

Diagnosing and Delivering Repairs for Fortran GPU Offload Compiler Defects

Spencer H. Bryngelson
Walker Department of Mechanical Engineering
Oden Institute for Computational Engineering and Sciences
The University of Texas at Austin
Austin, Texas, USA
shb@utexas.edu

Benjamin Wilfong Daniel J. Vickers
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia, USA
bwilfong3@gatech.edu, dvickers6@gatech.edu

***Abstract*—Many production simulation codes reach GPUs through directive-based Fortran offload, where compiler support is immature. Compiler suites are built largely from single-construct tests, leaving the clause combinations applications write thinly covered. We use a production flow solver as a compiler-validation instrument: golden files and grindtime detect a defect, metamorphic probes confirm it, and a C-to-Fortran differential and bisection localize it. Three months yielded 45 defects across the AMD, Cray, and Intel toolchains and 18 patches. About a dozen sit in Fortran lowering, the back end, or the linker, and more in the device runtime and the build system, all below any directive; several reproduce only under OpenACC. Fifteen give wrong answers with no diagnostic and four more change only performance, so 42% of what we found passes a compile-and-run suite. The wait users experience is dominated by delivery: Cray CCE 21.0.2 ships two defects whose upstream repairs predate its declared base by three and four months, and five merged fixes have reached no released compiler, 109 days after the last vendor drop.**

***Index Terms*—Compiler testing, OpenMP, GPU offload, Fortran, regression testing, LLVM**

I. Introduction

Directive-based offload maps loop nests onto accelerators, through OpenMP [1] or OpenACC. For Fortran on AMD GPUs, one of the two is the only vendor-supported path that spares the numerics a rewrite in C++. The compilers are new: AMD announced its offload-capable Fortran compiler in late 2024 and shipped it in ROCm in 2025 [2], [3].

Compiler correctness rests on suites of small programs. Random generators build inputs no human would write [4], and equivalence-modulo-inputs testing mutates dead code [5]. Directive suites check each OpenMP construct against the specification [6], [7]. In each, the unit under test is small and the construct under test usually stands alone.

MFC's kernels combine constructs rather than using one at a time. A single MFC kernel joins `target teams distribute parallel do` to `collapse`, `private`, `firstprivate`, and `reduction`, applies the result to derived types and assumed-shape dummy arguments, and calls device routines defined in other translation units. No single-construct suite forms that combination, so a defect that needs it goes undetected. Such a defect can also pass the front end and the linker, in which case it shows up only in the numerics.

SC26 Workshops, November 15-20, 2026, Chicago, Illinois, USA

Detecting these defects needs no new machinery in a project that already runs regression tests across several compilers, though acting on them is not free. Applications have long served to evaluate new platforms [8], [9]. The Multi-Component Flow Code (MFC) is an open-source compressible multi-phase solver [10], [11] whose toolchain runs 627 golden-file cases under five compilers on both major GPU vendors [9]. That work reported about 15 defects, most of them in OpenACC. This paper reports what the same suite finds once the OpenMP path is added alongside OpenACC, and what happens to a defect after it is found.

Most of the cost comes after detection. We filed a report only after establishing whether the defect belonged to the application or the compiler, and where it sat in a stack spanning a front end, an MLIR dialect, a shared IR builder, a device runtime, and a GPU back end. Answering those questions took most of the effort described here, and neither answer determined how long users waited for a repair; that was set by which organization could deliver one and how often it shipped.

II. MFC and its test suite

MFC solves the multi-component compressible Navier–Stokes equations with shock- and interface-capturing schemes, and adds sub-grid bubble models, phase change, surface tension, immersed boundaries, and chemistry [10], [11]. It holds about 200,000 lines of Fortran and emits OpenMP and OpenACC directives from one macro layer, a portability strategy developed over several generations of GPU hardware [12]–[14]. It runs at scale on Frontier [15], most recently at more than one quadrillion degrees of freedom on 9261 nodes [16].

The suite is golden-file based. Each of the 627 cases stores a reference solution, and a case passes only when every state variable matches to a fixed tolerance [9]. Any defect that moves a stored output beyond its comparison tolerance is detected, without anyone having to anticipate which case will fail.

The same source compiles under five compilers. MFC builds with Cray Fortran, AMD Flang, NVIDIA HPC, GNU, and Intel, under OpenACC and OpenMP offload, on AMD and NVIDIA hardware [14]. When one compiler disagrees with four, the defect is usually in that compiler, though four implementations

979-8-3195-1221-5/26/$31.00 ©2026 IEEE

can also agree on a reading of code that is undefined or non-conforming.

