CryptOpt: Automatic Optimization of Straightline Code

Joel Kuepper[1], David Wu[1], Andres Erbsen[2], Jason Gross[2], Owen Conoly[2], Chuyue Sun[3], Samuel Tian[2], Adam Chlipala[2], Chitchanok Chuengsatiansup[4], Daniel Genkin[5], Markus Wagner[6], Yuval Yarom[1]

[1] University of Adelaide, [2] Massachusetts Institute of Technology, [3] Stanford University,
[4] University of Melbourne [5] Georgia Tech, [6] Monash University

***Abstract*—Manual engineering of high-performance implementations typically consumes many resources and requires in-depth knowledge of the hardware. Compilers try to address these problems; however, they are limited by design in what they can do. To address this, we present CryptOpt, an automatic optimizer for long stretches of straightline code. Experimental results across eight hardware platforms show that CryptOpt achieves a speed-up factor of up to 2.56 over current off-the-shelf compilers.**

***Index Terms*—Automatic Performance Optimization, Search Based Software Engineering, Local Search, Elliptic Curve Cryptography**

I. Introduction

Superscalar processors in recent years have become very complex, and inherent CPU properties make it hard to reason about the performance of a given assembly code [3], [10], [13], [18], [22], [24]. As such, optimizing code performance for a specific CPU microarchitecture is not trivial. Typical compiler optimizations focus on control flow rather than long stretches of straightline code [2]. Moreover, they often utilize peephole optimizers [2], [7] with heuristics of replacement patterns to statically optimize small sections of code.

We present CryptOpt, an automatic optimizer for long stretches of straightline arithmetic code. CryptOpt recasts compilation as a combinatorial optimization problem, with runtime as the cost function, and utilizes techniques from the field of search-based software engineering [16] to optimize. We observe that rather simple techniques, such as a random local search (RLS) [14], can produce faster code than current off-the-shelf compilers, such as GCC and Clang, even when used with their highest optimization settings. Two observations are the key enablers for our approach. First, specializing in straightline code simplifies code analysis. This simplicity, in turn, enables CryptOpt to explore many optimization options, such as reordering operations, where conventional compilers, including GCC and Clang, tend to be more conservative. Second, by actually running the generated code on the target hardware we can optimize specifically for particular architectures, while treating the CPU as a black box, removing the need for complicated, error-prone, and lengthy modeling.

Cryptographic code typically follows the constant-time programming paradigm to mitigate timing side-channel attacks [4], [11], [21]. As such, it tends to contain long stretches of straightline arithmetic, which we use as our first use case: we use CryptOpt to generate high-performance cryptographic code,[1] optimized for eight CPU architectures, achieving speedups of up to 87% across platforms and up to 156% in single cases. We also show that we can optimize on a per-architecture level. That is, we can generate a solution on one platform (optimized for the same platform) that outperforms every other solution optimized on (and for) other platforms.

We believe that this technique can serve as a foundation for future engineering of high-performance implementations. Rather than employing reverse engineering and processor modeling, we employ search algorithms and performance measurements. That is, we simply run our optimizer on future processors to generate optimized code with minimal effort. CryptOpt is open-source, available at https://0xADE1A1DE.github.io/CryptOpt.

II. Overview

We now sketch CryptOpt's input language and outline how it works at a high level. Then, we focus on how a user would use it for their own architecture or for their own input functions. We conclude this section with a detailed description of the inner workings of CryptOpt.

A. Input Language

CryptOpt reads the description of an input function in an intermediate language (see below). This input language is sufficient to describe even large expressions including modular arithmetic using bitwise operations. It even allows to materialize ϕ-nodes via a conditional-move (cmovznz) operation.

Variable	x		
Binary integer	b		
Operand	e	$::=$	$x \mid b$
Operator	o	$::=$	$! \mid \& \mid * \mid + \mid - \mid << \mid = \mid >> \mid \sim \mid$ $\text{or} \mid \text{addcarryx} \mid \text{cmovznz} \mid \text{mulx} \mid$ $\text{static\_cast} \mid \text{subborrowx}$
Expression	E	$::=$	$\text{return } e \mid x, \ldots, x \leftarrow o(e, \ldots, e); E$

B. High-Level Concept

CryptOpt parses a function in the input language (specified as a JSON file) into an internal representation (IR). From this IR, CryptOpt derives a base implementation candidate in assembly (blue in Figure 1). At a high level, CryptOpt uses a RLS for optimizing the code. Figure 1 shows the basic step of

[1] The full version of this paper shows that this code is also formally verified [19].

