DOI:10.1145/2863701

Stochastic Program Optimization

By Eric Schkufza, Rahul Sharma, and Alex Aiken

Abstract

The optimization of short sequences of loop-free, fixed-point assembly code sequences is an important problem in high-performance computing. However, the competing constraints of transformation correctness and performance improvement often force even special purpose compilers to produce sub-optimal code. We show that by encoding these constraints as terms in a cost function, and using a Markov Chain Monte Carlo sampler to rapidly explore the space of all possible code sequences, we are able to generate aggressively optimized versions of a given target code sequence. Beginning from binaries compiled by `llvm -O0`, we are able to produce provably correct code sequences that either match or outperform the code produced by `gcc -O3`, `icc -O3`, and in some cases expert handwritten assembly.

1. INTRODUCTION

For many application domains there is considerable value in producing the most performant code possible. However, the traditional structure of a compiler's optimization phase is ill-suited to this task. Factoring the optimization problem into a collection of small subproblems that can be solved independently—although suitable for generating consistently good code—can lead to sub-optimal results. In many cases, the best possible code can only be obtained through the simultaneous consideration of mutually dependent issues such as instruction selection, register allocation, and target-dependent optimization.

Previous approaches to this problem have focused on the exploration of all possibilities within some limited class of code sequences. In contrast to a traditional compiler, which uses performance constraints to drive the generation of a single sequence, these systems consider multiple sequences and select the one that is best able to satisfy those constraints. An attractive feature of this approach is completeness: if a code sequence exists that meets the desired constraints it is guaranteed to be found. However, completeness also places practical limitations on the type of code that can be considered. These techniques are either limited to sequences that are shorter than the threshold at which many interesting optimizations take place or code written in simplified languages.

We overcome this limitation through the use of incomplete search: the competing requirements of correctness and performance improvement are encoded as terms in a cost function which is defined over the space of all loop-free x86_64 instruction sequences, and the optimization task is formulated as a cost minimization problem. While the search space is highly irregular and not amenable to exact optimization techniques, we show that the common approach of employing a Markov Chain Monte Carlo (MCMC) sampler to explore the function and produce low-cost samples is sufficient for producing high-quality code.

Although the technique sacrifices completeness it produces dramatic increases in the quality of the resulting code. Figure 1 shows two versions of the Montgomery multiplication kernel used by the OpenSSL RSA encryption library. Beginning from code compiled by `llvm -O0` (116 lines, not shown), our prototype stochastic optimizer STOKE produces code (right) that is 16 lines shorter and 1.6 times faster than the code produced by `gcc -O3` (left), and even slightly faster than the expert handwritten assembly included in the OpenSSL repository.

2. RELATED WORK

Although techniques that preserve completeness are effective within certain domains, their general applicability remains limited. The shortcomings are best highlighted in the context of the code sequence shown in Figure 1.

The code does the following: two 32-bit values, `ecx` and `edx`, are concatenated and multiplied by the 64-bit value `rsi` to produce a 128-bit product. The 64-bit values in `rdi` and `r8` are added to that product, and the result is stored in `r8` and `rdi`. The version produced by `gcc -O3` (left) implements the 128-bit multiplication as four 64-bit multiplications and a summation of the results. In contrast, the version produced by STOKE (right), uses a hardware intrinsic which requires that the inputs be permuted and moved to distinguished register locations so that the multiplication may be performed in a single step. The odd looking move on line 4 (right) produces the non-obvious but necessary side effect of zeroing the upper 32 bits of `rdx`.

Massalin's superoptimizer[12] explicitly enumerates sequences of code of increasing length and selects the first that behaves identically to the input sequence on a set of test cases. Massalin reports optimizing instruction sequences of up to length 12, but only after restricting the set of enumerable opcodes to between 10 and 15. In contrast, STOKE uses a large subset of the nearly 400 x86_64 opcodes, some with over 20 variations, to produce the 11 instruction kernel shown in Figure 1. It is unlikely that Massalin's approach would scale to an instruction set of this size.

Denali[9] and Equality Saturation,[17] achieve improved scalability by only considering code sequences that are equivalent to the input sequence; candidates are explored through successive application of equality preserving transformations. Because both techniques are goal-directed, they dramatically improve the number of primitive instructions and the length of sequences that can be considered in practice. However, both also rely heavily on expert knowledge. It is unclear whether an expert would know to encode

This work was supported by NSF grant CCF-0915766 and the Army High Performance Computing Research Center.