The suite also records grindtime, the wall time per grid point per time step, which detects regressions that leave numerical output unchanged. A defect that spills a private array to scratch alters no answer, so no golden file records it, but it moves grindtime against the baselines established for this solver across GPU generations [9], [13].

The suite runs on Frontier under the facility module stack, on AMD's HPCFund cluster, and on self-hosted runners. Moving the pinned compiler and ROCm versions is itself an experiment, and most Cray defects reported here surfaced when MFC moved from CCE 19.0.0 to CCE 21.0.2.

III. Detection

Figure 1 shows the six stages between a failing test and a repair. Stages 1–3 run unattended; stages 4–6 need a person, and reduction takes most of that time.

A. Metamorphic probes

Golden files detect a defect but do not isolate one construct, and writing a reference value by hand for every clause combination does not scale. We therefore generate probes that check themselves. Each probe computes one result two ways inside a single target region and compares the two, storing no predicted value. When two spellings of one computation disagree, the fault lies in the compiler, in the probe, or in our reading of the language.

Two generators produce the probes. The first emits 252 probes covering seven relations, which span whole-array and slice bodies, across three constructs and twelve problem sizes. The second emits 72 probes covering what a finite-volume solver actually writes: `collapse(2)` and `collapse(3)`, reductions, private and firstprivate arrays, nested parallelism, `simd`, allocatables, atomics, and complex arithmetic.

Because a non-conforming probe manufactures a fake defect, every mapped array has explicit shape and is never assumed-size, every `collapse(N)` nest is perfect, and no reduction variable is also private. Three genuine data races surfaced in our own probes and were fixed before any GPU run, and every probe passes on the host first.

B. Validating the harness against a live defect

Our first harness initialized both result arrays to zero. When a compiler drops an iteration, neither array is written, both stay zero, and the two compare equal. The harness was blind by construction to skipped iterations, the defect class it was built to find, and it reported 150 passes out of 150 on a compiler where that defect is live.

The second harness starts the two arrays at different sentinels, so an unwritten element cannot compare equal, and it reports *unwritten* separately from *mismatch*, separating a dropped iteration from wrong arithmetic.

We then ran it against a defect known to be live. On a drop where the dropped-iteration defect reproduces, the control fails 31 of 64, and the probes report unwritten counts matching the derived formula

$$N_{\text{unwritten}} = \big(N - (\lceil N/32 \rceil + 31)\big)\, n_{\text{col}}, \tag{1}$$

where N is the loop trip count and n_{col} the number of columns written per iteration. At $N = 64$ the harness reports 124 unwritten against 124 predicted, and at $N = 128$ it reports 372 against 372.

C. Probe results

On the current AMD beta compiler the array-operation probes fail 27 of 252 identically on gfx90a, gfx942, and gfx950, and the realistic-shape probes pass 72 of 72. All 27 failures fit eq. (1) with zero mismatches, so the three architectures share one defect and do not exhibit three; MI300X and MI355X behave as MI250X does.

Sizes at or below 33 never fail, as eq. (1) predicts. Masked assignment never fails while the three whole-array relations always do, which places the defect on the path that emits an opaque runtime call in the loop body.

IV. Attribution

The five-compiler matrix addresses application-versus-compiler cheaply, since agreement among four compilers against the fifth points to the fifth implementation, but it names no component within that compiler. On the AMD OpenMP path a sharper test is available, though it does not extend to OpenACC, which Clang does not implement. Clang and Flang share the device runtime, the offload plugin, and the AMDGPU back end, and they are meant to share device code generation through `OpenMPIRBuilder` [17], [18]. A line-for-line Fortran port of a C program therefore holds fixed the hardware, the runtime, and the versions of every shared downstream component. What differs is then the two front ends and the routes they take into the shared code. When C is correct on the same GPU under the same runtime and Fortran is not, the defect lies on the Fortran route, once the language-level differences a port cannot remove have been ruled out.

The LLVM offload suite ships C tests that already pass on the target hardware, so porting one to Fortran yields a Fortran test with a known-good reference and turns the differential into a test generator. Porting `offload/test/offloading/schedule.c`, which passes on gfx90a, produced a Fortran equivalent that fails.

A. Divergence at the shared builder

Clang implements device-specific behavior for a construct in its own GPU code generator, while Flang reaches `OpenMPIRBuilder`, which implements the host behavior and omits the device part. Four of the defects we found trace to that split. The front end accepts the clause, semantic checking passes, code generation discards it, and the program runs to completion and returns a plausible answer.

