RCE Compile: A Law of Compilation, Tested as a Normal Form Against Real Compilers

David Charlot, Open Interface Engineering

Research study · draft · Open Interface Engineering · research.openie.dev

Revision `5c50a85` (23 November 2026) · Web: research.openie.dev/papers/rce-compile/ · Archival LaTeX: /pdfs/latex/rce-compile.pdf (.tex)

Measurement. Research study, draft, not a journal final. Byte reproducibility (`E1`), `20`-kernel × **`15-tier`** Linux Zen 5 and Core i9-13900H `RAPL` core and package **`reported_j`** compilation vs. execution energy payback horizons N^* including pre-compiled **`librce_rt.rlib`** (`20.07`× lower compile energy, median $N^* = 1{,}079$) and `H4` Regime-B checks (**`E4-Energy`** / **`E4-Prelude`**), and `245`-program / `500`-mutant De Bruijn nameless cache soundness plus warm-cache avoided compilation **`reported_j`** (**`E5-Scale`**) are reported in Section 8, alongside the 12-compiler normal-form audit (**`R..G`**). No hardware power rail or wall meter was attached (**`measured_j = None`**, **`board_metered = false`**).

RCE Compile (a research project in the Reality Code Engine repository) asks whether one statement of what compilation is holds for every compiler, independent of the Reality Code Engine itself. Checked against twelve compiler families, the candidate holds as a standard a compiler can be measured against, not as a description of what compilers do. This study states that standard, the evidence for and against each of its seven parts, a measurable replacement for its convergence function, seven hypotheses with their falsifiers, and the experiments run so far: byte comparisons of compiler outputs on three machines, erasure counts alongside Linux Running Average Power Limit (RAPL) reported_j compilation cost and optimization payback horizons across 20 kernels and 15 compiler tiers (including a pre-compiled shared .rlib runtime crate that cuts emitted Rust -O3 compile reported_j by 20.1×), and name-free De Bruijn reuse hashing tested across 245 programs and 500 verified semantic mutants.

Contents

Abstract **2**

Terms and evidence marks **3**

1. Problem statement **5**

2. The candidate and its refinement **5**
2.1 Why the candidate fails as a description . . . 5
2.2 The refinement: a normal form with two bounds and a measured distance . . . 5
2.3 The normative statement . . . 5

3. The Compiler Form, R to G **6**
3.1 The seven parts . . . 6
3.2 Four equivalence levels . . . 6
3.3 What a gate must name . . . 7
3.4 Evidence per part across compilers . . . 7
3.5 Part-by-part findings . . . 9

4. The convergence function **10**
4.1 Landauer and Kolmogorov as conceptual bounds . . . 10
4.2 Why neither is computable or measurable as stated . . . 10
4.3 The measurable replacement . . . 11

5. The gear ladder and the 98/2 split **11**

6. Prior art **12**

7. Hypotheses and falsifiers **12**

8. Experiments **13**
8.1 E1: determinism of lowering, measured . . . 13
8.2 E2 to E4: protocols, E4's erasure counts, and E4-Energy (`reported_j` & payback horizons) . . . 17
8.3 E5: a name-free hash, measured . . . 21

9. Limits of this study **24**

10. Conclusion **24**

Appendix A. Claims checked **25**
A.1 Contradicted claims, with the correction . . . 25
A.2 Claims that partly hold, with the correction . . . 25
A.3 Confirmed claims . . . 26
A.4 Claims not found . . . 26

References **27**

Revision `5c50a85`, 23 November 2026. Research study, draft. Project repository: openIE-dev/rce, `compile/`, which holds the charter, the hypotheses and the experiments this study reports. Interactive observatory: `/living/rce-compile/`.

Abstract

RCE (Reality Code Engine) Compile is a research project that asks whether a single statement of what compilation is, here called a Law of Compilation, holds for any compiler, independent of the Reality Code Engine in whose repository the project lives. The candidate statement says that compilation is meaning-preserving lowering between canonical representations, ranked by a cost whose floor is Landauer's limit and whose target is the shortest equivalent program, gated by verification and compounded by reuse. This study checks that candidate against twelve families of production compilers and the research literature behind them. The candidate fails as a description: no compiler surveyed hashes a name-free representation, lowering in shipping compilers is neither total nor shown to preserve meaning, and no compiler accounts for energy. It holds up as a normative framework once three clauses are rewritten, and the study states it in that form: a normal form with two bounds, plus a measured distance from them. The convergence function cannot be computed or measured as stated, and the study replaces it with two counts that can be computed and compared with measured energy. Seven hypotheses with falsifiers follow, with the experiments run so far and the protocols of two more. Compiler outputs were compared byte for byte on three machines; some differences survive the toolchains' path remapping and have no traced cause, so the hypothesis that lowering is deterministic is neither supported nor falsified. Erasure and description length were first counted for 96 builds on a machine with no energy counter, and pre-registered Regime-B extensions then read Linux Running Average Power Limit (RAPL) on-chip `reported_j` (`board_metered = false`, no wall wattmeter) across 20 deterministic integer and fixed-point kernels and 15 compiler and runtime tiers (300 of 300 builds checksum-verified): lowering from Python to `clang -O3` recovers its compilation joules after a median of 2,490 runs, `-O0` to `-O3` pays back after a median of 53,464 to 102,720 runs, `-O2` to `-O3` never recovers its extra compile energy on 4 to 6 of the 20 kernels, pre-compiling RCE's standalone Rust runtime prelude once into a shared `librce_rt.rlib` crate cuts per-kernel emitted Rust `-O3` compilation `reported_j` by 20.07× (56.29 J to 2.81 J core, cross-checked at 15.44× on an idle dedicated Core i9-13900H host) and lowers its payback horizon against Node JavaScript 28.8× from 31,080 to 1,079 runs, and time multiplied by power explains far more Regime-B variance than static intermediate-representation size (adjusted R^2 0.9999 against 0.1154). A name-free hash, the part of the standard that no compiler surveyed has, was built first over binary objects and WebAssembly modules (16 kernels and 4 programs) and then scaled across 185 RCE Forms, 60 C, Rust and Python kernels, and 500 test-vector-verified semantic mutants: positional De Bruijn hashing produced 0 of 500 (0.00%) false-positive collisions while cutting cold `clang -O3` and `rustc -O3` compilation `reported_j` by 167.8× and 448.5× on renamed hits, whereas naive identifier-stripping collided on 150 of 150 (100.0%) variable-binding swaps. Almost every external source was seen only as a search result, and each citation says so.

In plain words

1. The question is whether all compilers obey one rule. Real compilers do not, so the rule is restated as a standard that any compiler can be measured against.
2. The standard has seven parts. Most compilers have some of them. None keeps track of energy, and none proves that it translates every program correctly.
3. Heavy compiler optimization spends chip energy up front to save chip energy per run. Across 20 test kernels on Linux chip counters (`reported_j`, not a wall plug meter), compiling to `clang -O3` instead of running Python pays back after about 2,500 runs, stepping from `-O2` to `-O3` never pays back on up to 30% of the kernels, and pre-compiling a shared `.rlib` runtime library cuts emitted Rust compile energy 20× so native lowering beats JavaScript after about 1,080 runs instead of 31,000.
4. A fingerprint of a program that replaces variable names by their binding positions — rather than merely erasing names — recognised renamed copies across 245 programs, avoided 168× to 449× of rebuild chip joules on warm hits, and made zero false matches across 500 bug-injected mutants where naive name-erasing collided on every swapped variable pair.

Terms and evidence marks

Every acronym on this page is either expanded where it is first used or listed here.

- AI, API: artificial intelligence; application programming interface
- AIR, ZIR, MIR: intermediate representations of the Zig compiler (analysed, untyped and machine-level)
- ANF: A-normal form, a program form in which every intermediate value is named
- APU, ARX, BST: accelerated processing unit; add-rotate-xor cipher structure; binary search tree
- ASR: Abstract Semantic Representation, the intermediate representation of LFortran
- AST: abstract syntax tree
- BF16, FP16, FP8, FP4: 16-, 8- and 4-bit floating-point formats
- CI, CRC, CRC32: continuous integration, the checks run on every change to a repository; cyclic redundancy check (32-bit IEEE 802.3 polynomial)
- CLIF: the intermediate representation of the Cranelift code generator
- CMOS: complementary metal-oxide-semiconductor, the usual chip technology
- CN101, MAX, XY.AI, MMA-Sim: product, platform, company and tool names used as they are
- CNF: conjunctive normal form, a standard form for logical formulas
- COBOL, SPARK, GHC, GCC, LLVM, MLIR, XLA, TVM, CUDA: names of languages and compilers (COBOL; SPARK, a verifiable subset of Ada; the Glasgow Haskell Compiler; the GNU Compiler Collection; the LLVM compiler infrastructure and its multi-level representation MLIR; the XLA and TVM compilers for tensor programs; NVIDIA's GPU platform)
- CPU, GPU, FPGA: central processing unit, graphics processing unit, field-programmable gate array
- DDC: diverse double-compiling, a test that rebuilds a compiler with an independent compiler and compares the bits
- DRAM, SRAM, DP: dynamic and static random-access memory; dynamic programming
- DWARF: the standard debug-information format
- ELF: Executable and Linkable Format, the binary format of Linux executables and object files
- EMI: equivalence modulo inputs, a compiler-testing method that changes code a given input never runs and checks the output stays the same
- FNV, IQR, JS, JSON: Fowler–Noll–Vo hash; interquartile range; JavaScript; JavaScript Object Notation
- GEMM: general matrix multiplication; cuBLAS and cuDNN are NVIDIA's libraries for it and for neural-network kernels
- GIMPLE, FIR, HLFIR, SOACS: the intermediate representations of GCC, of LLVM Flang (two levels) and of Futhark
- GNU, GNAT, GNATprove: the GNU project; the Ada compiler in GCC; the SPARK proof tool
- GODEBUG: the environment variable that turns on debugging settings of the Go toolchain and runtime
- GOPATH: Go's older workspace mode, used before Go modules
- HAL, CNKI, ACM DL, J-STAGE: national and publisher research databases (France, China, the ACM, Japan)
- HIR, THIR: the high-level and typed high-level representations of the Rust compiler, rustc
- HOL4, Rocq, Lean, Coq: proof assistants (Coq was renamed Rocq)

- HPC: high-performance computing
- IEEE 754: the floating-point standard of the Institute of Electrical and Electronics Engineers; NaN: not-a-number
- IR: intermediate representation, the form a compiler holds a program in between source and machine code
- ISLE: the rule language in which Cranelift's lowering rules are written
- LAPACK: the Linear Algebra Package, a Fortran library
- LCNF: Lean Compiler Normal Form
- LTO, LZ77: link-time optimisation; Lempel–Ziv 1977 sliding-window compression
- KB, MB, GB: kilobytes, megabytes, gigabytes
- MDL, MSR, NTT, OLS, OS: minimum description length, a two-part code length relative to a named class of models; model-specific register; number-theoretic transform; ordinary least squares; operating system
- ML compilers: compilers for neural-network tensor programs, such as XLA, TorchInductor, TVM, TensorRT, Triton and IREE
- MLGO, TENSAT, STOKE: a policy-driven LLVM optimisation framework, an equality-saturation optimiser for tensor graphs, and a stochastic superoptimiser
- NOT FOUND: an evidence mark, defined below
- NVML, RAPL, PP0: NVIDIA's management library and Intel's running average power limit interface (including its processor-core energy plane PP0), both of which report energy counters
- OSS: open-source software
- OTP: Open Telecom Platform, the name of the Erlang distribution and its release numbers
- POPL, PLDI, ICFP, OOPSLA, MLSys, ISSCC, ICML, PriSC: conferences and workshops in which cited papers appeared
- PP: the class of problems a polynomial-time randomised machine decides correctly more than half the time; PP-hard problems are believed to be far harder than NP-complete ones
- PTX, PYC, RLE: NVIDIA's virtual GPU instruction set; Python compiled bytecode; run-length encoding
- RCE: the Reality Code Engine
- SHA-256, SIMD, SWAR, UTF: a cryptographic hash function; single instruction, multiple data; SIMD within a register; Unicode Transformation Format
- SMT: satisfiability modulo theories; SMT solvers such as Z3 decide logical formulas over arithmetic, bit-vectors and memory
- SSA: static single assignment form
- STG, Cmm: the intermediate representations of GHC after Core
- TIRx, JACC, NVIDIA, AMD, ONNX, BEAM, DIKU: project, company and system names used as they are
- TZ, UTC, LC_ALL: environment variables for the time zone and locale; UTC is Coordinated Universal Time
- UB: undefined behaviour, source behaviour to which the language standard gives no meaning
- UBC, MIT, ORNL, NERSC, LBNL, HPE: the University of British Columbia, the Massachusetts Institute of Technology, Oak Ridge National Laboratory, the National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory and Hewlett Packard Enterprise
- ULP: unit in the last place
- UNCOL: Universal Computer Oriented Language, a 1958 proposal for one intermediate language for all compilers
- WARP: the function-identification format of Binary Ninja, a reverse-engineering tool, used as a name

Evidence marks. Each external statement carries the mark of how it was checked, and each entry in the reference list repeats it. **[F]** means read in the source itself. **[S]** means seen only in a search-result snippet or a secondary summary. **[K]** means stated from memory and not re-checked; the canonical location is given but was not opened. **[M]** means measured by the project on its own machine. **[NOT FOUND]** means searched for with no source located. Only two external items reached [F]: an index page and a set of repository commit counts. Most hosts that hold the primary texts (among them arxiv.org, compcert.org, gcc.gnu.org, ziglang.org, zenodo.org, pytorch.org and the UBC library) could not be reached from the research sessions, so an [S] item below means "a source exists and its abstract or snippet says this", not "the paper was read".

1. Problem statement

A compiler takes a program in one language and produces a program in another, usually closer to a machine. There are thousands of compilers, built by communities that rarely share vocabulary: systems languages, scientific computing, legacy business languages, high-integrity languages, functional languages and the compilers for neural-network tensor programs. Each community has its own account of what a correct and good compiler is. RCE Compile asks whether one statement covers all of them, independent of RCE, and if not, what the nearest statement is that does.

The candidate under test came out of an earlier discussion with an assistant system (Grok). It says: **compilation is semantics-preserving lowering between canonical representations, ranked by a cost whose floor is Landauer's limit and whose target is Kolmogorov-minimal, gated by verification, and compounded by reuse.** Three questions follow. Does it describe the compilers that exist? If not, can it be restated so that it is true of a correct compiler and checkable against a real one? And can its physical and mathematical terms be measured at all? The discussion that proposed the candidate also made 55 factual claims about the field; those are checked in Appendix A.

2. The candidate and its refinement

2.1 Why the candidate fails as a description

The candidate makes three kinds of claim: about representation ("canonical"), about lowering ("semantics-preserving"), and about cost ("floor is Landauer, target is Kolmogorov-minimal"). Each fails as a description of shipping compilers, for reasons given in full in §3 and §4:

- **No compiler surveyed hashes a name-free intermediate representation.** Every one keys its identity and its caches on named or textual content.
- **Lowering is neither total nor proved to preserve meaning.** Fuzzers have found more than 325 bugs (Csmith) and 1,602 bugs (the EMI project) in production C compilers [1] [2], and Zig's default self-hosted backend fails 24 of its 2,008 behaviour tests [3].
- **No compiler accounts for energy.** Every cost model found is in cycles, bytes or node counts.
- **The convergence function cannot be computed or measured as stated.** Kolmogorov complexity is uncomputable; even the minimum heat of one fixed circuit is PP-hard to compute [4]; and today's hardware sits 10^6 to 10^{11} times above the Landauer floor.

A statement that every compiler is claimed to obey, and that real compilers measurably break, is falsified as a descriptive law. No peer-reviewed source in English, Chinese, Russian or Japanese was found that names or proves a single law of compilation [NOT FOUND]; an English search for the phrase returns only the copyright sense of "compilation" [5].