Figure 1. Montgomery multiplication kernel from the OpenSSL RSA library. Compilations shown for `gcc -O3` (left) and a stochastic optimizer (right).

```
[r8:rdi] = rsi * [ecx:edx] + r8 + rdi

 1 # gcc -O3                        1 # STOKE
 2                                  2
 3 movq rsi, r9                     3 shlq 32, rcx
 4 movl ecx, ecx                    4 movl edx, edx
 5 shrq 32, rsi                     5 xorq rdx, rcx
 6 andl 0xffffffff, r9d             6 movq rcx, rax
 7 movq rcx, rax                    7 mulq rsi
 8 movl edx, edx                    8 addq r8, rdi
 9 imulq r9, rax                    9 adcq 0, rdx
10 imulq rdx, r9                   10 addq rdi, rax
11 imulq rsi, rdx                  11 adcq 0, rdx
12 imulq rsi, rcx                  12 movq rdx, r8
13 addq rdx, rax                   13 movq rax, rdi
14 jae .L0
15 movabsq 0x100000000, rdx
16 addq rdx, rcx
17 .L0:
18 movq rax, rsi
19 movq rax, rdx
20 shrq 32, rsi
21 salq 32, rdx
22 addq rsi, rcx
23 addq r9, rdx
24 adcq 0, rcx
25 addq r8, rdx
26 adcq 0, rcx
27 addq rdi, rdx
28 adcq 0, rcx
29 movq rcx, r8
30 movq rdx, rdi
```

the multiplication transformation shown in Figure 1, or whether a set of expert rules could ever cover the set of all possible interesting optimizations.

Bansal's peephole superoptimizer[2] automatically enumerates 32-bit x86 optimizations and stores the results in a database for later use. By exploiting symmetries between code sequences that are equivalent up to register renaming, Bansal was able to scale this approach to optimizations mapping code sequences of up to length 6 to sequences of up to length 3. The approach has the dual benefit of hiding the high cost of superoptimization by performing search once-and-for-all offline and eliminating the dependence on expert knowledge. To an extent, the use of a database also allows the system to overcome the low upper bound on instruction length through the repeated application of the optimizer along a sliding code window. Regardless, the kernel shown in Figure 1 has a property shared by many real world code sequences that no sequence of local optimizations will transform the code produced by `gcc -O3` into the code produced by STOKE.

Finally, Sketching[16] and Brahma[7] address the closely related component-based sequence synthesis problem. These systems rely on either a declarative specification, or a user-specified partial sequence, and operate on statements in simplified bit-vector calculi rather than directly on hardware instructions. Liang et al.[10] considers the task of learning code sequences from test cases alone, but at a similarly high level of abstraction. Although useful for synthesizing non-trivial code, the internal representations used by these systems preclude them from reasoning about the low-level performance properties of the code that they produce.

3. COST MINIMIZATION

Before describing our approach to x86_64 optimization, we discuss optimization as cost minimization in the abstract. To begin, we define a cost function with terms that balance the competing constraints of *transformation correctness* (eq(·)), and *performance improvement* (perf(·)). We refer to an input sequence as the *target* ($\mathcal{T}$) and a candidate compilation as a *rewrite* ($\mathcal{R}$), and say that a function $f(X; Y)$ takes inputs X and is defined in terms of Y

$$\text{cost}(\mathcal{R}; \mathcal{T}) = w_{\text{e}} \cdot \text{eq}(\mathcal{R}; \mathcal{T}) + w_{\text{p}} \cdot \text{perf}(\mathcal{R}) \qquad (1)$$

The eq(·) term measures the similarity of two code sequences and returns zero if and only if they are equal. For our purposes, code sequences are regarded as functions of registers and memory contents and we say they are equal if for all machine states that agree on the live inputs defined by the target, the two code sequences produce identical side effects on the live outputs defined by the target. Because optimization is undefined for ill-formed code sequences, it is unnecessary that eq(·) be defined for a target or rewrite that produce exceptional behavior. However nothing prevents us from doing so, and it would be straightforward to define eq(·) to preserve exceptional behavior as well.

In contrast, the perf(·) term quantifies the performance improvement of a rewrite; smaller values represent larger improvements. Crucially, the extent to which this term is accurate directly affects the quality of the resulting code.

