Compiler Backend Bring-Up: Blocking Frequency as the Ordering Principle for Instruction-Set Coverage

Brendan Sechter

2026-08-06

A compiler was 87 percent finished. It could not compile two thirds of the programs it was for.

Both numbers are correct. The first counts individual instructions the compiler knew how to translate. The second counts whole programs that would actually go through. **The gap between them is what this article is about**, and the reason it exists is simple enough to state in one sentence. **A program needs every instruction it uses, not most of them.** One missing instruction out of a hundred stops the whole thing, exactly as one missing link stops a chain.

That gap then destroyed a carefully reasoned plan. One working session before the measurement was taken, the author of this article had formally recommended what the next piece of work should be. The reasoning had no invalid step in it. The measurement showed the recommendation to be worth nothing at all, because the thing it would have unblocked **does not occur even once** in any program the compiler is meant to serve. The instrument that established this took about twenty minutes to build and two seconds to run.

The article reports that, and then reports four errors made while writing it, all four of which ran in the direction of a more striking result, and one of which was committed inside the paragraph warning against the other three.

What this is a case study of

The setting is compiler engineering, and a reader who has never written a compiler can follow the argument, because the shape of the problem is not specific to compilers.

The concrete project is Keleusma, whose compiler until now has emitted bytecode for a virtual machine, as described in the self-hosting strategy and its getting-started article. **Native code generation is the step after that one**, and it is where the ordering question first became expensive enough to measure. The lineage of the design sits in the stream-based compilers series and in the self-hosted silicon compiler. **None of that background is needed to follow what follows**, and the measurement stands on its own.

A **compiler backend** translates a program from an intermediate form into instructions a machine can run. The intermediate form has a fixed vocabulary of operations, here 66 of them. Translating each one is separate work. **Bringing up a backend therefore means choosing an order**, and the question of which operation to implement next is the ordering problem this article treats.

The ordering principle in common use ranks the remaining work by how hard each item is to build and by how tidily it fits the architecture. This article argues that principle is systematically wrong, and that the correct one ranks by **measured frequency of blocking** over a representative body of real programs. The argument is empirical, and nothing here

is asserted without measurement. An instrument was built over the 58 compilable programs of the shipped corpus, comprising 73,434 instruction instances across 496 compilation units, and the resulting distribution is reported in full, including the negative results and confidence bounds on them.

Anywhere a customer needs a conjunction of features rather than any one of them, the same trap is available, and the Pattern Extraction section at the end states it without reference to compilers.

Seven literatures touch this problem and none of them answers it

That absence is the gap the article fills. Each of the following traditions has something to say about coverage, and none supplies an ordering principle.

Tradition	Range surveyed here
Code generation and instruction selection	Sethi and Ullman 1970 to Ertl 1999
Verified and validated compilation	Necula 1997 to Lopes and colleagues 2021
Abstract interpretation	Cousot and Cousot 1977 to Blanchet and colleagues 2003
Worst-case resource analysis	Liu and Layland 1973 to Wilhelm and colleagues 2008
Corpus-based empirical study	Knuth 1971 to Ray and colleagues 2014
Test adequacy and mutation	DeMillo and colleagues 1978 to Inozemtseva and Holmes 2014
Submodular optimisation and prioritisation	Nemhauser, Wolsey and Fisher 1978 to Buchbinder and colleagues 2015

The reason none of them answers the question is the same in every case. They all reason about instructions, and the quantity that governs delivered capability is a property of **programs**.

The Coverage Ordering Problem

The ordering problem for instruction-set coverage during backend bring-up is the question of which subset of an instruction set a partial code generator should implement first, given a fixed implementation budget and a target population of programs that the generator is intended to serve. The problem permits several formalizations depending on the engineering tradition consulted.

The compiler-construction tradition treats the coverage property as an instruction-selection completeness question, where the generator must eventually cover the whole instruction set and the intermediate ordering is a matter of developer convenience. The staged-delivery tradition, in the form Boehm 1988 gave it, treats the coverage property as an incremental-capability question, in which each increment should deliver a demonstrable capability to some consumer. The verification tradition, which is the governing tradition for the present case because the Keleusma value proposition is definitive worst-case execution time and worst-case memory usage in the sense surveyed by Wilhelm and colleagues 2008 and by Puschner and Burns 2000, treats the coverage property as a soundness-preservation question, in the sense Leroy 2009 establishes for a realistic compiler and Pnueli and colleagues 1998 formulate as translation validation, in which every implemented instruction must be shown to preserve the semantics already proven on the bytecode artefact and every unimplemented instruction must be refused, never approximated.

The three traditions agree that the instruction set must eventually be covered in full and disagree about the intermediate ordering. The present article adopts the verification tradition for correctness obligations and argues that none of the three supplies an adequate ordering principle, because all three reason about instructions in isolation while the quantity that governs delivered capability is a property of programs, not of instructions.

Notation

The formal machinery below is worth the four symbols it costs, because the central result is a comparison between two averages that look interchangeable and are not. A reader who prefers prose can take the following and skip to the results. **One measure averages over instructions. The other averages over programs, and that one is what a user experiences. It is far lower.**

Let I denote the instruction set, with $|I| = 66$ in the present case. Let $S \subseteq I$ denote the subset the generator currently lowers, with $|S| = 39$ at the time of measurement. Let

$$C = \{c_1, c_2, \dots, c_M\}, \qquad M = 496$$

denote the corpus of compilation units, with each unit c_m a finite sequence of instruction instances drawn from I and $|c_m|$ its length. Write the total instance count

$$N = \sum_{m=1}^{M} |c_m| = 73{,}434$$

and the per-unit lowered count

$$n^{\mathrm{ok}}(c_m, S) = \big|\{\iota \in c_m : \iota \in S\}\big|.$$

Two coverage measures

The instruction-level coverage is the proportion of instruction instances the generator handles,

$$\rho^{\mathrm{inst}}(S) = \frac{1}{N} \sum_{m=1}^{M} n^{\mathrm{ok}}(c_m, S).$$

The unit-level coverage is the proportion of compilation units in which every instruction is handled. Writing the lowerability indicator

$$\chi(c_m, S) = \prod_{\iota \in c_m} \mathbf{1}[\iota \in S] = \mathbf{1}\big[n^{\mathrm{ok}}(c_m, S) = |c_m|\big]$$

the unit-level coverage is

$$\rho^{\mathrm{unit}}(S) = \frac{1}{M} \sum_{m=1}^{M} \chi(c_m, S).$$

The product is the whole difficulty. A sum forgives a missing term, since ninety-nine present out of a hundred still averages to 0.99. **A product does not forgive anything**, because a single zero anywhere sets the entire result to zero. The first measure is a sum over instructions and the second is an average of products over programs, which is why they can differ so widely while both being correct.

The product form of χ is the whole difficulty. The generator exploits structured control flow directly, in the sense of the structured program theorem of Bohm and Jacopini 1966, so that basic blocks fall out of the instruction stream without the control-flow-graph reconstruction Allen 1970 formalised, and the conjunction is therefore a property of the instruction multiset, and no recovered graph enters into it. Lowerability is a conjunction over the unit, so a single unimplemented instruction sets the indicator to zero regardless of how many instances were handled. The refusal is not a defect. It is the required behaviour under the verification tradition, since a generator that approximated an unimplemented instruction would emit native code whose semantics were never proven. The stance is the one Necula 1997 formalises as proof-carrying code, under which the artefact carries its own evidence, extended to a certifying compiler by Necula and Lee 1998, and the bytecode-level analogue is the bytecode verification treated by Leroy 2003 and specified for the Java virtual machine by the Oracle Java Virtual Machine Specification.

The two measures satisfy $\rho^{\mathrm{unit}}(S) \le \rho^{\mathrm{inst}}(S)$ with equality only in the degenerate cases where $S \supseteq \bigcup_m c_m$ or where every unit has unit length. Define the coverage gap

$$\Gamma(S) = \rho^{\mathrm{inst}}(S) - \rho^{\mathrm{unit}}(S)$$

which measures the extent to which an instance-level progress report overstates delivered capability.

Blocking sets and marginal gain

Define the blocking set of an instruction $\iota \notin S$ as the set of units that would become lowerable if and only if ι were added,

$$B(\iota, S) = \{c_m \in C : \chi(c_m, S) = 0 \ \wedge \ \chi(c_m, S \cup \{\iota\}) = 1\}$$

with the marginal unit-level gain

$$g(\iota, S) = \frac{|B(\iota, S)|}{M} = \rho^{\mathrm{unit}}(S \cup \{\iota\}) - \rho^{\mathrm{unit}}(S).$$

Let $n(\iota)$ denote the instance count of ι across the corpus,

$$n(\iota) = \sum_{m=1}^{M} \big|\{\iota' \in c_m : \iota' = \iota\}\big|.$$

The central structural claim is that g is not monotone in n. Formally, there exist instructions $\iota_a, \iota_b \notin S$ with

$$n(\iota_a) > n(\iota_b) \qquad \text{and} \qquad |B(\iota_a, S)| < |B(\iota_b, S)|$$

The results section exhibits such a pair at WORKSTREAM granularity, with a ratio exceeding three in the instance count and exceeding three in the opposite direction in the blocking count.

It does not exhibit one at instruction granularity, and cannot, because the supermodularity established below empties almost every instruction-level blocking set, so an instruction-level comparison would compare zeroes. The existence claim above is therefore stated for completeness of the formalism and is witnessed only after aggregation.

The ordering problem is therefore the selection

$$\iota^\star = \arg\max_{\iota \in I \setminus S} \lambda(\iota, S), \qquad \lambda(\iota, S) = \frac{|B(\iota, S)|}{\kappa(\iota)}$$

with $\kappa(\iota)$ the implementation cost and λ the leverage ratio.

That selection rule is not new, and naming it imports both a guarantee and the precise conditions under which the guarantee fails. Choosing repeatedly the candidate with the greatest newly covered mass per unit cost is the cost-weighted greedy heuristic for set covering, analysed by Johnson 1974 and Chvatal 1979, which established the logarithmic approximation ratio, sharpened by Slavik 1997 and shown to be essentially the best obtainable unless the complexity classes collapse by Lund and Yannakakis 1994 and Feige 1998. **The ordering problem posed here is set cover with costs, and the leverage ratio is the classical greedy rule for it.**

The guarantee attaches to the covering formulation, while the objective measured here falls outside it, a distinction the supermodularity section makes precise and which is the reason the greedy rule is offered below as a heuristic rather than as an approximation algorithm.

The ordering principle in common use is the degenerate variant

$$\iota^\star_{\text{naive}} = \arg\min_{\iota \in I \setminus S} \kappa(\iota)$$

which minimises cost alone while leaving $|B|$ unmeasured and therefore implicitly assumed uniform across instructions. The assumption is false in the case measured below.

Supermodularity, and why greedy carries no guarantee here

The ordering principle advocated here is greedy, and the question is whether greediness admits the usual formal justification. It does not, and the reason is structurally important.

Write $R_m \subseteq I$ for the set of distinct instructions appearing in unit c_m, so that $\chi(c_m, S) = \mathbf{1}[R_m \subseteq S]$ and

$$\rho^{\text{unit}}(S) = \frac{1}{M} \sum_{m=1}^{M} \mathbf{1}[R_m \subseteq S].$$

The objective is monotone non-decreasing,

$$S \subseteq S' \implies \rho^{\text{unit}}(S) \le \rho^{\text{unit}}(S')$$

since adding an instruction can only turn indicators from zero to one. The marginal gain of adding ι has the explicit form

$$g(\iota, S) = \frac{1}{M} \Big| \{ m : \iota \in R_m \ \wedge \ R_m \setminus \{\iota\} \subseteq S \} \Big|$$

and for $S \subseteq S'$ the qualifying set for S is contained in the qualifying set for S', since $R_m \setminus \{\iota\} \subseteq S$ implies $R_m \setminus \{\iota\} \subseteq S'$. Therefore

$$S \subseteq S' \subseteq I, \quad \iota \notin S' \implies g(\iota, S) \leq g(\iota, S')$$

which is **increasing** marginal returns. The objective is supermodular, not submodular.

The whole force of the error lies in a reversed inequality, so both forms belong side by side. The two properties, and the convex and concave structure that separates them, are set out in Lovasz 1983. Submodularity, the property that was asserted, is diminishing returns,

$$S \subseteq S' \implies g(\iota, S) \geq g(\iota, S') \qquad \text{(submodular, and FALSE here)}$$

$$S \subseteq S' \implies g(\iota, S) \leq g(\iota, S') \qquad \text{(supermodular, and true here)}$$

And the guarantee that submodularity would have bought belongs here too, since invoking it by name without stating it is what let the error pass. For a monotone submodular objective under a cardinality constraint the greedy algorithm satisfies

$$f(S_{\text{greedy}}) \geq \left(1 - \tfrac{1}{e}\right) f(S_{\text{OPT}}) \approx 0.632\, f(S_{\text{OPT}})$$

No such bound is available here. Under supermodularity the greedy choice can be arbitrarily bad, because an instruction with zero marginal gain today may have large gain once its companions are present, which is exactly the situation equation for $|B(\iota, S)| = 0$ above describes. An initial draft of this article asserted the opposite and invoked the Nemhauser, Wolsey and Fisher 1978 result to claim that greedy selection attains $\left(1 - \frac{1}{e}\right) \approx 0.632$ of the optimum under a cardinality constraint. That claim is false. The classical guarantee requires diminishing returns, whose tightness Nemhauser and Wolsey 1978 establish, whose dependence on curvature Conforti and Cornuejols 1984 characterise, whose geometric character Lovasz 1983 develops, whose budgeted variant Khuller and colleagues 1999 treat, and whose unconstrained non-monotone case Buchbinder and colleagues 2015 settle at one half. Maximisation of a monotone supermodular function admits no constant-factor approximation in general, the containment being of the set-cover family whose completeness Karp 1972 established and whose logarithmic inapproximability threshold Feige 1998 proved. The greedy ordering advocated here is therefore an empirically justified heuristic with no approximation guarantee attached, and the article says so rather than borrowing authority from a theorem that does not apply.

The correction is not merely a repair, because the increasing-returns structure explains an empirical fact the erroneous version could not. Under increasing returns the singleton blocking sets are typically empty,

$$\left|B(\iota, S)\right| = 0 \quad \text{for most } \iota \notin S, \qquad \text{while} \qquad \left|B(W, S)\right| \gg 0$$

because a unit requiring several unimplemented instructions is unblocked by none of them individually and by all of them jointly. This is precisely why instruction-level attribution is uninformative in this setting and why the analysis must aggregate to workstreams. The conjunction that makes χ a product, the conjunction that makes the objective supermodular, and the conjunction that empties the singleton blocking sets are one observation viewed three ways.

Workstream aggregation

Instructions that share a design prerequisite are implemented together, so the practically available choices are workstreams, not individual instructions. Let $W \subseteq I \setminus S$ denote a workstream. The workstream blocking set and counterfactual gain are

$$B(W,S) = \{c_m : \chi(c_m,S) = 0 \ \wedge \ \chi(c_m, S \cup W) = 1\}, \qquad \Delta\rho^{\text{unit}}(W) = \frac{|B(W,S)|}{M}.$$

Workstream-level counterfactuals are better identified than instruction-level ones, because the co-occurrence that defeats instruction-level attribution is largely within workstreams instead of across them. An instruction reading a data-segment slot co-occurs with an instruction writing one far more often than either co-occurs with a coroutine suspension.

The Code Generation Literature, and Where Ordering Is Absent From It

The instruction-selection literature is mature and supplies no ordering principle, which this section establishes instead of asserting, because the absence is the gap this article addresses.

The tradition begins with optimality results for restricted shapes. Sethi and Ullman 1970 gave optimal code generation for arithmetic expressions under a register-count constraint, and the result is optimal with respect to a fixed and fully implemented instruction set. Glanville and Graham 1978 introduced table-driven selection by parsing the intermediate representation against a machine grammar, and Fraser, Hanson and Proebsting 1992 reduced the approach to a practical generator generator. Ertl 1999 extended optimality from trees to directed acyclic graphs. At the extreme, Massalin 1987 searched the instruction space exhaustively for the shortest program computing a function. **The tradition also learned to synthesise the rules rather than write them, which matters because it changes what a lowering costs without changing what this article measures.** Davidson and Fraser 1984 generated peephole optimisations automatically from a machine description, Aho, Ganapathi and Tjiang 1989 reduced instruction selection to tree matching with dynamic programming, Bansal and Aiken 2006 harvested a peephole superoptimiser by enumeration over the target, and Schkufza, Sharma and Aiken 2013 replaced enumeration with stochastic search over loop-free binaries. **Synthesis lowers κ and leaves $|B|$ untouched**, so it changes the leverage ratio through its denominator only and reorders nothing that the numerator determines.

Every one of these results presupposes that the target instruction set is available in its entirety. None addresses the partial-implementation regime, because none has a reason to.

The supporting analyses are likewise complete-set analyses. Allen 1970 established control-flow analysis, Kildall 1973 the unified dataflow framework, Ferrante and colleagues 1987 the program dependence graph, Wegman and Zadeck 1991 conditional constant propagation, and Chaitin 1982 register allocation by graph colouring. Static single assignment form, introduced by Cytron and colleagues 1991, given a functional reading by Appel 1998, and reduced to a simple construction by Braun and colleagues 2013, is the representation the present generator relies on, since it models the operand stack as memory slots and delegates their promotion to registers to that machinery. The generator targets the intermediate representation of Lattner and Adve 2004 specified in the LLVM Language Reference.

The observation to draw is that this literature optimises within a covered instruction set and is silent on how to reach one. A backend author consulting it finds a great deal about how to lower an instruction well and nothing about which instruction to lower next. The silence is not an oversight, because in the settings these works address the instruction set is a fixed input, never a schedule. It becomes an oversight only when the set is being covered incrementally,

which is the universal condition of a backend under construction and a condition the literature treats as a transient state not worth theorising.

Verified and Validated Compilation as the Governing Constraint

The refusal discipline that makes χ a conjunction is inherited from the verified-compilation literature, and the strength of the inheritance determines how much of the ordering problem is forced and how much is chosen.

Two families of technique exist. The first proves the compiler correct once and for all, of which Leroy 2009 is the canonical instance for a realistic C compiler and Kumar and colleagues 2014 the analogous result for a functional language with a verified implementation down to machine code. The second validates each compilation run instead of the compiler, an approach Pnueli and colleagues 1998 introduced as translation validation, Sewell and colleagues 2013 applied to a verified operating-system kernel, and Kang and colleagues 2018 carried into the LLVM setting as credible compilation. Zhao and colleagues 2012 formalised the LLVM intermediate representation itself so that transformations over it may be verified, and Lopes and colleagues 2021 built bounded translation validation for LLVM into a practical tool.

Where full verification is unavailable, differential and randomised testing carries the load. Yang and colleagues 2011 found hundreds of defects in production C compilers by random program generation, Le and colleagues 2014 introduced equivalence modulo inputs as a general oracle for compiler testing, and Chen and colleagues 2016 applied coverage-directed differential testing across virtual machine implementations. The present generator sits in this second family. Its oracle is differential execution of the same bytecode on the reference virtual machine and on the lowered native code, which is the equivalence-modulo-inputs pattern with the input space supplied by hand rather than generated.

The consequence for the ordering problem is that partial implementation is not merely permitted but required to be explicit. A generator in this tradition cannot silently approximate. Every unimplemented instruction is a refusal, every refusal propagates to the whole compilation unit through the conjunction, and the ordering problem therefore acquires its characteristic structure directly from the verification stance rather than from any property of the instruction set. A generator willing to emit unverified approximations would face a smooth ordering problem with diminishing returns and would be able to invoke the classical greedy guarantee. The verification stance is what makes the objective supermodular and the guarantee unavailable, which is a cost of that stance and belongs plainly beside its benefits.

Static Analysis, Undecidability, and the Conservative Stance

The generator consumes shape information produced by an abstract interpretation in the sense of Cousot and Cousot 1977, which is also the framework within which Blanchet and colleagues 2003 built a static analyser for large safety-critical avionics software and within which Regehr and colleagues 2005 eliminated stack overflow in embedded software by bounding stack depth statically. The last of these is the closest published analogue to the native memory-bound problem the present workstream faces.

The conservative stance the language adopts, under which programs whose bounds cannot be proven are rejected, is forced by classical undecidability. Rice 1953 established that every non-trivial semantic property of programs is undecidable, and Landi 1992 sharpened the consequence for static analysis specifically. A verifier that admits only what it can prove will therefore reject some programs that are in fact well behaved, and the size of that gap is a design parameter and not a defect.

This bears on the ordering problem in a way not immediately obvious. The corpus measured below consists of programs the front end accepts, which is a population already shaped by the conservative stance. Instructions associated with constructs the verifier rejects cannot appear in the corpus at any frequency, so their measured blocking contribution is zero for a reason unrelated to demand. The measurement therefore estimates blocking frequency conditional on admissibility, and an instruction whose frequency would be high in an unconstrained population may measure at zero here. No such case is known to arise in the present data, but the conditioning is a real limitation of the method and not a hypothetical one.

Worst-Case Resource Analysis and the Native Transfer Problem

The reason the whole programme exists is the resource-bound guarantee, and the literature on that guarantee is what determines whether native lowering can preserve it.

Hard real-time scheduling, in the form Liu and Layland 1973 established, presupposes a worst-case execution time for each task. Obtaining that number is the subject of a substantial literature surveyed by Wilhelm and colleagues 2008 and earlier by Puschner and Burns 2000. The dominant bounding technique is implicit path enumeration, introduced by Li and Malik 1995, which expresses the worst-case path as an integer linear program over basic-block execution counts. The precision of any such bound on real hardware depends on microarchitectural modelling, for which Ferdinand and Wilhelm 1999 gave cache behaviour prediction and Reineke and colleagues 2007 characterised the predictability of cache replacement policies themselves.

The memory-bound side draws on the region and arena literature. Tofte and Talpin 1997 introduced region-based memory management with static inference of region lifetimes, Grossman and colleagues 2002 carried regions into a safe systems language, Hanson 1990 gave the practical arena discipline of allocation by object lifetime, and Berger and colleagues 2002 examined empirically when custom allocation actually pays. Keleusma’s arena model is of this family, and the native transfer question is whether a bound proven over the arena survives lowering to code whose stack frames are chosen by a register allocator in the tradition of Chaitin 1982.

A companion measurement, recorded in the project’s decision register rather than in a separate article, establishes that the per-function frame size is recoverable from the emitted object file, that it folds to a compile-time constant, and that it varies by a factor of roughly thirty depending on whether the middle-end promotion pass of Cytron and colleagues 1991 has run. That measurement is what makes the Regehr and colleagues 2005 approach applicable here, since it supplies the per-function weights their longest-path analysis consumes. The ordering relevance is that none of this work is unblocked by the instruction classes the naive ordering favoured.

Corpus-Based Empirical Study of Programs

The method employed here belongs to a tradition with a clear origin. Knuth 1971 collected and statically analysed a corpus of FORTRAN programs for the explicit reason that the profession’s beliefs about which language constructs mattered were untested, and found the distribution of constructs to be sharply different from the folklore. The paper is the direct methodological ancestor of the instrument described below, and its central move, which is to measure the population rather than reason about the language, is the move this article recommends generalising.

The tradition has continued. Basili and Weiss 1984 formalised the collection of valid software engineering data and the failure modes of informal collection. Richards and colleagues 2010 measured the dynamic behaviour of deployed JavaScript programs and found systematic divergence between what the language permits, what the literature assumed programs did, and

what programs actually did. Allamanis and Sutton 2013 and Dyer and colleagues 2013 built infrastructure for corpus mining at repository scale, and Ray and colleagues 2014 applied such infrastructure to language-level questions about code quality.

Two lessons from this tradition bear directly on the present measurement. The first is that the gap between permitted and actual is routinely large, which is exactly the gap the zero result below occupies, since the four instructions in question are fully supported by the language and used by no program in the corpus. The second concerns corpus composition, which dominates conclusions, a point Ray and colleagues 2014 and its subsequent reanalyses illustrate at length, and which the threats-to-validity section below treats as the principal limitation of this work, never as a formality.

Coverage Adequacy, Mutation, and What a Passing Suite Permits

The instrument reports a coverage figure, and the software-testing literature has spent decades establishing what coverage figures do and do not mean. The parallel is close enough to draw explicitly.

Zhu, Hall and May 1997 surveyed test coverage and adequacy criteria and the relationships among them. The decisive empirical result is Inozemtseva and Holmes 2014, which found that coverage is not strongly correlated with test-suite effectiveness once suite size is controlled, and therefore that a coverage percentage is a poor proxy for the property anyone actually wants. The structural analogy to the present case is exact. Instruction-level coverage ρ^{inst} is the attractive, easily computed, and largely uninformative measure. Unit-level coverage ρ^{unit} is the one that tracks delivered capability, and the two differ here by $\Gamma = 0.534$.

The testing literature also solved the selection problem this article poses, in its own setting and with the same formal object. Reducing a test suite to a subset preserving coverage is set cover, stated as such by Harrold, Gupta and Soffa 1993, and the ordering variant is test-case prioritisation, formalised by Rothermel and colleagues 2001 and surveyed with selection and minimisation by Yoo and Harman 2012. Safe selection, meaning selection that cannot discard a test capable of exposing a difference, is due to Rothermel and Harrold 1997. **The cautionary result belongs beside them.** Wong and colleagues 1995 found that minimising a suite while holding coverage constant can reduce fault-detection effectiveness, which is the same warning the present article issues in the opposite direction. **A coverage-preserving reduction and a coverage-maximising ordering are the same optimisation read forwards and backwards, and both are vulnerable to the measure standing in for the goal.**

The correctness discipline used throughout the development this article reports is mutation testing, introduced by DeMillo, Lipton and Sayward 1978 and surveyed by Jia and Harman 2011. Every structural assertion in the generator’s test suite carries a must-fire case, meaning a deliberate defect injected into the lowering that the assertion is required to detect, and a must-not-fire case, meaning a known-clean input on which it is required to stay silent. The empirical justification for treating mutants as a proxy for real defects is given by Andrews and colleagues 2005 and strengthened by Just and colleagues 2014, which found mutant detection to be correlated with real-fault detection to a degree that supports the practice.

The practice earned its place during this work instead of being adopted on authority. Of twenty-two mutations run across the increments this article reports, five failed to fire on first execution, and those five were of four distinct kinds.

One was a **null mutation**, where the mutated code is semantically identical to the original, so no test could distinguish it and none should. Changing an arithmetic shift to a logical one before a truncation that discards the differing bits is of this kind. A null mutation is not evidence of a coverage gap and treating it as one leads to writing a test that can never fail.

One was a **real coverage gap** in a target-specific case, where the hardware happened to define the behaviour that the undefined-behaviour rule permitted the compiler to exploit, so no behavioural test on that target could observe the defect. The response is a structural assertion over the emitted code.

Two were **vacuous tests**, whose test data carried a symmetry that concealed the asymmetry under test. In one, two branches of a differential case returned identical values. In the other, the callee of a cross-function call performed a commutative operation, so exchanging its arguments changed nothing. The response is redesigned input.

One was **genuinely unobservable**, a value no program in the language can read. The response is an explicit statement that the property is untested and untestable, which is the only honest option.

The four demand different responses and were nearly conflated. Only the mutation runs distinguished them, and the vacuous-test kind recurred three times across the work despite being actively watched for, which is the strongest available evidence that the failure is structural, so inattention does not explain it.

Submodular Optimisation, Coverage, and Prioritisation

The formal home of the ordering problem is the maximisation of a set function under a budget, and the relevant results have been stated above in the course of establishing which of them apply. The complementary literature is on prioritisation as an engineering practice rather than as an optimisation problem.

Karlsson and Ryan 1997 introduced a cost-value approach to prioritising requirements, in which candidate requirements are scored on value and on cost and ordered by the ratio, which is structurally the leverage ratio λ defined above. Their contribution for present purposes is the observation that practitioners reliably estimate cost and reliably fail to estimate value, so that a ratio-based method degenerates in practice to a cost-based one unless value is measured rather than elicited. That is precisely the failure this article documents in the compiler setting.

The regression-testing literature supplies the closest analogue with an empirical base. Elbaum and colleagues 2002 studied test-case prioritisation across a family of empirical studies and found that orderings informed by measured fault-detection history substantially outperform orderings informed by structural properties of the tests. The transferable finding is that a measured signal about outcomes beats a structural signal about artefacts, and that the margin is large enough to justify the measurement cost.

The Identification Problem for Blocking Attribution

The identification problem is the question of separating the blocking contribution of an individual instruction from the contributions of the instructions that co-occur with it. A unit blocked by four distinct unimplemented instructions is unblocked by none of them individually, so naive per-instruction attribution overstates every contribution. The overstatement obeys

$$\sum_{\iota \in I \setminus S} |B(\iota, S)| \;\le\; \Big| \bigcup_{\iota \in I \setminus S} B(\iota, S) \Big| \;\le\; M \cdot \left(1 - \rho^{\mathrm{unit}}(S)\right)$$

where the first inequality is an equality only when the blocking sets are pairwise disjoint, which is to say only when no unit is blocked by two distinct instructions. In the corpus below

the blocking sets are very far from disjoint, and the singleton blocking sets are in fact mostly empty, for the supermodularity reason established above.

The principled resolution of the attribution problem is the value of Shapley 1953, which distributes the total unblocking across contributors by averaging each contributor's marginal gain over all orders in which the contributors might be added. **It is the only attribution satisfying efficiency, symmetry, the null-contributor property and additivity together**, and Young 1985 showed that additivity may be replaced by a monotonicity requirement without changing the answer, which matters here because monotonicity in the marginal gain is the property an ordering argument actually wants. Owen 1972 gave the multilinear extension that makes the average tractable to reason about. Writing $\mathcal{W}$ for the set of workstreams, the Shapley attribution of workstream W is

$$\phi(W) = \sum_{T \subseteq \mathcal{W} \setminus \{W\}} \frac{|T|!\,(|\mathcal{W}| - |T| - 1)!}{|\mathcal{W}|!} \Big[\rho^{\text{unit}}\big(S \cup \bigcup T \cup W\big) - \rho^{\text{unit}}\big(S \cup \bigcup T\big) \Big]$$

which satisfies the efficiency property

$$\sum_{W \in \mathcal{W}} \phi(W) = \rho^{\text{unit}}\big(S \cup \bigcup \mathcal{W}\big) - \rho^{\text{unit}}(S) = 1 - \rho^{\text{unit}}(S)$$

so the attributions sum exactly to the total unblocking with no double counting, which is precisely the property the naive per-instruction count lacks. With $|\mathcal{W}| = 6$ the exact computation requires $2^6 = 64$ evaluations of ρ^{unit}, each a linear pass over the corpus, so the exact Shapley attribution is computationally trivial here and its omission is a matter of scope, since tractability was never the obstacle.

The article therefore reports two statistics with different properties instead of attempting a single attribution. The first is the instance count $n(\iota)$, an upper bound on effort saved that carries no unblocking claim whatsoever. The second reports the first-blocker distribution, which assigns each blocked unit to exactly one workstream and therefore partitions the blocked population without double counting,

$$\sum_{W} f(W) = M \cdot \big(1 - \rho^{\text{unit}}(S)\big), \qquad f(W) = \big|\{c_m : \text{first blocking instance of } c_m \in W\}\big|.$$

The first-blocker assignment is order-dependent within a unit. Writing π for the instruction-stream order and $f_\pi(W)$ for the resulting partition, the article reports f_π for the natural π and does not compute

$$\bar{f}(W) = \mathbb{E}_{\pi \sim \mathcal{U}}\big[f_\pi(W)\big]$$

over the uniform distribution on orders, nor the full blocking lattice over subsets of $I \setminus S$. This is stated outright, never left as an implicit claim to completeness. The first-blocker partition should be read as a cheap surrogate for ϕ, agreeing with it in ordering when one workstream dominates and unreliable when several are comparable.

Method

The instrument compiles every Keleusma source file in four corpus directories through the reference front end, walks the resulting instruction streams, classifies each instance as lowered or blocking, attributes blocking instances to a workstream by a total function over the instruction set, and classifies each compilation unit by the lowerability indicator χ.

Three properties bound the strength of the conclusions.