Consider `critical` inside a target region. The device runtime takes the lock on the lowest active lane of a wavefront, which is correct only when one lane per wavefront arrives.

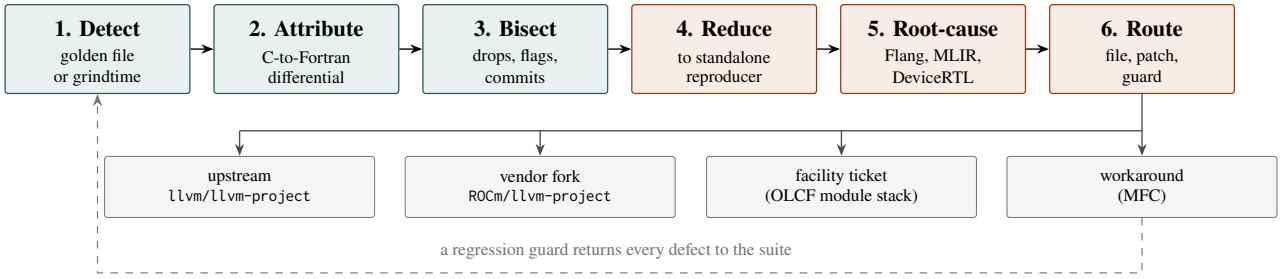


Fig. 1: The six-stage pipeline. Green stages run without human attention; rust stages require a person. Every route ends by adding a test that fails if the defect returns.

TABLE I: Result of *N* threads each incrementing a shared counter once inside a `critical` region in a target region on gfx90a.

Threads	Wavefronts	Flang	Clang
64	1	1	64
128	2	2	128
256	4	4	256

Clang guarantees that by wrapping the region in a loop over the hardware threads in the block and synchronizing between turns. `OpenMPIRBuilder::createCritical`, which Flang reaches, does not. The surviving count equals the number of wavefronts rather than the number of threads, as table I shows.

For `ordered`, the device path hands the loop to a static-loop runtime entry and drops the schedule arguments, and the device `__kmpc_ordered` is an empty stub. The ordering a dispatch loop would supply is therefore never emitted, while Clang switches to dispatch whenever `ordered` appears. For `schedule`, the device path hardcodes both chunk arguments to zero, which the runtime reads as unspecified. Every kind and chunk size emits a byte-identical call, and the mapping is always that of `schedule(static,1)`. Device-outlined target regions left un-inlined fail the same way, since Clang's device code generation inlines the loop and Flang's does not.

Enumerating the device behavior Clang implements outside `OpenMPIRBuilder` therefore yields a list of OpenMP constructs still to test in Fortran, and the C tests that establish the expected answer already exist upstream.

B. Bisection

Optimization flags are the cheapest thing to bisect over, though they do not identify a stage on their own, since optimization can expose or suppress a defect introduced earlier. Several defects here reproduce only under a specific flag combination. Upstream commits bisect in the usual way when the defect reproduces on a source build.

Bisecting over released vendor compilers is open to us only because AMD publishes its beta compilers as a dated archive [19]. The AFAR series is retained at a public URL, and no other compiler in our build matrix ships such an archive. A defect can be bisected across five releases spanning ten months, and whether a configuration ever worked can be settled against released binaries instead of a rebuilt compiler.

C. Reduction

Reduction converts a failing 200,000-line solver into a Fortran program of a few dozen lines. Automated reducers work well for C, given an oracle that separates the interesting failure from every other failure [20], [21]. For a wrong-answer offload defect the oracle has to run on a GPU, distinguish a wrong number from a crash, a hang, and an unrelated wrong number, and reject reductions that introduce undefined behavior. We did not build one, so reduction was manual, and it consumed more time than any other stage.

Case optimization compiles each simulation with its parameters embedded as Fortran `parameter` constants, which shrinks the kernel and often changes whether the defect fires, and that sensitivity is itself diagnostic. The solver is also split into modules with narrow interfaces, so a suspect kernel can usually be lifted into a driver program with a few dozen lines of setup.

Clause interactions surface at this stage. For one defect, `lastprivate` on a `distribute` construct aborted the compiler with a not-yet-implemented diagnostic, and adding `simd` to the same construct made the abort disappear, so a reduced test carrying `simd` would have reported nothing. The innermost operation with `simd` present is a `SimdOp`, which the data-sharing processor handles, while `omp::DistributeOp` falls through to the unimplemented case.