Using this notation, the connection to optimization is straightforward. We say that an optimization is any of the set of rewrites for which the perf(·) term is improved, and the eq(·) term is zero

$$\{\, r \mid \text{perf}(r) \le \text{perf}(\mathcal{T}) \wedge \text{eq}(r; \mathcal{T}) = 0 \,\} \qquad (2)$$

Discovering these optimizations requires the use of a cost minimization procedure. However, in general we expect cost functions of this form to be highly irregular and not amenable to exact optimization techniques. The solution to this problem is to employ the common strategy of using an MCMC sampler. Although a complete discussion of the technique is beyond the scope of this article, we summarize the main ideas here.

MCMC sampling is a technique for drawing elements from a probability density function in direct proportion to its value: regions of higher probability are sampled more often than regions of low probability. When applied to cost minimization, it has two attractive properties. In the limit, the most samples will be taken from the minimum (optimal) values of a function. And in practice, well before that behavior is observed, it functions as an intelligent hill climbing method which is robust against irregular functions that are dense with local minima.

A common method[6] for transforming an arbitrary function such as cost(·) into a probability density function is shown below, where β is a constant and Z is a partition

function that normalizes the resulting distribution: Although computing Z is generally intractable, the Metropolis–Hastings algorithm is designed to explore density functions such as $p(\cdot)$ without having to compute Z directly[8]

$$p(\mathcal{R};\mathcal{T}) = \frac{1}{Z}\exp(-\beta\cdot\text{cost}(\mathcal{R};\mathcal{T})) \tag{3}$$

The basic idea is simple. The algorithm maintains a current rewrite $\mathcal{R}$ and proposes a modified rewrite $\mathcal{R}^*$. The *proposal* $\mathcal{R}^*$ is then either accepted or rejected. If it is accepted, $\mathcal{R}^*$ becomes the current rewrite. Otherwise another proposal based on $\mathcal{R}$ is generated. The algorithm iterates until its computational budget is exhausted, and as long as the proposals are *ergodic* (sufficient to transform any code sequence into any other through some sequence of applications) the algorithm will in the limit produce a sequence of samples distributed in proportion to their cost.

This global property depends on the local acceptance criteria for a proposal $\mathcal{R} \to \mathcal{R}^*$, which is governed by the Metropolis–Hastings acceptance probability shown below. We use the notation $q(\mathcal{R}^*|\mathcal{R})$ to represent the proposal distribution from which a new rewrite $\mathcal{R}^*$ is sampled given the current rewrite, $\mathcal{R}$. This distribution is key to a successful application of the algorithm. Empirically, the best results are obtained for a distribution which makes both local proposals that make minor modifications to $\mathcal{R}$ and global proposals that induce major changes

$$\alpha(\mathcal{R}\to\mathcal{R}^*;\mathcal{T}) = \min\left(1, \frac{p(\mathcal{R}^*;\mathcal{T})\cdot q(\mathcal{R}|\mathcal{R}^*)}{p(\mathcal{R};\mathcal{T})\cdot q(\mathcal{R}^*|\mathcal{R})}\right) \tag{4}$$

The important properties of the acceptance criteria are the following: If $\mathcal{R}^*$ is better (has a higher probability/lower cost) than $\mathcal{R}$, the proposal is always accepted. If $\mathcal{R}^*$ is worse (has a lower probability/higher cost) than $\mathcal{R}$, the proposal may still be accepted with a probability that decreases as a function of the ratio between $\mathcal{R}^*$ and $\mathcal{R}$. This property prevents search from becoming trapped in local minima while remaining less likely to accept a move that is much worse than available alternatives. In the event that the proposal distribution is *symmetric*, $q(\mathcal{R}^*|\mathcal{R}) = q(\mathcal{R}|\mathcal{R}^*)$, the acceptance probability can be reduced to the much simpler Metropolis ratio, which is computed directly from cost(·):

$$\begin{aligned}\alpha(\mathcal{R}\to\mathcal{R}^*;\mathcal{T}) &= \min\left(1, \frac{p(\mathcal{R}^*;\mathcal{T})}{p(\mathcal{R};\mathcal{T})}\right)\\ &= \min\big(1, \exp(k)\big)\\ \text{where } k &= -\beta\cdot(\text{cost}(\mathcal{R}^*;\mathcal{T}) - \text{cost}(\mathcal{R};\mathcal{T}))\end{aligned} \tag{5}$$

4. X86_64 OPTIMIZATION

We now address the practical details of applying cost minimization to x86_64 optimization. As x86_64 is arguably the most complex instance of a CISC architecture, we expect this discussion to generalize well to other architectures. A natural choice for the implementation of the eq(·) term is the use of a *symbolic validator* (val(·)),[4] and a *binary indicator function* ($\mathbf{1}(\cdot)$), which returns one if its argument is true, and zero otherwise

$$\text{eq}(\mathcal{R};\mathcal{T}) = 1 - \mathbf{1}(\text{val}(\mathcal{T},\mathcal{R})) \tag{6}$$

However, the total number of invocations that can be performed per second using current symbolic validator technology is quite low. For even modestly sized code sequences, it is well below 1000. Because MCMC sampling is effective only insofar as it is able to explore sufficiently large numbers of proposals, the repeated computation of Equation (6) would drive that number well below a useful threshold.