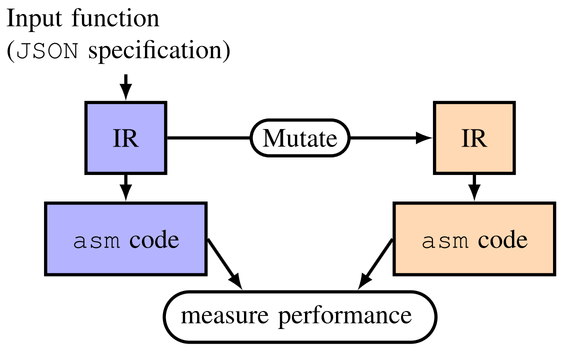

Fig. 1. CryptOpt high-level concept: Input specification is parsed into an IR and assembly code. The initial IR is then mutated (shown in orange). The performance of both candidates is then compared.

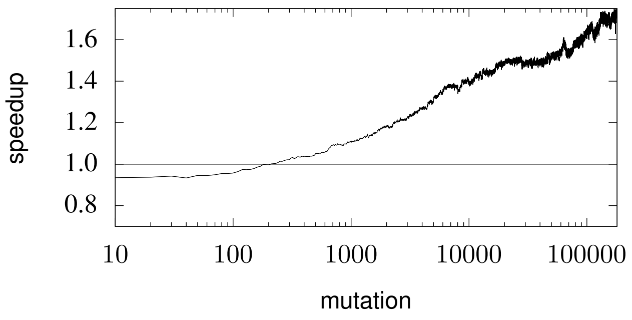

Fig. 2. Optimization progress of SIKEp434–square on Intel Core i7-10710U, showing the relative performance gain over Clang as a function of the number of tested mutations.

the RLS algorithm. Specifically, CryptOpt mutates the current IR, changing the instruction selected for implementing an IR operation or the order of operations to derive an alternative assembly implementation (orange in Figure 1). Next, CryptOpt measures the performance of both assembly implementations, i.e. of the previous best and of the mutant, and keeps the faster candidate. This *mutate-measure-select* step is repeated iteratively, resulting in an optimized implementation of the input function. While measuring the performance, CryptOpt also checks the functional result of the candidates against a C-compiled "ground truth."

C. User Workflow

The current application of CryptOpt is for optimizing cryptographic code. In particular, we focus on cryptographic primitives such as finite-field arithmetic, as widely used in elliptic-curve cryptography (e.g. in TLS and Bitcoin [8]) and postquantum cryptography (though schemes like SIKE [5]). To that end, we integrate with Fiat Cryptography [15], which can generate both a JSON specification of a field-arithmetic routine as well as a (decently optimized) C reference. In this particular use case, CryptOpt can be invoked with a "curve–method" combination, e.g. using the command `./CryptOpt --bridge fiat --curve curve25519 --method square`. Fiat Cryptography is then consulted internally to generate the required code. To use CryptOpt for other functions, which are not produced by Fiat Cryptography, the user should use the `--bridge manual` parameter and provide both a JSON specification of the input function and a C reference of the same function as the ground truth, e.g. `./CryptOpt --bridge manual --jsonFile ./example.json --cFile ./example.c`. Note that CryptOpt is not a compiler plugin. Rather, CryptOpt is a self-contained tool to generate x86-64 assembly from the input specification outlined in Section II-A.

While optimizing, the user receives regular status updates on the console, providing information on, e.g. the number of instructions used to implement the function in the current version, the number of memory spills, or the relative speedup compared to the C-compiled ground truth. Further, we also generate a PDF file showing how the optimization progresses over time. See Figure 2 for an example.