The instrument measures the reference compiler's output rather than source text, following the precedent Knuth 1971 set in measuring what programs actually contain, not what their authors assume, and the data-collection discipline Basili and Weiss 1984 formalise. This is the correct choice, because the generator consumes bytecode rather than source, and because the relationship between surface syntax and emitted instructions is not obvious in this language. A prior increment established that the four instructions named for ordinary arithmetic do not carry integer operands at all, and that all integer arithmetic is emitted as a checked form followed by a discard of two of its three results. Any instrument reasoning over source text would have miscounted that entirely.

The instrument excludes files the front end rejects. Five of the sixty-three files were rejected, all of them real-time operating system scripts referring to host functions registered by the embedding application rather than declared in the script. The exclusion is environmental and not linguistic, and including those files would add instances to the native application binary interface class, reinforcing the reported conclusion instead of qualifying it.

The instrument carries a guard assertion

$$\big(\text{files compiled} > 10\big) \ \wedge \ \big(N > 1000\big)$$

and fails loudly otherwise. The assertion exists because a path error in corpus discovery would silently yield a small clean sample, and every proportion reported below would then be noise presented as a measurement. The characteristic failure mode of a measurement instrument is a plausible number rather than an obvious error, a point Mytkowicz and colleagues 2009 make forcefully for performance measurement, where bias from incidental factors such as link order and environment size was found sufficient to reverse published conclusions.

Results

The corpus comprises sixty-three files, of which fifty-eight compiled, yielding $M = 496$ compilation units and $N = 73{,}434$ instruction instances. The implemented subset was $|S| = 39$ of $|I| = 66$.

This is the headline result and everything else in the article is either support for it or a consequence of it.

The two coverage measures diverge sharply.

$$\rho^{\text{inst}}(S) = \frac{64{,}116}{73{,}434} = 0.8731, \qquad \rho^{\text{unit}}(S) = \frac{168}{496} = 0.3387$$

$$\Gamma(S) = 0.8731 - 0.3387 = 0.5344$$

The generator handles the large majority of instruction instances and the minority of compilation units. A programme reporting eighty-seven percent coverage would be reporting a number no consumer of the generator can use, because no consumer executes an instruction instance in isolation.

The measurement repeated at a larger implemented subset, and a third level

The figures above were taken at $|S| = 39$. The subset has since reached $|S| = 46$, and repeating the measurement is instructive both because the numbers move and because it exposed a level of the conjunction the original analysis missed.

$$\rho^{\text{inst}}(S') = \frac{71{,}948}{73{,}434} = 0.9798, \qquad \rho^{\text{unit}}(S') = \frac{432}{496} = 0.8710, \qquad \Gamma(S') = 0.1088$$

The gap between the two measures has closed from 0.534 to 0.109, which is the expected behaviour, since as S approaches the union of the corpus's requirement sets, both measures approach one and the gap between them must vanish.

But a compilation unit is not what a consumer deploys. A consumer deploys a MODULE, and a module lowers only if every one of its compilation units does. That is the same conjunction applied once more, and it had been left out of the analysis entirely. Writing $\mathcal{P}$ for the corpus of programs and $u(P)$ for the compilation units of program P,

$$\rho^{\text{prog}}(S) = \frac{1}{|\mathcal{P}|} \sum_{P \in \mathcal{P}} \prod_{c \in u(P)} \chi(c, S)$$

and measured at $|S'| = 46$,

$$\rho^{\text{prog}}(S') = \frac{12}{58} = 0.2069.$$

So the three measures at the same implemented subset are 0.980, 0.871 and 0.207. The collapse from the middle figure to the outer one is a factor of

$$\frac{\rho^{\text{unit}}(S')}{\rho^{\text{prog}}(S')} = \frac{0.8710}{0.2069} = 4.21$$

and the article as first drafted quoted the middle one. That is the same category of error the article was written to describe, committed one level up, in the section reporting the error. The general form is that a conjunction exists at every level of aggregation the consumer's requirement spans, and an analysis must find the OUTERMOST one, not the first one that looks like a unit of work.

Ground truth, and why the original instrument could not supply it

The figures above at $|S| = 39$ came from a classification function that mirrored the lowering by hand. That mirror is a second implementation of the acceptance predicate, and it drifted, because the implemented set moved three times, each move requiring an edit to a list the lowering itself never reads.

The $|S'| = 46$ figures for ρ^{prog} come instead from calling the real entry point on every corpus program and counting successes, which cannot drift by construction. Where the two disagree the second is authoritative. **An instrument that mirrors the system under measurement measures the mirror**, and the defence is to call the system.

The blocking instances and first-blocker partition distribute as follows.

Workstream W	$\sum_{\iota \in W} n(\iota)$	$f_{\pi}(W)$
Data segment	7832	267
Native application binary interface	1057	9
Composites	331	28
Sub-coroutines	98	24
Typed arithmetic	0	0
Floating point and fixed point	0	0

At $|S'| = 46$, with the data segment implemented, the same table reads as follows.

Workstream W	$\sum_{\iota \in W} n(\iota)$	$f_{\pi}(W)$
Native application binary interface	1057	11
Composites	331	28
Sub-coroutines	98	25
Typed arithmetic	0	0

The data segment has left the table entirely, which is what implementing the leading blocker is supposed to look like. The native interface remains the instance-count leader and the blocked-unit laggard, so the non-monotonicity the article exhibits is not an artefact of one measurement.

The data-segment workstream accounts for

$$\frac{267}{496 - 168} = \frac{267}{328} = 0.814$$

of blocked units. The five most frequent individual blocking instructions are the data-segment slot write at $n = 3385$, the slot read at $n = 2109$, the indexed read at $n = 1456$, the verified native call at $n = 1057$, and the indexed write at $n = 882$.

The native application binary interface class instantiates the non-monotonicity claim at the workstream level. Writing W_a for the native-call workstream and W_b for the composite workstream, and extending n additively to workstreams,

$$n(W_a) = 1057 > 331 = n(W_b), \qquad f_{\pi}(W_a) = 9 < 28 = f_{\pi}(W_b)$$

so ranking by instance count places the native interface second and ranking by blocked units places it fourth. The instruction-level claim stated earlier is the same phenomenon one level down. It is exhibited here at workstream granularity because, for the supermodularity reason given above, the instruction-level blocking sets are mostly empty and the instruction-level comparison would be a comparison of zeroes. Its instances concentrate in a small number of large units already blocked for other reasons.

The sub-coroutine class deserves separate comment because its blocking count of twenty-four understates its architectural weight. Coroutines in the sense Conway 1963 introduced, and in the modern treatment of Moura and Ierusalimschy 2009, are the load-bearing primitive for the streaming execution model the language targets, so the workstream sits on the critical path for reasons the blocking count does not express. The ordering principle advocated here is a claim about marginal capability, leaving architectural sequencing untouched, and where a low-blocking workstream is a prerequisite for a high-blocking one the dependency order dominates. No such dependency is known to hold between coroutines and the data segment

in the present case, and the data-segment lowering does not appear to require suspension support. That is an engineering judgement about the two designs rather than a measured result, and it is the one load-bearing step in this article's recommendation that rests on judgement rather than on the instrument.

The independence null and the clustering coefficient

The unit-level collapse invites an obvious explanation, that unimplemented instructions are simply spread thinly across many units. That explanation is testable. **Comparing an observed count structure against an independence null is standard practice**, and the general statistic for departure from it is a test for overdispersion, treated by Dean 1992. **The expectation that the departure will be large is not a guess either.** Defects in software are known empirically to concentrate rather than distribute evenly, established by Fenton and Ohlsson 2000 and measured at scale by Ostrand and Weyuker 2002 and Ostrand, Weyuker and Bell 2005, who found small fractions of modules carrying most faults. **Unimplemented instructions are not defects, but the concentration argument transfers**, because both arise from uneven use of a shared vocabulary across a corpus. Under an independence null in which each instruction instance in a unit is lowered with probability $p = \rho^{\text{inst}}(S)$ independently, the expected unit-level coverage is

$$\rho_0^{\text{unit}} = \frac{1}{M} \sum_{m=1}^{M} p^{|c_m|}$$

and the clustering coefficient is the ratio of observed to null,

$$\Phi = \frac{\rho^{\text{unit}}(S)}{\rho_0^{\text{unit}}}.$$

Evaluating with the measured length distribution gives

$$\rho_0^{\text{unit}} = 6.152 \times 10^{-2}, \qquad \Phi = \frac{0.3387}{0.06152} = 5.51$$

so the blocking instructions are clustered, and by a factor of roughly five and a half, which is moderate. Repeating the calculation at $|S'| = 46$ gives $\rho_0^{\text{unit}} = 3.275 \times 10^{-1}$ and $\Phi = 2.66$, a weaker clustering, which is again the expected direction, since as fewer instructions block, those that remain have less opportunity to concentrate. Unimplemented instructions concentrate in units that use them repeatedly, which is the expected consequence of a data-segment-heavy workload, and the concentration is real but moderate.

A methodological near-miss worth reporting

The null above must be evaluated per unit, never at the mean unit length. Since $x \mapsto p^x$ is convex for $0 < p < 1$, the inequality of Jensen 1906 gives

$$\frac{1}{M} \sum_{m=1}^{M} p^{|c_m|} \;\geq\; p^{\bar{L}}, \qquad \bar{L} = \frac{1}{M} \sum_{m=1}^{M} |c_m| = 148.05$$

and the two quantities differ enormously here because the length distribution is heavily right-skewed, with mean 148.05 against median 69. Numerically,

$$p^{\,\bar{L}} = 0.8731^{148.05} = 1.884 \times 10^{-9} \qquad \text{against} \qquad \frac{1}{M}\sum_{m} p^{|c_m|} = 6.152 \times 10^{-2}$$

a discrepancy of more than seven orders of magnitude. Had the null been evaluated at the mean length, the clustering coefficient would have been reported as

$$\Phi_{\text{wrong}} = \frac{0.3387}{1.884 \times 10^{-9}} = 1.80 \times 10^{8}$$

instead of 5.51, and the article would have claimed an overwhelming clustering effect where the true effect is moderate. The error would have been in the direction of a more striking finding, which is the direction in which errors are least likely to be questioned.

Confidence bounds on the zero counts

Two instruction classes occur zero times as blockers. A zero count is not an assertion of impossibility, and the appropriate summary is an upper confidence bound and not a point estimate. For zero events observed in n independent trials, the rule of three stated by Hanley and Lippman-Hand 1983 and examined by Jovanovic and Levy 1997, a normal-approximation shortcut on the exact interval of Clopper and Pearson 1934, gives the approximate ninety-five percent upper bound

$$(1-p)^{n} = 0.05 \implies p = 1 - 0.05^{1/n} \approx \frac{\ln 20}{n} = \frac{2.996}{n}$$

$$\hat{p}_{\text{upper}} \approx \frac{3}{n}$$

which follows from solving $(1-p)^{n} = 0.05$ and using $\log(0.05) \approx -3$. Applied to the two units of account,

$$\hat{p}^{\text{inst}}_{\text{upper}} = \frac{3}{73{,}434} = 4.09 \times 10^{-5}, \qquad \hat{p}^{\text{unit}}_{\text{upper}} = \frac{3}{496} = 6.05 \times 10^{-3}$$

so the true per-instance rate of typed arithmetic in this population is below roughly four in one hundred thousand, and the per-unit rate is below roughly six in one thousand, each at the stated confidence. The independence assumption underlying the rule of three is violated here, since instruction instances within a unit are strongly dependent, and the per-unit bound is consequently the more defensible of the two. For a boundary proportion of this kind the interval of Wilson 1927, or the adjusted interval Agresti and Coull 1998 recommend in preference to the exact method, would be the more careful summary, and the rule of three is reported here because it is the form in which such a result is conventionally communicated rather than because it is the tightest available.

The Zero Result, and the Recommendation It Falsified

The second zero is the finding of consequence, because the working session immediately preceding the measurement closed with a formal recommendation that the next research spike address exactly that class. The reasoning was as follows. The four instructions are reachable only for byte, fixed-point and floating-point operands. The byte representation had just been

settled by a measurement contributed from a parallel development line. The remaining obstacle was that the instruction does not record which of the three types its operands carry, and the lowerings differ, since a byte addition must mask its result to eight bits while a fixed-point addition must not. Recovering operand types would therefore unblock the class. The recommendation observed further that the same type information is required by the composite workstream for field offsets, and concluded that operand type recovery was the highest-leverage remaining work by a clear margin.

Every step of that reasoning is correct. The conclusion is worthless. Writing W_{typed} for the class,

$$n(W_{\text{typed}}) = 0 \implies B(W_{\text{typed}}, S) = \emptyset \implies \Delta\rho^{\text{unit}}(W_{\text{typed}}) = 0$$

and the composite workstream it was partly justified by satisfies $f_\pi(W_{\text{composite}}) = 28$ against the data segment's 267, a ratio of

$$\frac{f_\pi(W_{\text{data}})}{f_\pi(W_{\text{composite}})} = \frac{267}{28} = 9.54.$$

The failure is not a reasoning error, and the distinction matters because the corrective differs. The argument contained no invalid step. It reasoned carefully about the structure of a problem while never asking how often the problem occurs. A dependency analysis answers what must be true before an instruction can be implemented. It does not answer whether anyone needs the instruction, and the two questions are independent. The recommendation was a well-constructed answer to a question nobody had asked. Meehl 1967 observed the general form of this hazard in comparing theory-testing practice across disciplines, namely that a methodology can be internally rigorous and systematically uninformative about the question of interest.

A refinement the same work later forced, which sharpens the lesson

The account above is correct and incomplete, and the missing part changes what a reader should take away.

Operand type recovery was dismissed on the strength of a zero. Some increments later, the composite workstream turned out to require exactly that capability, for an unrelated reason, namely that `Op::NewComposite` carries the composite kind, the field count and the total byte size, and NOT the per-field widths, while packing is tight and in declaration order. Writing a composite body therefore needs each field's width and no instruction records it. That is 18 compilation units, against the zero the original assessment measured.

So the measurement was right and the inference from it was too broad. It established that type recovery was worthless FOR THE INSTRUCTION CLASS PROPOSED, and the conclusion drawn was that type recovery was worthless. A capability is not made valueless by the emptiness of one demand for it, and a frequency measurement that redirects work away from a capability has said something about that demand rather than about the capability.

The sharpened statement is therefore in two parts rather than one. Measure demand before ordering by structure, because a dependency argument cannot see frequency. And scope the negative result to what was measured, because a zero counts occurrences of one thing and licenses no conclusion about anything else. The first part saved the work described here. The second part was learned by getting it wrong in the same programme, a few increments later, and having to correct a document that had recorded the over-broad version as a finding.

The corpus check that falsified it cost approximately twenty minutes to build and two seconds to run. The cost of the work it redirected cannot be stated with comparable confidence, because that work was never performed. **The numerator below is measured and the denominator is an estimate of a counterfactual**, so the ratio should be read as an order of magnitude and not as a quantity. Writing $\kappa_{\mathrm{measure}}$ for the measured cost and κ_{spike} for the estimated cost of the redirected spike,

$$\frac{\kappa_{\mathrm{measure}}}{\kappa_{\mathrm{spike}}} \sim 10^{-2}$$

so the measurement plausibly cost one or two orders of magnitude less than the work it redirected, and the redirected work would have delivered $\Delta\rho^{\mathrm{unit}} = 0$ exactly. Only the last of those three statements is a measurement. An earlier draft gave this ratio as 0.019 against a sixteen-hour denominator, presenting an invented figure to two significant figures inside a formal expression, which lent it a precision it had not earned.

Measurement Error and the Direction of Bias

This article reports **four** errors made in the course of producing it, and their common direction is the most transferable observation available. **Three are described below and the fourth is described inside the description of the third**, which is not an accident and is the point of the section.

The first is the convexity shortcut described above, **which is Jensen's inequality applied in the wrong direction**, since p^{x} is convex in x and the mean of the powers therefore exceeds the power of the mean rather than equalling it. It would have inflated a clustering coefficient by seven orders of magnitude. The second was a false modularity theorem, where the coverage objective was asserted to be submodular so that a classical approximation guarantee could be invoked, when the objective is supermodular and no such guarantee exists. The third occurred while assembling this article's references. Of ninety-one candidate digital object identifiers supplied from memory, four resolved to entirely different works and one did not resolve at all, an error rate of five and a half percent. One resolved to a paper on record allocation for drum storage in place of a paper on optimal code generation for expression trees, and one to a paper on a lambda calculus of objects in place of a paper on lightweight bytecode verification. Each of the four would have returned a successful response to a reachability check, so only comparison of the resolved title against the claimed title detected them.

A draft of this article reported this rate as near ten percent over forty-two candidates. That figure is the first three verification batches taken alone, which is the worse subsample, and the later batches were run after the sentence was written and never folded into it. The effect was to report a more alarming number than the data supported, in the section of the article devoted to the tendency to report more alarming numbers than the data support. The error is left visible here instead of silently repaired, because a corrected figure with no record of the correction would conceal the most instructive part of the episode.

All four errors ran toward a more striking or better-founded-looking result. The clustering error would have produced a dramatic finding, the modularity error would have supplied a theoretical guarantee for the article's central recommendation, the citation errors would have furnished authority, and the misstated defect rate would have made the cautionary point more forcefully. None ran in the direction of weakening a claim, and the fourth was committed while writing the paragraph warning against the first three.

This direction is documented at scale in the empirical-methods literature. Rosenthal 1979 described the suppression of null results and estimated the number of unpublished null studies required to overturn a published effect. Simmons, Nelson and Simonsohn 2011 demonstrated

that undisclosed flexibility in data collection and analysis suffices to produce statistically significant evidence for false propositions without any conscious dishonesty. Ioannidis 2005 derived the consequences for base rates of published findings, and Munafo and colleagues 2017 set out the corresponding reform programme. The mechanism in each case is not fraud but asymmetric scrutiny, in which a result the author expects and welcomes receives less checking than one that surprises or disappoints.

The engineering translation is direct. An error that weakens a claim tends to be noticed because it is unwelcome. An error that strengthens one is congenial and passes unexamined, and a formal apparatus is an efficient way to manufacture such errors because its conclusions carry borrowed authority. Three defences were used here and are recommended generally. Evaluate aggregates over the observed distribution, never at its mean, whenever the aggregating function is not affine. Test a formal claim by brute-force enumeration over instances small enough to check exhaustively before relying on it. Verify every citation against the resolved record rather than against memory, since the failure mode is a plausible reference and not a broken link.

Threats to Validity

The corpus is drawn from one project's own examples, standard library and self-hosted compiler stages, and is therefore not a sample of the programs the generator will eventually serve. Writing $\mathcal{D}_{\text{corpus}}$ for the empirical distribution over units and $\mathcal{D}_{\text{target}}$ for the eventual target population, every quantity reported above estimates

$$\mathbb{E}_{c \sim \mathcal{D}_{\text{corpus}}}[\cdot] \qquad \text{rather than} \qquad \mathbb{E}_{c \sim \mathcal{D}_{\text{target}}}[\cdot]$$

and no importance weighting is applied because the target distribution is not yet observable. The corpus over-represents the self-hosted compiler, a text-processing workload with heavy data-segment use, and under-represents the signal-processing and embedded-control workloads the roadmap names as target applications. A defensible reading is that the data segment leads for the near-term consumer and that the ranking should be re-measured when the target population changes. The sensitivity of conclusions to corpus composition is the standard caution of this literature, and Blackburn and colleagues 2006 made the point concrete in constructing a benchmark suite specifically because prior suites had shaped conclusions unrepresentatively, while Blackburn and colleagues 2016 set out the corresponding evaluation discipline.

The first-blocker attribution is order-dependent, as formalised above, and the full blocking lattice is not computed. For the leading result this is unlikely to matter, since

$$\frac{f_{\pi}(W_{\text{data}})}{\max_{W \neq W_{\text{data}}} f_{\pi}(W)} = \frac{267}{28} = 9.54$$

and an order-dependence artefact would have to be extreme to invert a margin approaching ten to one. For the ordering of the second, third and fourth workstreams, at 28, 24 and 9, the artefact could plausibly reorder them, and no claim about that ordering should be read as established. The exact cost is derived here instead of quoted, since it is the reason the shortcut is hard to defend. The Shapley value needs ρ^{unit} evaluated on every subset of workstreams,

$$\left|2^{\mathcal{W}}\right| = 2^{|\mathcal{W}|} = 2^{6} = 64$$

and each evaluation is one pass over 496 units. Since the exact Shapley attribution therefore requires only sixty-four corpus passes, the honest characterisation is that the surrogate was used for scope reasons and that the secondary ordering could be settled cheaply whenever it matters.

The cheapness is a property of the aggregation, since the method itself does not become cheaper, and the distinction bounds how far the result generalises. Six workstreams give sixty-four subsets. The same attribution at instruction granularity ranges over subsets of the unimplemented instruction set, which is not enumerable, and there the exact value is unavailable rather than merely unattempted. **The literature has an answer for that regime**, in the sampling estimator of Castro, Gomez and Tejada 2009, which averages marginal gains over randomly drawn permutations and converges at the usual rate in the number of samples, and in the closely related attribution scheme of Strumbelj and Kononenko 2013. **Neither was needed here and both would be needed for any instruction-level restatement of the same question.**

The zero results are subject to the corpus caveat and are bounded above rather than asserted absent, as quantified in the confidence-bound section. The correct inference is not that the typed arithmetic class will never matter but that it does not matter now, which is the question an ordering principle asks.

The measurement is static. It counts instruction instances in compiled units, never executions, so it estimates the difficulty of lowering a program and says nothing about the time a program spends in any instruction. For the ordering question this is the correct unit, since a refusal is a static property, but the resulting figures must not be read as execution profiles. Georges and colleagues 2007 set out why execution-time claims require a different and considerably more careful methodology than the one applied here, and no such claim is made.

Finally, the measurement was performed by the same party that made the recommendation it falsifies, using an instrument that party wrote. The guard assertion described in the method section addresses the most likely failure mode of that arrangement, which is a silently empty sample, but it does not address the possibility of a classification function whose workstream boundaries were drawn, unconsciously, to produce a tidy result. The classification is a total function over sixty-six instructions and is available for inspection, which is offered as mitigation rather than as a claim that the concern is closed.

Pattern Extraction

The abstract mechanic generalises beyond compiler backends to any incremental capability programme in which a consumer requires a conjunction of features rather than any one of them.

The product form has a name as well, since it is the founding object of a mature discipline. A system that functions only when every one of its components functions is a series system, and the consequences of that structure were worked out for reliability by Esary and Proschan 1963. **The attribution question this article poses has a direct counterpart there**, in the component importance measure of Birnbaum 1968, which ranks components by the probability that the system's functioning turns on that component alone. That is the same quantity as a blocking set, computed in a different vocabulary.

The mechanic has three parts. First, when a consumer requires every element of a set to be present, delivered capability is governed by the product form $\chi = \prod_i \mathbf{1}[\,\cdot\,]$ and not by the sum form $\rho^{\text{inst}} = \frac{1}{N}\sum_i \mathbf{1}[\,\cdot\,]$, so per-element progress measures overstate delivered capability by the gap Γ, which grows with the dispersion of the missing elements. **That claim has computable extremes**, and the counting argument is the elementary one that underlies

covering problems generally, in Karp 1972 and Feige 1998. With $Q = N(1 - \rho^{\text{inst}})$ missing instances distributed over M units, the number of blocked units is bounded by

$$\left\lceil Q / L_{\max} \right\rceil \;\le\; \left|\{m : \chi(c_m, S) = 0\}\right| \;\le\; \min(Q, M)$$

In the present case $Q = 9{,}318$ against $M = 496$, so maximal dispersion would block every unit, giving $\rho^{\text{unit}} = 0$ and $\Gamma = \rho^{\text{inst}} = 0.8731$. The observed 328 blocked units and $\Gamma = 0.5344$ therefore sit well inside the dispersed extreme, **which is the same fact the clustering coefficient of 5.51 reports and is an independent way of seeing it.** Second, the natural ordering heuristics available to an implementer, which are implementation cost κ, architectural elegance, and dependency depth, are all properties of elements considered in isolation, and none carries information about $|B(\iota, S)|$. Third, the corrective is a frequency measurement over a representative population of consumers, and the measurement satisfies

$$\kappa_{\text{measure}} \ll \kappa_{\text{work}}$$

characteristically by one to two orders of magnitude, because the cost of counting occurrences is nearly independent of the cost of the work being ordered.

The observation is old. Knuth 1971 measured a corpus of FORTRAN programs precisely because the profession's assumptions about which constructs mattered were untested, and Amdahl 1967 had already given the arithmetic showing that effort spent outside the dominant term is bounded above by a small number regardless of how well it is spent. Writing f for the fraction of the work the improvement touches and s for how much faster that fraction becomes,

$$\text{speedup} = \frac{1}{(1 - f) + f/s} \xrightarrow[s \to \infty]{} \frac{1}{1 - f}$$

so an improvement addressing half the work cannot exceed a factor of two **however good it is**, a bound Gustafson 1988 later reframed without dissolving. **The parallel to the present case is exact.** An instruction that blocks nothing is a term with $f = 0$, and no quality of implementation raises its contribution above zero. The present article adds only that the same asymmetry applies to feature ordering under a conjunctive consumer, that the dominant term is not visible from the structure of the work, and that the conjunction makes the objective supermodular so that no approximation guarantee is available to substitute for the measurement.

The mechanic carries a self-application. An implementer reasoning carefully about dependency structure produces recommendations that feel well-founded, because they are well-founded with respect to the question of what is required. The recommendation acquires unearned authority from the rigour of the dependency argument, and the missing question is not visible from inside that argument. The defence is procedural and not intellectual. Before accepting an ordering derived from structure, measure the frequency, and treat the structural argument as establishing feasibility rather than priority.

A second self-application concerns the analysis itself, and this article contains four instances of it, comprising a convexity shortcut, a false modularity theorem, a defect rate in citations supplied from memory, and a misstatement of that rate over the worse subsample. All ran toward a more striking or better-supported result, which is the direction least likely to attract scrutiny, and none was caught by reading. Each was caught by an execution, respectively a numerical evaluation whose answer was implausible, a brute-force enumeration over small instances, a query against an authoritative record, and a recount over the full sample rather

than the remembered one. That the fourth was committed inside the passage cautioning against the first three is the clearest evidence available that awareness of a bias does not confer immunity to it. The general defence is to test the apparatus on cases small enough to enumerate before trusting it on the case of interest.

The general statement is that dependency analysis and demand measurement answer different questions, that the first is more intellectually satisfying and more readily available to an implementer, and that ordering decisions made on the first alone are unreliable in a direction the analysis itself cannot reveal.

The Contemporary Literature

The seven traditions above are the ones the argument sits inside, and they were surveyed for what they establish. **This section surveys what has happened lately**, on the standing expectation that an article of this kind should double as a review of the contemporary literature. The reading is not neutral, since each body of work is placed by what it says about the ordering question.

The short version is that the question has become more urgent since it was posed, and no more answered.

The number of targets multiplied

Backend bring-up used to be a rare activity performed by a few vendors, and it is now routine. An open instruction set with a ratified extension mechanism means anyone can add instructions, and the surrounding literature covers custom extension design, automatic extension identification, application-specific processors, accelerator compilation and vectorisation.

- A proposed synthesis method for Application-Specific…
- Accelerating H.264/HEVC video slice processing using…
- Automatic complex instruction identification for…
- Efficient Compilation for Application Specific…
- FPGA-based SHA-3 acceleration on a 32-bit processor via…
- Fast and accurate power estimation for…
- ISA customization for application specific instruction…
- Implementing an Application-Specific Instruction-Set…
- Timing speculation-aware instruction set extension for…
- A Domain-Specific Compiler for a Parallel Multiresolution…
- A RISC-V instruction set processor-micro-architecture…
- A basic linear algebra compiler for structured matrices
- An Application-Specific Instruction Set Processor for…
- Application specific instruction set processor for sensor…
- Development of a Code Generation Support System in…
- Exploring Compiler Optimization Opportunities for the…
- Hardware implementation of a SHA-3 application-specific…
- Matlab to C Compilation Targeting Application Specific…
- Oolong: A Baseband processor extension to the RISC-V ISA
- Outer-Loop Auto-Vectorization for SIMD Architectures…
- Regression Test Suites Optimization for…
- SHA-3 Instruction Set Extension for A 32-bit RISC…
- Vectorization in PyPy’s Tracing Just-In-Time Compiler
- Video SIMDBench: Benchmarking the Compiler Vectorization…
- A Domain-Specific Language and Compiler for…
- A MATLAB Vectorizing Compiler Targeting…

- Application-Specific Instruction Set Processors for Video…
- Automatic generation of fast BLAS3-GEMM: A portable…
- Compiler Techniques for Efficient MATLAB to OpenCL Code…
- Compiler auto-vectorization of matrix multiplication…
- Design of an Application Specific Instruction Set…
- Domain specific compiler for coordinated signal…
- Instruction set extension and hardware acceleration for…
- Intermediate-Code Generation
- Machine-Code Generation
- Metacasanova: an optimized meta-compiler for…
- Metamodeling and Code Generation in the Hardware/Software…
- Polyhedral Compiler Technology in Collaboration with…
- SuperGraph-SLP Auto-Vectorization
- A compiler for cyber-physical digital microfluidic…
- A low-cost synthesizable RISC-V dual-issue processor core…
- An application specific instruction set processor (ASIP)…
- Automatic Configurable Hardware Code Generation for…
- CAnDL: a domain specific language for compiler analysis
- COpt: A High Level Domain-Specific Language to Generate…
- Design and Simulation Of 64-Bit Hybrid Processor…
- Design of RLWE Cryptoprocessor Based on…
- Dominance-based duplication simulation (DBDS): code…
- Optimization of Specific Instruction Set Processor for…
- System on Chip Implementation of Compiler Stack with a…
- A compiler architecture for domain-specific type error…
- An Application-Specific VLIW Processor with Vector…
- An Application-specific Instruction Set Processor for…
- An Efficient Application Specific Instruction Set…
- Application Specific Instruction Set Processor Design for…
- Compiler-Assisted Selection of Hardware Acceleration…
- Compiler-support for Critical Data Persistence in NVM
- Enhancing Python Compiler Error Messages via Stack
- Proposal of Scalable Vector Extension for Embedded RISC-V…
- Research on Instruction Set Architecture of 40-Bit…
- Translating CUDA to OpenCL for Hardware Generation using…
- A Compiler Comparison in the RISC-V Ecosystem
- Agile Autotuning of a Transprecision Tensor Accelerator…
- BOSON - Application-Specific Instruction Set Processor…
- Lightweight Cryptographic Instruction Set Extension on…
- Really Embedding Domain-Specific Languages into C++
- A RISC-V Post Quantum Cryptography Instruction Set…
- An Agile Instruction Set Extension Method Based on the…
- An Interval Compiler for Sound Floating-Point Computations
- Compact native code generation for dynamic languages on…
- Development of RISC-V Based Soft-core Processor with…
- Exploring the RISC-V Vector Extension for the Classic…
- MLIR: Scaling Compiler Infrastructure for Domain Specific…
- RISC-VTF: RISC-V Based Extended Instruction Set for…
- Relaxed Peephole Optimization: A Novel Compiler…
- SSA Form and Code Generation
- Variable Bit-Precision Vector Extension for RISC-V Based…
- A Compiler for Sound Floating-Point Computations using…
- A Pluggable Vector Unit for RISC-V Vector Extension
- A Trigonometric Function Instruction Set Extension Method…