2.2 The refinement: a normal form with two bounds and a measured distance

The candidate holds up much better as a normative framework: a statement of what a correct, well-ranked compiler must satisfy, against which every real compiler can be placed and its shortfall measured. Three clauses change. "Semantics-preserving" becomes a **declared contract**, because the established notion is refinement under a precondition and the precondition differs by compiler. "Canonical" becomes an **operation** the framework applies to any representation, because no compiler supplies it. And "Landauer floor and Kolmogorov target" becomes **two computable counts plus a measurement**. The result is a normal form in the sense of Hoare, He and Sampaio (§6), with two bounds and a measured distance from them, not a law every compiler already obeys.

2.3 The normative statement

The project's charter states the standard as follows; the text is quoted from it unchanged [6]:

> **Compiler Form.** A compiler is a partial function from declared inputs (source, flags, toolchain, and any build path after remapping) to a chain of representations $R_0 \to \ldots \to R_n$. Each step either stops or produces an output that a named checker K accepts as preserving a declared contract: a property class, an equivalence level, a linking assumption and a trusted computing base. Among accepted outputs, a

stated cost C ranks the candidates. Reuse U is keyed by a name-free hash of the representation together with its contract. Nothing is released unless G records K's acceptance. For every released output, the framework reports its distance from two bounds, the bits irreversibly overwritten per run (scaled by $k_B \cdot T \cdot \ln 2$) and a stated description-length proxy, next to the energy measured in the regime the hardware belongs to.

A *stop* is a lowering step that ends with a diagnostic and no output. A *declared input* is an input the compiler's contract lists. The statement drops "holds for any compiler" in favour of "every compiler can be placed against it". The project claims a law only for what its registered hypotheses (§7) survive. §3 sets out the parts the statement names, and §4 the two bounds.

3. The Compiler Form, R to G

3.1 The seven parts

The project calls the framework the **Compiler Form**. It has seven parts:

Part	Name	What the part requires
R	Representation	A canonical IR with a stated equivalence relation and a name-free hash
L	Lowering	Each step either stops (ends with a diagnostic and no output) or produces an output that preserves a declared contract
C	Cost	A stated cost that ranks equivalent candidates
K	Checker	A named checker that certifies the contract
U	Reuse	Earlier accepted results are reused, keyed by the name-free hash and the contract
E	Energy	Energy is accounted for, by counting or by meter, for each released output
G	Gate	Nothing is released unless the gate records the checker's acceptance

Two refinements, from the correctness and ML-compiler literature, make L, K and G precise enough to test.

3.2 Four equivalence levels

ML compilers define correctness as closeness. XLA, TorchInductor, TVM, TensorRT, Triton and IREE all test against a reference within an absolute and relative tolerance (PyTorch's float32 defaults are a relative tolerance of 1.3×10^{-6} and an absolute tolerance of 10^{-5}) [K], because fusion, tiling, split reductions and tensor-core accumulation reorder floating-point sums, and lowering to FP8 or FP4 changes outputs on purpose [K]. FlexAttention's FlashAttention-4 backward pass is reported to be nondeterministic when block sparsity is on [7]. If "preserving" meant identical output bits, these compilers would all violate L. A September 2026 preprint argues that bit-level results depend on the accumulation structure the compiler chooses, reconstructs that structure for cuBLAS across three GPU generations, and gives static checkers that decide bitwise equivalence across tens of thousands of autotuner configurations [8]; MMA-Sim gives a bit-accurate model of tensor and matrix cores on ten architectures [9]. Halide already shows the pattern at small scale: a schedule cannot change results except through named reassociating reductions [K].

The Form therefore requires every lowering to declare one of four **equivalence levels**, and its checker to certify that level:

Level	Meaning	What K must certify
Exact	Integer, Boolean or fixed-point; identical output bits	Bitwise equality
Bitwise within an accumulation class	Floating point, with the reduction order recorded in R; L and C choose only within the class	Same class, then bitwise equality
Within a declared error bound	Floating point, with an absolute, relative or ULP bound	The bound, by static error analysis or interval methods, not by sample tests
Declared lossy	Quantisation or precision lowering	Nothing about equality: it is a change of program, recorded and admitted by G as a separate decision

The default is "exact unless a checked contract says otherwise". That is stricter than every production ML compiler and consistent with the bitwise-class research.

3.3 What a gate must name

The classical correctness theorem is not "the output means the same as the input". CompCert's main theorem is **behaviour refinement under a precondition**: if compilation succeeds and the source has no undefined behaviour, every observable behaviour of the compiled code is a behaviour of the source [K]. Patterson and Ahmed show that "the compiler is correct" is ambiguous until the theorem says what target code the output may be linked with [10]. Secure compilation adds that a compiler can preserve meaning in CompCert's sense and still let an attacker at the target level break abstractions of the source: *full abstraction* (source-equivalent components compile to target-equivalent ones) and the *robust property preservation* criteria are distinct, stronger guarantees [11]. And CompCert's proof leaves the preprocessor, assembler and linker outside it; Csmith found silent miscompilations exactly there and none in the verified middle end [1].

From these, every gate in the Form must state three things:

1. **The property class it preserves**: trace properties, hyperproperties, a robust variant, or an equivalence.
2. **The linking assumption**: which other code the output may be combined with.
3. **The trusted base**: the semantics, proof kernel, assembler, hardware models or diverse compiler the guarantee relies on.

Fully abstract compilation usually needs run-time checks or isolation, so a stronger property class raises cost [K].

On 2 November 2026, before any gate was measured, the project's charter added two statements to these three [6]:

4. **The authority of its evidence**: the level that the evidence behind the checker's acceptance must have.
5. **The digests its verdict is bound to**: the SHA-256 (the *digest*) of the declared inputs, of the output it releases and of the checker that accepted it, and the revision of the contract it checked. A verdict bound to a digest applies only to the exact bytes it was made on.

The charter takes the levels of authority from the reconstruction ledgers of REA (Reverse Engineer Anything), an open-source reverse-engineering project [6]. Two levels are *observed* evidence: *pinned*, in which the compiler, named by the SHA-256 of its toolchain, is run on the declared inputs and its output compared byte for byte; and *replay*, in which an output recorded earlier and bound by its SHA-256 is compared byte for byte. *Proved* evidence is a checker's certificate for every input of its fragment: a proof that a proof kernel checks, a translation validator's verdict, or an SMT solver's finding that the negated contract cannot be satisfied. *Derived* evidence follows from a stated rule that cites its source, such as a documented gate behaviour quoted with its passage. *Inferred* evidence is the statement of a model, a heuristic or a stochastic proposer, or a summary that cites no record of a higher level. A gate accepts an output only when evidence of the authority it states holds and is bound to every digest it states; a verdict bound to other bytes, or evidence that cannot be read, releases nothing. Inferred evidence never makes a gate accept. Observed evidence shows that a contract held, or failed, on the inputs that were run; only proved evidence certifies it for every input of a fragment. A gate that does not state all five is recorded as a gate with an unstated guarantee.

3.4 Evidence per part across compilers

The survey covered Zig, Rust, Go, GCC and Clang/LLVM, Swift, Cranelift and WebAssembly (Wasm), Julia, the Fortran compilers (gfortran, ifx, nvfortran, LLVM Flang, LFortran), COBOL (gcobol and GnuCOBOL), Ada and SPARK, GHC and the other functional compilers (OCaml, Lean 4, Idris 2, Koka, Erlang, Futhark, Halide), and the ML compilers. Figure 1 and the table below summarise the evidence part by part. Almost every cell rests on [K] or [S].



Figure 1. Where each part of the Compiler Form is found, by compiler family. Filled: present as the Form requires. Half: present in part or only by configuration. Empty: absent. Erlang's code validator, a real gate, sits inside the functional row. Qualitative summary of the survey in §3.4 and §3.5; almost every cell rests on [K] or [S] evidence.

Part	Supported by	Qualified by	Falsified by
R	E-classes; Wasm's name-free binary; exact text round-trip in LLVM IR and CLIF; normal forms (GHC Core, LFortran ASR, Lean LCNF)	Every compiler is a chain of 3 to 6 IRs; round-trip holds only between an IR and its own bytes	No compiler hashes a name-free IR
L	Morris's commuting square; CompCert; CakeML; Halide's schedule separation	Refinement, not equality; only for UB-free sources; indexed by linking context; ML compilers use tolerance	Csmith, EMI and Alive2 bug counts; Zig's failing behaviour tests
C	Cranelift e-graph extraction; equality saturation; autotuners	Mostly local heuristics in cycles or bytes	No cost in joules; shortest is not cheapest
K	Type and borrow checkers; IR verifiers; Alive2; ISLE verification; the Lean kernel; SPARK	Checks the source more than the lowering; often debug-only; bounded; ML checks are sample tests	No production compiler checks all of its lowering
U	Content-addressed caches in Go, rustc, LLVM, Wasmtime, JAX, Triton, TVM; CakeML bootstrapping; DDC	Keys include names and paths; TensorRT-LLM dropped its stored engines	Name-free reuse is absent
E	Only research (Eel, MIT 2016) and external meters	Energy roughly tracks run time on CPUs	Absent in every compiler surveyed
G	rustc, Go, Wasm validation, Erlang's validator, SPARK when a project requires it	tsc emits despite type errors by default; LLVM and GCC gate on post-commit testing; ML gates are fallbacks	No universal gate

3.5 Part-by-part findings

R is absent everywhere as the Form states it. Every production compiler lowers through a chain of IRs rather than one canonical form. Zig goes AST → ZIR → AIR → per-backend MIR [3]; rustc goes AST → HIR → THIR → MIR → one of LLVM IR, Cranelift's CLIF or GCC's GIMPLE [12]; Go goes syntax → IR → SSA → Plan 9-style instructions [13]; GHC goes Core → STG → Cmm [K]. Canonical *forms* exist (GHC Core, LFortran's ASR, Futhark's SOACS, Lean's LCNF, Flang's FIR and HLFIR), and the e-classes of an e-graph are canonical *sets* of equivalent terms. The closest result to the "canonical representations" clause is Bryant's thesis, which treats compilation as normalisation to a normal form [14]. Two qualifications follow. Morris's correctness square needs only *some* semantics on each side, not canonicity [K]. And round-trip holds only between an IR and its own serialised form (LLVM's text and bitcode, CLIF text, MLIR's generic form), never back to source, because every compiler discards comments, names and syntactic sugar [K]. The falsifier is plain: GHC's interface fingerprints, OCaml's interface digests, rustc's stable hashes, Go's action IDs, Julia's source hashes, Lean's Lake traces and Taichi's kernel hashes are all hashes of named or textual content [K]. The Wasm binary comes closest, because names sit in an optional custom section [K]. So R has to become an operation the framework applies to any IR (rename bound names to positions, strip paths, then hash), not a property compilers are claimed to have. The project's experiment E5 implements that operation in two forms and measures them on a small corpus (§8.3).

L fails as a description; the bug counts are the evidence. Csmith found more than 325 previously unknown bugs in C compilers [1]. EMI found 147 confirmed bugs in its first 11 months and 1,602 GCC and LLVM bugs in total, 1,007 of them fixed [2]. Alive2, a bounded translation validator for LLVM, found 47 new bugs (28 fixed) and prompted 8 patches to the LLVM language reference; a follow-up encoding of LLVM's memory model found 21 more [15]. Zig's self-hosted x86-64 backend, the default for Debug builds since 0.15, passes 1,984 of 2,008 behaviour tests (the project's development log gives 1,987 against 1,980 for LLVM; the two sources disagree) [3], so lowering is not total in a shipping compiler. CompCert and CakeML show that L can hold as a theorem: CakeML carries its guarantee down to machine code for five instruction sets and has compiled itself inside the HOL4 logic [16]. Neither is a production compiler for the languages surveyed. L survives only as a norm: a step either produces output its checker accepts or stops, and production compilers are measured by how far they fall short.

C is present but local. Every compiler has cost models, but almost all are local heuristics: inline budgets, LLVM's target cost queries and scheduling models, GCC's cost hooks [K]. Cranelift's acyclic e-graph is the only production case found that ranks the members of an explicit equivalence class and extracts the cheapest, and it deliberately saturates once, within a budget [17]. Research equality saturation shows uneven gains: TENSAT reports up to 16% [18], and equality saturation inside XLA averages 3.45% with a 56.26% peak on one model [19]. Only one mechanised result about a pass's *performance* guarantee was found, for an inlining heuristic in Rocq [20]. Every cost found is in cycles, latency, bytes or node counts; none is in joules, and the shortest program is not the cheapest whenever unrolling, vectorising or table lookup makes code longer but faster [K].

K is the strongest part, but it guards the source more than the lowering. Type systems, Rust's borrow checker and new trait solver [21], GHC's Core Lint, LFortran's ASR verifier, MLIR op verifiers, Futhark's typed IR, Lean's kernel, SPARK's GNATprove proof levels [22] and Erlang's code validator [K] are all real checkers. Three qualifications matter. Checking of the lowering itself exists only in pieces: Alive2 for some LLVM peepholes, SMT verification of some Cranelift ISLE rules, the mechanised Wasm specification [K]. Many IR checkers are debug-only: Core Lint is opt-in and the LLVM verifier between passes is commonly off in release builds [K]. And bounded checkers can miss bugs: Alive2 unrolls loops to a bound [15], and EMI and Csmith find bugs but cannot certify their absence. The ML compilers' checks are tolerance tests on sample inputs, which are tests, not certificates.

U is the part most often present, but its keys hash names and paths. Go's build cache maps an action ID (a hash of every input to a step) to an output [23]; rustc uses incremental fingerprints; LLVM's ThinLTO cache, Wasmtime's module cache, JAX's compilation cache, Triton's kernel cache and TVM's tuning database are all keyed by hashes [K]. Reuse of verified work compounds trust: CakeML compiled its own verified binary inside the logic [16]; "Compiling by Proving" turns proofs into optimised rewrite rules [24]; and DDC counters the trusting-trust attack under stated assumptions [25]. Two qualifications remain. The keys hash names and paths, so equal programs with different names miss the cache. And TensorRT-LLM abandoned its serialised, verified-once engines for PyTorch as its only backend [26]: stored artefacts lost to a flexible eager path when hardware and models changed faster than the artefacts could be rebuilt.

E is absent everywhere. No compiler in any of the twelve families accounts for energy [K; NOT FOUND for any counterexample]. Energy is measured outside compilers: by machine (the June 2026 Green500 leader runs at

73.28 GFlops per watt [27]) and by counters such as RAPL and NVML [K]. Research on energy-aware compilation is modest: the choice of compiler flags shifts energy by tens of percent, and energy and run time are "roughly correlated, though not always" [28]; a 2025 Fortran study found "minimal difference" in energy between gfortran, ifx and ifort, tracking run time [29]. The only compiler found that treats energy as a first-class quantity is the research language Eel, which treats reversible operations as free and charges for bit creation and destruction [30]. Either E is not a necessary part of a law of compilation, or the whole field falls short of it. This study takes the second reading: E is a part the framework adds, supplied by an external meter.

G is present only in some compilers. rustc emits no code after a type or borrow error; Go refuses unused imports and variables; Wasm engines refuse an invalid module at load time; Erlang's validator checks generated code before writing it [K]. SPARK is a gate only when a project's CI requires GNATprove with no unproved checks; the compiler itself does not refuse [K]. Elsewhere G is off or partial. tsc emits JavaScript despite type errors unless `noEmitOnError` is set; LLVM and GCC ship from trunk on post-commit testing and reverts rather than a gate before merging; Zig analyses lazily, so code that is never referenced is never checked [K]. ML compilers have fallbacks rather than gates: a graph break in TorchDynamo falls back to eager Python unless `fullgraph=True` turns it into an error, and ONNX Runtime sends unsupported nodes to its CPU provider [K].

