OF TABLE XI.

Benchmark	Sub-benchmark	Performance	Binary Size
hashing	ComputeCrc32c (hot)	1.42	1.11
proto	Arena	1.44	2.87
swissmap	InsertHit (cold)	2.99	2.31
stl	Cord	2.79	5.81
libc	Memcpy	2.21	1.09
tcmalloc	empirical_driver	1.82	1.18

Getting started: `cpu-check-ae/scripts/build_pass.sh` builds the ITHICA pass, e.g., `arith|mem|memdiv|br|arithmem` etc.:

```
cd cpu-check-ae
scripts/build_pass.sh arith
```

Building the two programs. `build.sh` compiles each program with the chosen ITHICA pass. For cpu-check:

```
cd cpu-check-ae
scripts/build.sh --pass=CC      # baseline, no ITHICA
scripts/build.sh --pass=arith   # ITHICA pass
# knobs: --blocksize=1|2|4|8|0 --interleaving=1|2|4|8|0
# 0 used for "dep" (blocksize) and "max" (interleaving)
```

For Fleetbench:

```
cd fleetbench-ae
scripts/build.sh --bench=hashing --nopass      # baseline
scripts/build.sh --bench=hashing --pass=arith  # ITHICA
```

Since `proto` and `stl` take hours to build, pre-built binaries in `Binaries/` can be used for the next step ("Measuring") by supplying `--fb-dir=Binaries` to the measuring script.

Each `build.sh --pass=P` (for cpu-check) copies its binary into `built/P_bs<BS>_intr<I>_cpu_check`; each `--bench=B` (for FB) run extracts a self-contained bundle (with the benchmark's required runfiles) into `built/B_baseline` or `built/B_arith`.

Note: the `build.sh` scripts are independent of `build_pass.sh`: each rebuilds the ITHICA pass itself before compiling the program, and emits verbose logs, including a "Running on function" message per compiled function confirming that the pass is applied throughout the program.

Measuring. `measure.sh` runs the binaries in the directory specified with `--cc-dir=<dir>` (cpu-check) or `--fb-dir=<dir>` (Fleetbench). The directory must contain the uninstrumented baseline and at least one ITHICA-instrumented binary. Pre-built binaries in `Binaries/` can be used by supplying `--cc-dir=Binaries` (§H) to directly reproduce Table XI.

E1—For cpu-check: performance overhead and binary size increase (main result; ∼40 min for all Table VI configurations).

```
cd cpu-check-ae
scripts/measure.sh --sec=120 --cc-dir=<built|Binaries>
# -> results/cpucheck_results.txt   (table)
#    results/logs/*.out             (raw binary outputs)
```

Each binary runs for a fixed time (120 sec). Performance overhead is calculated as the ratio of CC's to the instrumented binary's throughput of rounds ("SUCCESSES" in logs) completed within the specified time.

Expected Result: Similar to Table XI, also on par with the average (Avg) ratios in Table VI. The exact ratios will vary based on the hardware used (§H). Binary sizes might differ too, due to logging and structural changes for better user experience, code readability, and homogeneity across passes. Binary sizes changed the most for `Mem`, `MemDiv`, and `Br` in the artifact (updated to 4.19×, 5.89×, and 2.25× respectively). These code changes do not affect performance or the functionality of the passes. We also include on GitHub the original versions of these passes that match the binary sizes of Tables VI/XI, under `ithica-artifact/cpu-check-ae/Original_*`.

E2—For Fleetbench: same metrics as E1, with vs. without `Arith` (∼25 min for all six benchmarks).

```
cd fleetbench-ae
scripts/measure.sh --sec=120 --fb-dir=<built|Binaries>
# -> results/fleetbench_results.txt (perf)
#    results/fleetbench_sizes.txt   (size)
#    results/logs/*.out             (raw binary outputs)
```

Each benchmark reports time per operation; the overhead is the ratio of the instrumented to the baseline median ns/op.

Expected Result: Similar to Table XII. Only one sub-benchmark from each of the six benchmark families is reported.

G. Experiment customization

- *Pass selection:* `arith`, `mem`, `memdiv`, `br`, plus the combinations `arithmem`, `arithmemdiv`, `arithmemdivbr`.
- *Pass configuration parameters*: `--blocksize=X`, `--interleaving=Y`, with `X`, `Y` $\in \{0, 1, 2, 4, 8\}$.
- *Run duration:* `scripts/measure.sh --sec=N` sets the per-binary wall-clock budget (default 120).
- *CPU set:* `measure.sh` pins one worker per available CPU (§H). To do it manually, run `cpu_check` with `-c<list>`.

H. Notes

- *Measurement precision and stability.* The exact ratios in Tables XI and XII depend on the underlying hardware (e.g., its core count, hyperthreading and AVX support). A host with at least 32 cores is recommended for comparable results (§C2). Ratios can also vary run-to-run on the same hardware, due to frequency scaling or co-location with other load.
- *Thread placement.* cpu-check sizes its worker pool from `std::thread::hardware_concurrency()` and spawns one worker per logical CPU, independent of cgroup/affinity limits. On a restricted host this over-spawns and distorts the measurement. The scripts pass an explicit `-c` list so exactly one worker runs per available CPU.
- *Pre-built binaries.* All evaluated programs come pre-built in `Binaries/`, avoiding the need to rebuild from source.