- An Efficient Application Specific Instruction Set…
- Audio Denoising Coprocessor Based on RISC-V Custom…
- Automatic compiler/interpreter generation from programs…
- Automating Cryptographic Code Generation
- Backward Graph Construction and Lowering in DL Compiler…
- Bratter: An Instruction Set Extension for Forward…
- Communications Signal Processing Using RISC-V Vector…
- Design of RISC Processor with IEEE754 Standard…
- Effective Performance Modeling and Domain-Specific…
- Efficient Support of the Scan Vector Model for RISC-V…
- Just-In-Time Compiler System in Aspect-Oriented…
- Lowering Barriers to Application Development With…
- MLIR-based code generation for GPU tensor cores
- OpenASIP 2.0: Co-Design Toolset for RISC-V…
- RVVRadar: A Framework for Supporting the Programmer in…
- “A Multi-Pass Compiler with Code Optimized Abstract…
- A RISC-V Instruction Set Extension for Flexible…
- A buffer overflow detection and defense method based on…
- An extension to the RISC-V instruction set architecture…
- Artifact for Lifting Code Generation of Cardiac…
- Automatically Localizing Dynamic Code Generation Bugs in…
- Building a domain-specific compiler for emerging…
- Design and Implementation of a Compiler Supporting RISC-V…
- Design, development and testing of a 16-bit reduced…
- Efficient Compiler Design for a Geometric Shape…
- Evaluating RISC-V Vector Instruction Set Architecture…
- Flexible and Efficient Implementation of CRYSTALS-KYBER…
- FlowPix: Accelerating Image Processing Pipelines on an…
- Implementation and Reliability Evaluation of a RISC-V…
- Integration of a Real-Time CCSDS 410.0-B-32…
- JIT Compiler Security through Low-Cost RISC-V Extension
- Lifting Code Generation of Cardiac Physiology Simulation…
- PEMBANGUNAN COMPILER DOMAIN SPECIFIC LANGUAGE SEBAGAI…
- PIMFlow: Compiler and Runtime Support for CNN Models on…
- RISC-V Instruction Set Architecture Extensions: A Survey
- Resource-efficient RISC-V Vector Extension Architecture…
- The Design of Optimized RISC Processor for Edge…
- Vectorized Nonlinear Functions with the RISC-V Vector…
- A Tensor Algebra Compiler for Sparse Differentiation
- An FPGA-Based RISC-V Instruction Set Extension and Memory…
- An MLIR-Based Compiler for Hardware Acceleration with…
- Compile-Time Analysis of Compiler Frameworks for Query…
- Compiler Testing with Relaxed Memory Models
- Configurable Loop Shuffling via Instruction Set Extensions
- Convex: A RISC-V Instruction Set Extension Scheme for…
- Designing RISC-V Instruction Set Extensions for…
- Enabling Fine-Grained Incremental Builds by Making…
- Evaluating and optimising compiler code generation for…
- Fast Template-Based Code Generation for MLIR
- Fully Automatic Compiler Retargeting and CV-X-IF Hardware…
- High Performance Instruction-Data Level Parallelism Based…
- Implementation of Application Specific Instruction set…
- Improving the Accuracy of Batik Classification using Deep…
- Intermediate-Code Generation

- LLVM Library for a Dedicated Processor Instruction Set –...
- Machine-Code Generation
- RVCE-FAL: A RISC-V Scalar-Vector Custom Extension for...
- Special Session: Reliability and Performance Evaluation...
- Whose Baseline Compiler is it Anyway?
- eCC++ : A Compiler Construction Framework for Embedded...
- A Domain-Specific Compiler for Embedded DSP Development...
- A Graph-Based Learning Framework for Compiler Loop...
- A RISC-V Vector Extension for Multi-word Arithmetic
- A Way to Identify Potential Functions for Vectorization...
- AI Edge Processor Using RISC - V Instruction Set...
- Accelerating Machine Learning using RISC-V Vector...
- Accelerating Machine Learning with RISC-V Vector...
- Accelerating NTT with RISC-V Vector Extension for Fully...
- Acceleration of McEliece Cryptosystem with Instruction...
- Analysis of the RISC-V Vector Extension for Vulkan...
- Application-Specific Instruction Set Processor
- Compiler-Like Code Generation for fUML: Reducing Overhead...
- Efficient TinyML Inference on a Fault-Tolerant RISC-V SoC...
- Eight-Bit Vector SoftFloat Extension for the RISC-V Spike...
- Fast Interpreter-Based Instruction Set Simulation for...
- Finding Bugs in MLIR Compiler Infrastructure via Lowering...
- Functional Validation of the RISC-V Unlimited Vector...
- Key Operator Vectorization for LeNet and ResNet Based on...
- Logic Gate Network Inference Acceleration with RISC-V...
- Low-Power Implementation of DSP Instruction Set Extension...
- Microarchitecture Design and Benchmarking of Custom SHA-3...
- Optimizing TinyEngine for the RISC-V Vector Extension
- Performance Evaluation of CNN using RISC-V Vector...
- RI-MAC: Optimising MAC Operation Using Custom RISC-V...
- RISC-TAE: Instruction Set Extension for Transformer Model...
- RISC-V SIMD Instructions - Vector Extension (Load/Store)
- RISC-V Processor Hardware Modelling with Custom...
- Research on RISC-V-based Edge Convolution Acceleration...
- SySTeC: A Symmetric Sparse Tensor Compiler
- Tensor Program Optimization for the RISC-V Vector...
- TinyML Unleashed: Accelerating TensorFlow Lite Micro...
- A Reinforcement Learning Environment for Automatic Code...
- An Embedded RISC-V Vector Extension for Edge-Oriented...
- CKTI: A Domain-Specific Compiler for Lowering CUDA...
- CREF-Lang: A domain-specific language and compiler for...
- Compiler-ASR: Bridging the IR-to-Assembly Gap for...
- Compiler-Assisted Instruction Fusion
- Design Space Exploration of RISC-V Vector Extension...
- Enabling Automatic Compiler-Driven Vectorization of...
- Evaluation and Benefit Modeling of Auto-Vectorization...
- Exploring Instruction Set Extension Emulation for...
- FPGA-Based ORB Accelerator: Effects of Compiler...
- FWHT-RVV: A RISC-V vector processor with FWHT instruction...
- Hikami: A Lightweight Hypervisor for Emulating RISC-V...
- Instruction Set Optimization for FM-Type Digital Signal...
- PERFORMANCE EVALUATION OF THE UETRV-PCORE USING RISC-V...
- TPDE: A Fast Adaptable Compiler Back-End Framework
- Thinking Fast and Correct: Automated Rewriting of...

- TinyGen: Portable and Compact Code Generation for Tiny…
- Vmxdotp: A RISC-V Vector ISA Extension for Efficient…

Every one of those projects faces the ordering question and none of the papers answers it. The extension-identification work comes closest, since it selects instructions by measured frequency over a corpus, **but it selects instructions to ADD TO A COMPLETE MACHINE for speed, not instructions to implement next in an incomplete compiler for capability.** The objective is a sum in that setting and a product in this one, which is the whole difference.

And one target the entire industry had to bring up at once

WebAssembly is the closest thing to a natural experiment. A new target appeared, and a great many implementations were brought up against it more or less simultaneously, producing work on runtimes, formal semantics, binary size, ahead-of-time compilation and runtime bugs.

- Adjustable-Cost Overlays for Runtime Compilation
- Augmenting JavaScript JIT with ahead-of-time compilation
- Bytecode-to-C Ahead-of-Time Compilation for Android…
- Error-tolerant processors: Formal specification and…
- Executable Semantics for the Formal Specification and…
- Formal Semantics of Runtime Monitoring, Verification…
- Is dynamic compilation possible for embedded systems?
- Javascript ahead-of-time compilation for embedded web…
- Runtime Value Numbering: A Profiling Technique to…
- Automated formal verification of the refined…
- Modular specification and verification of a…
- Testing-Based Formal Verification for Theorems and Its…
- Advanced ahead-of-time compilation for Javascript engine
- Ahead-of-time compilation of JavaScript programs
- DAME: Runtime-compilation for data movement
- Enhancing formal specification and verification of…
- Erratum to: Formal Description Techniques and Protocol…
- Formal Specification and Verification of Security…
- Formal verification of ABAP by Z specification
- Hyperhierarchy of Semantics - A Formal Framework for…
- KART – A Runtime Compilation Library for Improving HPC…
- Formal Specification and Verification of Self-Adaptive…
- HiPEAC compilation architecture
- Intrinsic Compilation Model to enhance Performance of…
- Reusing the Optimized Code for JavaScript Ahead-of-Time…
- A Customized Real-Time Compilation for Motion Control in…
- Floating-point Semantics of Analyzed Programs
- Formal Specification Technique in Smart Contract…
- Formal Specification and Verification of Smart Contracts
- Formal specification and verification
- Improved Ahead-of-time Compilation of Stack-based JVM…
- POSTER: Runtime Adaptations for Energy-Efficient VSLAM
- Towards a WebAssembly standalone runtime on GraalVM
- Analysis of WebAssembly as a Strategy to Improve…
- Conclusion: Debugging Blazor WebAssembly
- Introducing H, an Institution-Based Formal Specification…
- Runtime prediction of high-performance computing jobs…
- Synchronized Shared Memory and Procedural Abstraction…
- Targeting both Blazor Server and Blazor WebAssembly

- Valent-Blocks: Scalable High-Performance Compilation of…
- A Case Study in Formal Specification and Runtime…
- A Self-certifying Compilation Framework for WebAssembly
- Correction: A Case Study in Formal Specification and…
- Formal Specification and Verification of MQTT Protocol in…
- HERTI: A Reinforcement Learning-Augmented System for…
- On the Runtime and Energy Performance of WebAssembly: Is…
- Runtime Metric Analysis in NoSQL Database Performance…
- Twine: An Embedded Trusted Runtime for WebAssembly
- Vivienne: Relational Verification of Cryptographic…
- WAFL: Binary-Only WebAssembly Fuzzing with Fast Snapshots
- WebAssembly Module Internals: Sections and Memory Model
- Breaking the Vendor Lock
- Formal Verification of SUBLEQ Microcode implementing the…
- On JavaScript Ahead-of-Time Compilation Performance…
- Potential of WebAssembly for Embedded Systems
- WaTZ: A Trusted WebAssembly Runtime Environment with…
- WebAssembly versus JavaScript: Energy and Runtime…
- A Comprehensive Study of Bugs in Embedded WebAssembly…
- A Comprehensive Study of WebAssembly Runtime Bugs
- Automated WebAssembly Function Purpose Identification…
- CWASI: A WebAssembly Runtime Shim for Inter-function…
- Characterizing and Detecting WebAssembly Runtime Bugs
- Formal Specification and Verification of JDK’s Identity…
- Formal Verification Platform as a Service: WebAssembly…
- High-Performance Web Frontend Using WebAssembly
- Hybrid Execution: Combining Ahead-of-Time and…
- Of Ahead Time: Evaluating Disassembly of Android Apps…
- Optimizing Tensor Computations: From Applications to…
- Profile Guided Optimization Transfer-Learning for…
- Studying WebAssembly and comparison of its performance…
- Support for Just-in-Time Compilation of WebAssembly for…
- WaVe: a verifiably secure WebAssembly sandboxing runtime
- WasmSlim: Optimizing WebAssembly Binary Distribution via…
- When Function Inlining Meets WebAssembly…
- A Compilation of Experimental Binary Alloy Surface…
- A Comprehensive Trusted Runtime for WebAssembly With…
- A Semantics of Structures, Unions, and Underspecified…
- Accelerating Embedded WebAssembly Based on FPGA
- Ahead-of-time Compilation for Diverse Samplers of…
- Bringing Binary Exploitation at Port 80: Understanding C…
- Challenges of Multilingual Program Specification and…
- Characterizing Dynamic Memory Behavior in WebAssembly…
- Formal Specification and Verification of MQTT Protocol…
- SCALE-Ahead-Of-Time Compilation of CUDA for AMD GPUs
- TreeHouse: An MLIR-based Compilation Flow for Real-Time…
- WARDuino: An embedded WebAssembly virtual machine
- Wapplique: Testing WebAssembly Runtime via Execution…
- Wasm-Mutate: Fast and effective binary diversification…
- WebAssembly as a Fuzzing Compilation Target (Registered…
- A QUANTITATIVE ANALYSIS OF WEBASSEMBLY INTEGRATION…
- A Two-step Approach to Find Short Compilation…
- Adaptivity in AdaptiveCpp: Optimizing Performance by…
- Ahead of Time Generation for GPSA Protection in RISC-V…

- Application of WebAssembly for High-Performance…
- Archs: A WebAssembly Runtime for Cross-host Heterogeneous…
- Benchmarking WebAssembly for Embedded Systems
- Bringing Together Cross-ISA Checkpoint/Restoration and…
- CWAMR: REIMAGINING A CAPABILITYBASED WEBASSEMBLY RUNTIME…
- Calibro: Compilation-Assisted Linking-Time Binary Code…
- Detecting WebAssembly Runtime Bugs With Grammar-Guided…
- Distinguishability-Guided Test Program Generation for…
- Ductape: Optimizing Dynamically Typed Programs Using…
- Ensuring Reliability in Self-Adaptive Systems: A…
- ForMAt: Formal Verification of Scalable Multiply and…
- Formal Specification and Verification of Smart…
- FreeWavm: Enhanced WebAssembly Runtime Fuzzing Guided by…
- Furina: A Light-weight WebAssembly Runtime for ICS
- Hybrid WebAssembly-Container Orchestration in Embedded…
- HybridServe: Adaptive WebAssembly-Container Runtime…
- Investigating the Role of Formal Verification in Software…
- Lumos: Performance Characterization of WebAssembly as a…
- Performance and Usability Implications of Multiplatform…
- Research of WebAssembly usage for high-performance code…
- Runtime prediction model for high performance computing…
- Seamless Self-Healing in WebAssembly Container…
- Self-Hosted WebAssembly Runtime for Runtime-Neutral…
- Specification and Formal Verification of…
- Specification and Verification of a Formal Model for a…
- Typestates Specification and Verification in Frama-C
- WBSan: WebAssembly Bug Detection for Sanitization and…
- WebAssembly for Container Runtime: Are We There Yet?
- An Analysis of Modern Web Security Vulnerabilities Inside…
- QJWasm: A lightweight runtime system for efficient…
- Stack-based static WebAssembly binary slicing and…
- SurtGIS: A high-performance raster geospatial analysis…
- Unleashing Triton on CPUs: Compilation and Runtime…
- WARD: Efficient Memory Protection for WebAssembly on Tiny…
- Wasm-WCET: Worst-Case Execution-Time Analysis of…
- WasmWeaver: A Framework for Runtime-Aware WebAssembly…

The interesting absence is that none of it reports a bring-up ORDER. The specifications say what the instruction set is, the papers say how fast the result runs, and the sequence in which the implementers built it is not treated as a research object.

The middle of the compiler became shared infrastructure

Lowering is now typically staged through a series of intermediate representations instead of performed in one step, and the infrastructure for that is common property.

- LAYANAN CLOUD COMPUTING BERBASIS INFRASTRUCTURE AS A…
- LLVM parallel intermediate representation
- LLVM-based communication optimizations for PGAS programs
- PENERAPAN PEMROSESAN PARALEL UNTUK MENGUJI WAKTU…
- Modular SDN Compiler Design with Intermediate…
- The ARES High-Level Intermediate Representation
- Towards Automatic HBM Allocation Using LLVM: A Case Study…
- Multi-level physical hierarchy floorplanning using IC…
- A cost model for a graph-based…

- AIWC: OpenCL-Based Architecture-Independent Workload…
- CLACC: Translating OpenACC to OpenMP in Clang
- Compiler and language design for quantum computing…
- Function/Kernel Vectorization via Loop Vectorizer
- LLVM and the Automatic Vectorization of Loops Invoking…
- OP2-Clang: A Source-to-Source Translator Using Clang/LLVM…
- OpenMP GPU Offload in Flang and LLVM
- The CakeML Compiler Explorer
- User-Directed Loop-Transformations in Clang
- Comparing Mutation Testing at the Levels of Source Code…
- Design of Cyanobyte: An Intermediate Representation to…
- Flexible Runtime Reconfigurable Computing Overlay…
- Introducing multi-level parallelism, at coarse, fine and…
- LLHD: a multi-level intermediate representation for…
- Leveraging Compiler Intermediate Representation for…
- Replication Package for Paper: LLHD: A Multi-level…
- Robust Practical Binary Optimization at Run-time using…
- Static Neural Compiler Optimization via Deep…
- 3CPS: The Design of an Environment-Focussed Intermediate…
- A High Performance Sparse Tensor Algebra Compiler in MLIR
- Building SSA in a Compiler for PHP
- Extending LLVM IR for DPC++ Matrix Support: A Case Study…
- Facilitating CoDesign with Automatic Code Similarity…
- Flacc: Towards OpenACC support for Fortran in the LLVM…
- Functional Representations of SSA
- Improved Circuit Compilation for Hybrid MPC via Compiler…
- Integrating a functional pattern-based IR into MLIR
- Redundancy Elimination
- STERILIZER CHAMBER DESIGN WITH TELEGRAM-BASED INTERNET OF…
- Toward an Automated Hardware Pipelining LLVM Pass…
- An MLIR-based Compiler Flow for System-Level Design and…
- Caffeine: CoArray Fortran Framework of Efficient…
- Design of automatic aircraft parking system module…
- Logic Synthesis with Design Compiler
- Reinforcement Learning Strategies for Compiler…
- SPNC: An Open-Source MLIR-Based Compiler for Fast…
- ScaleHLS: A New Scalable High-Level Synthesis Framework…
- Towards Supporting Semiring in MLIR-Based COMET Compiler
- Unleashing the power of compiler intermediate…
- Use of Compiler Intermediate Representation for Reverse…
- ASPIRE: An Intermediate Representation for Abstract…
- INR-Arch: A Dataflow Architecture and Compiler for…
- Intermediate Representations
- LAGrad: Statically Optimized Differentiable Programming…
- MLIRSmith: Random Program Generation for Fuzzing MLIR…
- Experiences Building an MLIR-Based SYCL Compiler
- Fully integrating the Flang Fortran compiler with…
- Fuzzing MLIR Compiler Infrastructure via Operation…
- MLIR-Based Homomorphic Encryption Compiler for GPU
- MLIR-to-CGRA: A Versatile MLIR-Based Compiler Framework…
- Open-Source MLIR-Based Intermediate Representation for…
- Paddle-Mlir: A Compiler Based on MLIR for Accelerating…
- VCNN: A compiler of CNNs based on MLIR for multi-core…
- An Innovation Study on Luggage Wheel Design for Seamless…

- DESIL: Detecting Silent Bugs in MLIR Compiler…
- Enhancing Compiler Design for Machine Learning Workflows…
- MLIR: A Panacea for ML Compiler Challenges?
- P4IRS: An intermediate representation and compiler for…
- PolyMorphous: An MLIR-Based Polyhedral Compiler with Loop…
- The MLIR Transform Dialect
- Towards a Unified Multi-Target Mlir-Based Compiler: A…
- CombRewriter: Enabling Combinational Logic Simplification…
- Compiling Linear Algebra Workloads from C to Quantum…
- GeoIR-Compiler: A Geospatial Intermediate Representation…
- Quantum Circuit Synthesis from C via Multi-Level…
- Quantum Oracle Synthesis from HDL Designs via Multi Level…
- RV-IR: An MLIR-Based Architecture-Aware Intermediate…

Staging multiplies the number of lowering steps and does not touch the conjunction problem. A program still needs every step on its path. **A partially implemented lowering pipeline has the same product form as a partially implemented instruction set**, one level up, which suggests the measure generalises and does not suggest anyone has measured it.

Selection and synthesis moved on from the classics

The instruction-selection tradition surveyed earlier did not stop in 1999. Selection is surveyed as a field, superoptimisation became practical, peephole rules are generated and verified automatically, and program synthesis acquired solver-based and syntax-guided formulations.

- A Formal Approach based on Fuzzy Logic for the…
- An innovative approach for automatic generation…
- Counterexample-guided simulation framework for formal…
- Finding Good Compiler Optimization Sets - A Case-based…
- Keynote talk I: Syntax-guided synthesis
- PERANCANGAN PENGAMANAN SERVER SECARA OTOMATIS MENGGUNAKAN…
- The Correctness-Security Gap in Compiler Optimization
- Alive-FP: Automated Verification of Floating Point Based…
- Automatic Testbench Generation for Simulation-based…
- Automatic data layout generation and kernel mapping for…
- Counterexample-guided diagnosis
- A New Functional-Logic Compiler for Curry: Sprite
- Accurate quasi-P traveltimes in 3D transversely isotropic…
- Automatic Generation of the AADL ALISA Verification Plan…
- Automatic generation of simulation workflows for system…
- Counterexample-guided approach to finding numerical…
- HSS cluster-based direct solver for acoustic wave equation
- Look for the Proof to Find the Program…
- Automatic security verification of mobile app…
- Compiler optimization for scientific computation in C/C++
- Fast and flexible instruction selection with constraints
- Synthesizing an instruction selection rule library from…
- A Survey of Automatic Generation of Source Code Comments…
- Accurate quasi-SV traveltimes in 3D transversely…
- AliveInLean: A Verified LLVM Peephole Optimization…
- Alumni’s Perception on Program Specification of ELT…
- Automatic Verification of FSA Strategies via…
- Design and Development of Bridge AI bid Program based on…
- FrAngel: component-based synthesis with control structures

- Multi-target Compiler for the Deployment of Machine…
- An Empirical Study of Counterexample-Guided Fuzzing for…
- An approach to generate text-based IDEs for syntax…
- Artifact for article: Exact and Approximate Methods for…
- Automatic Generation of Multi-Objective Polyhedral…
- Automatic compiler optimization on embedded software…
- Boosting component-based synthesis with control structure…
- Dataflow-based pruning for speeding up superoptimization
- Exact and approximate methods for proving unrealizability…
- Grammar Filtering for Syntax-Guided Synthesis
- Performance Improvement of Kotlin Program in…
- Specification and automatic verification of trust-based…
- Automatic Verification of Data Summaries
- Can reactive synthesis and syntax-guided synthesis be…
- Compression Optimization For Automatic Verification of…
- Counterexample Guided Inductive Repair of Reactive…
- EMPIRICAL INVESTIGATION OF CLOUD, GRID AND VIRTUALIZATION…
- Instruction Code Selection
- Multi-modal program inference: a marriage of pre-trained…
- Open-Source Memory Compiler for Automatic RRAM Generation…
- Solver-based gradual type migration
- Special Issue on Syntax-Guided Synthesis Preface
- The Impact of Undefined Behavior on Compiler Optimization
- Thread-Aware Area-Efficient High-Level Synthesis Compiler…
- Towards a Domain-Extensible Compiler: Optimizing an Image…
- When Function Signature Recovery Meets Compiler…
- A Novel Counterexample-Guided Inductive Synthesis…
- Boosting Compiler Testing via Compiler Optimization…
- Can reactive synthesis and syntax-guided synthesis be…
- Cape: compiler-aided program transformation for HTM-based…
- Code Generation Techniques in Compiler Design: Conceptual…
- SRTuner: Effective Compiler Optimization Customization by…
- Specification-guided component-based synthesis from…
- Testing a PL/I Compiler Using Precomputation-based…
- Threaded Code Generation with a Meta-Tracing JIT Compiler
- Towards Automatic Property Generation for SoC Security…
- Accurate Compiler and Optimization Independent Function…
- An Automatic Generation and Verification Method of…
- Automatic Benchmark Generation for Object Constraint…
- Component-based specification, design and verification of…
- Counterexample Guided Knowledge Compilation for Boolean…
- Fast Compiler Optimization Flag Selection
- From SMT to ASP: Solver-Based Approaches to Solving…
- Introduction to Optimization
- Modular Component-Based Quantum Circuit Synthesis
- Neuroevolutionary Compiler Control for Code Optimization
- Optimization of production planning using integer linear…
- Optimization-Aware Compiler-Level Event Profiling
- Relational Solver for Java Generics Type System
- An Experimental Analysis of RL based Compiler…
- Association Rule Learning Based Approach to Automatic…
- DEVELOPING SITUATIONAL CONDITIONS AND PROGRAM CODES FOR…
- Passenger Queue Simulation Analysis and Optimization at…
- Reinforcement Learning and Data-Generation for…

- Revealing Compiler Heuristics Through Automated Discovery…
- Rule modeling for automatic verification of RDC-50…
- Automatic Generation of Assertions for Security…
- Automatic Generation of Assertions for Functional…
- Automatic Test Case Generation for Jasper App HDL…
- CTDip: a diversity-guided test program synthesis approach…
- CompilerDream: Learning a Compiler World Model for…
- Counterexample-Guided Inference of Modular Specifications
- LLM Compiler: Foundation Language Models for Compiler…
- MarQSim: Reconciling Determinism and Randomness in…
- Optimasi Pemanfaatan Air Menggunakan Program Solver di…
- Optimization-Directed Compiler Fuzzing for Continuous…
- Programming Assessment in E-Learning through Rule-Based…
- SAGE-HLS: Syntax-Aware AST-Guided LLM for High-Level…
- Accelerating Sparse Algebra with Program Synthesis
- Accelerating Syntax-Guided Program Synthesis by…
- An Ultralow-latency Constrained Quadratic Program (QP)…
- CHEHAB: Automatic Compiler Code Optimization for Fully…
- CPerfSmith: A Randomized C Program Generator for…
- Compiler-Runtime Co-operative Chain of Verification for…
- Distributionally robust optimization with…
- Hexcute: A Compiler Framework for Automating Layout…
- LLM-VeriOpt: Verification-Guided Reinforcement Learning…
- SecSwift, a Compiler-Based Framework for Software…
- Synthesizing Instruction Selection Back-Ends from ISA…
- Tensor Program Superoptimization through Cost-Guided…

This is the most direct engagement with the cost side of the ordering problem and it leaves the value side untouched. Synthesis makes an individual lowering cheaper to produce. In the notation used here it reduces κ and leaves $|B|$ exactly where it was, **so it moves the leverage ratio through the denominator and reorders nothing that the numerator determines.**

Verification became a family rather than a single system

The conservative stance this article depends on, in which anything not provably handled is refused, is no longer exotic. Verified compilation, translation validation, refinement checking for optimisations and certified static analysis are an active field with several mature systems.

- A verified type system for CakeML
- Abstract Interpretation with Infinitesimals
- An optimizing compiler for a purely functional…
- Certified Abstract Interpretation with Pretty-Big-Step…
- Correctness of Isabelle’s Cyclicity Checker
- Machine-Checked Verification of the Correctness and…
- Many-core compiler fuzzing
- Modular translation validation of a full-sized…
- Pilsner: a compositionally verified compiler for a…
- Session details: Session 4A: Compiler Correctness
- Towards a verified compiler prototype for the synchronous…
- Verification by abstract interpretation, soundness and…
- A Mechanical Soundness Proof for Subtyping Over Recursive…
- A new verified compiler backend for CakeML
- Abstract Interpretation of Supermodular Games
- Automated compiler optimization of multiple vector…

- Bounded Abstract Interpretation
- From Array Domains to Abstract Interpretation Under…
- LifeJacket: verifying precise floating-point…
- Mechanizing conventional SSA for a verified destruction…
- ModelPlex: verified runtime validation of verified…
- Static Analysis by Abstract Interpretation of the…
- Static analysis of Sequential Function Charts using…
- Termination-checking for LLVM peephole optimizations
- Verified construction of static single assignment form
- Verified lifting of stencil computations
- Verified peephole optimizations for CompCert
- A Verified CompCert Front-End for a Memory Model…
- A Verified Generational Garbage Collector for CakeML
- A formally verified compiler for Lustre
- A simple soundness proof for dependent object types
- Alive-Infer: data-driven precondition inference for…
- Automated Compiler Optimization of Multiple Vector…
- Automated Test Case Generation from OTS/CafeOBJ…
- Combining Forward and Backward Abstract Interpretation of…
- Handling Environments in a Nested Relational Algebra with…
- Lifting proof-relevant unification to higher dimensions
- Modeling Undefined Behaviour Semantics for Checking…
- Prototype implementation of the OpenGL ES 2.0 shading…
- Reduction of Workflow Nets for Generalised Soundness…
- Skeletal program enumeration for rigorous compiler testing
- Taming undefined behavior in LLVM
- Tutorial on Static Inference of Numeric Invariants by…
- Verified compilation of CakeML to multiple machine-code…
- A Proof Score Approach to Formal Verification of an…
- A Verified Compiler from Isabelle/HOL to CakeML
- A Verified Generational Garbage Collector for CakeML
- A fully verified container library
- A machine-checked correctness proof for Pastry
- CompCertS: A Memory-Aware Verified C Compiler Using a…
- Compiler-agnostic Translation Validation
- Compiler-aided Type Tracking for Correctness Checking of…
- Delta Debugging Type Errors with a Blackbox Compiler
- HHVM JIT: a profile-guided, region-based compiler for PHP…
- Mechanising a Type-Safe Model of Multithreaded Java with…
- Reconciling high-level optimizations and low-level code…
- Securing a compiler transformation
- Static Value Analysis of Python Programs by Abstract…
- Towards a verified Lustre compiler with modular reset
- VeriPhy: verified controller executables from verified…
- A verified protocol buffer compiler
- Testing and Verifying Parallel Programs Using Data…
- The verified CakeML compiler backend
- Verified compilation on a verified processor
- A Formal Proof of the Soundness of the Hybrid CPS Clock…
- Compiler and runtime support for continuation marks
- CoreJIT: a Replication Package for Article…
- Do you have space for dessert? a verified space cost…
- IMPLEMENTATION OF GENETIC ALGORITHM IN COLLEGE SCHEDULING…
- Implementation and Performance Evaluation of Omni Compiler

- Lost In Translation: Exposing Hidden Compiler…
- On a Machine-Checked Proof for Fraction Arithmetic over a…
- Proof pearl: Braun trees
- Testing static analyses for precision and soundness
- Towards compiler-aided correctness checking of adjoint…
- Translation from Visual to Layout-based Android Test…
- A minimalistic verified bootstrapped compiler (proof…
- Accessible Formal Methods for Verified Parser Development
- CoStar: a verified ALL(*) parser
- Compiler Module of Abstract Machine Code for Formal…
- Formally verified speculation and deoptimization in a JIT…
- Hyperchaining Optimizations for an LLVM-Based Binary…
- Lutsig: a verified Verilog compiler for verified circuit…
- Machine-checked ZKP for NP relations: Formally Verified…
- Parallelizing Compiler Translation Validation Using…
- Static Analysis of Endian Portability by Abstract…
- A machine-checked direct proof of the Steiner-lehmus…
- AV-AFL: A Vulnerability Detection Fuzzing Approach by…
- Certified abstract machines for skeletal semantics
- CirC: Compiler infrastructure for proof systems, software…
- DISTAL: the distributed tensor algebra compiler
- Gradual Soundness: Lessons from Static Python
- LocSeq: Automated Localization for Compiler Optimization…
- Machine-checked Verification of Cognitive Agents
- SMT-Based Translation Validation for Machine Learning…
- Simulink Model Static Analysis Results based on Abstract…
- The Trusted Computing Base of the CompCert Verified…
- Towards Verified Rounding Error Analysis for Stationary…
- Verifying optimizations of concurrent programs in the…
- A Control Flow based Static Analysis of GRAFCET using…
- A Hotspot-Driven Semi-automated Competitive Analysis…
- CompCert: A Journey through the Landscape of Mechanized…
- Completeness in Static Analysis by Abstract…
- Enhancing LLVM Optimizations for Linear Recurrence…
- Formalising Sharkovsky’s Theorem (Proof Pearl)
- Formally Verified Native Code Generation in an Effectful…
- Formally Verifying Optimizations with Block Simulations
- Improved Assistance for Interactive Proof (Keynote)
- Lazy Code Transformations in a Formally Verified Compiler
- PureCake: A Verified Compiler for a Lazy Functional…
- Syntax-Driven Translation
- Terms for Efficient Proof Checking and Parsing
- Translation Validation of Information Leakage of Compiler…
- Verified Propagation Redundancy and Compositional UNSAT…
- Verifying Term Graph Optimizations using Isabelle/HOL
- A Mechanised and Constructive Reverse Analysis of…
- A Safe Low-Level Language for Computer Algebra and Its…
- A Verified Compiler for a Functional Tensor Language
- Assuring Correctness, Testing, and Verification of…
- Automated Testing to Evaluate Employee Attendance System…
- Enhancing Translation Validation of Compiler…
- Extracting Invariants from Conditional Branches for…
- Leveraging LLVM OpenMP GPU Offload Optimizations for…
- Memory Simulations, Security and Optimization in a…

- PfComp: A Verified Compiler for Packet Filtering…
- Refinement of Parallel Algorithms Down to LLVM: Applied…
- Source code obfuscation with genetic algorithms using…
- Tinyrossa: A Compiler Framework for Vertical, Verified…
- Translation Validation for JIT Compiler in the V8…
- A Formal Verification Library Design for Behavioral…
- A Formally Verified Microcoded RISC-V Platform
- A Fully Automated Agent for End-to-End Code Translation…
- A Proof-Producing Compiler for Blockchain Applications
- A Universal Quantum Compiler GPT: Multi-Framework…
- CIRE: LLVM Analysis for Floating-Point Rounding Error…
- FormalGym: Deep Reinforcement Learning Agent Based Formal…
- Modeling and implementation of Common LISP functional…
- ORMorpher: An Interactive Framework for ORM Translation…
- Static analysis by abstract interpretation against data…
- Toward a Formally Verified Compiler for a Synchronous…
- A Calculus for Web Services Choreography: Formal…
- A Rose Tree Is Blooming (Proof Pearl)
- Brack: A Verified Compiler for Scheme via CakeML
- CODO: An Automated Compiler for Comprehensive Dataflow…
- CPP ’26 Artifact - Brack: A Verified Compiler for Scheme…
- Chariot: Compiler-Aware Heterogeneous Graph…
- Inductor-TV: Formal Methods for the Pytorch Compiler
- Making Time Observable: Compiler Correctness for…
- Test case sampling optimization for safety validation of…
- Testing, Credible Compilation, and Verification in the…
- Towards Verified Security: Formal Methods for LLM-Based…
- Verified VCG and Verified Compiler for Dafny

That matters to the argument in a specific way. The product form arises because a verifying compiler must refuse a program containing anything it cannot lower correctly. **The more normal that stance becomes, the more compilers exhibit the behaviour this article measures**, and the less the result depends on one project’s unusual strictness.