This observation motivates the definition of an approximation to the eq(·) term which is based on *test cases* (τ). Intuitively, we execute the proposal $\mathcal{R}^*$ on a set of inputs and measure "how close" the output matches the target for those same inputs by counting the number of bits that differ between live outputs (i.e., the Hamming distance). In addition to being much faster than using a theorem prover, this approximation of equivalence has the added advantage of producing a smoother landscape than the 0/1 output of a symbolic equality test; it provides a useful notion of "almost correct" that can help to guide search

$$\begin{aligned}\text{eq}'(\mathcal{R};\mathcal{T},\tau) = &\sum_{t\in\tau}\text{reg}(\mathcal{R};\mathcal{T},t) + \text{mem}(\mathcal{R};\mathcal{T},t)\\ &+\sum_{t\in\tau}\text{err}(\mathcal{R};\mathcal{T},t)\end{aligned} \tag{7}$$

In Equation (7), reg(·) is used to compare the *side effects* (eval(·)) that both functions produce on the *live register outputs* (ρ) defined by the target. These outputs can include general purpose, SSE, and condition registers. reg(·) computes the number of bits that the results differ by using the *population count function* (pop(·)), which returns the number of 1-bits in the 64-bit representation of an integer

$$\text{reg}(\mathcal{R};\mathcal{T},t) = \sum_{r\in\rho}\text{pop}(\text{eval}(\mathcal{T},r)\oplus\text{eval}(\mathcal{R},r)) \tag{8}$$

For brevity, we omit the definition of mem(·), which is analogous. The remaining term, err(·), is used to distinguish code sequences that exhibit undefined behavior, by counting and then penalizing the number of segfaults, floating-point exceptions, and reads from undefined memory or registers, that occur during execution of a rewrite. We note that sigsegv(·) is defined in terms of $\mathcal{T}$, which determines the set of addresses that may be successfully dereferenced by a rewrite for a particular test case. Rewrites must be run in a sandbox to ensure that this behavior can be detected safely at runtime. The extension to additional types of exceptions is straightforward

$$\begin{aligned}\text{err}(\mathcal{R};\mathcal{T},t) = &\ w_{ss}\cdot\text{sigsegv}(\mathcal{R};\mathcal{T},t)\\ &+ w_{sf}\cdot\text{sigfpe}(\mathcal{R};t)\\ &+ w_{ur}\cdot\text{undef}(\mathcal{R};t)\end{aligned} \tag{9}$$

The evaluation of eq′(·) may be implemented either by JIT compilation, or the use of a hardware emulator. In our experiments (Section 8) we have chosen the former, and shown the ability to dispatch test case evaluations at a rate

of between 1 and 10 million per second. Using this implementation, we define an optimized method for computing eq(·), which achieves sufficient throughput to be useful for MCMC sampling

$$\mathrm{eq}^*(\mathcal{R};\mathcal{T},\tau)=\begin{cases}\mathrm{eq}(\mathcal{R};\mathcal{T}), & \text{if } \mathrm{eq}'(\mathcal{R};\mathcal{T},\tau)=0\\ \mathrm{eq}'(\mathcal{R};\mathcal{T},\tau), & \text{otherwise}\end{cases}\qquad(10)$$

Besides improved performance, Equation (10) has two desirable properties. First, failed computations of eq(·) will produce a counterexample test case[4] that can be used to refine τ. Although doing so changes the search space defined by cost(·), in practice the number of failed validations that are required to produce a robust set of test cases that accurately predict correctness is quite low. Second, as remarked above, it improves the search space by smoothly interpolating between correct and incorrect code sequences.