D. Detailed Internals

We now describe the inner workings of CryptOpt, shown in Figure 3. The user invokes CryptOpt with the parameters described in Section II-C. Upon invocation, the provided parameters are parsed, and the control is handed to Optimizer. Optimizer selects the required Bridge—a module which generates both the required function description in a JSON format and a shared object (`*.so`-file) from the C reference, which is used for correctness checks while measuring. Optimizer then initializes Model, a component used as the single point of truth for the IR. The initialization procedure of Model also preprocesses the operations specified in the input description. This includes

- **Instruction scheduling**: analyzing the data flow and subsequently generating an initial ordering of the operations.
- **Instruction selection**: assigning a template to each operation specifying which x86-64 assembly instruction(s) implement it.

After that, Optimizer invokes Assembler to assemble the current IR. Assembler initializes Register Allocator, which is the component maintaining the virtual state of the CPU FLAGS register, general-purpose registers, and stack memory. The Register Allocator is initialized with caller-save registers holding (unknown) live values and calling-convention-based registers, holding function parameters. Assembler processes the operations from Model according to the current order, producing the x86-64 assembly according to the assigned template. While doing that, Assembler consults Register Allocator to get empty registers or modify the storage locations of intermediate values. Register Allocator will, in case it needs to spill a variable to memory, get the next operations from Model to determine which value is most suitable to spill. Assembler eventually returns the x86-64 assembly string to Optimizer, which stores it temporarily. Optimizer then invokes the mutate function on Model. This will (randomly) change the IR in one of two ways: change the order in which the operations are implemented, or change the template assigned to one of

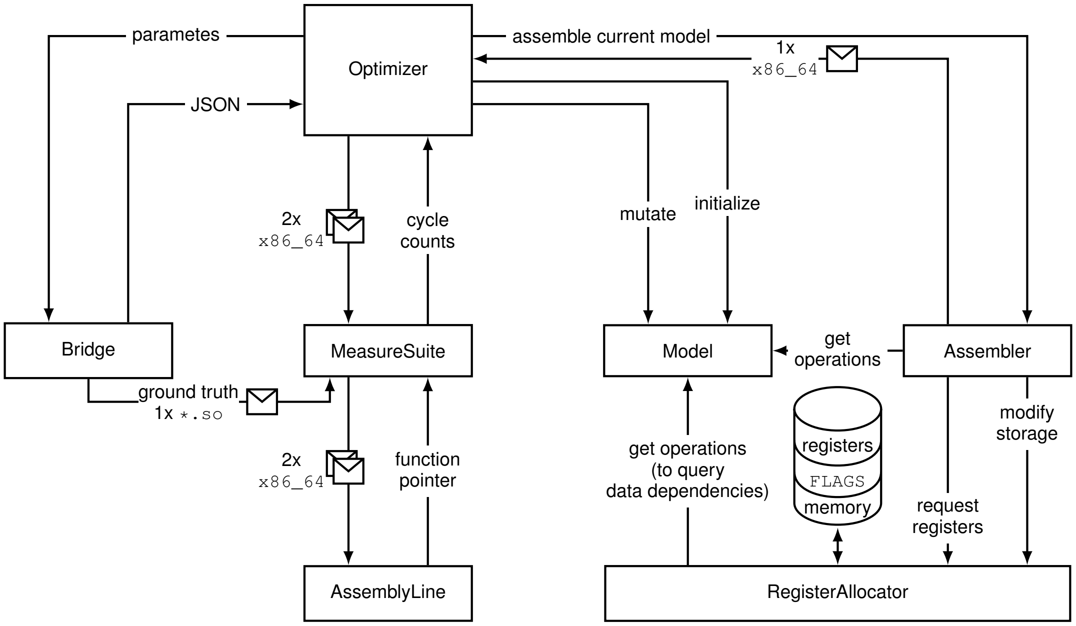

Fig. 3. Component diagram of CryptOpt.

the operations. Optimizer will then invoke Assembler again to generate the assembly code of the mutated IR. Optimizer then uses MeasureSuite to compare the performance of the implementation of the original IR with that of the mutated IR. MeasureSuite uses AssemblyLine [1] to assemble two x86-64 assembly codes and perform a version of the R3-validation [9], as outlined in Section III. Finally, MeasureSuite returns the measurement results to Optimizer, which compares the results and decides whether to accept the mutated IR as the new base implementation. Otherwise, Optimizer discards the mutation and proceeds to test another one. After a predefined number of mutations has been tried, CryptOpt writes the last base implementation to disk.