Testing the compiler became industrial, and it cannot answer this question

Compiler fuzzing is one of the genuine success stories of the last fifteen years. Random program generation, equivalence modulo inputs, differential testing and empirical studies of compiler bugs have found very large numbers of real defects.

- An Empirical Study of Bug Fixing Rate
- An Empirical Study on Real Bug Fixes
- Finding deep compiler bugs via guided stochastic program…
- Test program generation for mixed-signal integrated…
- An application of metamorphic testing for testing…
- An empirical study on how expert knowledge affects bug…
- Efficient Program Tracing and Monitoring Through Power…
- From Android Bug Reports to Android Bug Handling Process
- How Are Discussions Associated with Bug Reworking?
- Metamorphic testing for (graphics) compilers
- Random testing of C compilers based on test program…
- A XOR data compiler: Combined with physical unclonable…
- Are tweets useful in the bug fixing process? An empirical…
- Bug Propagation through Code Cloning: An Empirical Study
- Common Bug-Fix Patterns: A Large-Scale Observational Study

- Empirical Study on Software Bug Prediction
- Experience Report: Security Vulnerability Profiles of…
- Faster mutation analysis via equivalence modulo states
- Metamorphic Testing for Adobe Data Analytics Software
- PENERAPAN EIGENFACE UNTUK COMPUTER BASED TEST (CBT)…
- Understanding Key Features of High-Impact Bug Reports
- An Empirical Study of Multi-entity Changes in Real Bug…
- Empirical study on developer factors affecting tossing…
- Equivalence Class Testing
- Fault detection effectiveness of source test case…
- Finding missed compiler optimizations by differential…
- INSTRIM: Lightweight Instrumentation for Coverage-guided…
- Not all bug reopens are negative: A case study on eclipse…
- Preventing duplicate bug reports by continuously querying…
- Quality assurance of bioinformatics software
- Compiler bug isolation via effective witness test program…
- Coverage-Guided Learning-Assisted Grammar-Based Fuzzing
- Full-Speed Fuzzing: Reducing Fuzzing Overhead through…
- Haskell Compiler Testing Automation Based on…
- History-Guided Configuration Diversification for Compiler…
- Reinforcement-Learning-Based Test Program Generation for…
- Verifying Instruction Set Simulators using…
- An Empirical Study of Bug Bounty Programs
- Enhanced compiler bug isolation via memoized search
- On the relationship between bug reports and queries for…
- Demystifying the challenges and benefits of analyzing…
- EPF: An Evolutionary, Protocol-Aware, and Coverage-Guided…
- How to Better Distinguish Security Bug Reports (Using…
- Metamorphic Testing on the Continuum of Verification and…
- REST API Fuzzing by Coverage Level Guided Blackbox Testing
- ReFuzz: A Remedy for Saturation in Coverage-Guided Fuzzing
- Same Coverage, Less Bloat: Accelerating Binary-only…
- Similarity-Aware Architecture/Compiler Co-Designed…
- The forgotten role of search queries in IR-based bug…
- Type-Centric Kotlin Compiler Fuzzing: Preserving Test…
- Why Some Bug-bounty Vulnerability Reports are Invalid?
- CAGFuzz: Coverage-Guided Adversarial Generative Fuzzing…
- An Empirical Study of Aging Related Bug Prediction Using…
- An Empirical Study of the Bug Link Rate
- An empirical study of the effectiveness of IR-based bug…
- Application of Property-based Testing Tools for…
- Backend Bug Finder — a platform for effective compiler…
- CsmithEdge: more effective compiler testing by handling…
- Efficient Cross-Level Processor Verification using…
- Enriching Compiler Testing with Real Program from Bug…
- Fine-Grained Coverage-Based Fuzzing
- FitM: Binary-Only Coverage-Guided Fuzzing for Stateful…
- High-coverage metamorphic testing of concurrency support…
- Investigating Coverage Guided Fuzzing with Mutation…
- Metamorphic testing in bioinformatics software
- POWER: Program Option-Aware Fuzzer for High Bug Detection…
- Remgen: Remanufacturing a Random Program Generator for…
- SpinalFuzz: Coverage-Guided Fuzzing for SpinalHDL Designs
- Testing ocean software with metamorphic testing

- Unified HW/SW Coverage: A Novel Metric to Boost…
- Upstream bug management in Linux distributions
- Bug characterization in machine learning-based systems
- Compiler Test-Program Generation via Memoized…
- Coverage-Guided Fuzzing for Plan-Based Robotics
- Finding Unstable Code via Compiler-Driven Differential…
- Fuzzing Deep Learning Compilers with HirGen
- GrayC: Greybox Fuzzing of Compilers and Analysers for C
- Improve Model Testing by Integrating Bounded Model…
- Inferring test models from user bug reports using…
- JITfuzz: Coverage-guided Fuzzing for JVM Just-in-Time…
- Poster: BugOss: A Regression Bug Benchmark for Empirical…
- Program Reconditioning: Avoiding Undefined Behaviour When…
- Rainfuzz: Reinforcement-Learning Driven Heat-Maps for…
- RustSmith: Random Differential Compiler Testing for Rust
- Silent Compiler Bug De-duplication via Three-Dimensional…
- A study of common bug fix patterns in Rust
- An empirical study on the potential of word embedding…
- Automated SC-MCC test case generation using…
- Automatic program bug fixing by focusing on finding the…
- Boosting Compiler Testing by Injecting Real-World Code
- Bug numbers matter: An empirical study of effort-aware…
- CatchFuzz: Reliable active anti-fuzzing techniques…
- Compatible Branch Coverage Driven Symbolic Execution for…
- Compiler Bug Isolation via Enhanced Test Program Mutation
- Detecting Optimizing Compiler Bugs via History-Driven…
- Diffy: Data-Driven Bug Finding for Configurations
- Discretized optimization algorithms for finding the…
- FOX: Coverage-guided Fuzzing as Online Stochastic Control
- Fuzzing Command-line Interface by Edge Coverage Guided…
- Fuzzing JavaScript Interpreters with Coverage-Guided…
- Fuzzing guided by context-sensitive branch coverage
- History-driven Compiler Fuzzing via Assembling and…
- Industrial adoption of machine learning techniques for…
- Inside Bug Report Templates: An Empirical Study on Bug…
- Optimising Bcrypt Parameters: Finding the Optimal Number…
- RLGFuzz: Reinforcement Learning Guided Fuzzing with…
- Rust-twins: Automatic Rust Compiler Testing through…
- Rustlantis: Randomized Differential Testing of the Rust…
- Semantic-Type-Guided Bug Finding
- Testing Error Handling Code With Software Fault Injection…
- Testing the Unknown: A Framework for OpenMP Testing via…
- When Compiler Optimizations Meet Symbolic Execution: An…
- Automated Test Generation from Program Documentation…
- BoostPolyGlot: A Structured IR Generation-Based Fuzz…
- CBGF: Callback Coverage Guided Fuzzing
- Can We Enhance Bug Report Quality Using LLMs?: An…
- Compiler Optimization Testing Based on…
- DeepUIFuzz: A Guided Fuzzing Strategy for Testing UI…
- Differential Fuzzing Go Compilers using LLMs: A…
- Finding Compiler Bugs through Cross-Language Code…
- From Bug Reports to Workarounds: The Real-World Impact of…
- Fuzzing JavaScript JIT compilers with a high-quality…
- GrammLLM: Grammar-Guided LLM Test Generation for Compiler…

- HFuzz: Havoc Mode Guided Fuzzing
- Hybrid Equivalence/Non-Equivalence Testing
- Interleaving Large Language Models for Compiler Testing
- IntraFuzz: Coverage-Guided Intra-Enclave Fuzzing for…
- MALintent: Coverage Guided Intent Fuzzing Framework for…
- Metamorphic Relation Patterns for Metamorphic Testing…
- Research on coverage-guided fuzzing technique based on…
- SSFuzz: Synthesizing and scheduling bug-triggering code…
- Scalable SMT Sampling for Floating-Point Formulas via…
- Shepherd: High-Precision Coverage Inference for…
- Solsmith: Solidity Random Program Generator for Compiler…
- Symbolic MRD: Dynamic Memory, Undefined Behaviour, and…
- Synchronized Behavior Checking: A Method for Finding…
- Testing Autonomous Driving Systems Through Blind-Spot…
- An Empirical Comparison of Human and LLM-Assisted Bug…
- An exploratory study of bug-introducing changes…
- Automated Inference of Expressive Metamorphic Relations…
- Batch Me If You Can: Coverage-Guided RPKI Fuzzing at Scale
- Coverage-Guided Multi-Agent Harness Generation for Java…
- Detecting Compiler-Introduced Security Bugs via IR…
- Grammar-Aware Coverage-Guided Fuzzing with Grammarinator…
- How Effective Is Coverage-Guided Fuzzing to Test Deep…
- QEMI: A Quantum Software Stacks Testing Framework via…
- TYPEFUZZ: Type Coverage Directed JavaScript Engine…
- TenSure: Fuzzing Sparse Tensor Compilers (Registered…
- Towards Path-Aware Coverage-Guided Fuzzing
- Understanding Bug-Reproducing Tests: A First Empirical…
- Understanding and Finding JIT Compiler Performance Bugs
- WuppieFuzz: Coverage-Guided, Stateful REST API Fuzzing

And the entire apparatus is blind to the question asked here, for a precise reason. A fuzzer generates programs and checks whether the compiler handles them correctly. **It therefore tests what is implemented.** An unimplemented instruction is not a bug the fuzzer can find, because the compiler correctly refuses it. **The most sophisticated compiler-testing machinery ever built cannot tell an engineer what to implement next**, and nothing in that literature claims otherwise.

Timing analysis, which is the reason to go native at all

The motivation for native code generation in this project is worst-case resource behaviour, and that subject remains active, with multicore timing, cache predictability and time-predictable architecture carrying the effort.

- Calculation of worst-case execution time for multicore…
- Efficient Worst-Case Execution Time Analysis of Dynamic…
- FIFO Cache Analysis for WCET Estimation
- MRU Cache Analysis for WCET Estimation
- Operator-data type pair based execution environments…
- Time-Accurate ASM as a Refinement Scheme for Worst-Case…
- Worst-Case Execution Time Analysis for Many-Core…
- Architecture of a tool for automated testing the…
- Class-based query-optimization for minimizing worst-case…
- Combining loop unrolling strategies and code predication…
- Integration of Static Worst-Case Execution Time and Stack…
- On the Criticality of Probabilistic Worst-Case Execution…

- Predicting Worst-Case Execution Time Trends in Long-Lived…
- Replacing conjectures by positive knowledge: Inferring…
- On the use of static branch prediction to reduce the…
- PSO based optimization of worst-case execution time for…
- Static Worst-Case Execution Time Optimization using DPSO…
- Worst-Case Execution Time Testing via Evolutionary…
- Correction to: A compiler framework for the reduction of…
- Software UART: A Use Case for VSCPU Worst-Case Execution…
- Symbolic Execution and Recent Applications to Worst-Case…
- Reliability Test based on a Binomial Experiment for…
- Survey on Estimation and Optimization of Worst-case…
- Worst-case Execution Time Estimation of Legacy Vehicular…
- Deep Neural Network Approach to Estimate Early Worst-Case…
- Experiences from Adjusting Industrial Software for…
- Practical Examples of Timing Problems
- Timing Analysis Techniques
- Use of Measurements in Worst-Case Execution Time…
- CUDA Acceleration of Worst-Case Execution Time Analysis…
- Worst-Case Execution Time Estimation for Numerical…
- Analysis of benchmark program results of worst case…
- Design of DMS-RRIP replacement algorithm for L1-cache of…
- Worst Case Execution Time and Power Estimation of…
- Analyzing Data Flow and Control Flow of Multicore…
- Exact Worst-Case Execution-Time Analysis for Implicit…
- Utilizing Machine Learning Techniques for Worst-Case…
- Worst-Case Execution Time Analysis of Real-Time Robotic…
- Determining Worst-Case Execution Time Bounds for…
- Hardware/Software Co-Analysis for Worst Case Execution…
- Static Timing and Power Analysis of a RISC-V Pipelined…
- Worst-Case Execution Time Analysis of a Real-Time System…
- A Time-Predictable Multicore RISC-V Architecture for…

The relevant point is that the analysis is only as complete as the lowering. A bound that cannot be computed for a program the compiler refuses is not a bound at all, so the coverage question sits upstream of the timing question rather than beside it.

Measuring a corpus became routine, which strengthens the recommendation

This is the largest change and it cuts in the article's favour. Mining software repositories is a mature field with its own venue, and large-scale study of what code actually contains is ordinary work now.

- A Large Scale Study of License Usage on GitHub
- A large-scale study on the usage of Java's concurrent…
- Characteristics of Useful Code Reviews: An Empirical…
- Co-evolution of Infrastructure and Source Code - An…
- Code Ownership and Software Quality: A Replication Study
- Code coverage and test suite effectiveness: Empirical…
- Fuse: A Reproducible, Extendable, Internet-Scale Corpus…
- Graph-Based Statistical Language Model for Code
- License Usage and Changes: A Large-Scale Study of Java…
- Novice comprehension of Object-Oriented OO programs: An…
- Partitioning Composite Code Changes to Facilitate Code…
- Quality Questions Need Quality Code: Classifying Code…
- Query by example in large-scale code repositories

- A Taxonomy of Spanish Nouns, a Statistical Algorithm to…
- A large-scale empirical study on self-admitted technical…
- A large-scale study on repetitiveness, containment, and…
- License usage and changes: a large-scale study on gitHub
- Mining performance regression inducing code changes in…
- Mining the modern code review repositories
- A Large-Scale Study of the Impact of Feature Selection…
- A large-scale study of programming languages and code…
- An Empirical Study on Real Bugs for Machine Learning…
- Bug Characteristics in Blockchain Systems: A Large-Scale…
- Classifying Code Comments in Java Open-Source Software…
- Creating and Analyzing Source Code Repository Models - A…
- PRPA REPERCUSSIONS and IMPLICATIONS FOR REAL WORLD STUDY…
- Statistical Unigram Analysis for Source Code Repository
- Who Will Leave the Company?: A Large-Scale Industry Study…
- A benchmark study on the effectiveness of search-based…
- Analyzing False Positive Source Code Vulnerabilities…
- Large-scale analysis of the co-commit patterns of the…
- SLAMPA: Recommending Code Snippets with Statistical…
- Syntax, predicates, idioms — what really affects code…
- The Expansion of Source Code Abbreviations Using a…
- Why are Android apps removed from Google Play?
- A Convolutional Neural Network for Language-Agnostic…
- A Large-Scale Study About Quality and Reproducibility of…
- Branch Use in Practice: A Large-Scale Empirical Study of…
- Combining Program Analysis and Statistical Language Model…
- GreenSource: A Large-Scale Collection of Android Code…
- Impact of Stack Overflow Code Snippets on Software…
- git2net - Mining Time-Stamped Co-Editing Networks from…
- A Large-Scale Comparative Evaluation of IR-Based Tools…
- Big code != big vocabulary
- Building Intelligent Integrated Development Environment…
- Dockerfile Changes in Practice: A Large-Scale Empirical…
- Empirical Study about Class Change Proneness Prediction…
- How does combinatorial testing perform in the real world…
- How to Evaluate the Productivity of Software Ecosystem: A…
- Large-Scale Manual Validation of Bugfixing Changes
- Open Source Software (OSS) for Big Data
- Querying Big Source Code
- RTFM: Towards Understanding Source Code using Natural…
- The Software Heritage Graph Dataset
- Using Large-Scale Anomaly Detection on Code to Improve…
- A Large Scale Study of Long-Time Contributor Prediction…
- A Large-Scale Empirical Study of COVID-19 Themed GitHub…
- A large-scale study on human-cloned changes for automated…
- An Empirical Study of Real-World WebAssembly Binaries
- An Empirical Study on the Impact of Aspect-oriented…
- AndroidCompass: A Dataset of Android Compatibility Checks…
- CCEyes: An Effective Tool for Code Clone Detection on…
- Does Code Review Promote Conformance? A Study of…
- Machine Learning Approaches for Authorship Attribution…
- Predicting Design Impactful Changes in Modern Code…
- QScored: A Large Dataset of Code Smells and Quality…
- A Mechanism for Automatically Extracting Reusable and…

- A large-scale comparison of Python code in Jupyter…
- A large-scale dataset of (open source) license text…
- Geographic diversity in public code contributions
- How to improve deep learning for software analytics
- Low Level Source Code Vulnerability Detection Using…
- Mining the usage of reactive programming APIs
- A Large Scale Analysis of Semantic Versioning in NPM
- APR4Vul: an empirical study of automatic program repair…
- DeepSurveySim: Simulation Software and Benchmark…
- Enriching Source Code with Contextual Data for Code…
- Improving Code Completion by Solving Data Inconsistencies…
- Language usage analysis for EMF metamodels on GitHub
- Large Language Models for Code Obfuscation Evaluation of…
- Naturalness in Source Code Summarization. How Significant…
- Source Code Features and their Dependencies: An…
- Source Code Implied Language Structure Abstraction…
- Source Code Plagiarism Detection with Pre-Trained Model…
- A Large-Scale Empirical Study of Open Source License…
- A Methodology for Analysing Code Anomalies in Open-Source…
- An Empirical Analysis of Issue Templates Usage in…
- Can Large Language Model Detect Plagiarism in Source Code?
- EnStack: An Ensemble Stacking Framework of Large Language…
- How accessibility affects other quality attributes of…
- Large-Scale Analysis of GitHub and CVEs to Determine…
- Multi-faceted Code Smell Detection at Scale using…
- Naturalness of Attention: Revisiting Attention in Code…
- Propagating Large Language Models Programming Feedback
- CASTL: A Composable Source Code Query Language for…
- Can LLMs Generate Higher Quality Code Than Humans? An…
- CoUpJava: A Dataset of Code Upgrade Histories in…
- Combining Large Language Models with Static Analyzers for…
- Come for syntax, stay for speed, write secure code: an…
- Examining the impact of bias mitigation algorithms on the…
- GHALogs: Large-Scale Dataset of GitHub Actions Runs
- Generating Software Architecture Description from Source…
- Harnessing Large Language Models for Curated Code Reviews
- HyperAST: Incrementally Mining Large Source Code…
- Java Source Code Vulnerability Detection Using Large…
- Large Language Models for Computer Programming Education…
- OSS License Identification at Scale: A Comprehensive…
- Programming Large Language Models
- RETRACTION: Study on Large-Scale Promotion of…
- Towards understanding code review practices for…
- Understanding Feature Request Practice on GitHub via a…
- Wild SBOMs: a Large-scale Dataset of Software Bills of…
- rFocal: Run ‘FOCAL’ Language Source Code
- A Large-Scale Dataset of MCP Implementations on GitHub
- A Large-Scale Investigation Into the Loss of Pull Request…
- AI builds, We Analyze: An Empirical Study of AI-Generated…
- An Empirical Study of SBOM Usage Through GitHub Actions
- Automating Software Documentation with n8n and Large…
- Beyond Single Code Changes: An Empirical Study of…
- Empirical Study on Real-time System Modeling and Code…
- Floating-Point Usage on GitHub: A Large-Scale Study of…

- GeoAutoModuler: a knowledge–enhanced large language model…
- GitEvo: Code Evolution Analysis for Git Repositories
- HackRep: A Large-Scale Dataset of GitHub Hackathon…
- How AI Coding Agents Modify Code: A Large-Scale Study of…
- How challenging it is to identify real code authors: an…
- Integrating Large Language Models in Software Engineering…
- Modeling Sampling Workflows for Code Repositories
- OSSGameBench: A Large-Scale Dataset of Development…
- Running Large Language Models at Scale for Mining…
- Understanding Binary Code Similarity for Real-World…

The instrument this article describes took twenty minutes to build because that is now a twenty-minute job. When the instruction-selection classics were written it was not. **The recommendation to measure before ordering was expensive advice in 1978 and is nearly free in 2026**, which is the strongest argument for adopting it and is an argument the article did not previously make.

The surrogate-measure warning has been rediscovered everywhere

The specific failure this article reports, where an easily computed measure stands in for the property actually wanted, is not confined to compilers. Coverage as a proxy for suite quality, mutation scores, flaky tests, benchmarking methodology and replication have all produced the same warning.

- An Empirical Study on Effects of Code Visibility on Code…
- An Initiative to Improve Reproducibility and Empirical…
- An empirical study of bugs in test code
- Beyond code coverage and#x2014; An approach for test…
- Do Automatically Generated Unit Tests Find Real Faults?…
- Exploring Test Suite Diversification and Code Coverage in…
- Investigations about replication of empirical studies in…
- Mutation testing in practice using Ruby
- On the Benefits and Barriers When Adopting Software…
- Replication of Empirical Studies in Software Engineering…
- Using Artificial Bee Colony for Code Coverage Based Test…
- Using text clustering to predict defect resolution time…
- A detailed investigation of the effectiveness of whole…
- A large-scale study of call graph-based impact prediction…
- Assessing the Test Suite of a Large System Based on Code…
- Characterizing logging practices in Java-based open…
- Empirical study of correlation between mutation score and…
- Global vs. local models for cross-project defect…
- Relating Code Coverage, Mutation Score and Test Suite…
- System-Level Test Case Prioritization Using Machine…
- Techniques of Test Case Prioritization
- UML Associations - Reducing the Gap in Test Coverage…
- Using docker containers to improve reproducibility in…
- An Empirical Study of Activity, Popularity, Size…
- An Empirical Study of the Personnel Overhead of…
- An Empirical Study on the Cross-Project Predictability of…
- An empirical study of regression test suite reduction…
- An empirical study on the application of mutation testing…
- Analytics-Driven Load Testing: An Industrial Experience…
- Assessing and Improving the Mutation Testing Practice of…
- Could We Predict the Result of a Continuous Integration…

- Creating and Running Tests with Xamarin Test Cloud
- Extended firm mutation testing: A cost reduction…
- How effective are mutation testing tools? An empirical…
- Impact of Static and Dynamic Coverage on Test-Case…
- Integrating Tests into Your Builds
- Reinforcement learning for automatic test case…
- 12.4 - Machine Learning-Driven Test Case Prioritization…
- An Empirical Study of Flaky Tests in Android Apps
- An Industrial Application of Mutation Testing: Lessons…
- An empirical study of inadequate and adequate test suite…
- Are mutation scores correlated with real fault detection?
- Continuous Integration and Visual GUI Testing: Benefits…
- Generating Effective Test Suite for Multiparameter…
- Prediction of relatedness in stack overflow: deep…
- Program comprehension of domain-specific and…
- Redefining prioritization
- Reproducibility and credibility in empirical software…
- State of mutation testing at google
- Test case prioritization and selection technique in…
- Test prioritization in continuous integration environments
- The role and value of replication in empirical software…
- A Time Window based Reinforcement Learning Reward for…
- An Empirical Study of the Relationship between Continuous…
- An empirical study of the long duration of continuous…
- Combining Code and Requirements Coverage with Execution…
- Comprehending Test Code: An Empirical Study
- Meta-analysis for families of experiments in software…
- RETRACTED ARTICLE: The smell of fear: on the relation…
- Temporal Discounting in Software Engineering: A…
- Test Case Design and Test Case Prioritization using…
- TestCov: Robust Test-Suite Execution and Coverage…
- A study on the lifecycle of flaky tests
- Code review effectiveness: an empirical study on selected…
- Data Science and Empirical Software Engineering
- De-Flake Your Tests : Automatically Locating Root Causes…
- Effective Test Suite Optimization for Improving the…
- Empirical Software Engineering Experimentation with Human…
- Empirical Study of Restarted and Flaky Builds on Travis CI
- Empirical Study of Software Test Suite Evolution
- Learning-based prioritization of test cases in continuous…
- Multi-Armed Bandit Test Case Prioritization in Continuous…
- Resources for Reproducibility of Experiments in Empirical…
- Retraction Note: Retraction note to: The smell of fear…
- TABU Search Prioritized Ant Colony Metaheuristic…
- Test Case Prioritization in Continuous Integration…
- The Application Of Machine Learning In Test Case…
- The Evolution of Empirical Methods in Software Engineering
- A Review on Continuous Integration and Continuous…
- An Empirical Analysis of UI-Based Flaky Tests
- An Empirical Study of Flaky Tests in Python
- Assessment of off-the-shelf SE-specific sentiment…
- DeepCrime: mutation testing of deep learning systems…
- DeepOrder: Deep Learning for Test Case Prioritization in…
- Empirical evaluation of tools for hairy requirements…

- Extreme mutation testing in practice: An industrial case…
- Genetic programming for feature model synthesis: a…
- Industrial Scale Passive Testing with T-EARS
- Lessons Learnt on Reproducibility in Machine Learning…
- Locating faults with program slicing: an empirical…
- Reflections on the Empirical Software Engineering journal
- Release synchronization in software ecosystems
- Statement frequency coverage: A code coverage criterion…
- Test case selection and prioritization using machine…
- Understanding and improving the quality and…
- Weighted Reward for Reinforcement Learning based Test…
- When is Continuous Integration Useful? Empirical Study on…
- A Comprehensive Study on Code Coverage Analysis for…
- A Multi-Armed Bandit Approach for Test Case…
- A Qualitative Study on the Sources, Impacts, and…
- An Empirical Study of Flaky Tests in JavaScript
- An Improvement to Test Case Prioritization Techniques…
- Breaking bad? Semantic versioning and impact of breaking…
- Checked coverage for test suite reduction
- Cost-effective learning-based strategies for test case…
- Evaluating classifiers in SE research: the ECSER pipeline…
- Excluding code from test coverage: practices…
- Machine Learning Regression Techniques for Test Case…
- Mutation analysis and its industrial applications
- Patterns of Code-to-Test Co-evolution for Automated Test…
- Preempting flaky tests via non-idempotent-outcome tests
- Real world projects, real faults: evaluating spectrum…
- Reinforcement Learning Reward Function for Test Case…
- Revisiting the building of past snapshots — a replication…
- Static detection of equivalent mutants in real-time…
- The reproducibility of programming-related issues in…
- What do developer-repaired Flaky tests tell us about the…
- An Empirical Study of Greedy Test Suite Minimization…
- An Empirical Study of Regression Testing for Android Apps…
- An Empirical Study on the Correlation between Neuron…
- An Evaluation of Ranking-to-Learn Approaches for Test…
- An Improved Method for Test Case Prioritization in…
- Automated NFR testing in continuous integration…
- Comparative study of machine learning test case…
- DeepCrime: from Real Faults to Mutation Testing Tool for…
- Evaluation of Coverage Metrics for Assessing Test Suite…
- Guest editorial: special issue on empirical software…
- How Closely are Common Mutation Operators Coupled to Real…
- How Do Deep Learning Faults Affect AI-Enabled…
- Model vs system level testing of autonomous driving…
- Mutation Testing in Continuous Integration: An…
- Mutation Testing of Deep Reinforcement Learning Based on…
- Neural Network-Based Test Case Prioritization in…
- On factors that impact the relationship between code…
- Operationalizing validity of empirical software…
- Optimizing test case prioritization using machine…
- Parallel mutation testing for large scale systems
- Revisiting Machine Learning based Test Case…
- Revisiting the reproducibility of empirical software…

- Scalable and Accurate Test Case Prioritization in…
- Semantic Coverage: Measuring Test Suite Effectiveness
- Semantic-aware two-phase test case prioritization for…
- Syntactic Versus Semantic Similarity of Artificial and…
- Test Case Prioritization using Transfer Learning in…
- The Vocabulary of Flaky Tests in the Context of SAP HANA
- A Preliminary Framework for Optimising Test Case…
- An Empirical Study on Code Coverage of Performance Testing
- An Exploratory Study on Soft Skills present in Software…
- An extensive replication study of the ABLoTS approach for…
- Cost of Flaky Tests in Continuous Integration: An…
- Deployment and Integration of Machine Learning Methods…
- Examining ownership models in software teams
- Explainable Test Case Prioritization in Continuous…
- Exploring the Effectiveness of LLM based Test-driven…
- FlakeSync: Automatically Repairing Async Flaky Tests
- Leveraging Rough Sets for Enhanced Test Case…
- Machine Learning for Test Case Prioritization in…
- Machine Learning-based Test Case Prioritization using…
- Nonlinear Reinforcement Learning-Based Dynamic Test Case…
- On the use of contextual information for machine learning…
- Reinforcement learning for online testing of autonomous…
- Test Case Prioritization for Regression Testing Using…
- Test code refactoring unveiled: where and how does it…
- Towards enhancing the reproducibility of deep learning…
- Using rapid reviews to support software engineering…
- μPRL: A Mutation Testing Pipeline for Deep…
- A Defect Taxonomy for Infrastructure as Code: A…
- A Preliminary Study of Fixed Flaky Tests in Rust Projects…
- AI-Driven Test Case Generation for Continuous Integration…
- Achieving High-Integrity Software Quality and Security…
- An Effective GRU-Based Deep Learning Method for Test Case…
- An Empirical Study of Web Flaky Tests: Understanding and…
- An Environment Adaptation Agent of Reinforcement Learning…
- An Explainable Deep Learning Model in Improving Test Case…
- Attention Transfer Reinforcement Learning for Test Case…
- Dynamic Test Case Prioritization and Selection for…
- Evaluating Machine Learning-Based Test Case…
- How Does Test Code Differ from Production Code in Terms…
- Identifying and Mitigating Flaky Tests in JavaScript
- Intelligent Test Case Prioritization: A Review of Machine…
- Is code coverage of performance tests related to source…
- Mutation Operators for Mutation Testing of Angular Web…
- Mutation Testing for Industrial Robotic Systems
- Mutation Testing of Programs for Industrial Robots
- Opportunities and security risks of technical leverage: A…
- Optimizing Test Case Prioritization With Meta Deep…
- Ranking Relevant Tests for Order-Dependent Flaky Tests
- Rechecking Recheck Requests in Continuous Integration: An…
- Reimagining Studies' Replication: A Validity-Driven…
- Reproducibility Practices of Software Engineering…
- Reviewing Reproducibility in Software Engineering Research
- muttest: Mutation Testing
- A large-scale empirical study of configurations, errors…

- Adaptive and Explainable Test Case Prioritization in…
- Beyond Coverage: Automatic Test Suite Augmentation for…
- Can We Classify Flaky Tests Using Only Test Code? an…
- Fuzzy-Graph Contrastive Test Case Prioritization…
- Industrial Application of Deep Learning based Fault…
- Interruptibility of software developers and its…
- Is this build failure related to my patch? An empirical…
- Mitigating omitted variable bias in empirical software…
- Neural-MCTS Test Prioritization for Smart Contract…

Instruction-level coverage is a metric that became a target. It is the measure a bring-up effort naturally reports because it moves smoothly and always improves, and it is nearly uninformative about whether anything works.