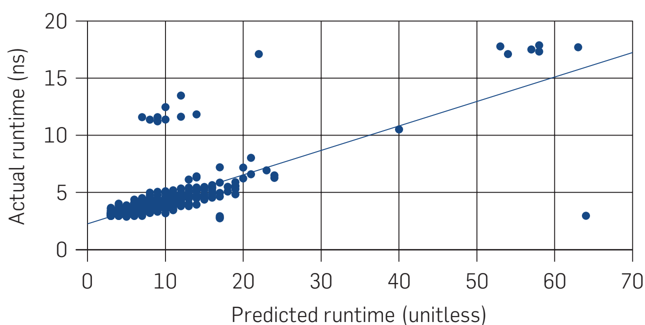
Similar considerations apply to the implementation of the perf(·) term. Although it seems natural to JIT compile both target and rewrite and compare runtimes, the amount of time required to execute a code sequence sufficiently many times to eliminate transient effects is prohibitively expensive. To account for this, we define a simple heuristic for approximating runtime which is based on a static approximation of the *average latencies* (lat(·)), of instructions

$$\mathrm{perf}(\mathcal{R})=\sum_{i\in\mathrm{inst}(\mathcal{R})}\mathrm{lat}(i)\qquad(11)$$

Figure 2 shows a reasonable correlation between this heuristic and actual runtimes for a representative corpus of code sequences. Outliers are characterized either by disproportionately high instruction level parallelism at the micro-op level or inconsistent memory access times. A more accurate model of the higher order performance effects introduced by a modern CISC processor is feasible if tedious to construct and would likely be necessary for more complex code sequences.

Regardless, this approximation is sufficient for the benchmarks that we consider (Section 8). Errors that result from this approach are addressed by recomputing perf(·) using the JIT compilation method as a postprocessing step. During search, we record the n lowest cost programs produced by MCMC sampling, rerank each based on their actual runtimes, and return the best result.

Figure 2. Predicted versus observed runtimes for selected code sequences. Outliers are characterized by instruction level parallelism and memory effects.



Finally, there is the implementation of MCMC sampling for x86_64 optimization. Rewrites are represented as loop-free sequences of instructions of length ℓ, where a distinguished token (UNUSED) is used to represent unused instruction slots. This simplification to sequences of bounded length is crucial, as it places a constant value on the dimensionality of the resulting search space.[1] The proposal distribution $q(\cdot)$ chooses from four possible moves: the first two minor and the last two major:

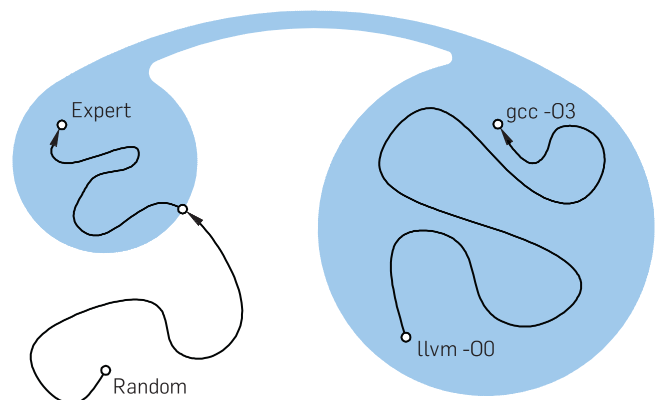
- **Opcode.** An instruction is randomly selected, and its opcode is replaced by a random opcode.
- **Operand.** An instruction is randomly selected and one of its operands is replaced by a random operand.
- **Swap.** Two lines of code are randomly selected and interchanged.
- **Instruction.** An instruction is randomly selected and replaced either by a random instruction or the UNUSED token. Proposing UNUSED corresponds to deleting an instruction, and replacing UNUSED by an instruction corresponds to inserting an instruction.

These moves satisfy the ergodicity property described in Section 3: any code sequence can be transformed into any other through repeated application of instruction moves. These moves also satisfy the symmetry property, and allow the use of Equation (5). To see why, note that the probabilities of performing all four moves are equal to the probabilities of undoing the transformations they produce using a move of the same type: opcode and operand moves are constrained to sample from identical equivalence classes before and after acceptance, and swap and instruction moves are unconstrained in either direction.