4. The convergence function

4.1 Landauer and Kolmogorov as conceptual bounds

The candidate's convergence function says that the cost of a compiled program has a floor at Landauer's limit and a target at Kolmogorov-minimal size. **Landauer's limit** is the minimum heat, $k_B T \ln 2$, released into surroundings at temperature T when one bit is erased: about 2.871 zeptojoules at 300 K (2.853 at 298.15 K) [K, arithmetic]. It bounds heat per bit *erased*, not per operation, and it is reached only in the infinitely slow limit; erasing faster costs more, roughly as $1/\tau$ for erasure time τ [K]. Single-bit experiments confirm the bound at that limit, in colloidal beads (Bérut et al., 2012) and in nanomagnetic memory (Hong et al., 2016) [31]. **Kolmogorov complexity** $K(x)$ is the length of the shortest program that prints x. Both are real physics or mathematics, and as bounds they say something true: no compiled program can dissipate less than its erasures require, and none can be shorter than the shortest equivalent program.

4.2 Why neither is computable or measurable as stated

The Kolmogorov target cannot be computed. $K(x)$ is uncomputable (Kolmogorov, Chaitin, Solomonoff); compressors give only upper bounds, and program equivalence, the constraint the target sits under, is undecidable in general by Rice's theorem [K]. The full-employment theorem for compiler writers makes this concrete: no compiler can always produce the smallest equivalent program, because that would decide halting [K]. Superoptimisers approach the target for short loop-free fragments under an SMT or test-vector check, but never with a guarantee.

The Landauer floor cannot be computed either, and it is far below today's costs. Computing the optimal heat functions for one fixed circuit is PP-complete according to the abstract of a 2026 paper, multiplicative approximation is NP-hard, and only additive approximation has a polynomial-time randomised algorithm [4]; the body reportedly argues PP-hardness, so membership in PP remains to be checked. Choosing among equivalent circuits, which is the compilation problem, sits on top of that. At the commonly quoted 45 nm figures, a 32-bit integer add costs about 0.1 picojoules, **about** 3.5×10^7 **times** $k_B T \ln 2$, or about 10^6 per erased bit if 32 bits are overwritten; a 640-picojoule DRAM read is **about** 2.2×10^{11} times the floor; and the overhead of one out-of-order instruction, about 70 picojoules, is **about** 2.5×10^{10} [32]. Today's gap is therefore 10^6 to 10^{11}, and almost all of it is wires, memory and control, not logical erasure.

The two terms overlap. Bennett showed that any computation can be made logically reversible, so the Landauer floor applies only to information a program deliberately discards [K]. Zurek, and Bennett and colleagues, showed that the minimum thermodynamic cost of erasing a string x is about $k_B T \ln 2 \cdot K(x)$, and that converting x to y costs about their information distance [K]. The Landauer cost of erasure is already a Kolmogorov quantity, so minimising both jointly counts the same thing twice. Three further problems remain. Shortness and low energy trade off, so a joint objective without a stated weighting is a Pareto front, not a function. Time is missing from the statement. And the energy spent compiling can outweigh the savings: in an October 2026 study, 42 of 44 configurations of model self-refinement used 4% to 886% more execution energy than the reference solutions, and the two exceptions broke even only after about 10^6 to 10^7 executions [33].

One result supports the idea at its core. Wolpert and Kolchinsky's "circuit Landauer loss" shows that two circuits computing the same function can have different minimum dissipation [34], and mismatch-cost results give strictly positive lower bounds for circuits [35]. Circuit structure, which is what a compiler chooses, does change the physical floor.

4.3 The measurable replacement

The replacement keeps both terms and makes each computable:

1. ε**, the erasure count**: the number of bits a run overwrites irreversibly, computed from the IR (bits overwritten or discarded without having been copied out). $\varepsilon \cdot k_B T \ln 2$ is the run's Landauer bound.
2. λ**, a declared description-length proxy**: IR node count, binary bytes, compressed bytes, or an MDL code length relative to a named model class. The proxy is declared before a measurement and not changed after it.

Both are computed under an equivalence checked on a decidable fragment, and reported next to **measured joules** in the energy regime the hardware belongs to:

Regime	Energy is proportional to	Example hardware
A	Counted operations	In-order cores, FPGAs
B	Time multiplied by power	Out-of-order CPUs, GPUs
C	Entropy production	Thermodynamic hardware, or a simulation of it; the only regime in which Landauer units apply directly

Regimes A and B are approximations: static power scales with time even on in-order cores [K]. The claim becomes testable: does lowering ε and λ predict lower measured energy in each regime? No published measurement of the erasure content of real compiled programs against their energy was found [NOT FOUND], so the result would be new evidence whichever way it comes out. Self-published preprints that claim convergence to the Landauer limit at a rate of $O(1/\sqrt{N})$ are unreviewed, and their rate is a statistic about estimating work, not a rate at which programs approach a floor [36]; they are not cited as support.

5. The gear ladder and the 98/2 split

RCE Compile orders its own compilation steps as a **gear ladder** of six rungs, G0 to G5, tried in order of cost. G0 reuses a stored result by its hash; G1 applies rewrite rules proved correct in advance; G2 extracts the cheapest form from an e-graph. These three are deterministic. G3 to G5 are stochastic proposers, and every output they produce must pass the gate before it is released; the charter defines G3 to G5 only as a group, and their separate definitions are open [6]. (The rungs are written G0 to G5; the Form's gate is the part named G.) The design principle attached to the ladder is a split: **at least 98% of the work deterministic, at most 2% stochastic, and everything gated.**

No source measures such a split for any compiler, by decision count, CPU time, cost or energy [NOT FOUND]. The verified systems sit at the two ends instead. Equality saturation, Souper, Cranelift and MLGO are fully deterministic at compile time; MLGO's policy is fixed and compiled into LLVM, and it decides one class of choices (inline or not, which register to evict) for gains of 0.3% to 7% [37]. At the other end, a stochastic author writes nearly everything behind a deterministic gate: in work reported in October 2026, a model wrote about 180 million lines of Rocq proof that the Rocq checker accepts or rejects in full [38], and AdaCore's GNAT Foundry: Intersection demonstrator has an agent write SPARK while deterministic tools produce all the release evidence [39]. By authoring effort that split is inverted, while the gate is total, so any claim about 98/2 must first say what is being counted. One taxonomy places hybrid workflows at 14% to 56% deterministic, but it is a classification, not a measurement [40].

The literature does support the two principles beneath the split. **Gate everything**: SuperCoder's best configuration still fails 5% of its test-checked outputs, and the strongest commercial baseline in that paper passes only 51.5% [41]; STOKE paired a stochastic search with an SMT validator [K]. **Put deterministic reuse first**: Compiled AI moves all stochastic work into a one-time compile phase and breaks even after about 17 runs [42], and Progressive Crystallization promotes recurring agent runs into deterministic workflows [43]. One caution carries over: in Veritas and CktFormalizer the gate checks the stochastic output against its own specification (CNF clauses, hardware types), while an error in the specification sits outside the gate [44] [45].

The 98/2 split is therefore a registered hypothesis, listed in §7, not a measured result. It is to be measured rung by rung with a declared unit and recorded by the gate. Candidate units are accepted decisions, accepted rewrites, CPU seconds and joules; each is reported separately, because they can disagree.

6. Prior art

No earlier source found names a single law of compilation (§2.1). The closest earlier claims each cover a few of the seven parts.

- **Morris (1973)**, following Burstall and Landin (1969) and refined by the ADJ group (Thatcher, Wagner and Wright, 1979 to 1981), made compiler correctness a commuting square of algebra homomorphisms: compiling then interpreting equals interpreting then encoding [46] [47]. That is L and K. Lowering as stated in the Form restates it and claims no novelty there.
- **Futamura (1971)** showed that specialising an interpreter on a program yields a compiled program, that specialising the specialiser on the interpreter yields a compiler, and that specialising it on itself yields a compiler generator [48]. **Ershov's** Novosibirsk school reached the same view as "mixed computation" [49]. That is L and U, and the strongest prior claim that compilers are derived rather than designed.
- **Hoare, He and Sampaio (Acta Informatica, 1993)** treated compilation as reducing a source program, by algebraic laws of programming, to a normal form that mirrors the target machine, with correctness following from the laws; the work fed into Hoare and He's *Unifying Theories of Programming* (1998) [50]. This is the closest prior work to calling compilation "governed by laws", and the closest to the normal-form framing this study adopts.
- **Cousot and Cousot (POPL 1977)** unified static analyses as fixpoints over lattices, and later extended this to program transformations [51]. That is K, and the precision ordering in C.
- **Eel (Tyagi, Lynch and Demaine, MIT, 2016)** is the only compiler found that ties code to an energy floor, by charging for bit creation and destruction [30]. That is E.

UNCOL (1958) tried and failed to give one universal R; MLIR answers that failure with many levels but holds no semantics [52]. E-graphs already combine canonical sets, cost-ranked extraction and sound rewrites in one structure [53].

Two primary sources have not yet been read in full: Hoare, He and Sampaio (1993) [50] and Ershov's "On the essence of compilation" (1977) [54]. Both are cited from memory [K]. Either may contain a precedent for the framing or the wording, and both must be read before RCE Compile claims priority for anything. The novel part that remains, on present evidence, is the combination: one statement binding a physical floor and an information-theoretic target to verified, reuse-compounding lowering, with energy as a ranked quantity.

7. Hypotheses and falsifiers

The project registered seven hypotheses on 1 November 2026, before any measurement of its own, each with a metric, a condition for support, a condition for falsification and an experiment [6]. The statements and falsifiers below are quoted from the registration. A *declared corpus* is a set of programs fixed, by path and hash, before the measurement that uses it. A change after registration is recorded as a dated row in the registration's table of changes. Besides the registration itself, that table holds five changes, all dated 2 November 2026: a verdict on the one E1 difference in Go that survived remapping on Linux, which adds the import path to H1's declared inputs (§8.1); a record that the first macOS run of E1 has no verdict yet (§8.1); a change to the metrics of H2 and H3, which now record the authority of each gate's evidence and the digests its verdict is bound to (§3.3, §8.2), made before either experiment has run; the attachment of experiment E5, with its predictions and falsifiers, to the fifth hypothesis below, before E5's first run (§8.3); and E5's first results, which support that hypothesis on E4's corpus by its registered conditions (§8.3).

	Statement	Falsified if	Experiment
H1	Lowering is a function of declared inputs. With declared inputs (source, flags, toolchain, remapped build path, fixed `SOURCE_DATE_EPOCH`), each compiler's output bytes are identical across directories and across machines.	Any byte difference survives remapping, across directories or across machines.	E1, run (§8.1)
H2	Gates can be classified. Gates differ by compiler and can be classified: some emit nothing by default when a check fails, others emit unless configured not to.	A compiler's documented gate behaviour differs from what the injected-fault test observes.	E2, protocol (§8.2)
H3	Every pass and flag has an equivalence level. Every optimisation pass and flag can be assigned one of the four equivalence levels.	A pass's observed output change exceeds its assigned level.	E3, protocol (§8.2)
H4	Erasure and description length against measured energy. The counted irreversible erasure ε plus a declared description-length proxy λ predict measured energy in regime C (or a simulation of it), but explain less variance than operation counts in regime A and than time multiplied by power in regime B.	ε and λ explain no variance in regime C, or explain more variance than the regime's own baseline in A or B, which would contradict the scale-gap analysis of the charter.	E4, erasure counts + Regime-B Linux RAPL `reported_j` & payback horizons on 20 kernels (§8.2); Regimes A and C open
H5	A name-free hash raises reuse without false hits. A name-free hash (bound names replaced by positions, path removed) raises reuse hits on renamed but equal programs, without false hits.	There is any false hit, or no gain in hits on the declared corpus.	E5, supported on E4's corpus and scaled to 245 programs + 500 verified semantic mutants (§8.3)
H6	Deterministic rungs decide at least 98%. The deterministic rungs G0 to G2 decide at least 98% of accepted decisions on a declared corpus, and every stochastic proposal is gated.	The measured stochastic share is above 2% in the declared unit, or any stochastic output is released without a recorded gate decision.	E6, planned; needs a compiler built on the gear ladder, which does not exist yet
H7	Partial lowering with stops admits no miscompilation inside its fragment. Lowering stated as partial, with a stop as an outcome, has a measurable stop rate, and gated lowering admits no miscompilation that EMI-style testing finds inside the checker's decidable fragment.	EMI-style testing finds a miscompilation inside the fragment the gate claims to cover.	E7, planned; no protocol yet

The conditions for support are the converse of each falsifier, with these specifics from the registration: H1 is supported if the differing-byte count is 0 for every output file in every pair; H2 if every compiler falls into one class and its observed behaviour matches its documented behaviour; H4 if ε and λ explain variance in regime C above a threshold declared in the E4 protocol, and less than the baseline in regimes A and B; H6 if the stochastic share is at most 2% in the declared unit and every stochastic output has a recorded gate decision; H7 if the stop rate is measured and no such miscompilation is found.

8. Experiments

8.1 E1: determinism of lowering, measured

Method. E1 tests H1 [6]. Each *configuration* is one compiler with one flag set, and names one or two small programs. For every program, the harness makes four pairs of builds, each build in a fresh directory that holds only the copied source: the same directory twice; two directories with absolute paths of different lengths; the same two directories with the toolchain's documented path remapping; and the same directory twice with `SOURCE_DATE_EPOCH` unset, at least 1.1 s apart. All builds but the last pair run with a declared environment (`LC_ALL=C`, `TZ=UTC`, `SOURCE_DATE_EPOCH=1735689600`), and every build gets new, empty compiler caches, so the second build of a pair is a real rebuild. For each pair the harness records, per output file, the number of byte positions at which the two files differ, and, for an ELF file, how many of those bytes fall in each section. The count is positional, so a path of another length that shifts the rest of a file counts every shifted byte. The remappings are `-ffile-prefix-map` for gcc, g++,

clang, clang++, gfortran and gnatmake, `--remap-path-prefix` for rustc, `-trimpath` for Go, `+deterministic` for erlc and `BUILD_PATH_PREFIX_MAP` for ocamlopt; Go 1.21 was described by its developers as "the first Go toolchain with perfectly reproducible builds" [55]. Julia has no remapping in the matrix: its remapped run writes the code without source lines. No remapping was found for Lean, `wasmtime compile`, Zig, GHC or GnuCOBOL, so their remapped comparison is not tested. `go build` is measured three times: in module mode, Go's default; in GOPATH mode with the source file named on the command line; and in GOPATH mode with the source placed in a `GOPATH` and built by its import path, `e1`. For each Go configuration the harness runs the remapped build once more in each directory with `GODEBUG=gocachehash=1`, which makes `go build` print every input it hashes into each action ID, and records the lines that name the build directory. The harness also records the SHA-256 of every output, so that runs on two machines can be compared where a tool's version string is the same on both. Results are written by the harness, and the write-up is generated from them by a script; no figure in it is typed.

Machines and toolchains. Three runs on 2 November 2026 (UTC). Two are on Linux x86-64 with 4 CPUs and Ubuntu 24.04.5. The *local machine* (Intel Xeon at 2.80 GHz) has gcc, g++ and gfortran 13.3.0, clang and clang++ 18.1.3, rustc 1.97.0 (including its `wasm32-wasip1` target), go 1.24.7, Lean 4.12.0 and Wasmtime 26.0.1. The *CI runner*, a GitHub-hosted runner (Intel Xeon Platinum 8573C) on which a workflow in the repository runs the same harness, has gcc, g++, gfortran and gnatmake 13.3.0, clang and clang++ 18.1.3, rustc 1.99.1, go 1.24.13, Zig 0.17.0, GHC 9.14.1, GnuCOBOL 3.1.2.0, OCaml 4.14.1, Erlang/OTP 25 and Julia 1.13.2; Lean and Wasmtime are not installed there. The third, the *macOS laptop*, is recorded in the repository's table of changes as a MacBook Pro (M5 Max) with macOS 27.2 on arm64; its results file records 18 CPUs and neither the CPU model nor the system release. On it `gcc`, `g++`, `clang` and `clang++` all run Apple clang 21.0.0, beside gfortran 16.2.0 (Homebrew), rustc 1.99.1 (Homebrew), go 1.27.3, Lean 4.35.0, Wasmtime 50.0.1, Zig 0.17.0, GHC 9.14.1, GnuCOBOL 3.2.0, OCaml 5.5.2 and Julia 1.13.3; `gnatmake`, `erlc` and rustc's WebAssembly target are not installed there. Its executables and object files are Mach-O files, not ELF, and a `-g` build of C, C++, Fortran or Rust leaves its debug information in a separate `.dSYM` bundle beside the executable. The file was written before the harness could attribute differing Mach-O bytes to regions of the file, so for this run E1 records how many bytes differ but not where.