III. Measurement

Measuring precise execution times of short stretches of code is challenging because it is affected by a large number of noise factors [3], [10], [18], [22], [24]. Bokhari et al. [9] compared validation approaches in the context of energy-consumption optimization. We adopt their R3-validation approach with two modifications. First, we do not restart the computer after each evaluation, because we do not observe any measurement drift over time. Second, we use a random scheduling of program variants instead of following a strict order of measurements to reduce effects of learned execution orders by the CPU.

Recall that we need to assess which of the two candidate x86-64 assembly implementations is faster. Our measurement procedure randomly selects one candidate and measures it in a tight loop. Then, we measure the check function (ground truth) in a tight loop. The number of iterations in each of those loops depends on the measured code: Slow stretches are repeated fewer times than fast stretches. With this dynamic, we can skew the nominal number of cycles measured into a range where the performance counter gives enough granularity. Empirically, aiming for roughly 10 000 cycles provides a good trade-off in terms of sensitivity because it amplifies the execution time differences enough to be detected, yet is robust enough not to be misled by the environmental factors.

TABLE I
Overview of target machines used in the experiments

CPU	μ-arch	Mainboard
AMD Ryzen Threadripper 1900X	Zen 1	ASUS ROG STRIX X399-E Gaming
AMD Ryzen 7 5800X	Zen 3	Gigabyte B550 AORUS ELITE V2
AMD Ryzen 9 5950X	Zen 3	Gigabyte X570 GAMING X
Intel Core i7-6770HQ	Skylake-H	Intel NUC6i7KYB
Intel Core i7-10710U	Comet Lake-U	Intel NUC10i7FNB
Intel Core i9-10900K	Comet Lake-S	Gigabyte H470 HD3
Intel Core i7-11700KF	Rocket Lake-S	ASRock Z590 Pro4
Intel Core i9-12900KF	Alder Lake-S	Micro-Star PRO Z690-A Wifi (MS-7D25)

For stability, we repeat this procedure multiple times, each with randomly selected input values and compare the median of these experiments to determine which of the candidates to keep. We empirically find that 31 repetitions provide sufficiently stable results on our systems. The user can adjust the number to suit the running environment, e.g. a higher number may be required for noisy systems.

IV. CryptOpt in the Real World

Evaluation setup. We evaluate CryptOpt on eight different platforms, summarized in Table I. On each machine, we generate x86-64 assembly code with CryptOpt for the multiply and square methods of nine different prime fields from Fiat Cryptography.

Optimization process. Optimization takes between 36 and 70 wall-clock hours to generate those 18 primitives, depending on the machine. The length of the produced code depends on the compiled primitive and varies from fewer than 100 instructions to almost 1000 instructions. The average length of the best implementations generated is shown in Table II.

TABLE II
AVERAGE INSTRUCTION COUNT, ROUNDED TO NEAREST INTEGER

Primitive	Multiply	Square
Curve25519	170	121
NIST P-224	221	219
NIST P-256	204	200
NIST P-384	710	698
SIKEp434	986	965
Curve448	588	405
NIST P-521	542	338
Poly1305	76	61
secp256k1	233	224

Optimized code performance. Table III shows the geometric mean over those eight platforms of the speedup of CryptOpt-generated code vs. machine code generated by off-the-shelf compilers Clang version 14.0.0 and GCC version 11.3.0 at the highest optimization settings (`-O3 -mtune=native -march=native`).

Side-channel resistance. Code generated by Fiat Cryptography is timing-side-channel-secure. CryptOpt will only optimize with different instruction scheduling, instruction selection, and register allocation. CryptOpt will not change algorithmic structures[2]. We implement Fiat IR's `cmovznz`-operation with Intel's dedicated `cmovCC` instruction. As such, code based on Fiat Cryptography optimized with CryptOpt is inherently timing-side-channel-secure.