Attribution and submodularity moved into machine learning

The Shapley machinery invoked for the attribution problem is no longer a game-theory curiosity. It is production tooling for model explanation and data valuation, and submodular maximisation acquired streaming and distributed algorithms.

- Randomized Composable Core-sets for Distributed…
- Risk Attribution Using the Shapley Value: Methodology and…
- Submodular maximization meets streaming: matchings…
- On distributed submodular maximization with limited…
- The Shapley Value as a Sustainable Cooperative Solution…
- A distributed algorithm for partitioned robust submodular…
- Bicriteria Distributed Submodular Maximization in a Few…
- Shapley Value of a Cooperative Game with Fuzzy Set of…
- Distributed Submodular Maximization on Partition Matroids…
- Distributed matroid-constrained submodular maximization…
- Greedily Excluding Algorithm for Submodular Maximization
- Streaming Non-Monotone Submodular Maximization…
- An Approximation Algorithm for Distributed Resilient…
- Collaboration Formation and Profit Sharing Between…
- Distributed Submodular Maximization with Bounded…
- Hodge decomposition and the Shapley value of a…
- Shapley Value Approximation with Divisive Clustering
- Streaming Submodular Maximization Under Noises
- The Shapley Value, a Crown Jewel of Cooperative Game…
- The Shapley and Position Values to Design Coalitional…
- A cooperative game theory application in chicks brood…
- An Improved Shapley Value Benefit Distribution Mechanism…
- Distributed Attack-Robust Submodular Maximization for…
- Distributed Maximization of Submodular and Approximately…
- Distributed Submodular Maximization with Parallel…
- Sequence submodular maximization meets streaming
- A Multi-pass Streaming Algorithm for Regularized…
- A Streaming Model for Monotone Lattice Submodular…
- Fast derivation of Shapley based feature importances…
- Fixed-size video summarization over streaming data via…
- Improved Prediction of Total Energy Consumption and…
- Multi-Pass Streaming Algorithms for Monotone Submodular…
- One-pass streaming algorithm for monotone lattice…
- Shapley-Value Data Valuation for Semi-supervised Learning
- Streaming algorithms for robust submodular maximization

- A Programming Approach for Worst-case Studies in…
- An Optimal Streaming Algorithm for Submodular…
- An optimal streaming algorithm for non-submodular…
- Application of an Improved Shapley Value Method in…
- CS-Shapley: Class-Wise Shapley Values for Data Valuation…
- Data Shapley Valuation for Efficient Batch Active Learning
- Distributed submodular maximization: trading performance…
- One-pass streaming algorithm for DR-submodular…
- Poster Abstract: Towards Shapley Value based Security…
- Regularized two-stage submodular maximization under…
- Resource-Aware Distributed Submodular Maximization: A…
- Shapley Value is an Equitable Metric for Data Valuation
- Streaming submodular maximization under d-knapsack…
- A Model-Agnostic Feature Attribution Approach to…
- Algorithms to estimate Shapley value feature attributions
- Cooperative game amongst prefabricated building chain…
- DASH: A Distributed and Parallelizable Algorithm for…
- Data valuation using Shapley value in machine learning
- Distributed strategy selection: A submodular set function…
- Erratum: Weighted shapley value: a cooperative game…
- Fair and Efficient Alternatives to Shapley-based…
- Integrating Staleness and Shapley Value Consistency for…
- Machine Learning for Data Center Optimizations: Feature…
- Streaming Algorithms for Constrained Submodular…
- Streaming Algorithms for Non-Submodular Maximization on…
- Streaming adaptive submodular maximization
- Weighted shapley value: A cooperative game theory for…
- Zoish: A Novel Feature Selection Approach Leveraging…
- A Semi-streaming Algorithm for Monotone Regularized…
- An Improved Space Semi-Streaming Algorithm for Submodular…
- An innovative machine learning workflow to research…
- Applications and Computation of the Shapley Value in…
- Approximation Algorithm for Connected Submodular Function…
- Blending Shapley values for feature ranking in machine…
- Bridging efficacy and efficiency: Innovations in Shapley…
- Codes for machine learning and Shapley value analysis
- DU-Shapley: A Shapley Value Proxy for Efficient Dataset…
- Deterministic Algorithm and Faster Algorithm for…
- Deterministic streaming algorithms for non-monotone…
- Efficient Shapley Value Driven Federated Learning System…
- Efficient Shapley performance attribution for…
- Explaining 3D Object Detection Through Shapley…
- Greedy algorithm for maximization of semi-monotone…
- Greedy+Singleton: An efficient approximation algorithm…
- Machine learning models with distinct Shapley value…
- Machine learning-based modelling, feature importance and…
- Multipass Streaming Algorithms for Regularized Submodular…
- Optimizing Shapley Value for Client Valuation in…
- SHapley Additive exPlanations (SHAP) for Efficient…
- Semi-streaming Algorithms for Submodular Function…
- Shapley value in machine learning modeling: optimizing…
- Shapley value: from cooperative game to explainable…
- Shapley-Based Data Valuation Method for the Machine…
- The Forward-Reverse Greedy Algorithm for Distributed…

- Why Shapley Value and Its Variants Are Useful in Machine…
- A Scalable and Efficient Intrusion Detection System Based…
- DERIVATIVE-BASED SHAPLEY VALUE FOR GLOBAL SENSITIVITY…
- Data Valuation Method Based on Federated Learning and…
- Data Valuation with Shapley-based Methods for Medical…
- Data valuation with Leave-One-Out (LOO) test and Shapley…
- Deterministic Algorithm and Faster Algorithm for…
- Fast Shapley Value Approximation Through Machine Learning…
- Heterogeneous Graph Data Valuation: A Shapley Value-based…
- Integrating Shapley Value and Least Core Attribution for…
- Localized Data Shapley: Accelerating Valuation for…
- Offline and Online Distributed Submodular Maximization…
- Online and Streaming Algorithms for Constrained…
- Optimizing Task Allocation in IT Project Management Using…
- Privacy-Preserving Feature Valuation in Vertical…
- Reward-Aware Shapley Compensation: a Probabilistic and…
- Shapley Patch Valuation Method for Histopathological…
- Shapley value-based data valuation for machine learning…
- Shapley-Based Data Valuation for Weighted k-Nearest…
- Shapley-Based Data Valuation with Mutual Information: A…
- Shapley-Value Based Feature Attribution for…
- Streaming Stochastic Submodular Maximization with…
- Streaming algorithms for non-monotone DR-submodular…
- Toward learnable and interpretable data Shapley valuation…
- k-Submodular Maximization Under Individual Knapsack…
- A Priority-Ordered Swapping Algorithm for Submodular…
- A Shapley-value cooperative game-based risk decision…
- A primal-dual algorithm for monotone submodular…
- A streaming algorithm for non-monotone regularized…
- Dynamic valuation of data assets via multi-agent…
- Energy-Based Model for Accurate Estimation of Shapley…
- Improving the accuracy and stability of privacy-aware…
- LTSV: Layered Type-Constrained Shapley Value for…
- Light Shapley: Improving the Scalability of Equitable…
- P30 - Möbius-Shapley: Native Feature Attribution for…
- Ripple Shapley: Data Influence Attribution in One…
- Shapley Value-Based Feature Attribution for Data Masking
- Streaming Submodular Maximization Under Matroid…
- Streaming submodular maximization with fairness…
- Submodular Maximization Subject to Uniform and Partition…

The transfer runs the wrong way for the present problem. Those literatures overwhelmingly assume a submodular or approximately submodular objective, because that is where the guarantees live. **The coverage objective here is supermodular**, which is exactly the regime the contemporary work sets aside, so the tooling is available and the guarantees are not.

The compiler started learning, which changes the corpus

Machine learning entered the compiler itself, in phase ordering, learned cost models and heuristic replacement, and large language models entered code production.

- Checking correctness of code generator architecture…
- Compiler-Directed Power Management for Superscalars
- S-compiler: A code vulnerability detection method

- A Compiler Approach for Exploiting Partial SIMD…
- A graph-based iterative compiler pass selection and phase…
- Clustering-Based Selection for the Exploration of…
- JavaScript Parallelizing Compiler for Exploiting…
- Compiler-Assisted Loop Hardening Against Fault Attacks
- On the Interactions Between Value Prediction and Compiler…
- Machine Learning in Compiler Optimization
- AutoPhase: Compiler Phase-Ordering for HLS with Deep…
- Nonio — modular automatic compiler phase selection and…
- Identifying Compiler and Optimization Options from Binary…
- Reliable Compilation Optimization Phase-ordering…
- Automatic Joint Optimization of Algorithm-Level…
- Identifying Compiler and Optimization Level in Binary…
- Memory Utilization and Machine Learning Techniques for…
- Towards Compile-Time-Reducing Compiler Optimization…
- A Novel Prediction Model for Compiler Optimization with…
- Automating reinforcement learning architecture design for…
- CARL: Compiler Assigned Reference Leasing
- Compiler Optimization Parameter Selection Method Based on…
- Compiler Support for Sparse Tensor Computations in MLIR
- Hybrid Approach based on Multi-agent System and Fuzzy…
- Language models can prioritize patches for practical…
- Object Intersection Captures on Interactive Apps to Drive…
- An Evaluation Method for Large Language Models' Code…
- Automated Program Repair in the Era of Large Pre-trained…
- Compiler Optimization for Quantum Computing Using…
- Is Your Code Generated by ChatGPT Really Correct?…
- Large Language Models for Automated Program Repair
- Work-in-Progress: Searching Optimal Compiler Optimization…
- A Comparative Evaluation of Prompting Strategies for Code…
- A Digital Twin Modeling Code Generation Framework based…
- A Multi-Expert Large Language Model Architecture for…
- A Survey of Optimized Compiler Using Advanced Machine…
- Advancements and Challenges of Large Language Model-Based…
- Assessing the Impact of Compiler Optimizations on GPUs…
- Automated C/C++ Program Repair for High-Level Synthesis…
- Chinese Generation and Security Index Evaluation Based on…
- CodeJudge: Evaluating Code Generation with Large Language…
- Compiler-Based Memory Encryption for Machine Learning on…
- Exponentially Expanding the Phase-Ordering Search Space…
- FormalEval: A Method for Automatic Evaluation of Code…
- LLM-based Control Code Generation using Image Recognition
- Large Language Models Meet Automated Program Repair…
- Large Language Models in Automated Repair of Haskell Type…
- Machine Learning Based Compiler Optimization Technique
- Machine Learning-Driven GCC Loop Unrolling Optimization…
- Research on Program Automatic Repair Method Combining…
- WhiteFox: White-Box Compiler Fuzzing Empowered by Large…
- A Systematic Review about Large Language Models (LLMs)…
- AI for Code: Reinforcement-Learned Compiler Optimizations…
- APICoder: A Multi-Role Large Language Model Framework for…
- AUTOMATED GENERATION AND EVALUATION OF VOCABULARY TESTS…
- Advancements in AI-Based Compiler Optimization Techniques…
- Applying Knowledge-Guided Deep Reinforcement Learning…

- Balancing Security and Correctness in Code Generation: An…
- Beyond Functional Correctness: An Empirical Evaluation of…
- CWEval: Outcome-driven Evaluation on Functionality and…
- Can AI Fix Buggy Code? Exploring the Use of Large…
- Causality-Aided Evaluation and Explanation of Large…
- Compiler-Assisted Optimization Using Neural Code…
- Compiler-R1: Towards Agentic Compiler Auto-tuning with…
- CompilerGPT: Leveraging Large Language Models for…
- DeCOS: Data-Efficient Reinforcement Learning for Compiler…
- Efficient program optimization through knowledge-enhanced…
- EvoAPR: Enhancing Large Language Models for Automatic…
- Exploiting Booster Pass Chain for Compiler Phase Ordering
- Finding Missed Code Size Optimizations in Compilers using…
- Finetune-Then-Merge: Democratizing Large Language Model…
- Holistic evaluation of LLM-Based Code Generation
- Hybrid Automated Program Repair by Combining Large…
- Implement Machine Learning to Schedule Instructions to…
- InstructRepair: Instruct Large Language Models With Rich…
- Knowledge Graph Based Repository-Level Code Generation
- LMFuzz: Program repair fuzzing based on large language…
- Large Language Model Generation Safety: A Comprehensive…
- LegoFuzz: Interleaving Large Language Models for Compiler…
- Leveraging Compilation Statistics for Compiler Phase…
- Model-Driven Quantum Code Generation Using Large Language…
- Navigating the SIMD Optimization Maze: A Reinforcement…
- Optimization, Machine Learning, and Fuzzy Logic
- Proving the Coding Interview: A Benchmark for Formally…
- ReAPR: Automatic program repair via retrieval-augmented…
- Reductive Analysis with Compiler-Guided Large Language…
- RepairBench: Leaderboard of Frontier Models for Program…
- Research on Compiler Optimization Technology Based on…
- Revisiting Unnaturalness for Automated Program Repair in…
- Role-Aware Intelligent Agent Framework for Enhanced Code…
- SPRoC: Semantics-Preserving Mutations for Robustness…
- Supporting Dynamic Program Sizes in Deep Learning-Based…
- T 3 : Multi-level Tree-based Automatic Program Repair…
- TOWARDS AUTONOMOUS CODE OPTIMIZATION: A REINFORCEMENT…
- Template-Guided Program Repair in the Era of Large…
- The Impact of Fine-Tuning Large Language Models on…
- The use of large language models for program repair
- Toward Green Code: Prompting Small Language Models for…
- Usage of Large Language Model for Code Generation Tasks…
- VEGA: Automatically Generating Compiler Backends using a…
- A Systematic Literature Review on Automated Program…
- A comparative analysis of the role of Large Language…
- Academic english writing generation-evaluation…
- An overview of evaluation and enhancement methods for…
- ArkTS code generation: A comprehensive evaluation with…
- Automatic Program Repair Using Large Language Models in…
- Automating code generation for a new ecosystem…
- Benchmarking Cross-Language Code Smell Detection with…
- Can test cases generated by large language models…
- Combining Static Code Analysis and Large Language Models…
- Exploring Generalizable Automated Program Repair With…

- From Natural Language to Interpretable Code: Automated…
- Large Language Model-Based Interactive Code Generation…
- Large Language Models for Code Translation: An In-Depth…
- Model-Agnostic Empirical Evaluation of Test-Driven Prompt…
- PredComp: Predicting Compiler Optimization Options with…
- PromptTone: A Dataset for Evaluating Large Language Model…
- Protean Compiler: An Agile Framework to Drive Fine-grain…
- RE-APR: Reasoning-Enhanced Automated Program Repair via…
- Rethinking Correctness and Efficiency in AI-Assisted Code…
- SAGE: A Compiler-assisted Reinforcement Learning-based…
- Technical Perspective: Fusing Large Language Models with…
- Towards defect-type-aware adaptive program repair: A…

The second of those affects this article's method rather than its argument. A corpus of real programs is the instrument's foundation, and a growing fraction of real programs is now machine-generated. **Whether generated code has the same instruction distribution as human code is an open empirical question**, and if it does not, then the corpus caveat this article already states becomes sharper rather than milder.

Ordering engineering work is a discipline, and it counts costs

There is a literature on deciding what to build next, covering technical debt prioritisation, requirements prioritisation and effort estimation.

- Accuracy Comparison of Analogy-Based Software Development…
- An Empirical Approach for Estimation of the Software…
- An Empirical Investigation on Effort Estimation in Agile…
- An empirical evaluation of ensemble adjustment methods…
- Analysis of task effort estimation accuracy based on use…
- Automated selection of a software effort estimation model…
- Empirical Application of Simulated Annealing Using…
- How do open source software (OSS) developers practice and…
- Prediction accuracy measurements as a fitness function…
- A large-scale empirical comparison of static and dynamic…
- A stability assessment of solution adaptation techniques…
- Assessment and Comparison of Fuzzy Based Test Suite…
- How do software development teams manage technical debt?…
- Negative results for software effort estimation
- Realistic assessment of software effort estimation models
- Systematic Mapping Study of Ensemble Effort Estimation
- Technical debt prioritization using predictive analytics
- AHP_GORE_PSR: Applying analytic hierarchy process in goal…
- An Empirical Comparison of Similarity Measures for…
- An Empirical Study of Technical Debt in Open-Source…
- An Evaluation of Selection Methods for Time-Aware Effort…
- DRank: A semi-automated requirements prioritization…
- Empirical Assessment of Machine Learning Models for…
- Empirical evaluation of fuzzy analogy for software…
- Entropy-based Framework Dealing with Error in Software…
- Evaluating Pred(p) and standardized accuracy criteria in…
- Fuzzy_MoSCoW: A fuzzy based MoSCoW method for the…
- Identifying self-admitted technical debt in open source…
- Looking for Peace of Mind? Manage Your (Technical) Debt…
- On the Evaluation of Effort Estimation Models
- Startups and Technical Debt: Managing Technical Debt with…

- Cuckoo search based hybrid models for improving the…
- Effective fault localization of automotive Simulink…
- Empirical evaluation of an entropy-based approach to…
- Flaws of Quantification Method as applied to Software…
- Identification and prioritization of SLR search tool…
- On the value of a prioritization scheme for resolving…
- Prioritizing solution-oriented software requirements…
- Project productivity evaluation in early software effort…
- Software Development Effort Estimation Using Random…
- Towards a functional requirements prioritization with…
- A novel online supervised hyperparameter tuning procedure…
- An Effort Estimation Support Tool for Agile Software…
- An Empirical Study on Technical Debt in a Finnish SME
- Analogy-Based Approaches to Improve Software Project…
- Business-Driven Technical Debt Prioritization
- Technical Debt Prioritization: A Search-Based Approach
- Tracy: A Business-Driven Technical Debt Prioritization…
- Usability Technical Debt in Software Projects: A…
- A Taste of the Software Industry Perception of Technical…
- A systematic literature review of technical debt…
- Continuous Debt Valuation Approach (CoDVA) for Technical…
- Empirical Evaluation of Mimic Software Project Data Sets…
- Evaluating the agreement among technical debt measurement…
- Improving Estimation Accuracy Prediction of Software…
- Long-Term Evaluation of Technical Debt in Open-Source…
- Profiling Developers Through the Lens of Technical Debt
- Using Extremely Simplified Functional Size Measures for…
- Wait for it: identifying “On-Hold” self-admitted…
- A Collaborative Effort-Benefit-Value Analysis Model to…
- A Stacking Ensemble-based Approach for Software Effort…
- A systematic literature review on Technical Debt…
- AI Techniques for Software Requirements Prioritization
- An Extreme Learning Machine based Approach for Software…
- Applying test case prioritization to software…
- Correction to: Wait for it: identifying “On-Hold”…
- Flower Pollination Algorithm for Software Effort…
- Measuring affective states from technical debt
- Neural Networks based Software Development Effort…
- Refactorings and Technical Debt in Docker Projects: An…
- Self-admitted technical debt practices: a comparison…
- Software effort estimation accuracy prediction of machine…
- Technical Debt Prioritization: Taxonomy, Methods Results…
- 23 shades of self-admitted technical debt: an empirical…
- Adopting DevOps Paradigm in Technical Debt Prioritization…
- An Empirical Study on Software Test Effort Estimation for…
- An empirical study on self-admitted technical debt in…
- An extended study on applicability and performance of…
- Asking about Technical Debt: Characteristics and…
- Blockchain-Based Software Effort Estimation: An Empirical…
- Characterizing Technical Debt in Evolving Open-source…
- DebtFree: minimizing labeling cost in self-admitted…
- Development effort estimation in free/open source…
- Empirical Research for Self-Admitted Technical Debt…
- Evaluation of Context-Aware Language Models and Experts…

- FIXME: synchronize with database! An empirical study of...
- Heterogeneous Graph Neural Networks for Software Effort...
- Identifying self-admitted technical debt in issue...
- MCBRank Method to Improve Software Requirements...
- On the documentation of self-admitted technical debt in...
- OurRank: A Software Requirements Prioritization Method...
- PriorTD: A Method for Prioritization Technical Debt
- Security Requirements Prioritization Techniques: A Survey...
- Self-Admitted Technical Debt and comments' polarity: an...
- Solution to CAD Designer Effort Estimation based on...
- Technical Debt, Software Evolution and Legacy
- Technical debt prioritization
- The Influence of Cost Drivers on Effort Estimation in...
- Toward prioritization of self-admitted technical debt: an...
- Use Case-Based Analytical Hierarchy Process Method for...
- A practical approach for technical debt prioritization...
- Automatic identification of self-admitted technical debt...
- Evaluating ensemble imputation in software effort...
- Exploring Technical Debt in Security Questions on Stack...
- Heterogeneous Ensemble Model to Optimize Software Effort...
- Improving Software Requirements Prioritization through...
- Much more than a prediction: Expert-based software effort...
- Software effort estimation using validated accuracy...
- Technical Debt Contagiousness Metrics for Measurement and...
- An Empirical Study on Self-Admitted Technical Debt in...
- Optimizing Software Effort Estimation Accuracy with a...
- Quantifying and characterizing clones of self-admitted...
- Technical Debt Tools: a Survey and an Empirical Evaluation
- The broken windows theory applies to technical debt
- The method of requirements prioritization in software...
- Agile Effort Estimation Improved by Feature Selection and...
- An Empirical Study on Software Developer-Related Factors...
- How do Community Smells Influence Self-Admitted Technical...
- Identifying Key Requirements Prioritization Criteria for...
- Negativity in self-admitted technical debt: how sentiment...
- Software effort estimation using validated accuracy...
- Swarm Intelligence for Software Effort Estimation: An...
- Understanding practitioners' reasoning and requirements...
- A Rigorous Empirical Benchmark of Machine Learning Models...
- An Empirical Study of Self-Admitted Technical Debt in...
- An empirical evaluation of white-box and black-box test...
- Effort-optimized, accuracy-driven labelling and...
- Hybrid Model for Improving the Accuracy of Software...
- Input perturbation robustness for software effort...
- Leveraging explainable AI for requirements prioritization...
- Reducing labeling effort in architecture technical debt...
- TagDebt: a bot to support technical debt management
- Test input prioritization for image segmentation: an...
- Towards Interpretable Ensemble Learning for Software...
- Understanding Self-Admitted Technical Debt in Test Code...
- VECTR: A Lightweight Requirements Prioritization Method...

Read for the ordering question, it is almost entirely a cost-side literature. Effort estimation predicts κ. Technical debt prioritisation ranks by remediation cost and interest.

Nothing found here defines a blocking-frequency notion, meaning a measure of how much delivered capability is gated on a single unbuilt item, which is the quantity this article argues should lead.

The runtime side, and reading machine code back

Two smaller bodies of work bound the subject. The choice between interpretation, just-in-time compilation and ahead-of-time compilation is the alternative to native lowering rather than a part of it.

- 14. Compilation of dictionaries of the Chinese language
- A Bytecode Interpreter for Secure Program Execution in…
- A Software-Managed Approach to Die-Stacked DRAM
- Formal Certification of Non-interferent Android Bytecode…
- Load Balancing in Decoupled Look-ahead: A Do-It-Yourself…
- The Simian concept: Parallel Discrete Event Simulation…
- Buddhist studies on Seokbosangjeol and tasks ahead…
- Compilation for Operation Execution Time Variability
- Programming GPUs with C++14 and Just-In-Time Compilation
- A Study of Virtual Machine Placement Optimization in Data…
- Adaptive just-in-time and relevant vector machine based…
- DRUT: An Efficient Turbo Boost Solution via Load…
- Fast Failure Erasure Encoding Using Just in Time…
- Just-In-Time GPU Compilation for Interpreted Languages…
- Corpus Compilation and Exploitation in Language…
- ClangJIT: Enhancing C++ with Just-in-Time Compilation
- Just-In-Time Compilation for Verilog
- POSTER: Tango: An Optimizing Compiler for Just-In-Time…
- JIT Leaks: Inducing Timing Side Channels through…
- PHPIL: Fuzzing the PHP Interpreter with Custom Bytecode
- Automatically exploiting the memory hierarchy of GPUs…
- Fractional Artificial Bee Chicken Swarm Optimization…
- Just-In-Time Compilation on ARM—A Closer Look at…
- Just-in-Time Compilation and Link-Time Optimization for…
- Just-in-time scheduling in identical parallel machine…
- Preprocessing and Compilation
- Quantum simulation with just-in-time compilation
- A Low-Level Virtual Machine Just-In-Time Prototype for…
- AHEAD-OF-TIME and JUST-IN-TIME technologies
- CHERI Performance Enhancement for a Bytecode Interpreter

- Hybrid Metaheuristic Technique for Optimization of…
- Reducing the Compilation Time of Quantum Circuits Using…
- Remote Just-in-Time Compilation for Dynamic Languages
- Resource Optimization based Virtual Machine Allocation…
- Reverse Engineering of Obfuscated Lua Bytecode via…
- Compilation Optimization Methods in the Customization of…
- Efficient Virtual Machine Allocation Technique Based on…
- Interactive Programming for Microcontrollers by…
- Accelerating Startup Time of React Native Applications by…
- An efficient load balance using virtual machine migration…
- FPGA-Accelerated Neural Network Inference via a…
- From Source to Bytecode: How.py Becomes.pyc
- Intelligent Power Grid Startup Scheme Based on Rule…
- Proteus: Portable Runtime Optimization of GPU Kernel…
- Trace-Based Bytecode Interpreter Visualization for…
- WebAssembly: How Low Can a Bytecode Go?
- AST, Bytecode, and the Space In Between: An Exploration…
- Designing quantum chemistry algorithms with just-in-time…
- Practical Python FPGA Acceleration with Fast Just-In-Time…
- Why Just-In-Time Compilation Matters: Evaluating Runtime… And the inverse problem of recovering structure from machine code has its own methods.
- Lifting Assembly to Intermediate Representation
- Zipr: Efficient Static Binary Rewriting for Security
- Evolving Exact Decompilation
- Towards Incremental Static Race Detection in OpenMP…
- CLIK on PLCs! Attacking Control Logic with Decompilation…
- Performance, Correctness, Exceptions: Pick Three
- How far we have come: testing decompilation correctness…
- PARCOACH Extension for Static MPI Nonblocking and…
- High-Precision Evaluation of Both Static and Dynamic…
- Beyond the C: Retargetable Decompilation using Neural…
- NemesisGuard: Mitigating interrupt latency side channel…
- Performant Binary Fuzzing without Source Code using…
- Static Local Concurrency Errors Detection in MPI-RMA…
- BIRD: A Binary Intermediate Representation for Formally…
- BREWasm: A General Static Binary Rewriting Framework for…

- Cross-Language Binary-Source Code Matching Based on Rust…
- Static Analysis of JNI Programs via Binary Decompilation
- dewolf: Improving Decompilation by leveraging User Surveys
- Automatically Mitigating Vulnerabilities in Binary…
- Binary Similarity Detection Based on Intermediate…
- Binary–Source Code Matching Based on Decompilation…
- dAngr: Lifting Software Debugging to a Symbolic Level
- Does Representation Matter? Evaluating IRs for LLM-based…
- LAPSE: Automatic, Formal Fault-Tolerant Correctness…
- LLM-assisted end-to-end binary decompilation: a…
- Large Language Models for Binary Decompilation…
- NOProbe: A NOP-Based Dynamic Binary Instrumentation…
- Representation learning for coincidental correctness in…

The second matters here for a methodological reason. The instrument measures the compiler's bytecode output rather than source text, which is a lifting problem in miniature, and the difficulties that literature documents are the reasons the measurement was kept deliberately shallow.

What the survey shows

The ordering question is asked far more often than it was, by far more people, and it still has no published principle. Custom instruction sets, a universal new target, staged lowering pipelines and accelerator back ends have all multiplied the occasions for asking it. The classics that did not answer it have been extended, automated and verified, and the extensions do not answer it either.

The one thing that changed decisively is the cost of finding out. Corpus measurement became ordinary, so the advice to measure before ordering costs almost nothing to follow.

And the blindness is structural rather than accidental. Compiler testing asks whether what exists is correct. Coverage tooling asks how much of what exists is exercised. Verification asks whether what exists is sound. **None of them asks what does not exist yet and what that absence costs**, because all three presuppose the artefact. The question this article asks falls in the gap between building a compiler and testing one, and that gap has no literature.

The Source Base

This survey rests on 3,155 records harvested from Crossref across sixty-one queries, filtered to 1,678 by removing homonym contamination and reduced to 1,650 by removing duplicate titles. The queries were written to span the argument rather than the subject, so each cluster corresponds to a claim above instead of to a keyword.

The filtering was not incidental and the failures explain the counts, because they are the reason the counts are what they are. A query on binary translation returned a large body of work on the static dielectric constants of binary liquid mixtures. A query on code corpora returned linguistic corpora. A query on interpreters returned the training of human interpreters. A venue-level filter alone was insufficient, since a study of industrial chiller faults reached the shortlist through the word empirical in a software-engineering venue. **Every**

contaminant listed here was found by reading a sample of each cluster and none by anticipating it, which is the same lesson the article's own citation-defect section reports in a different setting.

The period distribution is deliberate and is reported as a count rather than only as a fraction. The foundational base carries the derivations and stands at 111 references with a median year of 1997, of which 62 predate 2000. The contemporary base carries this survey. **Adding a contemporary survey lowers the primary fraction while leaving the primary count unchanged**, so the fraction on its own would read as a regression when it is the directive working.

One property of this survey's references matters here, because the article devotes a section to the opposite case. The citation defect reported earlier arose because identifiers were supplied from memory, so a digital object identifier could be paired with a title it did not belong to. **These were retrieved rather than recalled.** The identifier, title, author and year of each come from a single record, so the substitution failure is not available here by construction. **That is a structural guarantee and not a verified one**, and what verification can still add is resolvability, checked on a random sample of 120 of the 1,650, all of which resolve. **The foundational references were checked exhaustively instead**, at 86 of 86 digital object identifiers resolving to the claimed author and year.

What the source base cannot do is establish an absence. The claim that no published ordering principle exists is supported by not finding one across these queries, which is weaker than a proof and is stated that way in the Epistemic State below.

Epistemic State

Measured, and reproducible from the instrument. The corpus comprises 63 files of which 58 compiled, yielding 496 compilation units and 73,434 instruction instances. The implemented subset was 39 of 66 instructions. Instruction-level coverage was 0.8731 and unit-level coverage 0.3387, for a gap of 0.5344. The leading workstream blocked 267 units against 28 for the next, a ratio of 9.54. The instruction class the falsified recommendation would have unblocked occurs zero times. Of 91 candidate digital object identifiers supplied from memory, four resolved to different works and one did not resolve, an error rate of 5.5 percent.

Derived, and checkable from the definitions. That unit-level coverage cannot exceed instruction-level coverage follows from the product form. That the objective is supermodular and not submodular, so that the classical greedy approximation guarantee does not apply, is established by counterexample. The clustering coefficient of 5.51 and the confidence bounds on the zero counts follow from the stated method.

Assumed, and marked as such. The first-blocker attribution is order-dependent and the full blocking lattice was not computed, so **the ordering of the second, third and fourth workstreams is not established** and no claim about it should be read as such. The cost ratio between the measurement and the work it redirected is an order of magnitude rather than a quantity, because its numerator is measured and its denominator is an estimate of work never performed.

Corrected during writing, and left visible. An independence null was first evaluated at the mean compilation-unit length rather than per unit, which would have inflated a clustering coefficient by seven orders of magnitude. A modularity theorem was asserted that the objective does not admit. A citation defect rate was first reported as near ten percent over the worse subsample rather than 5.5 percent over the full one. **The conclusion that operand type recovery was worthless was itself too broad**, since the capability was later required by a different workstream for an unrelated reason, at 18 compilation units. The zero was correct and the inference from it was not.

What the article does not establish. Whether the corpus resembles the population the generator will eventually serve, since it is drawn from one project's own examples and over-represents a text-processing workload. Whether the classification of instructions into workstreams was drawn neutrally, since the party that made the falsified recommendation also wrote the instrument that falsified it. **The article offers the classification for inspection as mitigation and does not claim the concern is closed.**

The strongest claim the evidence supports is that on this corpus, at this implemented subset, a frequency measurement redirected work away from an item that would have delivered exactly zero. **The weakest link is corpus representativeness**, and a reader who doubts it should read the ordering as established for the present consumer rather than for the eventual one.