5. SEARCH STRATEGIES

An early version of the implementation described above was able to transform `llvm -O0` code into the equivalent of `gcc -O3` code, but was unable to produce results

Figure 3. Search space for the Montgomery multiplication benchmark: `O0` and `O3` codes are densely connected, whereas expert code is reachable only by an extremely low probability path.



that were competitive with expert hand-written code. The reason is suggested by Figure 3, which abstractly depicts the search space for the Montgomery multiplication benchmark shown in Figure 1. For loop-free code sequences, `llvm -O0` and `gcc -O3` differ primarily in stack use and instruction selection, but otherwise produce algorithmically similar results. Compilers are generally designed to compose many small local transformations: dead code elimination deletes an instruction, constant propagation changes a register to an immediate, and strength reduction replaces a multiplication by an add. Sequences of local optimizations such as these correspond to regions of equivalent code sequences that are densely connected by very short sequences of moves (often just one) and easily traversed by local search methods. Beginning from `llvm -O0` code, MCMC sampling will quickly improve local inefficiencies one by one and hill climb its way to the equivalent of `gcc -O3` code.

The code discovered by STOKE occupies an entirely different region of the search space. As remarked earlier, it represents a completely distinct algorithm for implementing the kernel at the assembly level. The only method for a local search procedure to produce code of this form from compiler generated code is to traverse the extremely low probability path that builds the code in place next to the original (all the while increasing its cost) only to delete the original code at the very end.

Although MCMC sampling is guaranteed to traverse this path in the limit, the likelihood of it doing so in any reasonable amount of time is so low as to be useless in practice. This observation motivates the division of cost minimization into two phases:

- A **synthesis** phase focused solely on correctness, which attempts to locate regions of equivalent code sequences that are distinct from the region occupied by the target.
- An **optimization** phase focused on performance, which searches for the fastest sequence within each of those regions.

These phases share the same implementation and differ only in starting point and acceptance function. Synthesis begins with a random code sequence, while optimization begins from a code sequence that is known to be equivalent to the target. Synthesis ignores the perf(·) term and uses Equation (10) as its cost function, whereas optimization uses both terms, which allows it to improve performance while also experimenting with "shortcuts" that (temporarily) violate correctness.

It is perhaps unintuitive that synthesis should be able to produce a correct rewrite from such an enormous search space in a tractable amount of time. In our experience, synthesis is effective precisely when it is possible to discover portions of a correct rewrite incrementally, rather than all at once. Figure 4 compares cost over time against the percentage of instructions that appear in the final rewrite for the Montgomery multiplication benchmark. As synthesis proceeds, the percentage of correct code increases in inverse proportion to the value of the cost function.

While this is encouraging and there are many code sequences that can be synthesized in pieces, there are many that cannot. Fortunately, even when synthesis fails, optimization is still possible. It must simply proceed only from the region occupied by the target as a starting point.

6. SEARCH OPTIMIZATIONS

Equation (10) is sufficiently fast for MCMC sampling, however its performance can be further improved. As described above, the eq* (·) term is computed by executing a proposal on test cases, noting the ratio in total cost with the current rewrite, and then sampling a random variable to decide whether to accept the proposal. However, by first sampling p, and then computing the maximum ratio that the algorithm will accept for that value, it is possible to terminate the evaluation of test cases as soon as that bound is exceeded and the proposal is guaranteed to be rejected

$$\begin{aligned} p &< \alpha(\mathcal{R} \to \mathcal{R}^*; \mathcal{T}) \\ &< \min(1, \exp(k)) \\ \text{where } k &= -\beta \cdot (\mathrm{cost}(\mathcal{R}^*; \mathcal{T}) - \mathrm{cost}(\mathcal{R}; \mathcal{T})) \\ \mathrm{eq}^*(\mathcal{R}^*; \mathcal{T}, \tau) &\le \mathrm{cost}(\mathcal{R}; \mathcal{T}, \tau) - \mathrm{perf}(\mathcal{R}^*) - \frac{\log(p)}{\beta} \end{aligned} \tag{12}$$

Figure 4. Cost over time versus percentage of instructions that appear in the final zero-cost rewrite for the Montgomery multiplication synthesis benchmark.

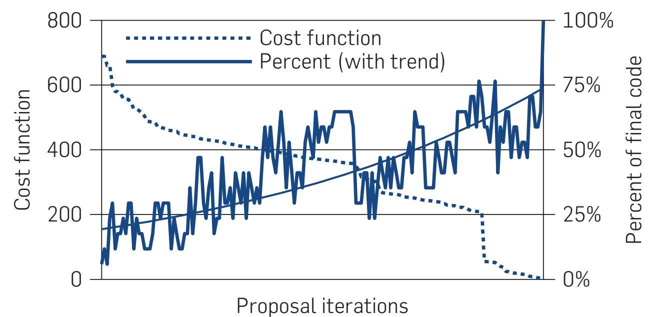


Figure 5. Proposals evaluated per second versus test cases evaluated prior to early termination, for the Montgomery multiplication synthesis benchmark. Cost function shown for reference.