Results, local machine. Fifteen configurations on 28 programs in all (two per configuration, one for Lean and one for Wasmtime).

Configuration	Same directory twice	Two directories	Two directories, remapped	Same directory, clock free
`gcc -O2 -g` (C)	identical (2 of 2)	differs (up to 2,131 bytes)	identical	identical
`gcc -O2` (C)	identical	identical	identical	identical
`clang -O2 -g` (C)	identical	differs (up to 2,191 bytes)	identical	identical
`g++ -O2 -g` (C++)	identical	differs (up to 20,602 bytes)	identical	identical
`clang++ -O2 -g` (C++)	identical	differs (up to 91,041 bytes)	identical	identical
`rustc -O -g` (Rust)	identical	differs (up to 2,441,330 bytes)	identical	identical
`rustc -O` (Rust)	identical	identical	identical	identical
`rustc` to WebAssembly, `-O -g`	identical	differs (up to 1,213,811 bytes)	identical	identical
`go build` (module mode)	identical	differs (up to 741,728 bytes)	identical	identical
`go build`, GOPATH mode	identical	differs (up to 743,688 bytes)	**differs (58 and 59 bytes)**	identical
`go build`, GOPATH mode, import path declared	identical	differs (up to 1,058,112 bytes)	identical	identical
`gfortran -O2 -g` (Fortran)	identical	differs (up to 2,368 bytes)	identical	identical
`gfortran -O2` (Fortran)	identical	identical	identical	identical
`lean -c` (Lean 4, 1 program)	identical	identical	not tested	identical
`wasmtime compile` (Cranelift, 1 program)	identical	identical	not tested	identical

Results, CI runner. Twenty-one configurations on 34 programs in all.

Configuration	Same directory twice	Two directories	Two directories, remapped	Same directory, clock free
`gcc -O2 -g` (C)	identical (2 of 2)	differs (up to 2,159 bytes)	identical	identical
`gcc -O2` (C)	identical	identical	identical	identical
`clang -O2 -g` (C)	identical	differs (up to 2,129 bytes)	identical	identical
`g++ -O2 -g` (C++)	identical	differs (up to 20,635 bytes)	identical	identical
`clang++ -O2 -g` (C++)	identical	differs (up to 91,003 bytes)	identical	identical
`rustc -O -g` (Rust)	identical	differs (up to 2,449,365 bytes)	identical	identical
`rustc -O` (Rust)	identical	identical	identical	identical
`rustc` to WebAssembly, `-O -g`	identical	differs (up to 1,249,142 bytes)	identical	identical
`go build` (module mode)	identical	differs (up to 1,061,816 bytes)	identical	identical
`go build`, GOPATH mode	identical	differs (up to 1,061,871 bytes)	**differs (59 and 60 bytes)**	identical
`go build`, GOPATH mode, import path declared	identical	differs (up to 751,516 bytes)	identical	identical
`gfortran -O2 -g` (Fortran)	identical	differs (up to 2,403 bytes)	identical	identical
`gfortran -O2` (Fortran)	identical	identical	identical	identical
`zig build-exe` (Zig, 1 program)	**differs (1,488,011 bytes)**	differs (3,790,312 bytes)	not tested	differs (1,858,521 bytes)
`zig build-exe -O ReleaseSmall` (Zig, 1 program)	identical	identical	not tested	identical
`ghc -O1` (Haskell, 1 program)	identical	identical	not tested	identical
`gnatmake -O2 -g` (Ada, 1 program)	identical	differs (26,349 bytes)	**differs (12,129 bytes)**	differs (3 bytes)
`cobc -x` (COBOL, 1 program)	**differs (28 bytes)**	differs (11,891 bytes)	not tested	differs (25,886 bytes)
`ocamlopt -g` (OCaml, 1 program)	identical	differs (164,436 bytes)	identical	identical
`erlc` (Erlang, 1 program)	identical	differs (197 bytes)	identical	identical
Julia `code_llvm` and `code_native` (1 program)	**differs (168 bytes)**	differs (144,945 bytes)	**differs (146 bytes)**	differs (168 bytes)

Results, macOS laptop. Twenty configurations on 32 programs in all; `gcc` and `g++` are Apple clang here.

Configuration	Same directory twice	Two directories	Two directories, remapped	Same directory, clock free
`gcc -O2 -g` (C)	**differs (up to 101 bytes)**	differs (up to 1,317 bytes)	**differs (up to 1,316 bytes)**	differs (up to 103 bytes)
`gcc -O2` (C)	identical (2 of 2)	identical	identical	identical
`clang -O2 -g` (C)	**differs (up to 38 bytes)**	differs (up to 1,318 bytes)	**differs (up to 1,319 bytes)**	differs (up to 101 bytes)
`g++ -O2 -g` (C++)	**differs (up to 38 bytes)**	differs (up to 126,164 bytes)	**differs (up to 126,165 bytes)**	differs (up to 134 bytes)
`clang++ -O2 -g` (C++)	**differs (up to 38 bytes)**	differs (up to 126,165 bytes)	**differs (up to 126,164 bytes)**	differs (up to 135 bytes)
`rustc -O -g` (Rust)	identical	differs (up to 105,506 bytes)	**differs (up to 105,506 bytes)**	identical
`rustc -O` (Rust)	identical	identical	identical	identical
`go build` (module mode)	identical	differs (up to 1,089,596 bytes)	identical	identical
`go build`, GOPATH mode	identical	differs (up to 1,089,588 bytes)	**differs (119 and 120 bytes)**	identical
`go build`, GOPATH mode, import path declared	identical	differs (up to 1,089,997 bytes)	identical	identical
`gfortran -O2 -g` (Fortran)	**differs (up to 38 bytes)**	differs (up to 1,015 bytes)	**differs (up to 1,014 bytes)**	differs (up to 103 bytes)
`gfortran -O2` (Fortran)	identical	identical	identical	identical

Configuration	Same directory twice	Two directories	Two directories, remapped	Same directory, clock free
`lean -c` (Lean 4, 1 program)	identical	identical	not tested	identical
`wasmtime compile` (Cranelift, 1 program)	identical	identical	not tested	identical
`zig build-exe` (Zig, 1 program)	**differs (402 bytes)**	differs (28,099 bytes)	not tested	differs (396 bytes)
`zig build-exe -O ReleaseSmall` (Zig, 1 program)	identical	identical	not tested	identical
`ghc -O1` (Haskell, 1 program)	identical	identical	not tested	identical
`cobc -x` (COBOL, 1 program)	identical	identical	not tested	differs (50 bytes)
`ocamlopt -g` (OCaml, 1 program)	identical	differs (669 bytes)	**differs (669 bytes)**	identical
Julia `code_llvm` and `code_native` (1 program)	**differs (148 bytes)**	differs (60,594 bytes)	**differs (114 bytes)**	differs (100 bytes)

Every cell marked "identical" holds for all programs of its configuration; every cell marked "differs" holds for every program of its configuration. A count is the differing bytes of one program, summed over its output files where it has several.

Findings.

1. **Same directory.** On the local machine no configuration differed between two builds in the same directory, with the clock fixed or free. On the CI runner three did, with the clock fixed: `zig build-exe` in its default mode, mostly in debug-information sections (`zig build-exe -O ReleaseSmall` was identical); `cobc -x`, in 28 bytes (20 of them in `.note.gnu.build-id`); and the Julia run, in 168 bytes of one LLVM IR file. For these three configurations the output is not determined by the inputs E1 declares. On the macOS laptop seven did: every `-g` configuration of C, C++ and Fortran, in 37 or 38 bytes of the executable at equal size (in 85 bytes of the executable and 16 of the DWARF file for `gcc -O2 -g` on `hello.c`); `zig build-exe` in its default mode, in 402 bytes; and the Julia run, in 148 bytes. `rustc -O -g` and the other twelve configurations were identical.
2. **Two directories.** On all three machines, every configuration that differed between two directories also had the build directory written into its output. The configurations identical across directories without remapping were `gcc -O2`, `rustc -O` and `gfortran -O2` on all three machines, Lean and Wasmtime on the local machine and the macOS laptop, `zig build-exe -O ReleaseSmall` and `ghc -O1` on the CI runner and the macOS laptop, and `cobc -x` on the macOS laptop. For the others, the build path is an undeclared input by default.
3. **Two directories, remapped.** On the local machine, remapping removed every difference except in Go's GOPATH mode, which still differed in 58 and 59 bytes, with files of equal size and no copy of the build directory left in them. On the CI runner the same configuration differed in 59 and 60 bytes, and two more configurations differed after remapping: `gnatmake -O2 -g`, whose remapped outputs still hold two copies of the build directory and differ in its two `.ali` files; and the Julia run, whose remapped run differs in 146 bytes of one LLVM IR file with no copy of the build directory left. In GOPATH mode with the import path declared, the remapped builds were identical across directories on both Linux machines. On the macOS laptop, remapping removed no copy of the build directory from the `-g` builds of C, C++, Fortran and Rust: three copies are counted in each program's outputs with and without remapping, and the differing byte counts with remapping are within one byte of those without. Go's GOPATH mode differed after remapping in 119 and 120 bytes at equal size with no copy of the build directory left, `ocamlopt -g` in 669 bytes with one copy, and the Julia run in 114 bytes with none; Go in module mode, and in GOPATH mode with the import path declared, was identical across directories when remapped.
4. **Clock.** Only `gnatmake -O2 -g` on the CI runner and `cobc -x` on the macOS laptop were identical in the same directory with `SOURCE_DATE_EPOCH` set but not with it unset, in 3 and 50 bytes; for them the clock is an input.
5. **Across machines.** Between the local machine and the CI runner, only configurations whose tool version string is the same on both are compared: the gcc, g++, clang, clang++ and gfortran configurations, seven configurations on 14 programs, by the SHA-256 of every output of one build on each machine, in directories with different absolute paths. With remapping, gcc, g++ and gfortran outputs were identical across the

two machines. `clang -O2 -g` and `clang++ -O2 -g` differed across machines for all four of their programs even with remapping, although on each machine their remapped builds were identical across directories. Go and rustc were not compared, because their version strings differ between the two machines, and the other toolchains were installed on only one of the two. The macOS laptop shares a version string with the CI runner only for Zig 0.17.0 and GHC 9.14.1, and with the local machine for no tool. The Zig and GHC outputs differ between the CI runner and the laptop, which build for different systems and instruction sets (Linux on x86-64, macOS on arm64), so that comparison does not test H1 across machines.

What this means for H1. The Go GOPATH-mode result is a byte difference that survives remapping across directories, and the project's repository records a verdict on it in the registration's table of changes, dated 2 November 2026 [6]. Its cause is named. Every differing byte lies in the `.note.go.buildid` and `.note.gnu.build-id` sections, and code and data are identical. The `gocachehash` trace shows the compile and link action IDs hashing the build directory, because GOPATH mode gives a package named by a file outside every `GOPATH` the import path `_` followed by its absolute directory, and `-trimpath` does not remove it; once the directory is written as a placeholder, the hashed inputs of the two directories are the same. With only that input changed, the package placed in a `GOPATH` and built by its import path, no hashed input names the directory and the remapped builds are byte-identical across directories, while the original configuration still differs. The verdict applies the registered conditions: the falsifier is a difference that survives remapping when the declared inputs are equal, and here they were not equal, since the import path is an input that `go build` derives from the directory. The result is therefore not a falsifier of H1. The import path can be named and controlled, so it is added to H1's declared inputs, and H1 holds for Go in GOPATH mode with the import path named; for a GOPATH-mode build of a file outside every `GOPATH`, `-trimpath` alone is not a remapping.

The other differences that survive remapping have no recorded verdict. On the CI runner, `gnatmake -O2 -g` and the Julia run differ across directories after remapping, and E1 does not establish whether a declared input differs between the two builds. Across machines, the `clang -O2 -g` and `clang++ -O2 -g` differences after remapping are differences between builds with the same tool version string; E1 records no trace of their cause, and the repository has not recorded a verdict on them. For the macOS laptop, the table of changes records that the run has no verdict yet, and what it establishes [6]. Its remapped builds named the build directory by a path the tools did not record: the system temporary directory, under `/var/folders`, is reached through the symbolic link `/var` to `/private/var`, and Go's trace names the directory with the `/private` prefix. A check on Linux of gcc, clang and rustc found that a remapping that names a linked path leaves the real path in the output. The macOS remapped comparisons therefore do not test H1's remapping condition. Since that run, E1 names the build directory by its real path, attributes differing Mach-O bytes to regions, and adds three configurations that run only on macOS: two in which clang compiles and links in two calls and the linker is told to drop the build directory from its debug map, one of them also setting `ZERO_AR_DATE=1`, which Apple's linker reads to write zero for object-file times [K]; and rustc with the same linker option. No rerun with these changes is in the repository yet; the table of changes names that rerun as the evidence for a verdict. Until those causes are traced, H1 is neither supported nor falsified by E1 as a whole. An earlier probe in the research sessions, with one program per language, had found the same-directory and cross-directory pattern for gcc, clang, rustc, Go in its default mode and gfortran on one machine [M].

Limits. The programs are small, one or two per language, so a difference that only a large program reaches is not tested. Link-time optimisation, profile-guided builds and parallel code generation beyond rustc's defaults are not in the matrix. E1 compares bytes; it does not check that meaning was preserved, which is H3 and H7. The macOS run says how many bytes differ in each file, not in which part of it.

8.2 E2 to E4: protocols, E4's erasure counts, and E4-Energy (`reported_j` & payback horizons)

E2 and E3 have protocols in the repository and have not been run. E4 has its original 96-row erasure-count run on a host without an energy counter [6], plus pre-registered Wave 4 and Wave 5 Linux RAPL `reported_j` extensions across 20 deterministic kernels and 15 compiler and runtime tiers (`board_metered = false`, `measured_j = None`).

E2, gate audit (H2). For each compiler, a fixed set of small programs, each with one injected fault and a correct twin that differs only at the fault: a type error for every compiler; a borrow error for rustc; an unused import or variable for Go; an invalid IR module for LLVM's `llc` and an invalid WebAssembly module for `wasmtime compile`; a failing contract (a Lean theorem whose proof does not check, an Ada precondition GNATprove cannot prove); and a generated-code fault for Erlang's validator. Each faulty program and its twin are built under the default flags

and under each flag set documented as changing the gate (for example `noEmitOnError` for tsc, `fullgraph=True` for TorchDynamo, `-Werror` for gcc and clang). The harness records the exit status, every artefact written with its SHA-256, and the first line of the diagnostic; the twin must build or the case is void. Each compiler and fault is classified as *emits nothing by default*, *emits by default and nothing when configured*, or *emits always*, and each gate's property class, linking assumption and trusted base is recorded from the compiler's own documentation, quoted, or as "not stated". The falsifier count is the number of cases where observed and documented behaviour differ. The tensor compilers' fallbacks to eager execution or to a CPU provider are recorded as a class of their own. Since the change of 2 November 2026, each record is evidence of authority pinned (§3.3): it is bound to the SHA-256 of the program, of the compiler's executable and of every artefact, and to the flag set, and a record whose digests do not match its case is void and the case is run again. Each gate's documented statements also include the authority of the evidence its checker accepts on and whether that acceptance is recorded bound to digests. A documented statement is derived evidence and is never promoted to an observation, so a compiler's class rests on the observed records; a gate that accepts on inferred evidence, such as a fallback chosen by a heuristic, is recorded with its acceptance as undecided.