Out of Scope

The design of the code generator itself, and the lowering strategy for any particular instruction. The worst-case execution time and memory analysis that motivates the project, which is a separate subject. Register allocation, instruction scheduling and peephole optimisation, none of which the ordering question touches. Dynamic profiling, since the measurement here is static by design and the two answer different questions. The exact Shapley attribution over the blocking lattice, which is affordable at 64 corpus passes and was not performed. Any claim about execution time, which would require a different and considerably more careful methodology. The Keleusma language itself and its implementation history, which are covered in the getting-started article and the self-hosting strategy.

Conclusion

Eighty-seven percent of the instructions and thirty-four percent of the programs are the same compiler on the same day. The first number is what a progress report contains and the second what a user experiences, and the distance between them is created entirely by the fact that a program needs every instruction it uses.

The practical rule is short. When a consumer requires a conjunction of features, per-feature progress overstates delivered capability, and the amount of the overstatement is not visible from inside the work. **Dependency analysis tells you what is required before something can be built. It tells you nothing about whether anyone needs it**, and those two questions are independent in a way that is easy to miss because the first is more satisfying to answer.

The measurement that settles it is usually trivial next to the work it redirects. Twenty minutes of instrument against a research spike, in this case, and the spike would have delivered nothing.

The last part is the least comfortable and the most useful. Four errors were made in producing this article and every one of them pointed toward a more striking result. None was found by rereading. Each was found by running something, whether a numerical evaluation whose answer was implausible, an enumeration over small cases, a query against an authoritative record, or a recount over the full sample rather than the remembered one. **The fourth was committed while writing the warning about the first three**, which is the clearest evidence available that knowing about a bias does not protect anyone from it.

References

Reference

- LLVM Language Reference

- Oracle Java Virtual Machine Specification

Related Post

- Related Post, Getting Started with Keleusma 0.2.2
- Related Post, The Self-Hosted Silicon Compiler
- Related Post, Keleusma's Self-Hosting Strategy
- Related Post, The Stream Processor as Compiler and the Compiler as Stream Processor

Research

- "A Multi-Pass Compiler with Code Optimized Abstract…
- μPRL: A Mutation Testing Pipeline for Deep…
- 12.4 - Machine Learning-Driven Test Case Prioritization…
- 14. Compilation of dictionaries of the Chinese language
- 23 shades of self-admitted technical debt: an empirical…
- 3CPS: The Design of an Environment-Focussed Intermediate…
- A basic linear algebra compiler for structured matrices
- A benchmark study on the effectiveness of search-based…
- A buffer overflow detection and defense method based on…
- A Bytecode Interpreter for Secure Program Execution in…
- A Calculus for Web Services Choreography: Formal…
- A Case Study in Formal Specification and Runtime…
- A Collaborative Effort-Benefit-Value Analysis Model to…
- A comparative analysis of the role of Large Language…
- A Comparative Evaluation of Prompting Strategies for Code…
- A Compilation of Experimental Binary Alloy Surface…
- A Compiler Approach for Exploiting Partial SIMD…
- A compiler architecture for domain-specific type error…
- A Compiler Comparison in the RISC-V Ecosystem
- A compiler for cyber-physical digital microfluidic…
- A Compiler for Sound Floating-Point Computations using…
- A Comprehensive Study of Bugs in Embedded WebAssembly…
- A Comprehensive Study of WebAssembly Runtime Bugs
- A Comprehensive Study on Code Coverage Analysis for…
- A Comprehensive Trusted Runtime for WebAssembly With…
- A Control Flow based Static Analysis of GRAFCET using…
- A Convolutional Neural Network for Language-Agnostic…
- A cooperative game theory application in chicks brood…
- A cost model for a graph-based…
- A Customized Real-Time Compilation for Motion Control in…
- A Defect Taxonomy for Infrastructure as Code: A…
- A detailed investigation of the effectiveness of whole…
- A Digital Twin Modeling Code Generation Framework based…
- A distributed algorithm for partitioned robust submodular…
- A Domain-Specific Compiler for a Parallel Multiresolution…
- A Domain-Specific Compiler for Embedded DSP Development…
- A Domain-Specific Language and Compiler for…
- A Formal Approach based on Fuzzy Logic for the…
- A Formal Proof of the Soundness of the Hybrid CPS Clock…
- A Formal Verification Library Design for Behavioral…
- A formally verified compiler for Lustre
- A Formally Verified Microcoded RISC-V Platform
- A Fully Automated Agent for End-to-End Code Translation…

- A fully verified container library
- A graph-based iterative compiler pass selection and phase…
- A Graph-Based Learning Framework for Compiler Loop…
- A High Performance Sparse Tensor Algebra Compiler in MLIR
- A Hotspot-Driven Semi-automated Competitive Analysis…
- A Large Scale Analysis of Semantic Versioning in NPM
- A Large Scale Study of License Usage on GitHub
- A Large Scale Study of Long-Time Contributor Prediction…
- A Large-Scale Comparative Evaluation of IR-Based Tools…
- A large-scale comparison of Python code in Jupyter…
- A large-scale dataset of (open source) license text…
- A Large-Scale Dataset of MCP Implementations on GitHub
- A large-scale empirical comparison of static and dynamic…
- A large-scale empirical study of configurations, errors…
- A Large-Scale Empirical Study of COVID-19 Themed GitHub…
- A Large-Scale Empirical Study of Open Source License…
- A large-scale empirical study on self-admitted technical…
- A Large-Scale Investigation Into the Loss of Pull Request…
- A Large-Scale Study About Quality and Reproducibility of…
- A large-scale study of call graph-based impact prediction…
- A large-scale study of programming languages and code…
- A Large-Scale Study of the Impact of Feature Selection…
- A large-scale study on human-cloned changes for automated…
- A large-scale study on repetitiveness, containment, and…
- A large-scale study on the usage of Java’s concurrent…
- A low-cost synthesizable RISC-V dual-issue processor core…
- A Low-Level Virtual Machine Just-In-Time Prototype for…
- A machine-checked correctness proof for Pastry
- A machine-checked direct proof of the Steiner-lehmus…
- A MATLAB Vectorizing Compiler Targeting…
- A Mechanical Soundness Proof for Subtyping Over Recursive…
- A Mechanised and Constructive Reverse Analysis of…
- A Mechanism for Automatically Extracting Reusable and…
- A Methodology for Analysing Code Anomalies in Open-Source…
- A minimalistic verified bootstrapped compiler (proof…
- A Model-Agnostic Feature Attribution Approach to…
- A Multi-Armed Bandit Approach for Test Case…
- A Multi-Expert Large Language Model Architecture for…
- A Multi-pass Streaming Algorithm for Regularized…
- A New Functional-Logic Compiler for Curry: Sprite
- A new verified compiler backend for CakeML
- A Novel Counterexample-Guided Inductive Synthesis…
- A novel online supervised hyperparameter tuning procedure…
- A Novel Prediction Model for Compiler Optimization with…
- A Pluggable Vector Unit for RISC-V Vector Extension
- A practical approach for technical debt prioritization…
- A Preliminary Framework for Optimising Test Case…
- A Preliminary Study of Fixed Flaky Tests in Rust Projects…
- A primal-dual algorithm for monotone submodular…
- A Priority-Ordered Swapping Algorithm for Submodular…
- A Programming Approach for Worst-case Studies in…
- A Proof Score Approach to Formal Verification of an…
- A Proof-Producing Compiler for Blockchain Applications
- A proposed synthesis method for Application-Specific…

- A Qualitative Study on the Sources, Impacts, and…
- A QUANTITATIVE ANALYSIS OF WEBASSEMBLY INTEGRATION…
- A Reinforcement Learning Environment for Automatic Code…
- A Review on Continuous Integration and Continuous…
- A Rigorous Empirical Benchmark of Machine Learning Models…
- A RISC-V Instruction Set Extension for Flexible…
- A RISC-V instruction set processor-micro-architecture…
- A RISC-V Post Quantum Cryptography Instruction Set…
- A RISC-V Vector Extension for Multi-word Arithmetic
- A Rose Tree Is Blooming (Proof Pearl)
- A Safe Low-Level Language for Computer Algebra and Its…
- A Scalable and Efficient Intrusion Detection System Based…
- A Self-certifying Compilation Framework for WebAssembly
- A Semantics of Structures, Unions, and Underspecified…
- A Semi-streaming Algorithm for Monotone Regularized…
- A Shapley-value cooperative game-based risk decision…
- A simple soundness proof for dependent object types
- A Software-Managed Approach to Die-Stacked DRAM
- A stability assessment of solution adaptation techniques…
- A Stacking Ensemble-based Approach for Software Effort…
- A streaming algorithm for non-monotone regularized…
- A Streaming Model for Monotone Lattice Submodular…
- A study of common bug fix patterns in Rust
- A Study of Virtual Machine Placement Optimization in Data…
- A study on the lifecycle of flaky tests
- A Survey of Automatic Generation of Source Code Comments…
- A Survey of Optimized Compiler Using Advanced Machine…
- A systematic literature review of technical debt…
- A Systematic Literature Review on Automated Program…
- A systematic literature review on Technical Debt…
- A Systematic Review about Large Language Models (LLMs)…
- A Taste of the Software Industry Perception of Technical…
- A Taxonomy of Spanish Nouns, a Statistical Algorithm to…
- A Tensor Algebra Compiler for Sparse Differentiation
- A Time Window based Reinforcement Learning Reward for…
- A Time-Predictable Multicore RISC-V Architecture for…
- A Trigonometric Function Instruction Set Extension Method…
- A Two-step Approach to Find Short Compilation…
- A Universal Quantum Compiler GPT: Multi-Framework…
- A Verified CompCert Front-End for a Memory Model…
- A Verified Compiler for a Functional Tensor Language
- A Verified Compiler from Isabelle/HOL to CakeML
- A Verified Generational Garbage Collector for CakeML
- A Verified Generational Garbage Collector for CakeML
- A verified protocol buffer compiler
- A verified type system for CakeML
- A Way to Identify Potential Functions for Vectorization…
- A XOR data compiler: Combined with physical unclonable…
- Abstract Interpretation of Supermodular Games
- Abstract Interpretation with Infinitesimals
- Academic english writing generation-evaluation…
- Accelerating Embedded WebAssembly Based on FPGA
- Accelerating H.264/HEVC video slice processing using…
- Accelerating Machine Learning using RISC-V Vector…

- Accelerating Machine Learning with RISC-V Vector…
- Accelerating NTT with RISC-V Vector Extension for Fully…
- Accelerating Sparse Algebra with Program Synthesis
- Accelerating Startup Time of React Native Applications by…
- Accelerating Syntax-Guided Program Synthesis by…
- Acceleration of McEliece Cryptosystem with Instruction…
- Accessible Formal Methods for Verified Parser Development
- Accuracy Comparison of Analogy-Based Software Development…
- Accurate Compiler and Optimization Independent Function…
- Accurate quasi-P traveltimes in 3D transversely isotropic…
- Accurate quasi-SV traveltimes in 3D transversely…
- Achieving High-Integrity Software Quality and Security…
- Adaptive and Explainable Test Case Prioritization in…
- Adaptive just-in-time and relevant vector machine based…
- Adaptivity in AdaptiveCpp: Optimizing Performance by…
- Adjustable-Cost Overlays for Runtime Compilation
- Adopting DevOps Paradigm in Technical Debt Prioritization…
- Advanced ahead-of-time compilation for Javascript engine
- Advancements and Challenges of Large Language Model-Based…
- Advancements in AI-Based Compiler Optimization Techniques…
- Agile Autotuning of a Transprecision Tensor Accelerator…
- Agile Effort Estimation Improved by Feature Selection and…
- Agresti and Coull 1998
- Ahead of Time Generation for GPSA Protection in RISC-V…
- AHEAD-OF-TIME and JUST-IN-TIME technologies
- Ahead-of-time Compilation for Diverse Samplers of…
- Ahead-of-time compilation of JavaScript programs
- Aho, Ganapathi and Tjiang 1989
- AHP_GORE_PSR: Applying analytic hierarchy process in goal…
- AI builds, We Analyze: An Empirical Study of AI-Generated…
- AI Edge Processor Using RISC - V Instruction Set…
- AI for Code: Reinforcement-Learned Compiler Optimizations…
- AI Techniques for Software Requirements Prioritization
- AI-Driven Test Case Generation for Continuous Integration…
- AIWC: OpenCL-Based Architecture-Independent Workload…
- Algorithms to estimate Shapley value feature attributions
- Alive-FP: Automated Verification of Floating Point Based…
- Alive-Infer: data-driven precondition inference for…
- AliveInLean: A Verified LLVM Peephole Optimization…
- Allamanis and Sutton 2013
- Allen 1970
- Alumni’s Perception on Program Specification of ELT…
- Amdahl 1967
- An Agile Instruction Set Extension Method Based on the…
- An Analysis of Modern Web Security Vulnerabilities Inside…
- An application of metamorphic testing for testing…
- An application specific instruction set processor (ASIP)…
- An Application-specific Instruction Set Processor for…
- An Application-Specific Instruction Set Processor for…
- An Application-Specific VLIW Processor with Vector…
- An approach to generate text-based IDEs for syntax…
- An Approximation Algorithm for Distributed Resilient…
- An Automatic Generation and Verification Method of…
- An Effective GRU-Based Deep Learning Method for Test Case…

- An Efficient Application Specific Instruction Set…
- An Efficient Application Specific Instruction Set…
- An efficient load balance using virtual machine migration…
- An Effort Estimation Support Tool for Agile Software…
- An Embedded RISC-V Vector Extension for Edge-Oriented…
- An Empirical Analysis of Issue Templates Usage in…
- An Empirical Analysis of UI-Based Flaky Tests
- An Empirical Approach for Estimation of the Software…
- An Empirical Comparison of Human and LLM-Assisted Bug…
- An Empirical Comparison of Similarity Measures for…
- An empirical evaluation of ensemble adjustment methods…
- An empirical evaluation of white-box and black-box test…
- An Empirical Investigation on Effort Estimation in Agile…
- An Empirical Study of Activity, Popularity, Size…
- An Empirical Study of Aging Related Bug Prediction Using…
- An Empirical Study of Bug Bounty Programs
- An Empirical Study of Bug Fixing Rate
- An empirical study of bugs in test code
- An Empirical Study of Counterexample-Guided Fuzzing for…
- An Empirical Study of Flaky Tests in Android Apps
- An Empirical Study of Flaky Tests in JavaScript
- An Empirical Study of Flaky Tests in Python
- An Empirical Study of Greedy Test Suite Minimization…
- An empirical study of inadequate and adequate test suite…
- An Empirical Study of Multi-entity Changes in Real Bug…
- An Empirical Study of Real-World WebAssembly Binaries
- An empirical study of regression test suite reduction…
- An Empirical Study of Regression Testing for Android Apps…
- An Empirical Study of SBOM Usage Through GitHub Actions
- An Empirical Study of Self-Admitted Technical Debt in…
- An Empirical Study of Technical Debt in Open-Source…
- An Empirical Study of the Bug Link Rate
- An empirical study of the effectiveness of IR-based bug…
- An empirical study of the long duration of continuous…
- An Empirical Study of the Personnel Overhead of…
- An Empirical Study of the Relationship between Continuous…
- An Empirical Study of Web Flaky Tests: Understanding and…
- An Empirical Study on Code Coverage of Performance Testing
- An Empirical Study on Effects of Code Visibility on Code…
- An empirical study on how expert knowledge affects bug…
- An Empirical Study on Real Bug Fixes
- An Empirical Study on Real Bugs for Machine Learning…
- An empirical study on self-admitted technical debt in…
- An Empirical Study on Self-Admitted Technical Debt in…
- An Empirical Study on Software Developer-Related Factors…
- An Empirical Study on Software Test Effort Estimation for…
- An Empirical Study on Technical Debt in a Finnish SME
- An empirical study on the application of mutation testing…
- An Empirical Study on the Correlation between Neuron…
- An Empirical Study on the Cross-Project Predictability of…
- An Empirical Study on the Impact of Aspect-oriented…
- An empirical study on the potential of word embedding…
- An Environment Adaptation Agent of Reinforcement Learning…
- An Evaluation Method for Large Language Models’ Code…

- An Evaluation of Ranking-to-Learn Approaches for Test…
- An Evaluation of Selection Methods for Time-Aware Effort…
- An Experimental Analysis of RL based Compiler…
- An Explainable Deep Learning Model in Improving Test Case…
- An exploratory study of bug-introducing changes…
- An Exploratory Study on Soft Skills present in Software…
- An extended study on applicability and performance of…
- An extension to the RISC-V instruction set architecture…
- An extensive replication study of the ABLoTS approach for…
- An Extreme Learning Machine based Approach for Software…
- An FPGA-Based RISC-V Instruction Set Extension and Memory…
- An Improved Method for Test Case Prioritization in…
- An Improved Shapley Value Benefit Distribution Mechanism…
- An Improved Space Semi-Streaming Algorithm for Submodular…
- An Improvement to Test Case Prioritization Techniques…
- An Industrial Application of Mutation Testing: Lessons…
- An Initiative to Improve Reproducibility and Empirical…
- An Innovation Study on Luggage Wheel Design for Seamless…
- An innovative approach for automatic generation…
- An innovative machine learning workflow to research…
- An Interval Compiler for Sound Floating-Point Computations
- An MLIR-based Compiler Flow for System-Level Design and…
- An MLIR-Based Compiler for Hardware Acceleration with…
- An optimal streaming algorithm for non-submodular…
- An Optimal Streaming Algorithm for Submodular…
- An optimizing compiler for a purely functional…
- An overview of evaluation and enhancement methods for…
- An Ultralow-latency Constrained Quadratic Program (QP)…
- Analogy-Based Approaches to Improve Software Project…
- Analysis of benchmark program results of worst case…
- Analysis of task effort estimation accuracy based on use…
- Analysis of the RISC-V Vector Extension for Vulkan…
- Analysis of WebAssembly as a Strategy to Improve…
- Analytics-Driven Load Testing: An Industrial Experience…
- Analyzing Data Flow and Control Flow of Multicore…
- Analyzing False Positive Source Code Vulnerabilities…
- Andrews and colleagues 2005
- AndroidCompass: A Dataset of Android Compatibility Checks…
- APICoder: A Multi-Role Large Language Model Framework for…
- Appel 1998
- Application of an Improved Shapley Value Method in…
- Application of Property-based Testing Tools for…
- Application of WebAssembly for High-Performance…
- Application Specific Instruction Set Processor Design for…
- Application specific instruction set processor for sensor…
- Application-Specific Instruction Set Processor
- Application-Specific Instruction Set Processors for Video…
- Applications and Computation of the Shapley Value in…
- Applying Knowledge-Guided Deep Reinforcement Learning…
- Applying test case prioritization to software…
- Approximation Algorithm for Connected Submodular Function…
- APR4Vul: an empirical study of automatic program repair…
- Architecture of a tool for automated testing the…
- Archs: A WebAssembly Runtime for Cross-host Heterogeneous…

- Are mutation scores correlated with real fault detection?
- Are tweets useful in the bug fixing process? An empirical…
- ArkTS code generation: A comprehensive evaluation with…
- Artifact for article: Exact and Approximate Methods for…
- Artifact for Lifting Code Generation of Cardiac…
- Asking about Technical Debt: Characteristics and…
- ASPIRE: An Intermediate Representation for Abstract…
- Assessing and Improving the Mutation Testing Practice of…
- Assessing the Impact of Compiler Optimizations on GPUs…
- Assessing the Test Suite of a Large System Based on Code…
- Assessment and Comparison of Fuzzy Based Test Suite…
- Assessment of off-the-shelf SE-specific sentiment…
- Association Rule Learning Based Approach to Automatic…
- Assuring Correctness, Testing, and Verification of…
- AST, Bytecode, and the Space In Between: An Exploration…
- Attention Transfer Reinforcement Learning for Test Case…
- Audio Denoising Coprocessor Based on RISC-V Custom…
- Augmenting JavaScript JIT with ahead-of-time compilation
- Automated C/C++ Program Repair for High-Level Synthesis…
- Automated compiler optimization of multiple vector…
- Automated Compiler Optimization of Multiple Vector…
- Automated formal verification of the refined…
- AUTOMATED GENERATION AND EVALUATION OF VOCABULARY TESTS…
- Automated Inference of Expressive Metamorphic Relations…
- Automated NFR testing in continuous integration…
- Automated Program Repair in the Era of Large Pre-trained…
- Automated SC-MCC test case generation using…
- Automated selection of a software effort estimation model…
- Automated Test Case Generation from OTS/CafeOBJ…
- Automated Test Generation from Program Documentation…
- Automated Testing to Evaluate Employee Attendance System…
- Automated WebAssembly Function Purpose Identification…
- Automatic Benchmark Generation for Object Constraint…
- Automatic compiler optimization on embedded software…
- Automatic compiler/interpreter generation from programs…
- Automatic complex instruction identification for…
- Automatic Configurable Hardware Code Generation for…
- Automatic data layout generation and kernel mapping for…
- Automatic Generation of Assertions for Functional…
- Automatic Generation of Assertions for Security…
- Automatic generation of fast BLAS3-GEMM: A portable…
- Automatic Generation of Multi-Objective Polyhedral…
- Automatic generation of simulation workflows for system…
- Automatic Generation of the AADL ALISA Verification Plan…
- Automatic identification of self-admitted technical debt…
- Automatic Joint Optimization of Algorithm-Level…
- Automatic program bug fixing by focusing on finding the…
- Automatic Program Repair Using Large Language Models in…
- Automatic security verification of mobile app…
- Automatic Test Case Generation for Jasper App HDL…
- Automatic Testbench Generation for Simulation-based…
- Automatic Verification of Data Summaries
- Automatic Verification of FSA Strategies via…
- Automatically exploiting the memory hierarchy of GPUs…

- Automatically Localizing Dynamic Code Generation Bugs in…
- Automatically Mitigating Vulnerabilities in Binary…
- Automating code generation for a new ecosystem…
- Automating Cryptographic Code Generation
- Automating reinforcement learning architecture design for…
- Automating Software Documentation with n8n and Large…
- AutoPhase: Compiler Phase-Ordering for HLS with Deep…
- AV-AFL: A Vulnerability Detection Fuzzing Approach by…
- Backend Bug Finder — a platform for effective compiler…
- Backward Graph Construction and Lowering in DL Compiler…
- Balancing Security and Correctness in Code Generation: An…
- Bansal and Aiken 2006
- Basili and Weiss 1984
- Batch Me If You Can: Coverage-Guided RPKI Fuzzing at Scale
- Benchmarking Cross-Language Code Smell Detection with…
- Benchmarking WebAssembly for Embedded Systems
- Berger and colleagues 2002
- Beyond code coverage and#x2014; An approach for test…
- Beyond Coverage: Automatic Test Suite Augmentation for…
- Beyond Functional Correctness: An Empirical Evaluation of…
- Beyond Single Code Changes: An Empirical Study of…
- Beyond the C: Retargetable Decompilation using Neural…
- Bicriteria Distributed Submodular Maximization in a Few…
- Big code != big vocabulary
- Binary Similarity Detection Based on Intermediate…
- Binary–Source Code Matching Based on Decompilation…
- BIRD: A Binary Intermediate Representation for Formally…
- Birnbaum 1968
- Blackburn and colleagues 2006
- Blackburn and colleagues 2016
- Blanchet and colleagues 2003
- Blending Shapley values for feature ranking in machine…
- Blockchain-Based Software Effort Estimation: An Empirical…
- Boehm 1988
- Bohm and Jacopini 1966
- Boosting Compiler Testing by Injecting Real-World Code
- Boosting Compiler Testing via Compiler Optimization…
- Boosting component-based synthesis with control structure…
- BoostPolyGlot: A Structured IR Generation-Based Fuzz…
- BOSON - Application-Specific Instruction Set Processor…
- Bounded Abstract Interpretation
- Brack: A Verified Compiler for Scheme via CakeML
- Branch Use in Practice: A Large-Scale Empirical Study of…
- Bratter: An Instruction Set Extension for Forward…
- Braun and colleagues 2013
- Breaking bad? Semantic versioning and impact of breaking…
- Breaking the Vendor Lock
- BREWasm: A General Static Binary Rewriting Framework for…
- Bridging efficacy and efficiency: Innovations in Shapley…
- Bringing Binary Exploitation at Port 80: Understanding C…
- Bringing Together Cross-ISA Checkpoint/Restoration and…
- Buchbinder and colleagues 2015
- Buddhist studies on Seokbosangjeol and tasks ahead…
- Bug Characteristics in Blockchain Systems: A Large-Scale…

- Bug characterization in machine learning-based systems
- Bug numbers matter: An empirical study of effort-aware…
- Bug Propagation through Code Cloning: An Empirical Study
- Building a domain-specific compiler for emerging…
- Building Intelligent Integrated Development Environment…
- Building SSA in a Compiler for PHP
- Business-Driven Technical Debt Prioritization
- Bytecode-to-C Ahead-of-Time Compilation for Android…
- Caffeine: CoArray Fortran Framework of Efficient…
- CAGFuzz: Coverage-Guided Adversarial Generative Fuzzing…
- Calculation of worst-case execution time for multicore…
- Calibro: Compilation-Assisted Linking-Time Binary Code…
- Can AI Fix Buggy Code? Exploring the Use of Large…
- Can Large Language Model Detect Plagiarism in Source Code?
- Can LLMs Generate Higher Quality Code Than Humans? An…
- Can reactive synthesis and syntax-guided synthesis be…
- Can reactive synthesis and syntax-guided synthesis be…
- Can test cases generated by large language models…
- Can We Classify Flaky Tests Using Only Test Code? an…
- Can We Enhance Bug Report Quality Using LLMs?: An…
- CAnDL: a domain specific language for compiler analysis
- Cape: compiler-aided program transformation for HTM-based…
- CARL: Compiler Assigned Reference Leasing
- CASTL: A Composable Source Code Query Language for…
- Castro, Gomez and Tejada 2009
- CatchFuzz: Reliable active anti-fuzzing techniques…
- Causality-Aided Evaluation and Explanation of Large…
- CBGF: Callback Coverage Guided Fuzzing
- CCEyes: An Effective Tool for Code Clone Detection on…
- Certified Abstract Interpretation with Pretty-Big-Step…
- Certified abstract machines for skeletal semantics
- Chaitin 1982
- Challenges of Multilingual Program Specification and…
- Characteristics of Useful Code Reviews: An Empirical…
- Characterizing and Detecting WebAssembly Runtime Bugs
- Characterizing Dynamic Memory Behavior in WebAssembly…
- Characterizing logging practices in Java-based open…
- Characterizing Technical Debt in Evolving Open-source…
- Chariot: Compiler-Aware Heterogeneous Graph…
- Checked coverage for test suite reduction
- Checking correctness of code generator architecture…
- CHEHAB: Automatic Compiler Code Optimization for Fully…
- Chen and colleagues 2016
- CHERI Performance Enhancement for a Bytecode Interpreter
- Chinese Generation and Security Index Evaluation Based on…
- Chvatal 1979
- CirC: Compiler infrastructure for proof systems, software…
- CIRE: LLVM Analysis for Floating-Point Rounding Error…
- CKTI: A Domain-Specific Compiler for Lowering CUDA…
- CLACC: Translating OpenACC to OpenMP in Clang
- ClangJIT: Enhancing C++ with Just-in-Time Compilation
- Class-based query-optimization for minimizing worst-case…
- Classifying Code Comments in Java Open-Source Software…
- CLIK on PLCs! Attacking Control Logic with Decompilation…

- Clopper and Pearson 1934
- Clustering-Based Selection for the Exploration of…
- Co-evolution of Infrastructure and Source Code - An…
- Code coverage and test suite effectiveness: Empirical…
- Code Generation Techniques in Compiler Design: Conceptual…
- Code Ownership and Software Quality: A Replication Study
- Code review effectiveness: an empirical study on selected…
- CodeJudge: Evaluating Code Generation with Large Language…
- Codes for machine learning and Shapley value analysis
- CODO: An Automated Compiler for Comprehensive Dataflow…
- Collaboration Formation and Profit Sharing Between…
- Combining Code and Requirements Coverage with Execution…
- Combining Forward and Backward Abstract Interpretation of…
- Combining Large Language Models with Static Analyzers for…
- Combining loop unrolling strategies and code predication…
- Combining Program Analysis and Statistical Language Model…
- Combining Static Code Analysis and Large Language Models…
- CombRewriter: Enabling Combinational Logic Simplification…
- Come for syntax, stay for speed, write secure code: an…
- Common Bug-Fix Patterns: A Large-Scale Observational Study
- Communications Signal Processing Using RISC-V Vector…
- Compact native code generation for dynamic languages on…
- Comparative study of machine learning test case…
- Comparing Mutation Testing at the Levels of Source Code…
- Compatible Branch Coverage Driven Symbolic Execution for…
- CompCert: A Journey through the Landscape of Mechanized…
- CompCertS: A Memory-Aware Verified C Compiler Using a…
- Compilation for Operation Execution Time Variability
- Compilation Optimization Methods in the Customization of…
- Compile-Time Analysis of Compiler Frameworks for Query…
- Compiler and language design for quantum computing…
- Compiler and runtime support for continuation marks
- Compiler auto-vectorization of matrix multiplication…
- Compiler bug isolation via effective witness test program…
- Compiler Bug Isolation via Enhanced Test Program Mutation
- Compiler Module of Abstract Machine Code for Formal…
- Compiler Optimization for Quantum Computing Using…
- Compiler optimization for scientific computation in C/C++
- Compiler Optimization Parameter Selection Method Based on…
- Compiler Optimization Testing Based on…
- Compiler Support for Sparse Tensor Computations in MLIR
- Compiler Techniques for Efficient MATLAB to OpenCL Code…
- Compiler Test-Program Generation via Memoized…
- Compiler Testing with Relaxed Memory Models
- Compiler-agnostic Translation Validation
- Compiler-aided Type Tracking for Correctness Checking of…
- Compiler-ASR: Bridging the IR-to-Assembly Gap for…
- Compiler-Assisted Instruction Fusion
- Compiler-Assisted Loop Hardening Against Fault Attacks
- Compiler-Assisted Optimization Using Neural Code…
- Compiler-Assisted Selection of Hardware Acceleration…
- Compiler-Based Memory Encryption for Machine Learning on…
- Compiler-Directed Power Management for Superscalars
- Compiler-Like Code Generation for fUML: Reducing Overhead…

- Compiler-R1: Towards Agentic Compiler Auto-tuning with…
- Compiler-Runtime Co-operative Chain of Verification for…
- Compiler-support for Critical Data Persistence in NVM
- CompilerDream: Learning a Compiler World Model for…
- CompilerGPT: Leveraging Large Language Models for…
- Compiling Linear Algebra Workloads from C to Quantum…
- Completeness in Static Analysis by Abstract…
- Component-based specification, design and verification of…
- Comprehending Test Code: An Empirical Study
- Compression Optimization For Automatic Verification of…
- Conclusion: Debugging Blazor WebAssembly
- Configurable Loop Shuffling via Instruction Set Extensions
- Conforti and Cornuejols 1984
- Continuous Debt Valuation Approach (CoDVA) for Technical…
- Continuous Integration and Visual GUI Testing: Benefits…
- Convex: A RISC-V Instruction Set Extension Scheme for…
- Conway 1963
- Cooperative game amongst prefabricated building chain…
- COpt: A High Level Domain-Specific Language to Generate…
- CoreJIT: a Replication Package for Article…
- Corpus Compilation and Exploitation in Language…
- Correction to: A compiler framework for the reduction of…
- Correction to: Wait for it: identifying “On-Hold”…
- Correction: A Case Study in Formal Specification and…
- Correctness of Isabelle’s Cyclicity Checker
- Cost of Flaky Tests in Continuous Integration: An…
- Cost-effective learning-based strategies for test case…
- CoStar: a verified ALL(*) parser
- Could We Predict the Result of a Continuous Integration…
- Counterexample Guided Inductive Repair of Reactive…
- Counterexample Guided Knowledge Compilation for Boolean…
- Counterexample-guided approach to finding numerical…
- Counterexample-guided diagnosis
- Counterexample-Guided Inference of Modular Specifications
- Counterexample-guided simulation framework for formal…
- CoUpJava: A Dataset of Code Upgrade Histories in…
- Cousot and Cousot 1977
- Coverage-Guided Fuzzing for Plan-Based Robotics
- Coverage-Guided Learning-Assisted Grammar-Based Fuzzing
- Coverage-Guided Multi-Agent Harness Generation for Java…
- CPerfSmith: A Randomized C Program Generator for…
- CPP ’26 Artifact - Brack: A Verified Compiler for Scheme…
- Creating and Analyzing Source Code Repository Models - A…
- Creating and Running Tests with Xamarin Test Cloud
- CREF-Lang: A domain-specific language and compiler for…
- Cross-Language Binary-Source Code Matching Based on Rust…
- CS-Shapley: Class-Wise Shapley Values for Data Valuation…
- CsmithEdge: more effective compiler testing by handling…
- CTDip: a diversity-guided test program synthesis approach…
- Cuckoo search based hybrid models for improving the…
- CUDA Acceleration of Worst-Case Execution Time Analysis…
- CWAMR: REIMAGINING A CAPABILITYBASED WEBASSEMBLY RUNTIME…
- CWASI: A WebAssembly Runtime Shim for Inter-function…
- CWEval: Outcome-driven Evaluation on Functionality and…