V. Related Work

Next, we describe other work in the broader realm of automatically optimizing code. We start with superoptimization, which targets the smallest pieces of code, and turn our focus then onto the field of peephole optimization, which considers larger chunks of instructions.

Superoptimization. In 1987, Henry Massalin coined the term "superoptimizer" to describe his tool for exhaustive enumeration of all possible programs to implement a given function [20]. Because exhaustive enumeration can require a huge computational effort, the key idea making this feasible is the use of a probabilistic test set, which rejects the majority of incorrect candidates. His superoptimizer was able to generate programs of 12 instructions after several hours of running (on a 16MHz 68020 computer). Since then, superoptimizers have evolved significantly: Souper [25] can synthesize new optimizations on the LLVM IR, but as such they cannot exploit target-specific properties. Denali [17] uses solvers to generate provably shortest programs, but it can only be applied to rather short program sequences in the range of tens of instructions. STOKE [26] and its extensions [27]–[29] can synthesize new programs from scratch and optimize them, only focusing on very small kernels of loops.

TABLE III
GEOMETRIC MEANS OF CRYPTOPT VS. OFF-THE-SHELF COMPILERS.

	Multiply		Square	
Curve	Clang	GCC	Clang	GCC
Curve25519	1.19	1.14	1.14	1.18
P-224	1.31	1.87	1.24	1.84
P-256	1.27	1.79	1.30	1.85
P-384	1.12	1.66	1.08	1.60
SIKEp434	1.30	1.70	1.29	1.83
Curve448	1.02	0.95	1.00	0.99
P-521	1.20	1.06	1.25	1.11
Poly1305	1.10	1.15	1.09	1.16
secp256k1	1.34	1.73	1.32	1.74

Peephole optimization. Instead of synthesizing new and optimal very short programs, peephole optimizers use a sliding window on instructions (the peephole) and replace sets of existing instructions with more performant alternatives [2], [7], [12]. The replacement is usually done based on a predefined rule set (applying only to short instruction sequences), which itself is based on heuristics for estimating which set of instructions is likely to be more performant or shorter than an alternative.

One possible next step can be to find those heuristics automatically [6], [23] and then to apply this new knowledge to the target code. However, those rules are still applied statically, i.e. without taking the actual effects on runtime into account, and they are typically applied over the entire code.

With CryptOpt, we overcome those limitations by first only applying a mutation locally and second by measuring the (side) effects of every mutation.

VI. Summary and Outlook

We present CryptOpt, a tool for generating optimized assembly code by combining simple techniques. As present, CryptOpt tackles distinctive characteristics of straightline cryptographic code, achieving a significant improvement over mainstream compilers. In the future, we expect that these techniques can be generalized to other domains of compilation. Moreover, it would be interesting to test if replacing RLS with more advanced optimization strategies would improve CryptOpt's run time and results.

Acknowledgments

This research was supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; the ARC DECRA DE200101577; the ARC Discovery Projects DP200102364 DP210102670; the Blavatnik ICRC at Tel-Aviv University; CSIRO's Data61; the National Science Foundation under grants CNS-1954712 and CNS-2130671; the National Science Foundation Expedition on the Science of Deep Specification award CCF-1521584; and gifts from Amazon Web Services, AMD, Cisco, Facebook, Google, Intel, and the Tezos Foundation.

[2] We do optimize with simple forms of strength reduction such as replacing multiplication by constants with a combination of additions and bit-shifts.

References

[1] 0xADE1A1DE, "Assemblyline," 2022. [Online]. Available: https://github.com/0xADE1A1DE/AssemblyLine

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, ser. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[3] A. R. Alameldeen and D. A. Wood, "Variability in architectural simulations of multi-threaded workloads," in *HPCA*, 2003, pp. 7–18.

[4] N. J. AlFardan and K. G. Paterson, "Lucky thirteen: Breaking the TLS and DTLS record protocols," in *IEEE SP*, 2013, pp. 526–540.