E3, equivalence-level audit (H3). Each pass or flag is assigned a level before the run: `-O0`, `-O2` and `-O3` exact; `-ffp-contract` settings within a declared error bound; reassociation (`-fassociative-math` with vectorisation, `-ffast-math`, `-Ofast`) within a declared error bound, with the bound stated; and, on GPU runners, split reductions and lowering to FP8 declared lossy. The corpus has integer kernels, exact by construction, and floating-point kernels with sums, dot products, polynomial evaluation and reductions, on fixed inputs that include subnormal numbers, signed zeros, infinities and NaNs (not-a-number values). The exact reference is the same computation in rational arithmetic, rounded once at the end; the declared program is the `-O0` build with strict IEEE 754 semantics. The metrics are the bits that differ, the largest ULP distance, the largest absolute and relative error, and whether a static check (an SMT solver for loop-free code, or an interval or error-analysis tool) certified the level. The static tool is still to be chosen. Since the change of 2 November 2026, each differential-testing record is pinned evidence, bound to the SHA-256 of the kernel's source, the compiler's executable, the built binary and the input: an observed change beyond the assigned level is a counterexample, which is H3's falsifier, and agreement on every input certifies nothing. A level counts as certified only on proved evidence, the static tool's verdict bound to the SHA-256 of the binary or IR it checked, of the tool and of the stated level; a level that only a heuristic or a model assigns is recorded as undecided. The tensor compilers and the FP8 and FP4 formats need GPU runners, which the project does not have yet.

E4, erasure and description length against measured energy (H4). The corpus is a set of programs, each compiled under several configurations that E3's checker accepts as equal at the exact level, so that energy differences come from lowering and not from a change of program. ε is counted by instrumenting each run: every write of a register or memory location adds the bits of the previous value that were not copied out before being overwritten. λ is computed with a proxy declared before the run. Joules are measured per run, with intervals, in regime A (an FPGA or in-order core with a supply meter; executed operations as the baseline), regime B (an x86 host with RAPL, or a GPU with NVML; time multiplied by power as the baseline) and regime C (a simulation of entropy production). Compile-time energy is logged too. The analysis is declared in advance: an ordinary least-squares model per regime, with the share of variance explained and a 95% bootstrap interval over programs; ε and λ explain variance when that interval excludes zero. Energy figures without a meter reading are labelled modelled.

E4, declared before any energy is read (2 November 2026) [6]. The corpus is eight kernels (`sumsq`, `collatz`, `sieve`, `sort`, `crc32`, `matmul`, `fib` and `popcount`), each a function `run(n)` on 32-bit unsigned integers that uses no library, written in C and in Rust: 16 programs, each built at six optimisation levels (`clang -O0`, `-O1`, `-O2`, `-O3`, `-Os` and `-Oz`; `rustc` at `opt-level` 0, 1, 2, 3, s and z), 96 rows in all. Because E3's checker does not exist yet, the configurations are shown equal by a weaker test: each gives the reference value of its kernel on five declared inputs, natively and as WebAssembly. ε is counted by an interpreter of WebAssembly written for E4 and pinned by its SHA-256, on each kernel built for WebAssembly at the row's level; it is therefore a count for the configuration, not for the native code whose energy is read, and an upper bound, at byte granularity, on the bits a run discards. The primary λ is the number of WebAssembly instructions in the kernel's module. Energy is read in regime B only, from RAPL on Linux and from `powermetrics` on macOS; both are counters reported by the chip or the operating system, not a meter on the supply, and no host of the harness has a counter for regime A or C. The analysis uses adjusted R^2, which can fall below zero, with a 95% interval from 2,000 resamples of whole programs; ε and λ explain variance when that interval lies above zero, and in regime B H4 is supported only if the interval of the baseline's adjusted R^2 minus theirs lies above zero, and falsified if it lies below.

E4, counts without energy. The harness ran on 2 November 2026 (UTC) on a Linux x86-64 machine (Intel Xeon at 2.80 GHz, 4 CPUs, Ubuntu 24.04.5) with clang 18.1.3 and rustc 1.97.0. That machine has no `/sys/class/powercap` interface, so every energy field of that first results file is recorded as unmeasured: the file holds no joules, no energy analysis is computed from it, and H4 was left untested there. All 16 programs gave the reference value on every declared input under every configuration, natively and as WebAssembly, and ε was counted for all 96 rows. Per call, ε ranges from 272,048 bits (`popcount` at `-O2` and `-O3` in C, and at `opt-level` 1, 2 and 3 in Rust) to 10,057,528 bits (Rust `matmul` at `opt-level=0`), a modelled Landauer bound at 300 K of 7.81×10^{-16} J to 2.89×10^{-14} J per call. Each of the 16 programs has its highest ε at level 0. Size optimisation does not always lower it: C `popcount` at `-Oz` counts 3,540,016 bits, about 13 times its count at `-O2`, and executes 488,404 WebAssembly instructions per call against 19,530.

E4-Energy (Wave 4 & Wave 5), Linux RAPL `reported_j`, compilation cost and payback horizon N^*. Before running (`PREREG_rce_compile_wave4.json`, SHA-256 `165a12cda4d9c304…`, and `PREREG_rce_compile_wave5_rlib.json`, SHA-256 `1ba4f28abfaa3783…`), a 20-kernel deterministic exact-integer and fixed-point suite was locked across C, Rust, WebAssembly, Python, and RCE (`openIE-dev/rce` at `1762fb4b`): `sha256_block`, `chacha20_block`, `q32_orbital`, `crc32_ieee`, `lz77_match`, `dijkstra_grid`, `ntt_butterfly`, `matmul_i32`, `bst_probe`, `sortnet_16`, `ringbuf_sim`, `token_bucket`, `fnv1a_sip`, `sieve_primes`, `collatz_bounded`, `popcount_swar`, `edit_levenshtein`, `rle_codec`, `modexp_mont` and `kalman_q16`. All builds and runs executed on a 32-CPU Linux x86-64 host (AMD Ryzen AI Max+ 395, Zen 5, Ubuntu Linux 7.0; GCC 15.2.0, Clang 21.1.8, rustc 1.100.0, Node v22.22.1, Python 3.15.0) pinned via `sched_setaffinity` to CPU 0, the domain lead CPU of `/sys/class/powercap/intel-rapl:0:0/energy_uj` (core `MSR_CORE_ENERGY_STAT`), while simultaneously logging `/sys/class/powercap/intel-rapl:0/energy_uj` (APU package; `basis = "reported"`, `board_metered = false`, `measured_j = None`). All 300 kernel-by-tier cells (20 kernels × 15 tiers: 280 in Wave 4 plus 20 linked against the pre-compiled `librce_rt.rlib` crate in Wave 5) matched their 64-bit unsigned batch checksums over 2,000 seeds with 0 mismatches [M].

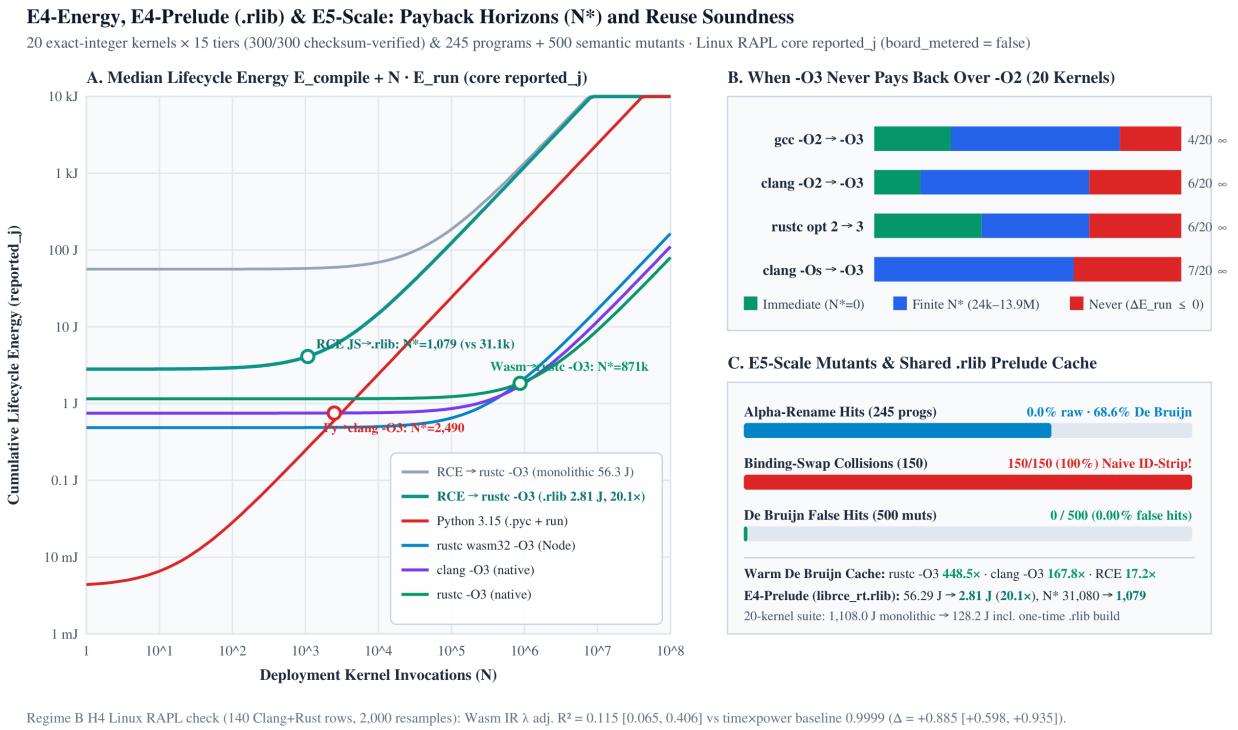

Figure 2. Pre-registered Wave 4 E4-Energy, Wave 5 E4-Prelude (`librce_rt.rlib`), and E5-Scale measurements (Linux RAPL core `reported_j`; `board_metered = false`). (A) Median cumulative lifecycle energy $E_{compile} + N \cdot E_{run}$ over $N \in [1, 10^8]$ kernel invocations across seven highlight tiers. (B) Share of the 20 kernels where stepping from level 2 (or `-Os`) to level 3 dominates immediately ($N^* = 0$), pays back at finite N^*, or never recovers its compile cost ($\Delta E_{run} \leq 0$). (C) E5-Scale soundness across 500 verified semantic mutants and warm De Bruijn cache savings [M].

Compiler / Lowering Tier	Median E_{compile} Core (J)	Median E_{compile} Pkg (J)	Median Build (ms)	Artifact (B)	Median E_{run} Core (µJ)	Median E_{run} Pkg (µJ)	Median Run (ns)
python3 py_compile + run	0.0041	0.056	0.47	3,546	241.74	2,685.98	23,035.4
gcc -O0	0.2904	4.892	40.16	16,384	4.17	53.24	442.5
gcc -O2	0.4723	7.528	61.51	16,320	1.81	27.24	231.5
gcc -O3	0.4955	7.890	65.78	16,320	1.75	24.36	213.7
clang -O0	0.4929	9.038	75.87	16,432	4.36	52.72	438.2
clang -Os	0.5917	10.800	88.63	16,400	1.75	22.51	190.9
clang -O2	0.6391	11.459	96.09	16,368	1.27	19.61	162.3
clang -O3	0.7500	11.479	96.16	16,368	1.10	15.47	132.9
rustc -C opt-level=0	0.6450	11.639	96.58	4,567,276	13.34	159.57	1,333.5
rustc -C opt-level=2	1.1040	19.838	163.65	4,533,388	1.35	19.54	171.2
rustc -C opt-level=3	1.1526	20.059	166.42	4,533,552	0.79	13.29	108.5
rustc wasm32 -O3 (node)	0.4853	8.132	67.12	1,601	1.64	20.95	157.3
rce check + 5-target emit (node JS)	0.8513	11.644	97.94	618,402	2,519.99	38,343.84	319,638.5
rce check + emit rust -> rustc -O3	56.2925	896.930	7,473.83	5,190,208	1,306.95	15,049.10	123,497.6
rce check + emit rust -> rustc -O3 (shared .rlib)	2.8053	27.045	304.10	5,303,456	1,212.03	8,864.82	94,571.7

Optimization payback horizons $N^* = \lceil \Delta E_{\text{compile}} / \Delta E_{\text{run}} \rceil$**.** Lowering from Python bytecode to `clang -O3` or `rustc -C opt-level=3` pays back on all 20 kernels after a median of **2,490** invocations (range 648 to 10,141) and **5,263** invocations (851 to 16,692). Within native compilers, stepping from unoptimised `-O0` / `opt-level=0` to `-O2` or `-O3` pays back on all 20 kernels, with median finite N^* of **38,262** (`rustc 0 -> 2`), **53,464** (`rustc 0 -> 3`), **61,637** (`clang -O0 -> -O2`), **81,004** (`clang -O0 -> -O3`) and **102,720** (`gcc -O0 -> -O3`). By contrast, stepping from `-O2` (or `-Os`) to `-O3` frequently fails to pay back at any deployment scale: $\Delta E_{\text{run}} \leq 0$ ($N^* = \infty$) on **6 of 20** kernels (30%) for `clang -O2 -> -O3`, **6 of 20** (30%) for `rustc 2 -> 3`, **7 of 20** (35%) for `clang -Os -> -O3`, and **4 of 20** (20%) for `gcc -O2 -> -O3`, while the kernels that do gain require a median of **106,920** to **244,165** invocations (and up to $\mathbf{3.87{\times}10^6}$ on Clang `sha256_block` and $\mathbf{1.39{\times}10^7}$ on GCC) before `-O3` recovers its extra compile joules. `rustc --target wasm32-unknown-unknown -C opt-level=3` builds a 1.6 KB portable module in 0.485 J core and executes in Node at 1.64 µJ per call, so native `rustc -O3` needs a median of **871,412** invocations to amortise its extra 0.667 J of native link and codegen work. For RCE (`1762fb4b`), `rce check` plus emitting all five scalar targets (`core`, `rust`, `python`, `js`, `scala`) costs a median of 0.851 J core (97.9 ms), whereas compiling the standalone RCE value-runtime prelude inline inside each emitted Rust file under `rustc -O3` costs 56.29 J core and requires a median of **31,080** invocations (18 of 20 finite, 2 never) to beat running the emitted JS directly in Node.