- Cytron and colleagues 1991
- DAME: Runtime-compilation for data movement
- dAngr: Lifting Software Debugging to a Symbolic Level
- DASH: A Distributed and Parallelizable Algorithm for…
- Data Science and Empirical Software Engineering
- Data Shapley Valuation for Efficient Batch Active Learning
- Data Valuation Method Based on Federated Learning and…
- Data valuation using Shapley value in machine learning
- Data valuation with Leave-One-Out (LOO) test and Shapley…
- Data Valuation with Shapley-based Methods for Medical…
- Dataflow-based pruning for speeding up superoptimization
- Davidson and Fraser 1984
- De-Flake Your Tests : Automatically Locating Root Causes…
- Dean 1992
- DebtFree: minimizing labeling cost in self-admitted…
- DeCOS: Data-Efficient Reinforcement Learning for Compiler…
- Deep Neural Network Approach to Estimate Early Worst-Case…
- DeepCrime: from Real Faults to Mutation Testing Tool for…
- DeepCrime: mutation testing of deep learning systems…
- DeepOrder: Deep Learning for Test Case Prioritization in…
- DeepSurveySim: Simulation Software and Benchmark…
- DeepUIFuzz: A Guided Fuzzing Strategy for Testing UI…
- Delta Debugging Type Errors with a Blackbox Compiler
- DeMillo and colleagues 1978
- Demystifying the challenges and benefits of analyzing…
- Deployment and Integration of Machine Learning Methods…
- DERIVATIVE-BASED SHAPLEY VALUE FOR GLOBAL SENSITIVITY…
- Design and Development of Bridge AI bid Program based on…
- Design and Implementation of a Compiler Supporting RISC-V…
- Design and Simulation Of 64-Bit Hybrid Processor…
- Design of an Application Specific Instruction Set…
- Design of automatic aircraft parking system module…
- Design of Cyanobyte: An Intermediate Representation to…
- Design of DMS-RRIP replacement algorithm for L1-cache of…
- Design of RISC Processor with IEEE754 Standard…
- Design of RLWE Cryptoprocessor Based on…
- Design Space Exploration of RISC-V Vector Extension…
- Design, development and testing of a 16-bit reduced…
- Designing quantum chemistry algorithms with just-in-time…
- Designing RISC-V Instruction Set Extensions for…
- DESIL: Detecting Silent Bugs in MLIR Compiler…
- Detecting Compiler-Introduced Security Bugs via IR…
- Detecting Optimizing Compiler Bugs via History-Driven…
- Detecting WebAssembly Runtime Bugs With Grammar-Guided…
- Determining Worst-Case Execution Time Bounds for…
- Deterministic Algorithm and Faster Algorithm for…
- Deterministic Algorithm and Faster Algorithm for…
- Deterministic streaming algorithms for non-monotone…
- DEVELOPING SITUATIONAL CONDITIONS AND PROGRAM CODES FOR…
- Development effort estimation in free/open source…
- Development of a Code Generation Support System in…
- Development of RISC-V Based Soft-core Processor with…
- dewolf: Improving Decompilation by leveraging User Surveys
- Differential Fuzzing Go Compilers using LLMs: A…

- Diffy: Data-Driven Bug Finding for Configurations
- Discretized optimization algorithms for finding the…
- DISTAL: the distributed tensor algebra compiler
- Distinguishability-Guided Test Program Generation for…
- Distributed Attack-Robust Submodular Maximization for…
- Distributed matroid-constrained submodular maximization…
- Distributed Maximization of Submodular and Approximately…
- Distributed strategy selection: A submodular set function…
- Distributed Submodular Maximization on Partition Matroids…
- Distributed Submodular Maximization with Bounded…
- Distributed Submodular Maximization with Parallel…
- Distributed submodular maximization: trading performance…
- Distributionally robust optimization with…
- Do Automatically Generated Unit Tests Find Real Faults?…
- Do you have space for dessert? a verified space cost…
- Dockerfile Changes in Practice: A Large-Scale Empirical…
- Does Code Review Promote Conformance? A Study of…
- Does Representation Matter? Evaluating IRs for LLM-based…
- Domain specific compiler for coordinated signal…
- Dominance-based duplication simulation (DBDS): code…
- DRank: A semi-automated requirements prioritization…
- DRUT: An Efficient Turbo Boost Solution via Load…
- DU-Shapley: A Shapley Value Proxy for Efficient Dataset…
- Ductape: Optimizing Dynamically Typed Programs Using…
- Dyer and colleagues 2013
- Dynamic Test Case Prioritization and Selection for…
- Dynamic valuation of data assets via multi-agent…
- eCC++ : A Compiler Construction Framework for Embedded…
- Effective fault localization of automotive Simulink…
- Effective Performance Modeling and Domain-Specific…
- Effective Test Suite Optimization for Improving the…
- Efficient Compilation for Application Specific…
- Efficient Compiler Design for a Geometric Shape…
- Efficient Cross-Level Processor Verification using…
- Efficient program optimization through knowledge-enhanced…
- Efficient Program Tracing and Monitoring Through Power…
- Efficient Shapley performance attribution for…
- Efficient Shapley Value Driven Federated Learning System…
- Efficient Support of the Scan Vector Model for RISC-V…
- Efficient TinyML Inference on a Fault-Tolerant RISC-V SoC…
- Efficient Virtual Machine Allocation Technique Based on…
- Efficient Worst-Case Execution Time Analysis of Dynamic…
- Effort-optimized, accuracy-driven labelling and…
- Eight-Bit Vector SoftFloat Extension for the RISC-V Spike…
- Elbaum and colleagues 2002
- Empirical Application of Simulated Annealing Using…
- Empirical Assessment of Machine Learning Models for…
- Empirical evaluation of an entropy-based approach to…
- Empirical evaluation of fuzzy analogy for software…
- Empirical Evaluation of Mimic Software Project Data Sets…
- Empirical evaluation of tools for hairy requirements…
- EMPIRICAL INVESTIGATION OF CLOUD, GRID AND VIRTUALIZATION…
- Empirical Research for Self-Admitted Technical Debt…
- Empirical Software Engineering Experimentation with Human…

- Empirical Study about Class Change Proneness Prediction...
- Empirical study of correlation between mutation score and...
- Empirical Study of Restarted and Flaky Builds on Travis CI
- Empirical Study of Software Test Suite Evolution
- Empirical study on developer factors affecting tossing...
- Empirical Study on Real-time System Modeling and Code...
- Empirical Study on Software Bug Prediction
- Enabling Automatic Compiler-Driven Vectorization of...
- Enabling Fine-Grained Incremental Builds by Making...
- Energy-Based Model for Accurate Estimation of Shapley...
- Enhanced compiler bug isolation via memoized search
- Enhancing Compiler Design for Machine Learning Workflows...
- Enhancing formal specification and verification of...
- Enhancing LLVM Optimizations for Linear Recurrence...
- Enhancing Python Compiler Error Messages via Stack
- Enhancing Translation Validation of Compiler...
- Enriching Compiler Testing with Real Program from Bug...
- Enriching Source Code with Contextual Data for Code...
- EnStack: An Ensemble Stacking Framework of Large Language...
- Ensuring Reliability in Self-Adaptive Systems: A...
- Entropy-based Framework Dealing with Error in Software...
- EPF: An Evolutionary, Protocol-Aware, and Coverage-Guided...
- Equivalence Class Testing
- Erratum to: Formal Description Techniques and Protocol...
- Erratum: Weighted shapley value: a cooperative game...
- Error-tolerant processors: Formal specification and...
- Ertl 1999
- Esary and Proschan 1963
- Evaluating and optimising compiler code generation for...
- Evaluating classifiers in SE research: the ECSER pipeline...
- Evaluating ensemble imputation in software effort...
- Evaluating Machine Learning-Based Test Case...
- Evaluating Pred(p) and standardized accuracy criteria in...
- Evaluating RISC-V Vector Instruction Set Architecture...
- Evaluating the agreement among technical debt measurement...
- Evaluation and Benefit Modeling of Auto-Vectorization...
- Evaluation of Context-Aware Language Models and Experts...
- Evaluation of Coverage Metrics for Assessing Test Suite...
- EvoAPR: Enhancing Large Language Models for Automatic...
- Evolving Exact Decompilation
- Exact and approximate methods for proving unrealizability...
- Exact Worst-Case Execution-Time Analysis for Implicit...
- Examining ownership models in software teams
- Examining the impact of bias mitigation algorithms on the...
- Excluding code from test coverage: practices...
- Executable Semantics for the Formal Specification and...
- Experience Report: Security Vulnerability Profiles of...
- Experiences Building an MLIR-Based SYCL Compiler
- Experiences from Adjusting Industrial Software for...
- Explainable Test Case Prioritization in Continuous...
- Explaining 3D Object Detection Through Shapley...
- Exploiting Booster Pass Chain for Compiler Phase Ordering
- Exploring Compiler Optimization Opportunities for the...
- Exploring Generalizable Automated Program Repair With...

- Exploring Instruction Set Extension Emulation for…
- Exploring Technical Debt in Security Questions on Stack…
- Exploring Test Suite Diversification and Code Coverage in…
- Exploring the Effectiveness of LLM based Test-driven…
- Exploring the RISC-V Vector Extension for the Classic…
- Exponentially Expanding the Phase-Ordering Search Space…
- Extended firm mutation testing: A cost reduction…
- Extending LLVM IR for DPC++ Matrix Support: A Case Study…
- Extracting Invariants from Conditional Branches for…
- Extreme mutation testing in practice: An industrial case…
- Facilitating CoDesign with Automatic Code Similarity…
- Fair and Efficient Alternatives to Shapley-based…
- Fast and accurate power estimation for…
- Fast and flexible instruction selection with constraints
- Fast Compiler Optimization Flag Selection
- Fast derivation of Shapley based feature importances…
- Fast Failure Erasure Encoding Using Just in Time…
- Fast Interpreter-Based Instruction Set Simulation for…
- Fast Shapley Value Approximation Through Machine Learning…
- Fast Template-Based Code Generation for MLIR
- Faster mutation analysis via equivalence modulo states
- Fault detection effectiveness of source test case…
- Feige 1998
- Fenton and Ohlsson 2000
- Ferdinand and Wilhelm 1999
- Ferrante and colleagues 1987
- FIFO Cache Analysis for WCET Estimation
- Finding Bugs in MLIR Compiler Infrastructure via Lowering…
- Finding Compiler Bugs through Cross-Language Code…
- Finding deep compiler bugs via guided stochastic program…
- Finding Good Compiler Optimization Sets - A Case-based…
- Finding Missed Code Size Optimizations in Compilers using…
- Finding missed compiler optimizations by differential…
- Finding Unstable Code via Compiler-Driven Differential…
- Fine-Grained Coverage-Based Fuzzing
- Finetune-Then-Merge: Democratizing Large Language Model…
- FitM: Binary-Only Coverage-Guided Fuzzing for Stateful…
- Fixed-size video summarization over streaming data via…
- FIXME: synchronize with database! An empirical study of…
- Flacc: Towards OpenACC support for Fortran in the LLVM…
- FlakeSync: Automatically Repairing Async Flaky Tests
- Flaws of Quantification Method as applied to Software…
- Flexible and Efficient Implementation of CRYSTALS-KYBER…
- Flexible Runtime Reconfigurable Computing Overlay…
- Floating-point Semantics of Analyzed Programs
- Floating-Point Usage on GitHub: A Large-Scale Study of…
- Flower Pollination Algorithm for Software Effort…
- FlowPix: Accelerating Image Processing Pipelines on an…
- Formal Certification of Non-interferent Android Bytecode…
- Formal Semantics of Runtime Monitoring, Verification…
- Formal specification and verification
- Formal Specification and Verification of JDK’s Identity…
- Formal Specification and Verification of MQTT Protocol in…
- Formal Specification and Verification of MQTT Protocol…

- Formal Specification and Verification of Security…
- Formal Specification and Verification of Self-Adaptive…
- Formal Specification and Verification of Smart Contracts
- Formal Specification and Verification of Smart…
- Formal Specification Technique in Smart Contract…
- Formal verification of ABAP by Z specification
- Formal Verification of SUBLEQ Microcode implementing the…
- Formal Verification Platform as a Service: WebAssembly…
- FormalEval: A Method for Automatic Evaluation of Code…
- FormalGym: Deep Reinforcement Learning Agent Based Formal…
- Formalising Sharkovsky's Theorem (Proof Pearl)
- Formally Verified Native Code Generation in an Effectful…
- Formally verified speculation and deoptimization in a JIT…
- Formally Verifying Optimizations with Block Simulations
- ForMAt: Formal Verification of Scalable Multiply and…
- FOX: Coverage-guided Fuzzing as Online Stochastic Control
- FPGA-Accelerated Neural Network Inference via a…
- FPGA-Based ORB Accelerator: Effects of Compiler…
- FPGA-based SHA-3 acceleration on a 32-bit processor via…
- Fractional Artificial Bee Chicken Swarm Optimization…
- FrAngel: component-based synthesis with control structures
- Fraser, Hanson and Proebsting 1992
- FreeWavm: Enhanced WebAssembly Runtime Fuzzing Guided by…
- From Android Bug Reports to Android Bug Handling Process
- From Array Domains to Abstract Interpretation Under…
- From Bug Reports to Workarounds: The Real-World Impact of…
- From Natural Language to Interpretable Code: Automated…
- From SMT to ASP: Solver-Based Approaches to Solving…
- From Source to Bytecode: How.py Becomes.pyc
- Full-Speed Fuzzing: Reducing Fuzzing Overhead through…
- Fully Automatic Compiler Retargeting and CV-X-IF Hardware…
- Fully integrating the Flang Fortran compiler with…
- Function/Kernel Vectorization via Loop Vectorizer
- Functional Representations of SSA
- Functional Validation of the RISC-V Unlimited Vector…
- Furina: A Light-weight WebAssembly Runtime for ICS
- Fuse: A Reproducible, Extendable, Internet-Scale Corpus…
- Fuzzing Command-line Interface by Edge Coverage Guided…
- Fuzzing Deep Learning Compilers with HirGen
- Fuzzing guided by context-sensitive branch coverage
- Fuzzing JavaScript Interpreters with Coverage-Guided…
- Fuzzing JavaScript JIT compilers with a high-quality…
- Fuzzing MLIR Compiler Infrastructure via Operation…
- Fuzzy-Graph Contrastive Test Case Prioritization…
- Fuzzy_MoSCoW: A fuzzy based MoSCoW method for the…
- FWHT-RVV: A RISC-V vector processor with FWHT instruction…
- Generating Effective Test Suite for Multiparameter…
- Generating Software Architecture Description from Source…
- Genetic programming for feature model synthesis: a…
- GeoAutoModuler: a knowledge–enhanced large language model…
- Geographic diversity in public code contributions
- GeoIR-Compiler: A Geospatial Intermediate Representation…
- Georges and colleagues 2007
- GHALogs: Large-Scale Dataset of GitHub Actions Runs

- git2net - Mining Time-Stamped Co-Editing Networks from…
- GitEvo: Code Evolution Analysis for Git Repositories
- Glanville and Graham 1978
- Global vs. local models for cross-project defect…
- Gradual Soundness: Lessons from Static Python
- Grammar Filtering for Syntax-Guided Synthesis
- Grammar-Aware Coverage-Guided Fuzzing with Grammarinator…
- GrammLLM: Grammar-Guided LLM Test Generation for Compiler…
- Graph-Based Statistical Language Model for Code
- GrayC: Greybox Fuzzing of Compilers and Analysers for C
- Greedily Excluding Algorithm for Submodular Maximization
- Greedy algorithm for maximization of semi-monotone…
- Greedy+Singleton: An efficient approximation algorithm…
- GreenSource: A Large-Scale Collection of Android Code…
- Grossman and colleagues 2002
- Guest editorial: special issue on empirical software…
- Gustafson 1988
- HackRep: A Large-Scale Dataset of GitHub Hackathon…
- Handling Environments in a Nested Relational Algebra with…
- Hanley and Lippman-Hand 1983
- Hanson 1990
- Hardware implementation of a SHA-3 application-specific…
- Hardware/Software Co-Analysis for Worst Case Execution…
- Harnessing Large Language Models for Curated Code Reviews
- Harrold, Gupta and Soffa 1993
- Haskell Compiler Testing Automation Based on…
- HERTI: A Reinforcement Learning-Augmented System for…
- Heterogeneous Ensemble Model to Optimize Software Effort…
- Heterogeneous Graph Data Valuation: A Shapley Value-based…
- Heterogeneous Graph Neural Networks for Software Effort…
- Hexcute: A Compiler Framework for Automating Layout…
- HFuzz: Havoc Mode Guided Fuzzing
- HHVM JIT: a profile-guided, region-based compiler for PHP…
- High Performance Instruction-Data Level Parallelism Based…
- High-Performance Web Frontend Using WebAssembly
- High-Precision Evaluation of Both Static and Dynamic…
- High-coverage metamorphic testing of concurrency support…
- Hikami: A Lightweight Hypervisor for Emulating RISC-V…
- HiPEAC compilation architecture
- History-driven Compiler Fuzzing via Assembling and…
- History-Guided Configuration Diversification for Compiler…
- Hodge decomposition and the Shapley value of a…
- Holistic evaluation of LLM-Based Code Generation
- How accessibility affects other quality attributes of…
- How AI Coding Agents Modify Code: A Large-Scale Study of…
- How Are Discussions Associated with Bug Reworking?
- How challenging it is to identify real code authors: an…
- How Closely are Common Mutation Operators Coupled to Real…
- How do Community Smells Influence Self-Admitted Technical…
- How Do Deep Learning Faults Affect AI-Enabled…
- How do open source software (OSS) developers practice and…
- How do software development teams manage technical debt?…
- How does combinatorial testing perform in the real world…
- How Does Test Code Differ from Production Code in Terms…

- How effective are mutation testing tools? An empirical...
- How Effective Is Coverage-Guided Fuzzing to Test Deep...
- How far we have come: testing decompilation correctness...
- How to Better Distinguish Security Bug Reports (Using...
- How to Evaluate the Productivity of Software Ecosystem: A...
- How to improve deep learning for software analytics
- HSS cluster-based direct solver for acoustic wave equation
- Hybrid Approach based on Multi-agent System and Fuzzy...
- Hybrid Automated Program Repair by Combining Large...
- Hybrid Equivalence/Non-Equivalence Testing
- Hybrid Execution: Combining Ahead-of-Time and...
- Hybrid Metaheuristic Technique for Optimization of...
- Hybrid Model for Improving the Accuracy of Software...
- Hybrid WebAssembly-Container Orchestration in Embedded...
- HybridServe: Adaptive WebAssembly-Container Runtime...
- HyperAST: Incrementally Mining Large Source Code...
- Hyperchaining Optimizations for an LLVM-Based Binary...
- Hyperhierarchy of Semantics - A Formal Framework for...
- Identification and prioritization of SLR search tool...
- Identifying and Mitigating Flaky Tests in JavaScript
- Identifying Compiler and Optimization Level in Binary...
- Identifying Compiler and Optimization Options from Binary...
- Identifying Key Requirements Prioritization Criteria for...
- Identifying self-admitted technical debt in issue...
- Identifying self-admitted technical debt in open source...
- Impact of Stack Overflow Code Snippets on Software...
- Impact of Static and Dynamic Coverage on Test-Case...
- Implement Machine Learning to Schedule Instructions to...
- Implementation and Performance Evaluation of Omni Compiler
- Implementation and Reliability Evaluation of a RISC-V...
- Implementation of Application Specific Instruction set...
- IMPLEMENTATION OF GENETIC ALGORITHM IN COLLEGE SCHEDULING...
- Implementing an Application-Specific Instruction-Set...
- Improve Model Testing by Integrating Bounded Model...
- Improved Ahead-of-time Compilation of Stack-based JVM...
- Improved Assistance for Interactive Proof (Keynote)
- Improved Circuit Compilation for Hybrid MPC via Compiler...
- Improved Prediction of Total Energy Consumption and...
- Improving Code Completion by Solving Data Inconsistencies...
- Improving Estimation Accuracy Prediction of Software...
- Improving Software Requirements Prioritization through...
- Improving the accuracy and stability of privacy-aware...
- Improving the Accuracy of Batik Classification using Deep...
- Inductor-TV: Formal Methods for the Pytorch Compiler
- Industrial adoption of machine learning techniques for...
- Industrial Application of Deep Learning based Fault...
- Industrial Scale Passive Testing with T-EARS
- Inferring test models from user bug reports using...
- Inozemtseva and Holmes 2014
- Input perturbation robustness for software effort...
- INR-Arch: A Dataflow Architecture and Compiler for...
- Inside Bug Report Templates: An Empirical Study on Bug...
- INSTRIM: Lightweight Instrumentation for Coverage-guided...
- Instruction Code Selection

- Instruction set extension and hardware acceleration for…
- Instruction Set Optimization for FM-Type Digital Signal…
- InstructRepair: Instruct Large Language Models With Rich…
- Integrating a functional pattern-based IR into MLIR
- Integrating Large Language Models in Software Engineering…
- Integrating Shapley Value and Least Core Attribution for…
- Integrating Staleness and Shapley Value Consistency for…
- Integrating Tests into Your Builds
- Integration of a Real-Time CCSDS 410.0-B-32…
- Integration of Static Worst-Case Execution Time and Stack…
- Intelligent Power Grid Startup Scheme Based on Rule…
- Intelligent Test Case Prioritization: A Review of Machine…
- Interactive Programming for Microcontrollers by…
- Interleaving Large Language Models for Compiler Testing
- Intermediate Representations
- Intermediate-Code Generation
- Intermediate-Code Generation
- Interruptibility of software developers and its…
- IntraFuzz: Coverage-Guided Intra-Enclave Fuzzing for…
- Intrinsic Compilation Model to enhance Performance of…
- Introducing H, an Institution-Based Formal Specification…
- Introducing multi-level parallelism, at coarse, fine and…
- Introduction to Optimization
- Investigating Coverage Guided Fuzzing with Mutation…
- Investigating the Role of Formal Verification in Software…
- Investigations about replication of empirical studies in…
- Ioannidis 2005
- Is code coverage of performance tests related to source…
- Is dynamic compilation possible for embedded systems?
- Is this build failure related to my patch? An empirical…
- Is Your Code Generated by ChatGPT Really Correct?…
- ISA customization for application specific instruction…
- Java Source Code Vulnerability Detection Using Large…
- Javascript ahead-of-time compilation for embedded web…
- JavaScript Parallelizing Compiler for Exploiting…
- Jensen 1906
- Jia and Harman 2011
- JIT Compiler Security through Low-Cost RISC-V Extension
- JIT Leaks: Inducing Timing Side Channels through…
- JITfuzz: Coverage-guided Fuzzing for JVM Just-in-Time…
- Johnson 1974
- Jovanovic and Levy 1997
- Just and colleagues 2014
- Just-in-Time Compilation and Link-Time Optimization for…
- Just-In-Time Compilation for Verilog
- Just-In-Time Compilation on ARM—A Closer Look at…
- Just-In-Time Compiler System in Aspect-Oriented…
- Just-In-Time GPU Compilation for Interpreted Languages…
- Just-in-time scheduling in identical parallel machine…
- k-Submodular Maximization Under Individual Knapsack…
- Kang and colleagues 2018
- Karlsson and Ryan 1997
- Karp 1972
- KART – A Runtime Compilation Library for Improving HPC…

- Key Operator Vectorization for LeNet and ResNet Based on…
- Keynote talk I: Syntax-guided synthesis
- Khuller and colleagues 1999
- Kildall 1973
- Knowledge Graph Based Repository-Level Code Generation
- Knuth 1971
- Kumar and colleagues 2014
- LAGrad: Statically Optimized Differentiable Programming…
- Landi 1992
- Language models can prioritize patches for practical…
- Language usage analysis for EMF metamodels on GitHub
- LAPSE: Automatic, Formal Fault-Tolerant Correctness…
- Large Language Model Generation Safety: A Comprehensive…
- Large Language Model-Based Interactive Code Generation…
- Large Language Models for Automated Program Repair
- Large Language Models for Binary Decompilation…
- Large Language Models for Code Obfuscation Evaluation of…
- Large Language Models for Code Translation: An In-Depth…
- Large Language Models for Computer Programming Education…
- Large Language Models in Automated Repair of Haskell Type…
- Large Language Models Meet Automated Program Repair…
- Large-Scale Analysis of GitHub and CVEs to Determine…
- Large-scale analysis of the co-commit patterns of the…
- Large-Scale Manual Validation of Bugfixing Changes
- Lattner and Adve 2004
- LAYANAN CLOUD COMPUTING BERBASIS INFRASTRUCTURE AS A…
- Lazy Code Transformations in a Formally Verified Compiler
- Le and colleagues 2014
- Learning-based prioritization of test cases in continuous…
- LegoFuzz: Interleaving Large Language Models for Compiler…
- Leroy 2003
- Leroy 2009
- Lessons Learnt on Reproducibility in Machine Learning…
- Leveraging Compilation Statistics for Compiler Phase…
- Leveraging Compiler Intermediate Representation for…
- Leveraging explainable AI for requirements prioritization…
- Leveraging LLVM OpenMP GPU Offload Optimizations for…
- Leveraging Rough Sets for Enhanced Test Case…
- Li and Malik 1995
- License Usage and Changes: A Large-Scale Study of Java…
- License usage and changes: a large-scale study on gitHub
- LifeJacket: verifying precise floating-point…
- Lifting Assembly to Intermediate Representation
- Lifting Code Generation of Cardiac Physiology Simulation…
- Lifting proof-relevant unification to higher dimensions
- Light Shapley: Improving the Scalability of Equitable…
- Lightweight Cryptographic Instruction Set Extension on…
- Liu and Layland 1973
- LLHD: a multi-level intermediate representation for…
- LLM Compiler: Foundation Language Models for Compiler…
- LLM-assisted end-to-end binary decompilation: a…
- LLM-based Control Code Generation using Image Recognition
- LLM-VeriOpt: Verification-Guided Reinforcement Learning…
- LLVM and the Automatic Vectorization of Loops Invoking…

- LLVM Library for a Dedicated Processor Instruction Set –...
- LLVM parallel intermediate representation
- LLVM-based communication optimizations for PGAS programs
- LMFuzz: Program repair fuzzing based on large language...
- Load Balancing in Decoupled Look-ahead: A Do-It-Yourself...
- Localized Data Shapley: Accelerating Valuation for...
- Locating faults with program slicing: an empirical...
- LocSeq: Automated Localization for Compiler Optimization...
- Logic Gate Network Inference Acceleration with RISC-V...
- Logic Synthesis with Design Compiler
- Long-Term Evaluation of Technical Debt in Open-Source...
- Look for the Proof to Find the Program...
- Looking for Peace of Mind? Manage Your (Technical) Debt...
- Lopes and colleagues 2021
- Lost In Translation: Exposing Hidden Compiler...
- Lovasz 1983
- Low Level Source Code Vulnerability Detection Using...
- Low-Power Implementation of DSP Instruction Set Extension...
- Lowering Barriers to Application Development With...
- LTSV: Layered Type-Constrained Shapley Value for...
- Lumos: Performance Characterization of WebAssembly as a...
- Lund and Yannakakis 1994
- Lutsig: a verified Verilog compiler for verified circuit...
- Machine Learning Approaches for Authorship Attribution...
- Machine Learning Based Compiler Optimization Technique
- Machine Learning for Data Center Optimizations: Feature...
- Machine Learning for Test Case Prioritization in...
- Machine Learning in Compiler Optimization
- Machine learning models with distinct Shapley value...
- Machine Learning Regression Techniques for Test Case...
- Machine learning-based modelling, feature importance and...
- Machine Learning-based Test Case Prioritization using...
- Machine Learning-Driven GCC Loop Unrolling Optimization...
- Machine-checked Verification of Cognitive Agents
- Machine-Checked Verification of the Correctness and...
- Machine-checked ZKP for NP relations: Formally Verified...
- Machine-Code Generation
- Machine-Code Generation
- Making Time Observable: Compiler Correctness for...
- MALintent: Coverage Guided Intent Fuzzing Framework for...
- Many-core compiler fuzzing
- MarQSim: Reconciling Determinism and Randomness in...
- Massalin 1987
- Matlab to C Compilation Targeting Application Specific...
- MCBRank Method to Improve Software Requirements...
- Measuring affective states from technical debt
- Mechanising a Type-Safe Model of Multithreaded Java with...
- Mechanizing conventional SSA for a verified destruction...
- Meehl 1967
- Memory Simulations, Security and Optimization in a...
- Memory Utilization and Machine Learning Techniques for...
- Meta-analysis for families of experiments in software...
- Metacasanova: an optimized meta-compiler for...
- Metamodeling and Code Generation in the Hardware/Software...