[5] R. Azarderakhsh, M. Campagna, C. Costello, L. D. Feo, B. Hess, A. Jalali, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, G. Pereira, J. Renes, V. Soukharev, and D. Urbanik, "Supersingular isogeny key encapsulation – submission to the NIST post-quantum standardization project, round 2," 2019. [Online]. Available: https://sike.org

[6] S. Bansal and A. Aiken, "Automatic generation of peephole superoptimizers," in *ASPLOS*, 2006, pp. 394–403.

[7] S. D. Bergmann, "Compilers," in *Encyclopedia of Information Systems*, 2003, pp. 141–170.

[8] Bitcoin Core, "libsecp256k1 - optimized C library for ECDSA signatures and secret/public key operations on curve secp256k1," 2022. [Online]. Available: https://github.com/bitcoin-core/secp256k1

[9] M. A. Bokhari, B. Alexander, and M. Wagner, "Towards rigorous validation of energy optimisation experiments," in *GECCO*, 2020, pp. 1232–1240.

[10] M. A. Bokhari, L. Weng, M. Wagner, and B. Alexander, "Mind the gap - a distributed framework for enabling energy optimisation on modern smart-phones in the presence of noise, drift, and statistical insignificance," in *IEEE CEC*, 2019, pp. 1330–1337.

[11] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "FaCT: A flexible, constant-time programming language," in *SecDev*, 2017, pp. 69–76.

[12] K. D. Cooper and L. Torczon, "Chapter 11 - instruction selection," in *Engineering a Compiler (Second Edition)*, 2012, pp. 597–638.

[13] C. Curtsinger and E. D. Berger, "STABILIZER: statistically sound performance evaluation," in *ASPLOS*, 2013, pp. 219–228.

[14] B. Doerr and F. Neumann, *Theory of evolutionary computation: Recent developments in discrete optimization*. Springer, 2019.

[15] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, "Simple high-level code for cryptographic arithmetic - with proofs, without compromises," in *IEEE SP*, 2019, pp. 1202–1219.

[16] M. Harman and B. F. Jones, "Software engineering using metaheuristic innovative algorithms: workshop report," *Inf. Softw. Technol.*, vol. 43, no. 14, pp. 905–907, 2001.

[17] R. Joshi, G. Nelson, and K. H. Randall, "Denali: A goal-directed superoptimizer," in *PLDI*, 2002, pp. 304–314.

[18] T. Kalibera, L. Bulej, and P. Tuma, "Benchmark precision and random initial state," in *SPECTS*, 2005, pp. 853–862.

[19] J. Kuepper, A. Erbsen, J. Gross, O. Conoly, C. Sun, S. Tian, D. Wu, A. Chlipala, C. Chuengsatiansup, D. Genkin, M. Wagner, and Y. Yarom, "CryptOpt: Verified compilation with random program search for cryptographic primitives," ArXiv abs/2211.10665, 2022.

[20] H. Massalin, "Superoptimizer - A look at the smallest program," in *ASPLOS*. ACM, 1987, pp. 122–126.

[21] D. Molnar, M. Piotrowski, D. Schultz, and D. A. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *ICISC*, 2005, pp. 156–168.

[22] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *ASPLOS*, 2009, pp. 265–276.

[23] G. Pekhimenko and A. D. Brown, "Efficient program compilation through machine learning techniques," in *Software Automatic Tuning, From Concepts to State-of-the-Art Results*. Springer, 2010, pp. 335–351.

[24] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan, "Thread tranquilizer: Dynamically reducing performance variation," *ACM TACO*, vol. 8, no. 4, pp. 46:1–46:21, 2012.

[25] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, J. Taneja, and J. Regehr, "Souper: A synthesizing superoptimizer," Arxiv abs/1711.04422, 2017.

[26] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *ASPLOS*, 2013, pp. 305–316.

[27] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic optimization of floating-point programs with tunable precision," in *PLDI*, 2014, pp. 53–64.

[28] R. Sharma, E. Schkufza, B. R. Churchill, and A. Aiken, "Data-driven equivalence checking," in *OOPSLA*. ACM, 2013, pp. 391–406.

[29] R. Sharma, E. Schkufza, B. R. Churchill, and A. Aiken, "Conditionally correct superoptimization," in *OOPSLA*, 2015, pp. 147–162.