E4-Prelude (Wave 5), pre-compiled `librce_rt.rlib` shared runtime crate. Because `rce emit --target rust` embeds the standalone `src/emit/rt.rs` value runtime verbatim at the top of every emitted `.rs` program, monolithic builds re-optimise `2,286` lines (`143.4 KB`) of `Val` arithmetic, copy-on-write collections and formatting helpers on every kernel compile. Under `PREREG_rce_compile_wave5_rlib.json` (`1ba4f28abfaa3783…`), that runtime prelude (plus `DISPLAY_ESCAPE` and a zero-cost `RceProgramHooks` trait dispatch vtable) was factored into `rce_rt.rs` and compiled once with `rustc --crate-type=rlib -C opt-level=3`: on Zen 5 CPU 0, building `librce_rt.rlib` (`5,409,636 B`, byte-identical SHA-256 `ecfdc64c8d5f141e…` across 5 of 5 trials) cost **66.0344 J** core (**575.648 J** package, **7,103.8 ms**; package idle drift `3.40%`, `49.825 W` to `51.517 W`). Linking each kernel via `extern crate rce_rt; use rce_rt::*;` and `--extern rce_rt=librce_rt.rlib` (`20 of 20` exact 2,000-seed batch checksum matches, `0` mismatches) cuts median per-kernel compile cost from **56.2925 J** core (`7,473.83 ms`) to **2.8053 J** core (`27.045 J` package, **304.10 ms**, `5,303,456 B` unstripped / `944,544 B` stripped) — a **20.07×** reduction in `compile_core_reported_j` and **24.58×** wall-time speedup against Wave 4 (**15.85×** lower core energy and **16.43×** faster than same-session monolithic `44.4513 J` / `4,996.61 ms`), reducing total 20-kernel suite compile energy (including the one-time `66.03 J .rlib` build) **8.65×** from **1,108.01 J** to **128.17 J** (`U + E` compounding). Against `rce check + 5-target emit` (`node` JS), median finite payback N^* drops **28.8×** from **31,080** to **1,079** invocations (17 of 20 finite, range 404 to 5,650, 3 never; **1,974** with 18 of 20 finite against same-session Node JS) when `librce_rt.rlib` ships pre-built with the toolchain, **3,110** invocations (range 1,294 to 9,326; **10.0×** lower) when amortised

across 20 kernels (+3.302 J core per kernel), and **37,744** invocations (range 18,198 to 79,159) if built cold for a single kernel (breaking even against inline compilation as soon as two or more kernels share librce_rt.rlib). To rule out co-tenant package noise on the Zen 5 host, the full 20-kernel comparison was replicated on an idle dedicated 20-thread Intel Core i9-13900H Linux x86-64 workstation (load average 1.19 to 2.35, idle package power 4.461 W to 4.756 W, 6.62% drift; one-time librce_rt.rlib 111.82 J PP0 core, 142.63 J package, 6,581.8 ms, byte-identical across 5 of 5 trials): linking librce_rt.rlib cut median per-kernel compile energy from **84.4024 J** PP0 (104.560 J package, 4,397.44 ms) to **5.4661 J** PP0 (6.851 J package, 277.47 ms), a **15.44**× PP0 core and **15.26**× full-socket package reduction (15.85× speedup) [M].

H4 Regime-B check on Linux RAPL. Across the 140 Clang and Rust rows with freestanding WebAssembly IR modules (40 kernel-by-language series, 2,000 whole-kernel bootstrap resamples), static WebAssembly IR description length λ (ir_nodes and wasm_bytes) explains an adjusted R^2 of **0.1154** (95% interval [**0.0654, 0.4058**]), while the Regime-B baseline (run_wall_s × mean_core_power_w) explains **0.9999** ([**0.9987, 1.0000**]), a positive gap of **+0.8845** ([**+0.5979, +0.9352**]) [M]. That matches H4's Regime-B prediction on Linux RAPL, alongside the earlier macOS macmon runs (HYPOTHESES.md); H4 as a whole remains open until Regimes A and C are tested and quiet dedicated hosts are compared against shared-host runs (non-pinned cores on this host carried co-tenant load averages of 18.97 to 36.42).

E6 and E7 (H6 and H7) are planned, with no protocol yet. E6 needs a compiler built on the gear ladder, which does not exist. E5 has run in two stages, and §8.3 reports both.

8.3 E5: a name-free hash, measured

Method. E5 tests H5 and implements part R of the Compiler Form as two operations [6]. Its protocol and harness are dated 2 November 2026, before its first run, and the registration records E5's predictions and falsifiers under H5 on the same date. Each operation hashes a build output after removing what names the identifiers a program binds.

- *At the machine-code level*, a function identifier in the style of WARP, following the parts of WARP's published description that the E5 protocol quotes, is computed over unlinked object files: x86-64 ELF objects, arm64 Mach-O objects and WebAssembly objects. Each function is split into basic blocks; every instruction that a relocation record (a note telling the linker to patch an address there) points into is replaced by zero bytes; no-operation instructions and moves of a full-width register to itself are left out; each block is hashed, and the function's identifier is a hash of its blocks' hashes. Two keys come from it: the identifier of the kernel's function run, and a hash of the sorted identifiers of every function in the object, which no name enters.
- *At the IR level*, a hash of linked WebAssembly modules modulo alpha-equivalence (two programs are alpha-equivalent when a one-to-one renaming of bound identifiers turns one into the other) writes each module in a canonical form and hashes it with SHA-256. In that form a type index becomes the signature it names, locals are numbered by position and first use, functions are ordered by the shape of the call graph, independent definitions are sorted, and custom sections, such as the one that holds names, are left out. One key covers the whole module and one the part that run reaches. The protocol credits the idea of hashing modulo alpha-equivalence to Maziarz and colleagues (PLDI 2021) and states that its canonical form is not their algorithm.

Two baseline keys stand for the two kinds of cache key described under U in §3.5: the SHA-256 of the output bytes, and the SHA-256 of the source text and build command with the build directory removed. H5's metric names the compiler's own key; E5 reads no compiler's cache key and uses the two baselines in its place, a substitution the table of changes records. E5 compares keys; it runs no cache.

Corpus and pairs. The corpus is E4's 16 programs at their six levels, each with four outputs (an x86-64 object, an arm64 Mach-O object for C only, a WebAssembly object, and E4's linked WebAssembly module), and E1's C and Rust programs (hello.c, sort.c, hello.rs and generic.rs) at the same six levels, as x86-64 objects only. Each source is built from directory A, its same bytes from directory B, and a renamed copy from directory N; the three paths have equal length. The renamer gives every identifier the program binds a new name derived from the SHA-256 of the old one, and removes comments. The evidence that a renamed source is the same program is its behaviour: the linked modules of all three variants give E4's reference value on every declared input (96 of 96 in each variant), and each of E1's four programs prints the same bytes before and after renaming. A against B (*two directories*) and A against N (*renamed*) are declared equal. Every pair of levels of one toolchain, the C and Rust builds of one kernel at matching levels, and every pair of builds of two different kernels are declared not equal, as are the *mutations*: each linked module against itself with one opcode changed to another of its class, one constant

changed or one memory offset changed.

A *hit* is an equal key on a two-directory or renamed pair. A *false hit* is an equal key on builds of two different kernels, or an IR hash left unchanged by a mutation. An equal key on two levels, or on the C and Rust builds of one kernel, is reported and not counted as false, because each such pair computes one function. H5 is supported if, on the renamed pairs, every R hash has a higher hit rate than both baselines and there is no false hit; it is falsified by any false hit, or if no R hash has a higher hit rate than both baselines on the renamed pairs.

Machine. E5 ran on 2 November 2026 (UTC) on a Linux x86-64 machine (Intel Xeon at 2.80 GHz, 4 CPUs, Ubuntu 24.04.5) with clang 18.1.3 and rustc 1.97.0. It built and hashed 360 x86-64 objects, 144 Mach-O objects, 288 WebAssembly objects and 288 linked modules. The instruction boundaries found by its x86-64 decoder agreed with those of LLVM's `llvm-objdump` in all 386 functions of the x86-64 objects built in directory A.

Results. Equal keys, out of the pairs compared. The R hash is the machine-code identifier for objects and the IR hash for linked modules.

Comparison	Output	Pairs	Output bytes	Source key	R hash, whole output	R hash, `run`
Two directories	x86-64 objects, E4's kernels	96	87	96	96	96
Two directories	x86-64 objects, E1's programs	24	24	24	24	no `run`
Two directories	arm64 Mach-O objects	48	48	48	48	48
Two directories	WebAssembly objects	96	87	96	96	96
Two directories	linked WebAssembly modules	96	87	96	**87**	**87**
Renamed	x86-64 objects, E4's kernels	96	44	0	96	96
Renamed	x86-64 objects, E1's programs	24	17	12	24	no `run`
Renamed	arm64 Mach-O objects	48	24	0	48	48
Renamed	WebAssembly objects	96	44	0	96	96
Renamed	linked WebAssembly modules	96	76	0	**87**	**87**

Findings.

1. **No false hit.** No R hash was equal on any of 26,208 comparisons of builds of two different kernels (4,032 pairs per output and key, 1,008 for the Mach-O objects), and none of 9,328 mutations of 96 linked modules (4,712 of an opcode, 3,564 of a constant and 1,052 of a memory offset) left the IR hash unchanged.
2. **A gain on renamed pairs.** On the renamed pairs every R hash, in each of its nine combinations of output and key, had a higher hit rate than both baselines: 100% for the machine-code identifiers, against 45.8% to 70.8% for output bytes and 0% to 50.0% for the source key; 90.6% for the IR hashes, against 79.2% and 0%.
3. **The verdict.** By its registered conditions, H5 is supported on this corpus. The table of changes records the verdict together with the predictions that did not hold, and limits it to this corpus, two toolchains and one IR [6].
4. **Misses at the IR level.** The IR hashes were unequal on 9 of the 96 two-directory pairs and 9 of the 96 renamed pairs: the linked Rust modules of all eight kernels at `opt-level=0`, and Rust `sort` at `opt-level=1`. In each, only the data part of the canonical form differs, and each module holds one copy of its build directory's path. The registered IR hash treats data bytes as constants and does not remove the build path, which the Form's R removes. A diagnostic key added after the first run, which writes that path in the data as a placeholder and is not used for the verdict, is equal on all 96 two-directory pairs and on 92 of the 96 renamed pairs. The four left are Rust `fib`, `matmul`, `sieve` and `sort` at `opt-level=0`, again in their data. For `fib`, a check outside the harness found the two data segments differing in the column numbers of the location records that Rust's overflow messages carry: the renamed identifiers are longer, so the same expression starts at a later column.
5. **Other predictions that did not hold.** The source key was equal on 12 of the 24 renamed pairs of E1's programs, because the two hello programs bind no identifier and the renamer left them unchanged. The machine-code identifier of `run` in the WebAssembly objects was equal for the C and the Rust build of `fib` at five matching levels (1, 2, 3, s and z) and of `popcount` at s, 6 of 48 pairs; each such pair computes one function, so it is not a false hit. On the linked modules the IR hashes differ in all 48 C and Rust pairs; the functions part of their canonical forms is equal in 6 of the 48, and the memory part in none.

6. **Levels.** Across pairs of optimisation levels of one toolchain, the R hashes were equal on 10.0% to 21.7% of pairs and the output bytes on 10.0% to 12.5%, as registered: unequal on most pairs, with equal pairs reported and not counted.

What E5 does not show. An equal machine-code identifier establishes no equivalence on its own: a zeroed instruction can call or load a different symbol once the object is linked, and the identifier does not see which. WARP's constraints, which tell such functions apart, are not implemented, so a machine-code hit is a candidate for reuse that a checker still has to accept, and the protocol gives these identifiers no property class. An equal IR hash means that the two linked modules are alpha-equivalent, barring a SHA-256 collision, which the protocol takes as the exact level for their integer results under the WebAssembly semantics; it says nothing about the native code built from the same source. Unequal hashes say nothing about inequivalence: pairs of levels compute one function under different hashes, and two alpha-equivalent modules whose functions tie in the canonical order could be ordered differently, since ties are counted, not resolved. If an IR hash keyed reuse at rung G0, the protocol's gate statement would be equivalence at the exact level of the module, linked by its import and export names by a host that does not read custom sections, with E5's own readers, canonicaliser and hashing code, Python's hashing libraries and the WebAssembly specification as the trusted base. That no false hit occurred among these pairs and mutations does not show that the hashes have none on other programs: the initial E5 corpus is 16 kernels and 4 programs, from two toolchains that both lower through LLVM, run on one machine, and the IR is WebAssembly, not LLVM IR. And since E5 reads no compiler's own cache key, it does not show that any compiler's cache would gain hits.

E5-Scale (Wave 4): 185 `W1` RCE Forms, 60 C/Rust/Python kernels, 500 semantic mutants, and avoided `reported_j`. To test H5 beyond 20 small files and measure the compilation joules a name-free cache avoids, E5 was scaled by more than 12× (`PREREG_rce_compile_wave4.json`) across **245 base programs**: all **185** emitted Forms of the `W1` RCE Form corpus (`openie-research-experiments/rce/bench/form-space/results/forms.jsonl`, evaluated with `rce fmt nameless` at `1762fb4b`) plus the **60** C, Rust and Python kernel sources of E4-Energy (`20 × 3`). Four cache keys were compared across formatting/comment shifts, systematic alpha-renaming, benign independent-constant or event reordering, **29,890** unordered cross-program pairs, and **500** single-site semantic mutations (150 variable-binding swaps such as `x - y` to `y - x`, 150 binary-operator changes, 125 constant changes, and 75 branch/guard changes), every one verified non-equivalent on concrete execution vectors before hashing [M].

Corpus / Adversarial Set	Pairs / Mutants	Raw SHA-256	Comment/Whitespace-Stripped	Naive AST ID-Strip	De Bruijn / Nameless
`185 W1` RCE Forms · alpha-rename (`rce fmt nameless`)	185 hits	0 (0.0%)	50 (27.0%)	185 (100.0%)	**128 (69.2%)**
`60` C/Rust/Python kernels · formatting & comment shift	60 hits	0 (0.0%)	60 (100.0%)	60 (100.0%)	**60 (100.0%)**
`60` C/Rust/Python kernels · alpha-rename	60 hits	0 (0.0%)	0 (0.0%)	60 (100.0%)	**40 (66.7%)**
`60` C/Rust/Python kernels · constant-order swap	60 hits	0 (0.0%)	0 (0.0%)	0 (0.0%)	**60 (100.0%)**
Soundness: `150` variable-binding swap mutants (false hits)	150 bugs	0	0	**150 (100.0%, unsound)**	**0 (0.00%)**
Soundness: `350` operator + literal + guard mutants (false hits)	350 bugs	0	0	0 (0.0%)	**0 (0.00%)**
Soundness: All `500` verified semantic mutants (false hits)	500 bugs	0 (0.0%)	0 (0.0%)	**150 (30.0%, unsound)**	**0 (0.00%)**
Soundness: All unordered cross-program pairs (`245` bases)	29,890 pairs	0	0	—	**0 (0.00%)**

Three empirical results stand out:

1. **Why De Bruijn positions beat naive identifier-stripping.** Replacing every user identifier with a single anonymous `ID` token gives a 100% hit rate on alpha-renaming, but **collides on 150 of 150 (100.0%) variable-binding swap bugs** (`x - y` and `y - x` both project to `ID - ID`), yielding a 30.0% overall false-hit rate across the 500 verified bugs. Replacing each bound occurrence by its binding slot (`v0 - v1` vs `v1 - v0`) yields **0 of 500 (0.00%)** false-positive collisions and **0 of 29,890** cross-program collisions, supporting H5 at scale.
2. **Real formatter behaviour of `rce fmt nameless` at `1762fb4b`.** Because `rce fmt nameless` is designed to emit valid, checkable RCE source text with comments preserved (`nameless.rs`), its raw stdout changes when `#` comments are inserted (0 of 185 on comment-added formatting shifts, and 128 of 185 on alpha-renaming

where 50 zero-binder or foreign-field Forms fell back to comment shifts) and when adjacent `event` declarations swap order (0 of 23, since `e0, e1` follow declaration order); stripping `#` comments before hashing `rce fmt nameless` restores 185 of 185 (100%) on comment shifts.

3. **Avoided compilation `reported_j`.** Sitting a warm De Bruijn source cache in front of alpha-renamed rebuilds on pinned CPU 0 lowers per-kernel work from **1.166 J** core (143.8 ms) to **2.60 mJ** (288.1 µs) on `rustc -C opt-level=3` (**448.5×** reduction, avoiding 1.163 J core per hit), from **0.682 J** core (82.0 ms) to **4.07 mJ** (342.0 µs) on `clang -O3` (**167.8×** reduction, avoiding 0.678 J core per hit), and from **78.71 mJ** core (9.85 ms) to **4.59 mJ** (0.74 ms) for `rce check` plus 5-target `rce emit` (**17.15×** reduction) [M]. Readers can test live snippets and inspect all 300 E4-Energy cells in the companion observatory at /living/rce-compile/.