D. Root cause

Fortran offload passes through the Flang front end, the MLIR OpenMP dialect, `OpenMPIRBuilder`, the device runtime, and the AMDGPU back end [17], [18], [22]. The layer that reports a symptom is often not the one at fault, and reading the IR at each layer boundary is what narrows it. The resulting patches are small: across the 18 we submitted, the median patch adds 62 lines. One mechanical patch is the exception, touching 94 TableGen and test files.

Whatever the destination, the last step is to add a test to the application suite that fails if the defect returns. Defects repaired in a vendor beta can reappear in the next drop, and workarounds adopted while waiting for a repair are otherwise never removed.

TABLE II: The 45 defects, classified by the diagnostic the toolchain emits. Silent wrong answers and performance-only defects pass tests that check only compilation and exit status.

Class	Count	Diagnostic emitted
Silent wrong answer	15	none
Crash, hang, or abort	16	abort or GPU fault
Performance only	4	none
Build and toolchain	7	link or configure error
Unimplemented feature	3	not-yet-implemented
Total	**45**	

V. What we found

Between 2026-05-19 and 2026-08-10 we reported 45 defects: 28 filed publicly against `llvm/llvm-project` and `ROCm/llvm-project`, 13 against Cray CCE through the facility, and 4 against Intel `ifx`. We submitted 18 patches. Table II classifies the defects by the diagnostic the toolchain emits.

Fifteen silent wrong answers and four performance defects make 19 of 45, or 42 %, for which the toolchain emits no diagnostic at all.

Classifying the same defects by the layer that holds the cause rather than by the symptom gives a different picture. Roughly half sit in directive lowering, where a clause is accepted and then dropped or mislowered, and those are specific to the model that spells the clause. The rest are not. They sit in Fortran lowering, in the device runtime, in the AMDGPU back end, in the linker, or in the build system, and they reach a user through whichever model happens to be compiled. One is host code with no offload at all. Three of the Cray reproducers are written in OpenACC rather than OpenMP, among them the AGPR-copy-MFMA assertion of table III, whose workaround costs 61 % of a kernel's throughput, and an archived set of twelve more covers `!$acc declare` alone. The programming model determines how a defect is reached but not where it lives, and a repair below the directive layer serves both models at once.

Three of the Cray reports arise from one rule violation: a `defaultmap` clause overrides an explicit specification. It privatizes a scalar carrying an explicit `map(tofrom:)`, zeroes an array made device-resident and pushed with `target update to`, and sends a variable listed in `private()` to a present-table lookup that aborts. Two share a device-IR signature and the third is host-side, so filing them as three tickets would have split one rule violation across three triage queues.

We also record the leads that were not defects, so that they are not re-derived. Two of the 45 proved not to be compiler defects and were closed, and two more duplicated an existing upstream issue. One crash we saw a single time never reproduced in twenty further attempts, and we did not file it. One apparent regression turned out to compare two different compilers, because one ROCm release ships classic Flang and the next ships the LLVM front end.

TABLE III: Cray CCE 21.0.2 back-end defects and the dates their upstream repairs merged to `main`, relative to 2025-12-12, the date of `llvmorg-21.1.8`.

Defect	Fixed	vs. cutoff	Size
AGPR-copy-MFMA	2025-08-16	−4 mo.	3 lines
Promote-alloca	2025-09-10	−3 mo.	1 opcode
InstCombine	2026-02-15	+2 mo.	3 lines

TABLE IV: Outcome of the 18 patches submitted, as of 2026-08-10, with merge latency measured from submission to merge.

Outcome	Count	Merge latency (d)
Merged and retained	5	0.1, 1, 3, 8, 13
Merged, later reverted	1	12
Closed without merging	6	–
Open at time of writing	6	–
Total	**18**	

VI. Delivery latency

Where to file a defect is a judgment about latency. A defect in Flang, the OpenMP dialect, `OpenMPIRBuilder`, or the device runtime belongs upstream, where one repair reaches every consumer. A defect present only in the vendor fork, or one whose repair users need sooner than upstream can supply it, belongs to the fork. Defects in the module stack go to the facility. When no repair will arrive before the science must run, the remaining option is a workaround in the application.

A. Repairs absent from the shipped compiler

Cray CCE 21.0.2 states that it is built on an LLVM 21 base corresponding to the tag `llvmorg-21.1.8`, which upstream dates 2025-12-12. That tag is public, so reachability of any upstream commit from it is decidable without vendor cooperation. Table III lists its three AMDGPU back-end defects against the upstream commits that repair them.