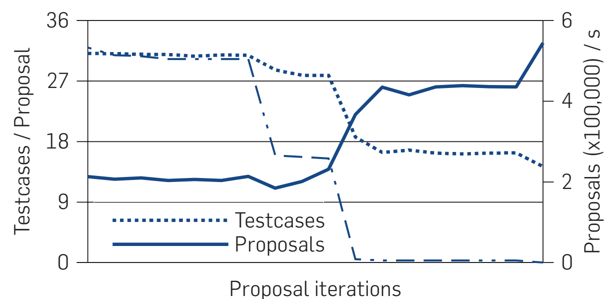


Figure 5 shows the result of applying this optimization during synthesis for the Montgomery multiplication benchmark. As the value of the cost function decreases, so too does the average number of test cases that must be evaluated prior to early termination. This produces a considerable increase in the number of proposals evaluated per second, which at peak falls between 100,000 and 1 million.

A more important improvement stems from the observation that the definition of reg(·) in Equation (8) is unnecessarily strict. An example is shown in Figure 6 for a target in which register al is live out. A rewrite that produces the inverse of the desired value in al is assigned the maximum possible cost in spite of the fact that it produces the correct value, only in the incorrect location: dl. We can improve this term by rewarding rewrites that produce correct (or nearly correct) values in incorrect locations. The relaxed definition shown below examines all registers of equivalent *bit-width* (bw(·)), selects the one that most closely matches the value of the target register, and assigns a small *misalignment penalty* (w_m) if the selected register differs from the original. Using this definition, the rewrite is assigned a cost of just w_m

$$\begin{aligned} \text{reg}'(\mathcal{R};\mathcal{T},\tau) &= \sum_{r\in\rho} \min_{r'\in \text{bw}(r)} \text{R}(r,r';\tau) \\ \text{R}(r,r';\tau) &= \text{pop}(\text{eval}(\mathcal{T},r)\oplus \text{val}(\mathcal{R},r')) \\ &\quad + w_m \cdot \mathbf{1}(r\neq r') \end{aligned} \tag{13}$$

Although it is possible to relax the definition of memory equality analogously, the time required to compute this term grows quadratically with the size of the target's memory footprint. Although this approach suffices for our experiments, a more efficient implementation is necessary for more complex code sequences.

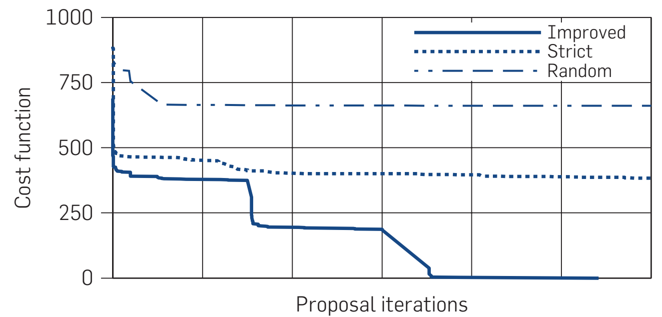
Figure 7 shows the result of using these improved definitions during synthesis for the Montgomery multiplication benchmark. In the amount of time required for the relaxed cost function to converge, the original strict version obtains a minimum cost that is only slightly superior to a purely random search. The dramatic improvement can be explained as an implicit parallelization of the search procedure. Accepting correct values in arbitrary locations allows a rewrite to simultaneously explore as many alternate computations as can fit within a code sequence of length ℓ.

Figure 6. Strict versus relaxed equality functions for a target in which ax is live out and the correct result appears in an incorrect location.

	al	bl	cl	dl
$\text{eval}(\mathcal{T},\cdot)$	1111	0000	0000	0000
$\text{eval}(\mathcal{R},\cdot)$	0000	1000	1100	1111
$\delta = \text{eval}(\mathcal{T},\text{al}) \oplus \text{eval}(\mathcal{R},\cdot)$	1111	0111	0011	0000
$\text{pop}(\delta)$	4	3	2	0
$w_m \cdot \mathbf{1}(\text{al} \neq \cdot)$	0	1	1	1

$$\begin{aligned} \text{reg}(\mathcal{T},\mathcal{R},\tau) &= 4 \\ \text{reg}'(\mathcal{T},\mathcal{R},\tau) &= \min(4, 3+1, 2+1, 1) \\ &= 1 \end{aligned}$$

Figure 7. Strict versus relaxed cost functions for the Montgomery multiplication synthesis benchmark. Random search results shown for reference.