9. Limits of this study

- **Almost all external evidence is [S] or [K].** The hosts that hold the primary texts could not be reached from the research sessions, the GitHub commits API with a path filter was refused, and the national databases (ACM DL, CNKI, J-STAGE, eLibrary.ru, HAL) were reached only through general web search. Before publication beyond a draft, the primary texts need reading, in this order: Hoare, He and Sampaio (1993); Ershov (1977); the theorem statement in the CompCert manual; arXiv:2610.33193, to check PP-complete against PP-hard; arXiv:2609.11356; and the Zig release notes.
- **The negative search for a named law is open in some languages.** Native terms for "law of compilation" in Chinese, Russian, French and German were not searched one by one.
- **The per-part table is qualitative.** It summarises architecture and documentation; it is not a measurement of each compiler.
- **Energy readings are Regime-B on-chip `reported_j`, not supply wattmeter `measured_j`.** E4-Energy records Linux RAPL core and package `reported_j` (`board_metered = false`, `measured_j = None`) on a 32-CPU Zen 5 host whose non-pinned cores carried co-tenant workloads, with the Wave 5 `librce_rt.rlib` compile-energy ratio replicated on an idle dedicated `Core i9-13900H` Linux workstation (`4.46 W` idle package, `15.44×` PP0 and `15.26×` package reduction, §8.2); Regimes A (in-order/FPGA supply shunt) and C (thermodynamic entropy production) remain unmeasured, so H4 as a three-regime hypothesis is not yet closed.
- **Corpus scope.** E1 builds one or two programs per language; E4-Energy covers 20 self-contained integer and fixed-point kernels (plus the shared `librce_rt.rlib` prelude crate) without whole-program link-time optimisation or GPU kernels; and E5-Scale's source-level De Bruijn canonicaliser covers RCE Forms (`185`) plus single-file C, Rust and Python kernels (`60`), not full multi-translation-unit C++ or LLVM IR (§8.3).
- **The 98/2 split is registered, not yet measured.** It is hypothesis H6 (§5, §7), and its measurement is planned rung by rung with a declared unit.

10. Conclusion

Measured against real compilers, the candidate splits into two kinds of part. Representation, cost, checker, reuse and gate describe practice that exists, unevenly, somewhere in the field. Lowering, the convergence function and energy describe a standard no shipping compiler meets. That split is the reason to restate the candidate as a normal form: placing gcc, rustc, Go, GHC, XLA and the rest against one declared contract, with path-stripped determinism, named gates, explicit equivalence levels and a measured distance from two computable bounds, gives compilers from these communities a common scale. Two empirical gaps in the initial draft now carry Regime-B data. First, E5-Scale shows across 245 programs and 500 verified semantic mutants why part R must replace bound identifiers by De Bruijn positions rather than merely stripping names: positional hashing yields 0 of 500 false hits and cuts cold `clang -O3` and `rustc -O3` rebuild `reported_j` by 167.8× and 448.5×, whereas naive identifier-stripping collides on 100% of variable-binding swaps (§8.3). Second, E4-Energy and E4-Prelude show across 20 exact-integer kernels and 15 compiler tiers that compilation joules materially reshape which optimisation tier wins: lowering from Python to `clang -O3` pays back after a median of 2,490 runs, `-O0` to `-O3` needs 53,464 to 102,720 runs, `-O2` to `-O3` never recovers its extra compile joules on 20% to 30% of the kernels, factoring RCE's emitted Rust runtime prelude into a pre-compiled `librce_rt.rlib` crate cuts per-kernel compile `reported_j` by 20.07× (56.29 J to 2.81 J core) and lowers its payback against Node JS from 31,080 to 1,079 runs, and in Regime B time multiplied by power explains far more variance in RAPL `reported_j` than static WebAssembly IR size (0.9999 against 0.1154, §8.2). With hardware 10^6 to 10^{11} above the floor, ranking compilers by lifecycle `reported_j` (and supply `measured_j` when hardware shunt meters are attached) while keeping Landauer as a physical lower bound

turns part E into an operational decision rule. The 98/2 principle needs the same treatment: the unit it is counted in must be chosen and published before the first measurement.

Appendix A. Claims checked

The discussion in which the candidate was proposed (§1) made 55 factual claims about the field. Each was checked against the sources the research sessions could reach. "Confirmed" means the source says what the claim says; "partly" means the source exists but the claim overstates, misdates or misattributes it; "contradicted" means the source says something incompatible; "not found" means no source was located. Nearly every confirmation is [S], so "confirmed" means confirmed against snippet text.

Topic	Claims	Confirmed	Partly	Contradicted	Not found
Correctness and verification	9	4	3	0	2
Optimisation and model-assisted compilation	11	2	3	2	4
Physics and thermodynamic hardware	9	4	3	0	2
Systems languages and industry	10	4	3	1	2
HPC, legacy and high-integrity compilers	7	4	0	0	3
ML compilers	8	2	2	1	3
The law itself	1	0	0	0	1
Total	**55**	**20**	**14**	**4**	**17**

Figures were wrong more often than existence. Most named papers, products and deals exist, but the numbers attached to them were misattributed, rounded in the claim's favour, moved between comparisons, or not found at all. The thermodynamic-compilation thread is the weakest: its headline paper and its largest numbers were not found, and the nearest matches are self-published preprints.

A.1 Contradicted claims, with the correction

Claim	Correction	Mark
TENSAT gives XLA 3.45% on average and 56% on NasRNN	The figures come from a different paper, "Mind the Abstraction Gap" (OOPSLA 2025) [19]; TENSAT reports up to 16% [18]	[S]
Japan's "SlimeCompiler" uses non-commutative rings for a 1.5× to 3× gain	A self-published preprint (December 2025) with a benchmark labelled "simulated" and one program at 1.2×; the method is read/write-set dependence analysis; no Japanese origin was found [56]	[S]
Mojo went fully open source on 23 October 2026	It was open-sourced on 18 August 2026; whether the MAX platform is covered is disputed [57]	[S]
Triton has 300 to 500 commits a month	159 to 267 a month, April to October 2026 [58]	[F]

A.2 Claims that partly hold, with the correction

Claim	Correction	Mark
Bryant's 2024 UBC thesis shows that compositional correctness reduces to confluence	The thesis recasts correctness of one ANF compiler model as confluence and proves "several related but weaker properties" [14]	[S]
October 2026 work on weakest-precondition-preserving compilation, mechanised in Rocq	A PriSC 2026 talk at POPL in January 2026 [59]; the Rocq mechanisation and the October date were not found	[S], [NOT FOUND]
"The Next 700 Compiler Correctness Theorems" unifies compiler correctness	It gives a framework for stating and comparing theorems and argues that "correct" is ambiguous until the linking context is named; it is not one theorem [10]	[S]
Meta's coreset pass ordering beats -Oz by about 4.5% on about 5,000 repositories	4.7% on 4,683 repositories [60]	[S]
"Compiled AI" (Stanford, April 2026): 96%, break-even at 17 runs, a 57-fold saving at 1,000 runs	Led by XY.AI Labs; 96% and about 17 runs confirmed; the 57-fold figure was not found [42]	[S], [NOT FOUND]

Claim	Correction	Mark
Model-written Rocq proofs of crash-freedom (October 2026) at about $40 per kilobyte, with 30-fold reuse savings	About 180 million lines of Rocq for 239 OSS-Fuzz entry points in 123 libraries, for inputs under 64 MB [61]; the cost and reuse figures were not found, and Anthropic's research index lists no such post [62]	[S], [F]
Berkeley Lab's generative thermodynamic computing gives an advantage above 10^{11}	The paper exists and is shown in digital simulation [63]; the 10^{11} figure was not found	[S], [NOT FOUND]
Normal Computing's CN101 thermodynamic chip	Taped out July 2025, shown at Hot Chips 2026; it is digital pseudorandom CMOS, and no measured joules were found [64]	[S]
Three energy regimes: linear in operations, linear in time, entropy production	The first two are approximations, because static power scales with time; only the third is in units where Landauer appears directly	[K]
Qualcomm acquired Modular in July 2026	Announced 24 June, closed 29 July, about $3.1 to 3.9 billion in stock [65]	[S]
MLIR has no upstream path to NVIDIA or AMD GPUs	A minimal upstream path exists (the gpu, nvvm and rocdl dialects); production-grade paths are downstream [66]	[S], [K]
NVIDIA acquired Groq for about $20 billion (December 2025)	A non-exclusive licence plus hires; Groq continues independently [67]	[S]
TensorRT-LLM 1.2 (March 2026) removed the TensorRT backend and made PyTorch the default	PyTorch is the sole backend; the March date was not found [26]	[S], [NOT FOUND]
CuTe-based FlashAttention-4 is 1.6× to 3.2× faster on Blackwell	That range is FlexAttention's FlashAttention-4 backend against its own Triton backend (GB200, forward pass); FlashAttention-4 itself is up to 1.3× over cuDNN and 2.7× over Triton on B200 [68]	[S]

A.3 Confirmed claims

Twenty claims were confirmed, nearly all against snippet text [S]: "Compiling by Proving" (September 2025) [24]; CktFormalizer's "the compiler is the verifier", from a third-party review only [45]; Veritas, correct only relative to its CNF clauses [44]; EMI testing finding more than 1,000 bugs [2]; the MLGO gains of 0.3% to 1.5% in queries per second and 3% to 7% in size [37]; SuperCoder's 95% and 1.46×, judged by tests, not proof [41]; the Landauer figure of about 2.87 zeptojoules at 300 K [K, arithmetic]; the PP-completeness abstract, with the caveat in §4.2 [4]; the June 2026 codon-optimisation paper on thermodynamic hardware [69]; the 4% to 886% energy cost of model self-refinement [33]; TypeScript 7's 8× to 12× faster builds, from Microsoft's own benchmarks [70]; Mojo 1.0 on 11 August 2026 [71]; WebAssembly 3.0 in September 2025 [72]; the OctoAI, CentML, Brium and Fermyon acquisitions [73]; ORNL's JACC, from ORNL's own sources [74]; LFortran compiling LAPACK and fpm, while still alpha [75]; gcobol in GCC 15 and 16 [76]; AdaCore's GNAT Foundry: Intersection, from secondary sources [39]; FlexAttention at about 60% of FlashAttention-3 on Hopper [77]; and Triton's BF16 GEMM at 62% of cuBLAS on B200, the low end of 62% to 70% [78].

A.4 Claims not found

No source was located for the following seventeen claims. None of them is stated as fact anywhere in this study.

Claim	What was found instead
Debian is over 95% reproducible, and its "forky" release makes reproducibility a hard requirement	One machine at 97.38% [79]
AutoPhase is 16% better than -O3	AutoPhase targets cycle counts in high-level synthesis; its figures were not checked
"AI as a Compiler" (September 2026), 0.8× to 3.34× over autotuned Triton	A summary of a Triton-to-PTX system with a verifier reports 1.37× and 1.10× [80]
An October 2026 hybrid-automation study cut model-usage cost by 48%	Nearby figures measure other things
Synapze: $2.03 against $8 per correct application	Nothing relevant
G1 uses Z3-certified rewrites	Not matched to any system; Alive2 and Souper check LLVM rewrites with SMT, while egg does not certify its rules
HPE Labs (January 2026), "Thermodynamically Optimal Compilation with Explicit Convergence Rates", 99.93%	Only unreviewed, self-published preprints [36]
Extropic's 146× and a product named "Z1/T"	The public figure is about 10,000×, from a GPU simulation [81]

Claim	What was found instead
Zig's self-hosted backend: 10 s and 2.8 GB against 75 s and 9.6 GB on LLVM	One user reports 15 s against 84 s; the direction holds
67% of respondents use Wasm in production	41%, from a 2023 survey [82]
Julia 1.14 cuts load and precompile time by 30% to 50%	The 30% figure belongs to 1.13 [83]
LLVM Flang is production-ready on NERSC workloads	NERSC warns that LLVM "isn't fully compatible with all vendor software" [84]
GnuCOBOL 4.0	Version 3.2 (July 2023) is current [85]
TIRx is within 4% of cuBLAS on FP16 matrix multiplication	The TIRx post exists; no benchmark figure was visible [86]
A September 2026 survey: hardware outruns portable abstractions, with three workarounds	Nearest is an August 2026 essay, "Retire the Abstractions" [87]
MLIR has 500 to 800 commits a month	A proxy count gives 225 to 458 a month; the exact count was refused [58]
A Landauer floor and a Kolmogorov target are parts of an established compiler law	No compiler-correctness or compiler-design source ties either to a compiler law

References

Each entry gives the evidence mark for how the source was checked. [S] entries were seen only as search results or secondary summaries; the paper itself was not read.

1. [1] X. Yang, Y. Chen, E. Eide and J. Regehr, "Finding and Understanding Bugs in C Compilers," PLDI 2011. Preprint Seen only as a search-result snippet or secondary summary [S].
2. [2] C. Sun, V. Le, Z. Su and colleagues, EMI project page (equivalence modulo inputs), University of California, Davis. https://web.cs.ucdavis.edu/~su/emi-project/ Seen only as a search-result snippet or secondary summary [S].
3. [3] Zig Software Foundation, Zig 0.15.1 release notes. https://ziglang.org/download/0.15.1/release-notes.html Seen only as a search-result snippet or secondary summary [S]. The host could not be reached from the research sessions.
4. [4] A. Condon, N. Kornerup and co-authors, "The Thermodynamic Complexity of Circuits," arXiv:2610.33193, October 2026. https://arxiv.org/abs/2610.33193 Seen only as a search-result snippet or secondary summary [S]. Only the abstract was seen; whether the problem is in PP must be checked in the full text.
5. [5] USLegal, "Compilation" (legal definition). https://definitions.uslegal.com/c/compilation/ Seen only as a search-result snippet or secondary summary [S].
6. [6] openIE-dev/rce, `compile/`: CHARTER.md, HYPOTHESES.md and the experiments. https://github.com/openIE-dev/rce/tree/main/compile The project's own repository.
7. [7] liora.io, "PyTorch's FlexAttention with FlashAttention-4." https://liora.io/en/pytorchs-flexattention-with-flashattention-4-is-a-game-changer Seen only as a search-result snippet or secondary summary [S]. Secondary source.
8. [8] "Taming Bitwise Behavior in GPU Kernels with Tensor Core: Black-Box Reconstruction, Compiler Enforcement, and Static Verification," arXiv:2609.11356, September 2026. https://arxiv.org/pdf/2609.11356 Seen only as a search-result snippet or secondary summary [S]. Provisional: only a partly garbled conclusion was visible.
9. [9] "MMA-Sim," a bit-accurate model of tensor and matrix cores, arXiv:2511.10909. https://arxiv.org/html/2511.10909v1 Seen only as a search-result snippet or secondary summary [S].
10. [10] D. Patterson and A. Ahmed, "The Next 700 Compiler Correctness Theorems (Functional Pearl)," ICFP 2019. ICFP 2019 page Seen only as a search-result snippet or secondary summary [S].
11. [11] M. Patrignani, A. Ahmed and D. Clarke, "Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work," ACM Computing Surveys 51(6), 2019. DBLP record Seen only as a search-result snippet or secondary summary [S].
12. [12] Rust project, rustc dev guide, "Overview of the compiler." https://rustc-dev-guide.rust-lang.org/overview.html Cited from memory; the page was not opened in the research sessions [K].
13. [13] Go project, cmd/compile README. https://go.dev/src/cmd/compile/README Cited from memory; the page was not opened in the research sessions [K].
14. [14] L. Bryant (supervised by W. J. Bowman), "Compilation as Normalization: A Multi-language Semantics Approach to Compiler Correctness," MSc thesis, University of British Columbia, November 2024. UBC Open