Two of the three repairs merged to upstream `main` four and three months before that tag was cut. Neither is reachable from the `release/21.x` branch on which the tag sits, and a search of that branch finds no cherry-pick of either. Both repairs were therefore available upstream in time and absent from the shipped compiler. The third merged after the tag and is legitimately absent. Because CCE 21.0.2 is built on an upstream stabilized release branch, it carries only the repairs that branch received, and for a back end under active development those arrive more slowly than they land on `main`. The history does not say whether the release branch should have carried them or the vendor should have cherry-picked them. Backporting the two commits, both small and both carrying upstream regression tests, would have removed both defects from the release branch.

B. Merge latency against drop cadence

Six of the 18 patches merged, with a median latency of 5.5 d and a range from 2.4 h to 13 d, and one of those six was later reverted. Six were closed without merging, in most cases because a maintainer preferred a different repair for the same defect, which leaves the defect report standing.

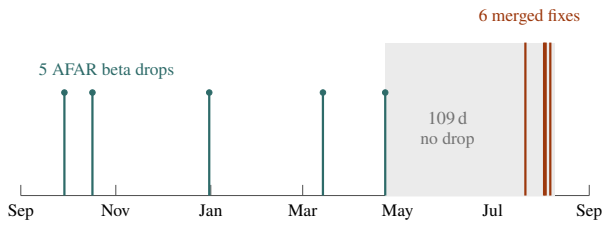


Fig. 2: AMD beta compiler drops and upstream patch merges. The shaded band is the interval since the most recent drop.

The compiler on the machine is a vendor beta and changes only when the vendor publishes a drop, so the drop cadence sets when a repair reaches users. Figure 2 places the five AMD beta drops released since 2025-09-29 against the dates our upstream patches merged. Intervals between drops are 18, 75, 73, and 40 d, a median of 56.5 d, an order of magnitude above the 5.5 d median merge latency.

Our six fixes merged between 2026-07-22 and 2026-08-07, so at the cutoff each had accumulated between 3 d and 19 d of waiting, and none had reached a released compiler. Five of the six remain upstream and need only a vendor build. The sixth was reverted, and it must reland before any drop can carry it. How long the five will wait is not estimable from four intervals. The most recent drop appeared 2026-04-23, and the 109 d since then already exceeds every completed interval in the series.

One of the six merged and was reverted two days later over an unrelated fault in another application, and it has not relanded, so that defect is live again. Counting merged patches therefore overstates delivered repairs, and only a regression test in the application detects the reversal.

Cutting a release from a moving branch can also expose a defect that no upstream release ever shipped. One miscompile that stamps false no-wrap flags on array addressing reached users because two consecutive drops were cut from a window in which the commit introducing the flags was present and the commit correcting them was not. The repair existed upstream before those drops were published, so we asked the vendor to rebase past it and added a workaround to hold the campaign together until they do.

VII. The cost to users

Two link-time defects in CCE 20 and CCE 21 crash `lld` during device link-time optimization. CCE 20 does not link MFC on Frontier at all, and CCE 21 links only with AGPRs disabled at link time. The three-line upstream repair absent from CCE 21 is already written and tested. Without it, MFC must disable AGPRs on the unified register file of gfx90a, which raises scratch from 36 B to 1060 B per lane and slows its most register-hungry kernel by 61 %.

CCE 21 discards a dynamically indexed store into a one-based private array with no diagnostic, which corrupted MFC's viscous flux by about a quarter of the term. A single `defaultmap` clause zeroed device-resident arrays; removing it moved MFC's OpenMP suite from 566 of 627 passing with 60 aborts to 622 of 627 with none, matching the OpenACC arm, which sat at 622 of 627 throughout. CCE accepts `!DIR$ INLINENEVER` on a device routine, then overrides it with the opposite attribute, silently disabling 51 existing sites in MFC.

A `firstprivate` fixed-size array is copied in through the Fortran runtime assignment helper rather than as a value copy. That spills about 35 kB per work item to scratch and runs 30 to 50 times slower. On an older release the same lowering fails to link, because the helper is absent on the device. Un-inlined device regions cost 212 vector registers against 80 for the C equivalent and cut occupancy from 6 to 2.

MFC carries seven workarounds, each tagged with the defect it evades so that it can be removed once the repair lands, and most of them compensate for a defect already fixed somewhere else. They include a reversed stride against the no-wrap miscompile, a host-capture change to restore lost viscosity, and an added flag on the AMD offload path.