- Metamorphic Relation Patterns for Metamorphic Testing…
- Metamorphic testing for (graphics) compilers
- Metamorphic Testing for Adobe Data Analytics Software
- Metamorphic testing in bioinformatics software
- Metamorphic Testing on the Continuum of Verification and…
- Microarchitecture Design and Benchmarking of Custom SHA-3…
- Mining performance regression inducing code changes in…
- Mining the modern code review repositories
- Mining the usage of reactive programming APIs
- Mitigating omitted variable bias in empirical software…
- MLIR-based code generation for GPU tensor cores
- MLIR-Based Homomorphic Encryption Compiler for GPU
- MLIR-to-CGRA: A Versatile MLIR-Based Compiler Framework…
- MLIR: A Panacea for ML Compiler Challenges?
- MLIR: Scaling Compiler Infrastructure for Domain Specific…
- MLIRSmith: Random Program Generation for Fuzzing MLIR…
- Model vs system level testing of autonomous driving…
- Model-Agnostic Empirical Evaluation of Test-Driven Prompt…
- Model-Driven Quantum Code Generation Using Large Language…
- Modeling and implementation of Common LISP functional…
- Modeling Sampling Workflows for Code Repositories
- Modeling Undefined Behaviour Semantics for Checking…
- ModelPlex: verified runtime validation of verified…
- Modular Component-Based Quantum Circuit Synthesis
- Modular SDN Compiler Design with Intermediate…
- Modular specification and verification of a…
- Modular translation validation of a full-sized…
- Moura and Ierusalimschy 2009
- MRU Cache Analysis for WCET Estimation
- Much more than a prediction: Expert-based software effort…
- Multi-Armed Bandit Test Case Prioritization in Continuous…
- Multi-faceted Code Smell Detection at Scale using…
- Multi-level physical hierarchy floorplanning using IC…
- Multi-modal program inference: a marriage of pre-trained…
- Multi-Pass Streaming Algorithms for Monotone Submodular…
- Multi-target Compiler for the Deployment of Machine…
- Multipass Streaming Algorithms for Regularized Submodular…
- Munafo and colleagues 2017
- Mutation analysis and its industrial applications
- Mutation Operators for Mutation Testing of Angular Web…
- Mutation Testing for Industrial Robotic Systems
- Mutation Testing in Continuous Integration: An…
- Mutation testing in practice using Ruby
- Mutation Testing of Deep Reinforcement Learning Based on…
- Mutation Testing of Programs for Industrial Robots
- muttest: Mutation Testing
- Mytkowicz and colleagues 2009
- Naturalness in Source Code Summarization. How Significant…
- Naturalness of Attention: Revisiting Attention in Code…
- Navigating the SIMD Optimization Maze: A Reinforcement…
- Necula 1997
- Necula and Lee 1998
- Negative results for software effort estimation
- Negativity in self-admitted technical debt: how sentiment…

- NemesisGuard: Mitigating interrupt latency side channel…
- Nemhauser and Wolsey 1978
- Nemhauser, Wolsey and Fisher 1978
- Neural Network-Based Test Case Prioritization in…
- Neural Networks based Software Development Effort…
- Neural-MCTS Test Prioritization for Smart Contract…
- Neuroevolutionary Compiler Control for Code Optimization
- Nonio — modular automatic compiler phase selection and…
- Nonlinear Reinforcement Learning-Based Dynamic Test Case…
- NOProbe: A NOP-Based Dynamic Binary Instrumentation…
- Not all bug reopens are negative: A case study on eclipse…
- Novice comprehension of Object-Oriented OO programs: An…
- Object Intersection Captures on Interactive Apps to Drive…
- Of Ahead Time: Evaluating Disassembly of Android Apps…
- Offline and Online Distributed Submodular Maximization…
- On a Machine-Checked Proof for Fraction Arithmetic over a…
- On distributed submodular maximization with limited…
- On factors that impact the relationship between code…
- On JavaScript Ahead-of-Time Compilation Performance…
- On the Benefits and Barriers When Adopting Software…
- On the Criticality of Probabilistic Worst-Case Execution…
- On the documentation of self-admitted technical debt in…
- On the Evaluation of Effort Estimation Models
- On the Interactions Between Value Prediction and Compiler…
- On the relationship between bug reports and queries for…
- On the Runtime and Energy Performance of WebAssembly: Is…
- On the use of contextual information for machine learning…
- On the use of static branch prediction to reduce the…
- On the value of a prioritization scheme for resolving…
- One-pass streaming algorithm for DR-submodular…
- One-pass streaming algorithm for monotone lattice…
- Online and Streaming Algorithms for Constrained…
- Oolong: A Baseband processor extension to the RISC-V ISA
- OP2-Clang: A Source-to-Source Translator Using Clang/LLVM…
- Open Source Software (OSS) for Big Data
- Open-Source Memory Compiler for Automatic RRAM Generation…
- Open-Source MLIR-Based Intermediate Representation for…
- OpenASIP 2.0: Co-Design Toolset for RISC-V…
- OpenMP GPU Offload in Flang and LLVM
- Operationalizing validity of empirical software…
- Operator-data type pair based execution environments…
- Opportunities and security risks of technical leverage: A…
- Optimasi Pemanfaatan Air Menggunakan Program Solver di…
- Optimising Bcrypt Parameters: Finding the Optimal Number…
- Optimization of production planning using integer linear…
- Optimization of Specific Instruction Set Processor for…
- Optimization, Machine Learning, and Fuzzy Logic
- Optimization-Aware Compiler-Level Event Profiling
- Optimization-Directed Compiler Fuzzing for Continuous…
- Optimizing Shapley Value for Client Valuation in…
- Optimizing Software Effort Estimation Accuracy with a…
- Optimizing Task Allocation in IT Project Management Using…
- Optimizing Tensor Computations: From Applications to…
- Optimizing test case prioritization using machine…

- Optimizing Test Case Prioritization With Meta Deep…
- Optimizing TinyEngine for the RISC-V Vector Extension
- ORMorpher: An Interactive Framework for ORM Translation…
- OSS License Identification at Scale: A Comprehensive…
- OSSGameBench: A Large-Scale Dataset of Development…
- Ostrand and Weyuker 2002
- Ostrand, Weyuker and Bell 2005
- OurRank: A Software Requirements Prioritization Method…
- Outer-Loop Auto-Vectorization for SIMD Architectures…
- Owen 1972
- P30 - Möbius-Shapley: Native Feature Attribution for…
- P4IRS: An intermediate representation and compiler for…
- Paddle-Mlir: A Compiler Based on MLIR for Accelerating…
- Parallel mutation testing for large scale systems
- Parallelizing Compiler Translation Validation Using…
- PARCOACH Extension for Static MPI Nonblocking and…
- Partitioning Composite Code Changes to Facilitate Code…
- Passenger Queue Simulation Analysis and Optimization at…
- Patterns of Code-to-Test Co-evolution for Automated Test…
- PEMBANGUNAN COMPILER DOMAIN SPECIFIC LANGUAGE SEBAGAI…
- PENERAPAN EIGENFACE UNTUK COMPUTER BASED TEST (CBT)…
- PENERAPAN PEMROSESAN PARALEL UNTUK MENGUJI WAKTU…
- PERANCANGAN PENGAMANAN SERVER SECARA OTOMATIS MENGGUNAKAN…
- Performance and Usability Implications of Multiplatform…
- Performance Evaluation of CNN using RISC-V Vector…
- PERFORMANCE EVALUATION OF THE UETRV-PCORE USING RISC-V…
- Performance Improvement of Kotlin Program in…
- Performance, Correctness, Exceptions: Pick Three
- Performant Binary Fuzzing without Source Code using…
- PfComp: A Verified Compiler for Packet Filtering…
- PHPIL: Fuzzing the PHP Interpreter with Custom Bytecode
- Pilsner: a compositionally verified compiler for a…
- PIMFlow: Compiler and Runtime Support for CNN Models on…
- Pnueli and colleagues 1998
- Polyhedral Compiler Technology in Collaboration with…
- PolyMorphous: An MLIR-Based Polyhedral Compiler with Loop…
- Poster Abstract: Towards Shapley Value based Security…
- Poster: BugOss: A Regression Bug Benchmark for Empirical…
- POSTER: Runtime Adaptations for Energy-Efficient VSLAM
- POSTER: Tango: An Optimizing Compiler for Just-In-Time…
- Potential of WebAssembly for Embedded Systems
- POWER: Program Option-Aware Fuzzer for High Bug Detection…
- Practical Examples of Timing Problems
- Practical Python FPGA Acceleration with Fast Just-In-Time…
- PredComp: Predicting Compiler Optimization Options with…
- Predicting Design Impactful Changes in Modern Code…
- Predicting Worst-Case Execution Time Trends in Long-Lived…
- Prediction accuracy measurements as a fitness function…
- Prediction of relatedness in stack overflow: deep…
- Preempting flaky tests via non-idempotent-outcome tests
- Preprocessing and Compilation
- Preventing duplicate bug reports by continuously querying…
- Prioritizing solution-oriented software requirements…
- PriorTD: A Method for Prioritization Technical Debt

- Privacy-Preserving Feature Valuation in Vertical…
- Profile Guided Optimization Transfer-Learning for…
- Profiling Developers Through the Lens of Technical Debt
- Program comprehension of domain-specific and…
- Program Reconditioning: Avoiding Undefined Behaviour When…
- Programming Assessment in E-Learning through Rule-Based…
- Programming GPUs with C++14 and Just-In-Time Compilation
- Programming Large Language Models
- Project productivity evaluation in early software effort…
- PromptTone: A Dataset for Evaluating Large Language Model…
- Proof pearl: Braun trees
- Propagating Large Language Models Programming Feedback
- Proposal of Scalable Vector Extension for Embedded RISC-V…
- Protean Compiler: An Agile Framework to Drive Fine-grain…
- Proteus: Portable Runtime Optimization of GPU Kernel…
- Prototype implementation of the OpenGL ES 2.0 shading…
- Proving the Coding Interview: A Benchmark for Formally…
- PRPA REPERCUSSIONS and IMPLICATIONS FOR REAL WORLD STUDY…
- PSO based optimization of worst-case execution time for…
- PureCake: A Verified Compiler for a Lazy Functional…
- Puschner and Burns 2000
- QEMI: A Quantum Software Stacks Testing Framework via…
- QJWasm: A lightweight runtime system for efficient…
- QScored: A Large Dataset of Code Smells and Quality…
- Quality assurance of bioinformatics software
- Quality Questions Need Quality Code: Classifying Code…
- Quantifying and characterizing clones of self-admitted…
- Quantum Circuit Synthesis from C via Multi-Level…
- Quantum Oracle Synthesis from HDL Designs via Multi Level…
- Quantum simulation with just-in-time compilation
- Query by example in large-scale code repositories
- Querying Big Source Code
- Rainfuzz: Reinforcement-Learning Driven Heat-Maps for…
- Random testing of C compilers based on test program…
- Randomized Composable Core-sets for Distributed…
- Ranking Relevant Tests for Order-Dependent Flaky Tests
- Ray and colleagues 2014
- RE-APR: Reasoning-Enhanced Automated Program Repair via…
- Real world projects, real faults: evaluating spectrum…
- Realistic assessment of software effort estimation models
- Really Embedding Domain-Specific Languages into C++
- ReAPR: Automatic program repair via retrieval-augmented…
- Rechecking Recheck Requests in Continuous Integration: An…
- Reconciling high-level optimizations and low-level code…
- Redefining prioritization
- Reducing labeling effort in architecture technical debt…
- Reducing the Compilation Time of Quantum Circuits Using…
- Reduction of Workflow Nets for Generalised Soundness…
- Reductive Analysis with Compiler-Guided Large Language…
- Redundancy Elimination
- Refactorings and Technical Debt in Docker Projects: An…
- Refinement of Parallel Algorithms Down to LLVM: Applied…
- Reflections on the Empirical Software Engineering journal
- ReFuzz: A Remedy for Saturation in Coverage-Guided Fuzzing

- Regehr and colleagues 2005
- Regression Test Suites Optimization for…
- Regularized two-stage submodular maximization under…
- Reimagining Studies’ Replication: A Validity-Driven…
- Reineke and colleagues 2007
- Reinforcement Learning and Data-Generation for…
- Reinforcement learning for automatic test case…
- Reinforcement learning for online testing of autonomous…
- Reinforcement Learning Reward Function for Test Case…
- Reinforcement Learning Strategies for Compiler…
- Reinforcement-Learning-Based Test Program Generation for…
- Relating Code Coverage, Mutation Score and Test Suite…
- Relational Solver for Java Generics Type System
- Relaxed Peephole Optimization: A Novel Compiler…
- Release synchronization in software ecosystems
- Reliability Test based on a Binomial Experiment for…
- Reliable Compilation Optimization Phase-ordering…
- Remgen: Remanufacturing a Random Program Generator for…
- Remote Just-in-Time Compilation for Dynamic Languages
- RepairBench: Leaderboard of Frontier Models for Program…
- Replacing conjectures by positive knowledge: Inferring…
- Replication of Empirical Studies in Software Engineering…
- Replication Package for Paper: LLHD: A Multi-level…
- Representation learning for coincidental correctness in…
- Reproducibility and credibility in empirical software…
- Reproducibility Practices of Software Engineering…
- Research of WebAssembly usage for high-performance code…
- Research on Compiler Optimization Technology Based on…
- Research on coverage-guided fuzzing technique based on…
- Research on Instruction Set Architecture of 40-Bit…
- Research on Program Automatic Repair Method Combining…
- Research on RISC-V-based Edge Convolution Acceleration…
- Resource Optimization based Virtual Machine Allocation…
- Resource-Aware Distributed Submodular Maximization: A…
- Resource-efficient RISC-V Vector Extension Architecture…
- Resources for Reproducibility of Experiments in Empirical…
- REST API Fuzzing by Coverage Level Guided Blackbox Testing
- Rethinking Correctness and Efficiency in AI-Assisted Code…
- RETRACTED ARTICLE: The smell of fear: on the relation…
- Retraction Note: Retraction note to: The smell of fear…
- RETRACTION: Study on Large-Scale Promotion of…
- Reusing the Optimized Code for JavaScript Ahead-of-Time…
- Revealing Compiler Heuristics Through Automated Discovery…
- Reverse Engineering of Obfuscated Lua Bytecode via…
- Reviewing Reproducibility in Software Engineering Research
- Revisiting Machine Learning based Test Case…
- Revisiting the building of past snapshots — a replication…
- Revisiting the reproducibility of empirical software…
- Revisiting Unnaturalness for Automated Program Repair in…
- Reward-Aware Shapley Compensation: a Probabilistic and…
- rFocal: Run ‘FOCAL’ Language Source Code
- RI-MAC: Optimising MAC Operation Using Custom RISC-V…
- Rice 1953
- Richards and colleagues 2010

- Ripple Shapley: Data Influence Attribution in One…
- RISC-TAE: Instruction Set Extension for Transformer Model…
- RISC-V Instruction Set Architecture Extensions: A Survey
- RISC-V SIMD Instructions - Vector Extension (Load/Store)
- RISC-VTF: RISC-V Based Extended Instruction Set for…
- RISC-V Processor Hardware Modelling with Custom…
- Risk Attribution Using the Shapley Value: Methodology and…
- RLGFuzz: Reinforcement Learning Guided Fuzzing with…
- Robust Practical Binary Optimization at Run-time using…
- Role-Aware Intelligent Agent Framework for Enhanced Code…
- Rosenthal 1979
- Rothermel and colleagues 2001
- Rothermel and Harrold 1997
- RTFM: Towards Understanding Source Code using Natural…
- Rule modeling for automatic verification of RDC-50…
- Running Large Language Models at Scale for Mining…
- Runtime Metric Analysis in NoSQL Database Performance…
- Runtime prediction model for high performance computing…
- Runtime prediction of high-performance computing jobs…
- Runtime Value Numbering: A Profiling Technique to…
- Rust-twins: Automatic Rust Compiler Testing through…
- Rustlantis: Randomized Differential Testing of the Rust…
- RustSmith: Random Differential Compiler Testing for Rust
- RV-IR: An MLIR-Based Architecture-Aware Intermediate…
- RVCE-FAL: A RISC-V Scalar-Vector Custom Extension for…
- RVVRadar: A Framework for Supporting the Programmer in…
- S-compiler: A code vulnerability detection method
- SAGE-HLS: Syntax-Aware AST-Guided LLM for High-Level…
- SAGE: A Compiler-assisted Reinforcement Learning-based…
- Same Coverage, Less Bloat: Accelerating Binary-only…
- Scalable and Accurate Test Case Prioritization in…
- Scalable SMT Sampling for Floating-Point Formulas via…
- SCALE-Ahead-Of-Time Compilation of CUDA for AMD GPUs
- ScaleHLS: A New Scalable High-Level Synthesis Framework…
- Schkufza, Sharma and Aiken 2013
- Seamless Self-Healing in WebAssembly Container…
- SecSwift, a Compiler-Based Framework for Software…
- Securing a compiler transformation
- Security Requirements Prioritization Techniques: A Survey…
- Self-Admitted Technical Debt and comments' polarity: an…
- Self-admitted technical debt practices: a comparison…
- Self-Hosted WebAssembly Runtime for Runtime-Neutral…
- Semantic Coverage: Measuring Test Suite Effectiveness
- Semantic-Type-Guided Bug Finding
- Semantic-aware two-phase test case prioritization for…
- Semi-streaming Algorithms for Submodular Function…
- Sequence submodular maximization meets streaming
- Session details: Session 4A: Compiler Correctness
- Sethi and Ullman 1970
- Sewell and colleagues 2013
- SHA-3 Instruction Set Extension for A 32-bit RISC…
- Shapley 1953
- SHapley Additive exPlanations (SHAP) for Efficient…
- Shapley Patch Valuation Method for Histopathological…

- Shapley Value Approximation with Divisive Clustering
- Shapley value in machine learning modeling: optimizing…
- Shapley Value is an Equitable Metric for Data Valuation
- Shapley Value of a Cooperative Game with Fuzzy Set of…
- Shapley value-based data valuation for machine learning…
- Shapley Value-Based Feature Attribution for Data Masking
- Shapley value: from cooperative game to explainable…
- Shapley-Based Data Valuation for Weighted k-Nearest…
- Shapley-Based Data Valuation Method for the Machine…
- Shapley-Based Data Valuation with Mutual Information: A…
- Shapley-Value Based Feature Attribution for…
- Shapley-Value Data Valuation for Semi-supervised Learning
- Shepherd: High-Precision Coverage Inference for…
- Silent Compiler Bug De-duplication via Three-Dimensional…
- Similarity-Aware Architecture/Compiler Co-Designed…
- Simmons, Nelson and Simonsohn 2011
- Simulink Model Static Analysis Results based on Abstract…
- Skeletal program enumeration for rigorous compiler testing
- SLAMPA: Recommending Code Snippets with Statistical…
- Slavik 1997
- SMT-Based Translation Validation for Machine Learning…
- Software Development Effort Estimation Using Random…
- Software effort estimation accuracy prediction of machine…
- Software effort estimation using validated accuracy…
- Software effort estimation using validated accuracy…
- Software UART: A Use Case for VSCPU Worst-Case Execution…
- Solsmith: Solidity Random Program Generator for Compiler…
- Solution to CAD Designer Effort Estimation based on…
- Solver-based gradual type migration
- Source Code Features and their Dependencies: An…
- Source Code Implied Language Structure Abstraction…
- Source code obfuscation with genetic algorithms using…
- Source Code Plagiarism Detection with Pre-Trained Model…
- Special Issue on Syntax-Guided Synthesis Preface
- Special Session: Reliability and Performance Evaluation…
- Specification and automatic verification of trust-based…
- Specification and Formal Verification of…
- Specification and Verification of a Formal Model for a…
- Specification-guided component-based synthesis from…
- SpinalFuzz: Coverage-Guided Fuzzing for SpinalHDL Designs
- SPNC: An Open-Source MLIR-Based Compiler for Fast…
- SPRoC: Semantics-Preserving Mutations for Robustness…
- SRTuner: Effective Compiler Optimization Customization by…
- SSA Form and Code Generation
- SSFuzz: Synthesizing and scheduling bug-triggering code…
- Stack-based static WebAssembly binary slicing and…
- Startups and Technical Debt: Managing Technical Debt with…
- State of mutation testing at google
- Statement frequency coverage: A code coverage criterion…
- Static analysis by abstract interpretation against data…
- Static Analysis by Abstract Interpretation of the…
- Static Analysis of Endian Portability by Abstract…
- Static Analysis of JNI Programs via Binary Decompilation
- Static analysis of Sequential Function Charts using…

- Static detection of equivalent mutants in real-time…
- Static Local Concurrency Errors Detection in MPI-RMA…
- Static Neural Compiler Optimization via Deep…
- Static Timing and Power Analysis of a RISC-V Pipelined…
- Static Value Analysis of Python Programs by Abstract…
- Static Worst-Case Execution Time Optimization using DPSO…
- Statistical Unigram Analysis for Source Code Repository
- STERILIZER CHAMBER DESIGN WITH TELEGRAM-BASED INTERNET OF…
- Streaming adaptive submodular maximization
- Streaming Algorithms for Constrained Submodular…
- Streaming algorithms for non-monotone DR-submodular…
- Streaming Algorithms for Non-Submodular Maximization on…
- Streaming algorithms for robust submodular maximization
- Streaming Non-Monotone Submodular Maximization…
- Streaming Stochastic Submodular Maximization with…
- Streaming submodular maximization under d-knapsack…
- Streaming Submodular Maximization Under Matroid…
- Streaming Submodular Maximization Under Noises
- Streaming submodular maximization with fairness…
- Strumbelj and Kononenko 2013
- Studying WebAssembly and comparison of its performance…
- Submodular maximization meets streaming: matchings…
- Submodular Maximization Subject to Uniform and Partition…
- SuperGraph-SLP Auto-Vectorization
- Support for Just-in-Time Compilation of WebAssembly for…
- Supporting Dynamic Program Sizes in Deep Learning-Based…
- SurtGIS: A high-performance raster geospatial analysis…
- Survey on Estimation and Optimization of Worst-case…
- Swarm Intelligence for Software Effort Estimation: An…
- Symbolic Execution and Recent Applications to Worst-Case…
- Symbolic MRD: Dynamic Memory, Undefined Behaviour, and…
- Synchronized Behavior Checking: A Method for Finding…
- Synchronized Shared Memory and Procedural Abstraction…
- Syntactic Versus Semantic Similarity of Artificial and…
- Syntax, predicates, idioms — what really affects code…
- Syntax-Driven Translation
- Synthesizing an instruction selection rule library from…
- Synthesizing Instruction Selection Back-Ends from ISA…
- SySTeC: A Symmetric Sparse Tensor Compiler
- System on Chip Implementation of Compiler Stack with a…
- System-Level Test Case Prioritization Using Machine…
- Systematic Mapping Study of Ensemble Effort Estimation
- T 3 : Multi-level Tree-based Automatic Program Repair…
- TABU Search Prioritized Ant Colony Metaheuristic…
- TagDebt: a bot to support technical debt management
- Taming undefined behavior in LLVM
- Targeting both Blazor Server and Blazor WebAssembly
- Technical Debt Contagiousness Metrics for Measurement and…
- Technical debt prioritization
- Technical debt prioritization using predictive analytics
- Technical Debt Prioritization: A Search-Based Approach
- Technical Debt Prioritization: Taxonomy, Methods Results…
- Technical Debt Tools: a Survey and an Empirical Evaluation
- Technical Debt, Software Evolution and Legacy

- Technical Perspective: Fusing Large Language Models with…
- Techniques of Test Case Prioritization
- Template-Guided Program Repair in the Era of Large…
- Temporal Discounting in Software Engineering: A…
- Tensor Program Optimization for the RISC-V Vector…
- Tensor Program Superoptimization through Cost-Guided…
- TenSure: Fuzzing Sparse Tensor Compilers (Registered…
- Termination-checking for LLVM peephole optimizations
- Terms for Efficient Proof Checking and Parsing
- Test Case Design and Test Case Prioritization using…
- Test case prioritization and selection technique in…
- Test Case Prioritization for Regression Testing Using…
- Test Case Prioritization in Continuous Integration…
- Test Case Prioritization using Transfer Learning in…
- Test case sampling optimization for safety validation of…
- Test case selection and prioritization using machine…
- Test code refactoring unveiled: where and how does it…
- Test input prioritization for image segmentation: an…
- Test prioritization in continuous integration environments
- Test program generation for mixed-signal integrated…
- TestCov: Robust Test-Suite Execution and Coverage…
- Testing a PL/I Compiler Using Precomputation-based…
- Testing and Verifying Parallel Programs Using Data…
- Testing Autonomous Driving Systems Through Blind-Spot…
- Testing Error Handling Code With Software Fault Injection…
- Testing ocean software with metamorphic testing
- Testing static analyses for precision and soundness
- Testing the Unknown: A Framework for OpenMP Testing via…
- Testing, Credible Compilation, and Verification in the…
- Testing-Based Formal Verification for Theorems and Its…
- The Application Of Machine Learning In Test Case…
- The ARES High-Level Intermediate Representation
- The broken windows theory applies to technical debt
- The CakeML Compiler Explorer
- The Correctness-Security Gap in Compiler Optimization
- The Design of Optimized RISC Processor for Edge…
- The Evolution of Empirical Methods in Software Engineering
- The Expansion of Source Code Abbreviations Using a…
- The forgotten role of search queries in IR-based bug…
- The Forward-Reverse Greedy Algorithm for Distributed…
- The Impact of Fine-Tuning Large Language Models on…
- The Impact of Undefined Behavior on Compiler Optimization
- The Influence of Cost Drivers on Effort Estimation in…
- The method of requirements prioritization in software…
- The MLIR Transform Dialect
- The reproducibility of programming-related issues in…
- The role and value of replication in empirical software…
- The Shapley and Position Values to Design Coalitional…
- The Shapley Value as a Sustainable Cooperative Solution…
- The Shapley Value, a Crown Jewel of Cooperative Game…
- The Simian concept: Parallel Discrete Event Simulation…
- The Software Heritage Graph Dataset
- The Trusted Computing Base of the CompCert Verified…
- The use of large language models for program repair

- The verified CakeML compiler backend
- The Vocabulary of Flaky Tests in the Context of SAP HANA
- Thinking Fast and Correct: Automated Rewriting of…
- Thread-Aware Area-Efficient High-Level Synthesis Compiler…
- Threaded Code Generation with a Meta-Tracing JIT Compiler
- Time-Accurate ASM as a Refinement Scheme for Worst-Case…
- Timing Analysis Techniques
- Timing speculation-aware instruction set extension for…
- TinyGen: Portable and Compact Code Generation for Tiny…
- TinyML Unleashed: Accelerating TensorFlow Lite Micro…
- Tinyrossa: A Compiler Framework for Vertical, Verified…
- Tofte and Talpin 1997
- Toward a Formally Verified Compiler for a Synchronous…
- Toward an Automated Hardware Pipelining LLVM Pass…
- Toward Green Code: Prompting Small Language Models for…
- Toward learnable and interpretable data Shapley valuation…
- Toward prioritization of self-admitted technical debt: an…
- Towards a Domain-Extensible Compiler: Optimizing an Image…
- Towards a functional requirements prioritization with…
- Towards a Unified Multi-Target Mlir-Based Compiler: A…
- Towards a verified compiler prototype for the synchronous…
- Towards a verified Lustre compiler with modular reset
- Towards a WebAssembly standalone runtime on GraalVM
- Towards Automatic HBM Allocation Using LLVM: A Case Study…
- Towards Automatic Property Generation for SoC Security…
- TOWARDS AUTONOMOUS CODE OPTIMIZATION: A REINFORCEMENT…
- Towards Compile-Time-Reducing Compiler Optimization…
- Towards compiler-aided correctness checking of adjoint…
- Towards defect-type-aware adaptive program repair: A…
- Towards enhancing the reproducibility of deep learning…
- Towards Incremental Static Race Detection in OpenMP…
- Towards Interpretable Ensemble Learning for Software…
- Towards Path-Aware Coverage-Guided Fuzzing
- Towards Supporting Semiring in MLIR-Based COMET Compiler
- Towards understanding code review practices for…
- Towards Verified Rounding Error Analysis for Stationary…
- Towards Verified Security: Formal Methods for LLM-Based…
- TPDE: A Fast Adaptable Compiler Back-End Framework
- Trace-Based Bytecode Interpreter Visualization for…
- Tracy: A Business-Driven Technical Debt Prioritization…
- Translating CUDA to OpenCL for Hardware Generation using…
- Translation from Visual to Layout-based Android Test…
- Translation Validation for JIT Compiler in the V8…
- Translation Validation of Information Leakage of Compiler…
- TreeHouse: An MLIR-based Compilation Flow for Real-Time…
- Tutorial on Static Inference of Numeric Invariants by…
- Twine: An Embedded Trusted Runtime for WebAssembly
- Type-Centric Kotlin Compiler Fuzzing: Preserving Test…
- TYPEFUZZ: Type Coverage Directed JavaScript Engine…
- Typestates Specification and Verification in Frama-C
- UML Associations - Reducing the Gap in Test Coverage…
- Understanding and Finding JIT Compiler Performance Bugs
- Understanding and improving the quality and…
- Understanding Binary Code Similarity for Real-World…

- Understanding Bug-Reproducing Tests: A First Empirical…
- Understanding Feature Request Practice on GitHub via a…
- Understanding Key Features of High-Impact Bug Reports
- Understanding practitioners' reasoning and requirements…
- Understanding Self-Admitted Technical Debt in Test Code…
- Unified HW/SW Coverage: A Novel Metric to Boost…
- Unleashing the power of compiler intermediate…
- Unleashing Triton on CPUs: Compilation and Runtime…
- Upstream bug management in Linux distributions
- Usability Technical Debt in Software Projects: A…
- Usage of Large Language Model for Code Generation Tasks…
- Use Case-Based Analytical Hierarchy Process Method for…
- Use of Compiler Intermediate Representation for Reverse…
- Use of Measurements in Worst-Case Execution Time…
- User-Directed Loop-Transformations in Clang
- Using Artificial Bee Colony for Code Coverage Based Test…
- Using docker containers to improve reproducibility in…
- Using Extremely Simplified Functional Size Measures for…
- Using Large-Scale Anomaly Detection on Code to Improve…
- Using rapid reviews to support software engineering…
- Using text clustering to predict defect resolution time…
- Utilizing Machine Learning Techniques for Worst-Case…
- Valent-Blocks: Scalable High-Performance Compilation of…
- Variable Bit-Precision Vector Extension for RISC-V Based…
- VCNN: A compiler of CNNs based on MLIR for multi-core…
- Vectorization in PyPy's Tracing Just-In-Time Compiler
- Vectorized Nonlinear Functions with the RISC-V Vector…
- VECTR: A Lightweight Requirements Prioritization Method…
- VEGA: Automatically Generating Compiler Backends using a…
- Verification by abstract interpretation, soundness and…
- Verified compilation of CakeML to multiple machine-code…
- Verified compilation on a verified processor
- Verified construction of static single assignment form
- Verified lifting of stencil computations
- Verified peephole optimizations for CompCert
- Verified Propagation Redundancy and Compositional UNSAT…
- Verified VCG and Verified Compiler for Dafny
- Verifying Instruction Set Simulators using…
- Verifying optimizations of concurrent programs in the…
- Verifying Term Graph Optimizations using Isabelle/HOL
- VeriPhy: verified controller executables from verified…
- Video SIMDBench: Benchmarking the Compiler Vectorization…
- Vivienne: Relational Verification of Cryptographic…
- Vmxdotp: A RISC-V Vector ISA Extension for Efficient…
- WAFL: Binary-Only WebAssembly Fuzzing with Fast Snapshots
- Wait for it: identifying "On-Hold" self-admitted…
- Wapplique: Testing WebAssembly Runtime via Execution…
- WARD: Efficient Memory Protection for WebAssembly on Tiny…
- WARDuino: An embedded WebAssembly virtual machine
- Wasm-Mutate: Fast and effective binary diversification…
- Wasm-WCET: Worst-Case Execution-Time Analysis of…
- WasmSlim: Optimizing WebAssembly Binary Distribution via…
- WasmWeaver: A Framework for Runtime-Aware WebAssembly…
- WaTZ: A Trusted WebAssembly Runtime Environment with…

- WaVe: a verifiably secure WebAssembly sandboxing runtime
- WBSan: WebAssembly Bug Detection for Sanitization and…
- WebAssembly as a Fuzzing Compilation Target (Registered…
- WebAssembly for Container Runtime: Are We There Yet?
- WebAssembly Module Internals: Sections and Memory Model
- WebAssembly versus JavaScript: Energy and Runtime…
- WebAssembly: How Low Can a Bytecode Go?
- Wegman and Zadeck 1991
- Weighted Reward for Reinforcement Learning based Test…
- Weighted shapley value: A cooperative game theory for…
- What do developer-repaired Flaky tests tell us about the…
- When Compiler Optimizations Meet Symbolic Execution: An…
- When Function Inlining Meets WebAssembly…
- When Function Signature Recovery Meets Compiler…
- When is Continuous Integration Useful? Empirical Study on…
- WhiteFox: White-Box Compiler Fuzzing Empowered by Large…
- Who Will Leave the Company?: A Large-Scale Industry Study…
- Whose Baseline Compiler is it Anyway?
- Why are Android apps removed from Google Play?
- Why Just-In-Time Compilation Matters: Evaluating Runtime…
- Why Shapley Value and Its Variants Are Useful in Machine…
- Why Some Bug-bounty Vulnerability Reports are Invalid?
- Wild SBOMs: a Large-scale Dataset of Software Bills of…
- Wilhelm and colleagues 2008
- Wilson 1927
- Wong and colleagues 1995
- Work-in-Progress: Searching Optimal Compiler Optimization…
- Worst Case Execution Time and Power Estimation of…
- Worst-Case Execution Time Analysis for Many-Core…
- Worst-Case Execution Time Analysis of a Real-Time System…
- Worst-Case Execution Time Analysis of Real-Time Robotic…
- Worst-Case Execution Time Estimation for Numerical…
- Worst-case Execution Time Estimation of Legacy Vehicular…
- Worst-Case Execution Time Testing via Evolutionary…
- WuppieFuzz: Coverage-Guided, Stateful REST API Fuzzing
- Yang and colleagues 2011
- Yoo and Harman 2012
- Young 1985
- Zhao and colleagues 2012
- Zhu, Hall and May 1997
- Zipr: Efficient Static Binary Rewriting for Security
- Zoish: A Novel Feature Selection Approach Leveraging…