Collections Seen only as a search-result snippet or secondary summary [S]. The host could not be reached; the abstract was not read verbatim.
15. [15] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu and J. Regehr, "Alive2: Bounded Translation Validation for LLVM," PLDI 2021. PDF Seen only as a search-result snippet or secondary summary [S].
16. [16] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens and M. Norrish, "The Verified CakeML Compiler Backend," Journal of Functional Programming 29, 2019. Kent repository Seen only as a search-result snippet or secondary summary [S].
17. [17] Bytecode Alliance, "Cranelift: an acyclic e-graph mid-end" (accepted RFC). https://github.com/bytecodealliance/rfcs/blob/main/accepted/cranelift-egraph.md Cited from memory; the page was not opened in the research sessions [K].
18. [18] Y. Yang, P. M. Phothilimthana, Y. R. Wang, M. Willsey, S. Roy and J. Pienaar, "Equality Saturation for Tensor Graph Superoptimization," MLSys 2021. Google Research page Seen only as a search-result snippet or secondary summary [S].
19. [19] "Mind the Abstraction Gap: Bringing Equality Saturation to Real-World ML Compilers," OOPSLA 2025. SPLASH 2025 page Seen only as a search-result snippet or secondary summary [S].
20. [20] Formal performance and compile-time guarantees for an inlining heuristic, mechanised in Rocq, arXiv:2608.20137, August 2026. https://arxiv.org/pdf/2608.20137 Seen only as a search-result snippet or secondary summary [S].
21. [21] Rust project, "Enabling the next-generation trait solver on nightly," Rust blog, 21 August 2026. https://blog.rust-lang.org/2026/08/21/enabling-next-solver-on-nightly Seen only as a search-result snippet or secondary summary [S].
22. [22] AdaCore, "Migrating C to Ada/SPARK with agentic AI: a practical playbook," March 2026. https://www.adacore.com/blog/migrating-c-to-ada-spark-with-agentic-ai-a-practical-playbook Seen only as a search-result snippet or secondary summary [S].
23. [23] Go project, cmd/go documentation, "Build and test caching." https://pkg.go.dev/cmd/go#hdr-Build_and_test_caching Cited from memory; the page was not opened in the research sessions [K].
24. [24] Zhao, Hildenbrandt, Conejero and Zhao, "Compiling by Proving: Language-Agnostic Automatic Optimization from Formal Semantics," arXiv:2509.21793, September 2025. https://arxiv.org/abs/2509.21793v1 Seen only as a search-result snippet or secondary summary [S].
25. [25] D. A. Wheeler, "Fully Countering Trusting Trust through Diverse Double-Compiling," PhD dissertation, George Mason University, 2009. arXiv:1004.5534 Seen only as a search-result snippet or secondary summary [S].
26. [26] NVIDIA, TensorRT-LLM, "Migration Guide: TensorRT Backend Removed." https://nvidia.github.io/TensorRT-LLM/legacy/tensorrt-backend-removal.html Seen only as a search-result snippet or secondary summary [S].
27. [27] TOP500, Green500 list, June 2026. https://top500.org/lists/green500/green500-june-2026/ Seen only as a search-result snippet or secondary summary [S].
28. [28] J. Pallister, S. J. Hollis and J. Bennett, "Identifying Compiler Options to Minimise Energy Consumption for Embedded Platforms," The Computer Journal, 2015. arXiv:1303.6485 Seen only as a search-result snippet or secondary summary [S].
29. [29] Comparison of gfortran, ifx and ifort energy use, Cloud Computing and Data Science (Wiser Pub), 2025. https://ojs.wiserpub.com/index.php/CCDS/article/view/6280 Seen only as a search-result snippet or secondary summary [S].
30. [30] J. S. Tyagi, J. Lynch and E. D. Demaine, "Toward an Energy Efficient Language and Compiler for (Partially) Reversible Algorithms," Reversible Computation 2016. arXiv:1605.08475 Seen only as a search-result snippet or secondary summary [S].
31. [31] J. Hong, B. Lambson, S. Dhuey and J. Bokor, "Experimental test of Landauer's principle in single-bit operations on nanomagnetic memory bits," Science Advances 2, 2016; the article also reviews the colloidal-bead test of Bérut et al., Nature 483, 2012. PMC4795654 Seen only as a search-result snippet or secondary summary [S].
32. [32] M. Horowitz, "Computing's Energy Problem (and what we can do about it)," ISSCC 2014. Slides Seen only as a search-result snippet or secondary summary [S]. The ratios to the Landauer bound are arithmetic done in the research sessions [K].
33. [33] "Green Code or Costly Code? Evaluating the Energy Costs of Iterative LLM-Based Code Optimizations," arXiv:2610.32891, October 2026. https://arxiv.org/abs/2610.32891 Seen only as a search-result snippet or

secondary summary [S].
34. [34] D. H. Wolpert and A. Kolchinsky, "The thermodynamics of computing with circuits," arXiv:1806.04103. https://arxiv.org/abs/1806.04103v2 Seen only as a search-result snippet or secondary summary [S].
35. [35] "Minimal thermodynamic cost of computing with circuits," arXiv:2504.04031. https://arxiv.org/html/2504.04031v3 Seen only as a search-result snippet or secondary summary [S].
36. [36] "On the Rate of Convergence to the Landauer Limit," self-published Zenodo record, March 2026. Zenodo 18922172 Seen only as a search-result snippet or secondary summary [S]. Unreviewed. Listed so that it is not mistaken for support.
37. [37] Google Research, "MLGO: a machine learning framework for compiler optimization," 2022. Google Research blog Seen only as a search-result snippet or secondary summary [S].
38. [38] AlphaSignal, report on model-written Rocq proofs of crash-freedom for 239 OSS-Fuzz entry points, October 2026. https://alphasignal.ai/news/anthropic-s-claude-proves-239-open-source-libraries-crash-free-exposing-fuzzing Seen only as a search-result snippet or secondary summary [S]. Secondary source; no primary post was found.
39. [39] Wevolver, "Guarding AI with deterministic trust" (AdaCore GNAT Foundry: Intersection), 2026. https://www.wevolver.com/article/guarding-ai-with-deterministic-trust Seen only as a search-result snippet or secondary summary [S]. Secondary source.
40. [40] T. Tunguz, "Hybrid state-machine agents." https://tomtunguz.com/hybrid-state-machine-agents/index.md Seen only as a search-result snippet or secondary summary [S]. A classification, not a measurement.
41. [41] A. Wei et al., "SuperCoder," arXiv:2505.11480, v3 January 2026. https://arxiv.org/pdf/2505.11480 Seen only as a search-result snippet or secondary summary [S].
42. [42] Trooskens, Karlsberg et al., "Compiled AI: Deterministic Code Generation for LLM-Based Workflow Automation," arXiv:2604.05150, April 2026. https://arxiv.org/abs/2604.05150 Seen only as a search-result snippet or secondary summary [S].
43. [43] "Progressive Crystallization," arXiv:2607.07052, July 2026. https://arxiv.org/pdf/2607.07052 Seen only as a search-result snippet or secondary summary [S].
44. [44] "Veritas: Deterministic Verilog Code Synthesis from LLM-Generated Conjunctive Normal Form," arXiv:2506.00005, 2025. https://arxiv.org/html/2506.00005 Seen only as a search-result snippet or secondary summary [S].
45. [45] Moonlight, review of "CktFormalizer: Autoformalization of Natural Language into Circuit Representations." https://www.themoonlight.io/de/review/cktformalizer-autoformalization-of-natural-language-into-circuit-representations Seen only as a search-result snippet or secondary summary [S]. Third-party review only; no arXiv record or author list was found.
46. [46] F. L. Morris, "Advice on Structuring Compilers and Proving Them Correct," POPL 1973, pp. 144–152. Bibliographic record Seen only as a search-result snippet or secondary summary [S]. The paper itself was not read.
47. [47] J. W. Thatcher, E. G. Wagner and J. B. Wright, "More on Advice on Structuring Compilers and Proving Them Correct," Theoretical Computer Science, 1981. Springer Seen only as a search-result snippet or secondary summary [S].
48. [48] Y. Futamura, "Partial Evaluation of Computation Process: An Approach to a Compiler-Compiler," 1971; English reprint in Higher-Order and Symbolic Computation 12, 1999. Summarised in Wikipedia, "Partial evaluation" Seen only as a search-result snippet or secondary summary [S]. The original was not read; sources disagree on its 1971 venue.
49. [49] A. P. Ershov, "Mixed computation: potential applications and problems for study," Theoretical Computer Science 18, 1982, pp. 41–67. Bibliographic record Seen only as a search-result snippet or secondary summary [S].
50. [50] C. A. R. Hoare, J. He and A. Sampaio, "Normal form approach to compiler design," Acta Informatica 30, 1993. Not read [K]. Must be read before any claim of priority.
51. [51] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," POPL 1977. Authors' page Seen only as a search-result snippet or secondary summary [S].
52. [52] C. Lattner et al., "MLIR: A Compiler Infrastructure for the End of Moore's Law," arXiv:2002.11054, 2020. https://arxiv.org/pdf/2002.11054 Seen only as a search-result snippet or secondary summary [S].
53. [53] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock and P. Panchekha, "egg: Fast and Extensible Equality Saturation," POPL 2021. NSF PAR copy Seen only as a search-result snippet or secondary summary

[S].
54. [54] A. P. Ershov, "On the essence of compilation," IFIP Working Conference on Formal Description of Programming Concepts, 1977. Not read [K]. Must be read before any claim of priority.
55. [55] R. Cox, "Perfectly Reproducible, Verified Go Toolchains," Go blog, 28 August 2023. https://go.dev/blog/rebuild Seen only as a search-result snippet or secondary summary [S].
56. [56] "SlimeCompiler: Commutative Ring Analysis for Compiler Optimization and Quality Assurance," self-published Zenodo record, December 2025. Zenodo 17946400 Seen only as a search-result snippet or secondary summary [S]. Unreviewed.
57. [57] Dataconomy, "Qualcomm open-sources Modular AI platform," 19 August 2026. https://dataconomy.com/2026/08/19/qualcomm-open-sources-modular-ai-platform-challenge-nvidia/ Seen only as a search-result snippet or secondary summary [S].
58. [58] GitHub search API, commit counts for triton-lang/triton and llvm/llvm-project by month, April to October 2026, queried with `gh api search/commits` on 1 November 2026. Read in the source [F]. The index covers the default branch and can lag, so the counts are lower bounds.
59. [59] POPL 2026, event listing (PriSC 2026 talk "WP-Preserving Compilation," Abate, Elsheikh et al.). https://popl26.sigplan.org/search/all/events Seen only as a search-result snippet or secondary summary [S].
60. [60] Liang, Stone, Tian et al., "Learning Compiler Pass Orders using Coreset and Normalized Value Prediction," ICML 2023. arXiv:2301.05104 Seen only as a search-result snippet or secondary summary [S].
61. [61] AI Weekly, "Anthropic and Theorem use Claude to verify 123 OSS libraries," October 2026. https://aiweekly.co/alerts/anthropic-theorem-use-claude-to-verify-123-oss-libraries Seen only as a search-result snippet or secondary summary [S].
62. [62] Anthropic, research index. https://www.anthropic.com/research Read in the source [F]. It lists no October 2026 post on this work.
63. [63] S. Whitelam, "Generative thermodynamic computing," Physical Review Letters 136, 037101, 2026. arXiv:2506.15121 Seen only as a search-result snippet or secondary summary [S].
64. [64] Normal Computing, "CN101: A Digital Thermodynamic Computer for Generative AI," arXiv:2608.00754. https://arxiv.org/pdf/2608.00754 Seen only as a search-result snippet or secondary summary [S].
65. [65] RuntimeWire, report on the Qualcomm acquisition of Modular and the Mojo source release, 2026. https://runtimewire.com/article/chris-lattner-open-sources-mojo-qualcomm-modular Seen only as a search-result snippet or secondary summary [S].
66. [66] LLVM project, MLIR "gpu" dialect documentation. https://mlir.llvm.org/docs/Dialects/GPU/ Seen only as a search-result snippet or secondary summary [S].
67. [67] AIN, "Nvidia is acquiring the assets of startup Groq," 26 December 2025. https://en.ain.ua/2025/12/26/nvidia-is-acquiring-the-assets-of-startup-groq/ Seen only as a search-result snippet or secondary summary [S].
68. [68] Lambda, "FlashAttention-4 gives the NVIDIA Blackwell platform its most optimized attention kernel yet." https://lambda.ai/blog/flashattention-4-gives-the-nvidia-blackwell-platform-its-most-optimized-attention-kernel-yet Seen only as a search-result snippet or secondary summary [S].
69. [69] Jelincic and Walker, codon optimisation on thermodynamic hardware, arXiv:2606.17327, June 2026. https://arxiv.org/abs/2606.17327 Seen only as a search-result snippet or secondary summary [S].
70. [70] Microsoft, "Announcing TypeScript 7.0," TypeScript blog, 8 July 2026. https://devblogs.microsoft.com/typescript/announcing-typescript-7.0/ Seen only as a search-result snippet or secondary summary [S].
71. [71] Techzine, "Mojo programming language reaches version 1.0," August 2026. https://www.techzine.eu/news/devops/143570/mojo-programming-language-reaches-version-1-0/ Seen only as a search-result snippet or secondary summary [S].
72. [72] x-cmd blog, WebAssembly 3.0 announcement summary, September 2025. https://www.x-cmd.com/blog/250924 Seen only as a search-result snippet or secondary summary [S].
73. [73] GeekWire, "Chip giant Nvidia acquires OctoAI," 2024. https://www.geekwire.com/2024/chip-giant-nvidia-acquires-octoai-a-seattle-startup-that-helps-companies-run-ai-models/ Seen only as a search-result snippet or secondary summary [S].
74. [74] Oak Ridge National Laboratory, "ORNL debuts JACC for performance-portable Julia on HPC systems," June 2026. https://www.ornl.gov/news/ornl-debuts-jacc-performance-portable-julia-hpc-systems Seen only as a search-result snippet or secondary summary [S].
75. [75] LFortran, "LFortran compiles LAPACK," February 2026. https://lfortran.org/blog/2026/02/lfortran-compiles-lapack/ Seen only as a search-result snippet or secondary summary [S].

76. [76] Phoronix, "GCC 15 merges COBOL," 2025; Arch Linux package gcc-gcobol 16.1. Phoronix; Arch Seen only as a search-result snippet or secondary summary [S].
77. [77] PyTorch, "FlexAttention + FlashAttention-4: fast and flexible." https://pytorch.org/blog/flexattention-flashattention-4-fast-and-flexible/ Seen only as a search-result snippet or secondary summary [S].
78. [78] "Evaluating CUDA Tile for AI Workloads on Hopper and Blackwell GPUs," arXiv:2604.23466. https://arxiv.org/pdf/2604.23466 Seen only as a search-result snippet or secondary summary [S].
79. [79] debian-vote mailing list, May 2025 (one machine at 97.38% reproducible). https://lists.debian.org/debian-vote/2025/05/msg00123.html Seen only as a search-result snippet or secondary summary [S].
80. [80] Digg summary of an "AI-as-compiler" system translating Triton to PTX with a verifier. https://digg.com/ai/g2dvabpv Seen only as a search-result snippet or secondary summary [S]. Secondary source; the original could not be reached.
81. [81] Extropic, denoising thermodynamic models, arXiv:2510.23972. https://arxiv.org/html/2510.23972v1 Seen only as a search-result snippet or secondary summary [S].
82. [82] The New Stack, WebAssembly adoption survey figures (2023). https://thenewstack.io/whos-leading-webassembly-adoption-so-far-vendors/ Seen only as a search-result snippet or secondary summary [S].
83. [83] AlternativeTo, "Julia 1.13 released," September 2026. https://alternativeto.net/news/2026/9/julia-1-13-released-with-major-speed-gains-in-precompilation-startup-and-garbage-collection/ Seen only as a search-result snippet or secondary summary [S].
84. [84] NERSC documentation, base compilers. https://docs.nersc.gov/development/compilers/base/ Seen only as a search-result snippet or secondary summary [S].
85. [85] GnuCOBOL project page. https://gnucobol.sourceforge.io/ Seen only as a search-result snippet or secondary summary [S].
86. [86] Apache TVM, "TIRx," TVM blog, 22 June 2026. https://tvm.apache.org/2026/06/22/tirx Seen only as a search-result snippet or secondary summary [S]. The page could not be reached; no benchmark figure was visible.
87. [87] Hazy Research, "Retire the Abstractions," 5 August 2026. https://hazyresearch.stanford.edu/blog/2026-08-05-retire-the-abstractions Seen only as a search-result snippet or secondary summary [S].