VIII. Generalizing the method

The defect counts above are specific to MFC and to these three compilers. The procedures that produced them depend only on the preconditions table V lists, alongside the settings that satisfy them. Two of them need no accelerator and no directive-based programming model, and one of the five is specific to OpenMP.

A. Auditing a compiler against its own declared base

When a vendor names an upstream base tag, its integration debt becomes measurable by anyone. Take a defect that reproduces in release R, an upstream commit c that repairs it, and the base ref T that R declares. Reachability of c from T is not by itself decisive; it must be read together with the merge date of c and the date T was cut. If c is unreachable and merged after T, the repair was unavailable when R was built. If c is unreachable and merged before T, it was available and was not carried. If c is reachable, it is present under that history, and the defect must have another cause. Reachability must be tested against the branch the tag sits on rather than upstream `main`, since a backport is a cherry-pick carrying a different hash. A title search over the branch catches most cherry-picks. It will miss a squashed or rewritten one, and it will miss an equivalent repair written independently, so a negative result is not proof that the repair is absent.

The test runs against the public upstream repository alone, without a GPU, vendor cooperation, or additional source access. Facilities could run it at acceptance time to learn which known repair commits a candidate compiler lacks; whether the associated defect still reproduces has to be settled with a reproducer, since an equivalent repair may have arrived by another route. Applied to the three CCE defects of table III, it finds two whose repairs were available and one whose repair postdates the declared base.

B. The stage at which a repair stalls

Counting defects measures discovery, which was not the step that delayed users in any of the cases here. What a user waits

TABLE V: Procedures derived from this study, their prerequisites, and other settings that satisfy those prerequisites.

Procedure	What it exploits, and what a site needs to reuse it	Where else the precondition holds
Application oracles	Reference solutions and a throughput metric catch defects that produce no diagnostic. Needs stored reference outputs and a cost metric stable enough that a change of a few percent is readable.	A regression-tested application being ported to new hardware or a new compiler
Front-end differential	Compares two producers of a shared lowering component, which exposes behavior either one implements above the shared line. Needs two front ends nominally sharing a component and a pair of programs the two should compile alike. Our instance is OpenMP-only, since Clang implements no OpenACC.	MLIR dialects with several producers; two languages over one back end
Self-checking probes	One result computed two ways in a single region, so a disagreement is a defect in the probe, the language semantics, or the compiler, and no expected answer is stored. Needs a construct with two conforming spellings and enough care that the probe itself conforms.	Language or directive features with redundant spellings
Oracle validation	An oracle whose two arms hold the same initial value cannot see an omission that leaves both at that value, so it must be run against a defect known to be live before a pass is believed. Needs such a defect and a way to reach it.	Differential and metamorphic oracles whose arms can share an initial state
Base-tag audit	A compiler built on an upstream release branch carries only the repairs that branch received; reachability of the repair commit from the declared tag, read together with its merge date, distinguishes a repair that postdates the base from one that predates it and was not carried. Needs a declared base tag, public upstream history, an identified repair commit, and a reproducer.	LLVM downstreams, distribution kernels, and other snapshotted dependencies that declare a base

on is a repair that exists somewhere and has not arrived, and each of our three toolchains stalled somewhere different. The CCE repairs were never carried onto the branch the vendor builds from, and a cherry-pick would close that gap. The AMD repairs are merged in the vendor's own public branch but appear in no published build, and cutting a drop would close that gap. The remaining case is a repair that postdates a base the vendor has already frozen, where the gap cannot close until the next release. Each calls for a different request, and a report that names which one applies is easier for a vendor to act on than one that names only the defect. Applying the classification needs a vendor-intermediated stack and a public upstream; it does not need Fortran, OpenMP, or a GPU.

IX. Conclusions

A production flow solver, run continuously across three vendor toolchains, found 45 defects in three months, 42 % of which a compile-and-run suite records as passes. Four of the silent ones share a single structural cause, which identifies where to test next.

The delay users experience is set after the repair exists. Two Cray defects had upstream repairs months before the release that still carried them, and five of our own merged fixes await a vendor build. Users absorb the delay as maintained workarounds and undetected wrong answers, and in one case as a compiler that will not link the application at all.

Detection costs nothing beyond golden-file tests already running across several compilers, and the same matrix performs the initial triage of application against compiler. The later stages cost more: bisection over vendor releases depends on an archive one vendor publishes and the others do not [19], and reduction stayed manual and was the most expensive stage in human time.

A. Limitations of this work

We observe one application, so the mix of defects reflects the constructs MFC writes. Its OpenMP path is newer than its OpenACC path and received more attention here, so the split between the two models reflects our effort as much as the compilers. Our coverage of the AMD path is deepest, because that toolchain archives its betas [19] and publishes its source, and the same bisection is unavailable for the proprietary compilers we test. Reduction is manual and dominates the human cost, and we did not build the device-side oracle that would let a reducer run unattended. Counts of reports are not counts of root causes, as the three faces of one Cray defect show. The delivery estimate rests on four intervals between drops, too few to fix the cadence firmly.

B. Prospects for extension

An automated reducer needs the device-side oracle described in section III, and the metamorphic probes already supply most of it. The shared-builder result of section IV gives a directed search, since the constructs it implicates are enumerable and upstream already has C tests for them. The backport check of table III is mechanical wherever a vendor states a base tag, and running it at each release would tell a facility, before deployment, which known defects a compiler still carries.

Acknowledgments

We thank the participants and mentors of the OLCF Frontier and Open Hackathons, the MFC contributors, and the compiler and system engineers at AMD, HPE, and the OLCF who triaged the defect reports discussed here.

This work was supported by the Predictive Science Academic Alliance Program (PSAAP Award DE-NA0004263, "Center for Multiscale Modeling of Multiphase Combustion (CM3C)")

managed by the NNSA (National Nuclear Security Administration) Office of Advanced Simulation. SHB acknowledges support from the U.S. Department of Defense, Office of Naval Research under grant numbers N00014-22-1-2519 and N00014-24-1-2094, the Army Research Office under grant number W911NF-23-10324, the Department of Energy under DOE DE-NA0003525 (Sandia National Labs, subcontract), the Oak Ridge Associated Universities (ORAU) Ralph E. Powe Junior Faculty Enhancement Award, and hardware gifts from NVIDIA and AMD.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725 (allocation CFD154, PI Bryngelson). This work used Delta and DeltaAI at the National Center for Supercomputing Applications, Bridges2 at the Pittsburgh Supercomputing Center, and ACES at Texas A&M University through allocations PHY210084 and PHY240200 (PI Bryngelson) from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296. Additional computations were performed on the AMD HPC Fund research cluster.

Data and code availability

MFC is available at https://github.com/MFlowCode/MFC under the MIT license. The reproducers, analyses, and probe generators discussed here are at https://github.com/sbryngelson/compiler-bugs, tagged `v1.0.1` for this article. Defect reports and patches against upstream LLVM and the vendor fork are public at https://github.com/llvm/llvm-project and https://github.com/ROCm/llvm-project.

Appendix A
Artifact description

The artifact is the reproducer corpus at https://github.com/sbryngelson/compiler-bugs, organized by toolchain and tagged `v1.0.1` at the state described here. Each case is a directory holding a standalone Fortran reproducer, a README recording the verified steps, the compiler versions on which it reproduces, the expected and observed output, and links to the upstream and downstream tickets. Controls sit beside the reproducers, isolating the trigger: the array lower bound, the clause that must be present, or the flag combination required. Several cases include the patch that repairs them and the extracted device IR or disassembly that establishes the mechanism.

The metamorphic generators and their recorded results for gfx90a, gfx942, gfx950, and upstream Flang are included. Reproducing the AMD cases needs an AMD GPU and one of the archived beta drops. AMD retains those publicly at https://repo.radeon.com/rocm/misc/flang/, so each defect can be run against the compiler it was filed against rather than a later one in which it may already be repaired. For cases that also affect upstream LLVM, build from source at the commit named in the report. Two classes of case need no GPU: one is checked statically from the shipped device runtime archive, and the backport claims of table III are verified with `git merge-base` against a public tag.

References

[1] OpenMP Architecture Review Board, "OpenMP application programming interface, version 6.0," 2024.

[2] J. Chang, B. Cornille, M. Klemm, and J. Potyka, "Introducing AMD's next-generation Fortran compiler," AMD ROCm Blog, 2024.

[3] B. Cornille *et al.*, "Implementing OpenMP offload support in the AMD next generation Fortran compiler," in *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2025.

[4] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, pp. 283–294.

[5] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 216–226.

[6] J. M. Diaz, S. Pophale, O. Hernandez, D. E. Bernholdt, and S. Chandrasekaran, "OpenMP 4.5 validation and verification suite for device offload," in *International Workshop on OpenMP*, 2018, pp. 82–95.

[7] A. Jarmusch, F. Cabarcas, S. Pophale, A. Kallai, J. Doerfert, L. Peyralans, S. Lee, J. Denny, and S. Chandrasekaran, "CI/CD efforts for validation, verification and benchmarking OpenMP implementations," in *IEEE International Conference on Cluster Computing Workshops*, 2024.

[8] W. Elwasif, S. Bastrakov, S. H. Bryngelson, M. Bussmann, S. Chandrasekaran, F. Ciorba, M. A. Clark, A. Debus, W. Godoy, N. Hagerty, J. Hammond, D. Hardy, J. A. Harris, O. Hernandez, B. Joo, S. Keller, P. Kent, H. Le Berre, D. Lebrun-Grandie, E. MacCarthy, V. G. M. Vergara, B. Messer, R. Miller, S. Oral, J.-G. Piccinali, A. Radhakrishnan, O. Simsek, F. Spiga, K. Steiniger, J. Stephan, J. E. Stone, C. Trott, R. Widera, and J. Young, "Early application experiences on a modern GPU-accelerated Arm-based HPC platform," in *HPC Asia '23*, ser. International Workshop on Arm-based HPC: Practice and Experience (IWAHPCE), Singapore, 2023, pp. 35–49.

[9] B. Wilfong, A. Radhakrishnan, H. A. Le Berre, T. Prathi, S. Abbott, and S. H. Bryngelson, "Testing and benchmarking emerging supercomputers via the MFC flow solver," in *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2025, arXiv:2509.13575.

[10] S. H. Bryngelson, K. Schmidmayer, V. Coralic, J. C. Meng, K. Maeda, and T. Colonius, "MFC: An open-source high-order multi-component, multi-phase, and multi-scale compressible flow solver," *Computer Physics Communications*, vol. 266, p. 107396, 2021.

[11] B. Wilfong, H. A. Le Berre, A. Radhakrishnan, A. Gupta, D. J. Vickers, D. Vaca-Revelo, D. Adam, H. Yu, H. Lee, J. R. Chreim *et al.*, "MFC 5.0: An exascale many-physics flow solver," *Computer Physics Communications*, p. 110055, 2026.

[12] A. Radhakrishnan, H. Le Berre, and S. H. Bryngelson, "Scalable GPU accelerated simulation of multiphase compressible flow," in *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, Dallas, TX, USA, 2022, pp. 1–3.

[13] A. Radhakrishnan, H. Le Berre, B. Wilfong, J.-S. Spratt, M. Rodriguez Jr., T. Colonius, and S. H. Bryngelson, "Method for portable, scalable, and performant GPU-accelerated simulation of multiphase compressible flow," *Computer Physics Communications*, vol. 302, p. 109238, 2024.

[14] B. Wilfong, A. Radhakrishnan, H. A. Le Berre, S. Abbott, R. D. Budiardja, and S. H. Bryngelson, "OpenACC offloading of the MFC compressible multiphase flow solver on AMD and NVIDIA GPUs," in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Atlanta, Georgia, USA: ACM, 2024, pp. 1923–1933.

[15] S. Atchley, C. Zimmer, J. Lange, D. Bernholdt, V. Melesse Vergara, T. Beck, M. Brim, R. Budiardja, S. Chandrasekaran, M. Eisenbach *et al.*, "Frontier: Exploring exascale," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023.

[16] B. Wilfong, A. Radhakrishnan, H. Le Berre, D. Vickers, T. Prathi, N. Tselepidis, B. Dorschner, R. Budiardja, B. Cornille, S. Abbott, F. Schäfer, and S. H. Bryngelson, "Simulating many-engine spacecraft: Exceeding 1 quadrillion degrees of freedom via information geometric regularization," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2025, pp. 14–24.

[17] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *IEEE/ACM International Symposium on Code Generation and Optimization*, 2021, pp. 2–14.

[18] J. Doerfert, A. Patel, J. Huber, S. Tian, J. M. M. Diaz, B. Chapman, and G. Georgakoudis, "Co-designing an OpenMP GPU runtime and optimizations for near-zero overhead execution," in *IEEE International Parallel and Distributed Processing Symposium*, 2022, pp. 504–514.

[19] Advanced Micro Devices, "ROCm AFAR Fortran compiler drops," https://repo.radeon.com/rocm/misc/flang/, 2026, dated archive of beta compiler releases.

[20] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012, pp. 335–346.

[21] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[22] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization*, 2004, pp. 75–86.