7. STOKE

STOKE is a prototype implementation of the ideas described above.[a] STOKE runs a binary created by a standard compiler (in our experiments, `llvm -O0`) under instrumentation to generate test cases for a loop-free target of interest and broadcasts both target and test cases to a set of synthesis threads. After a fixed time, those threads report back with a set of validated rewrites, and a new set of optimization threads are used to improve those rewrites. Of the results, the top 20% most performant are re-ranked based on actual runtime, and the best is returned to the user. STOKE generates 32 test cases for each target and synthesis and optimization are executed in parallel on an 8 core 3.5 GHz Intel i7–4770K with a computational budget of 15 min.

STOKE generates test cases using Intel's PinTool.[11] It executes the binary provided by the user, and for every invocation of the target records both initial register state and the set of values dereferenced from memory. For each test case, the set of addresses dereferenced by the target are used to define the sandbox in which candidate rewrites are executed. Dereferences of invalid addresses are trapped and replaced by instructions that produce a constant zero value; reads from undefined locations and floating-point exceptions are handled analogously.

STOKE uses a sound procedure to validate the equality of loop-free code sequences.[2] Both target and rewrite are converted into SMT formulae in the quantifier free theory of bit-vector arithmetic used by Z3,[5] producing a query that asks whether both sequences produce the same side effects on live outputs when executed from the same initial machine state. Depending on type, registers are modeled as between 8- and 256-bit vectors, and memory is modeled as two vectors: a 64-bit address and an 8-bit value (x86_64 is byte addressable).

STOKE asserts the constraint that both sequences agree on the initial machine state with respect to the live inputs defined by the target. For each instruction in the target, it also asserts a constraint that encodes the transformation it represents on the machine state and chains these together to

[a] See https://stoke.stanford.edu.

produce a constraint that describes the state of the live outputs defined by the target. An analogous set of constraints are asserted for the rewrite, and for all pairs of memory accesses at addresses $\mathbf{addr}_1$ and $\mathbf{addr}_2$, STOKE adds an additional constraint that relates their values: $\mathbf{addr}_1 = \mathbf{addr}_2 \Rightarrow \mathbf{val}_1 = \mathbf{val}_2$.

Using these constraints, STOKE asks Z3 if there exists an initial machine state that causes the two sequences to produce different results. If the answer is "no," then the sequences are determined to be equal. If the answer is "yes," then Z3 produces a witness that is converted to a new test case.

STOKE makes two simplifying assumptions to keep runtimes tractable. It assumes that stack addresses are represented exclusively as constant offsets from rsp. This allows stack locations to be treated as named variables in `llvm -O0` code, which exhibits heavy stack traffic. Additionally, it treats 64-bit multiplication and division as uninterpreted functions which are constrained by a small set of special-purpose axioms. Whereas Z3 diverges when reasoning about two or more such operations, our benchmarks contain up to four per sequence.

8. EVALUATION

In addition to the Montgomery multiplication kernel, we evaluate STOKE on benchmarks drawn from both the literature and high-performance codes. Performance improvements and runtimes are summarized in Figure 8. Beginning from binaries compiled using `llvm -O0`, STOKE consistently produces rewrites that match the performance of code produced by `gcc` and `icc` (the two compilers produce essentially identical results). In several cases, the rewrites are comparable in performance to handwritten assembly.

Gulwani et al.[7] identifies Hacker's Delight[18]—a collection of techniques for encoding complex algorithms as small loop-free code sequences—as a source of benchmarks (p01–p25) for program synthesis and optimization. For brevity, we focus only on a kernel for which STOKE discovers an algorithmically distinct rewrite. Figure 9 shows the "Cycle Through 3 Values" benchmark, which takes an input x and transforms it to the next value in the sequence ⟨a, b, c⟩. The most natural implementation of this function is a sequence of conditional assignments, but for ISAs without the required intrinsics, the implementation shown is cheaper than one that uses branches. For x86_64, which has conditional move intrinsics, this is an instance of premature optimization. While STOKE is able to rediscover the optimal implementation, `gcc` and `icc` transcribe the code as written.

Single-precision Alpha X Plus Y (SAXPY) is a level 1 vector operation in the Basic Linear Algebra Subsystems Library.[3] It makes heavy use of heap accesses and presents the opportunity to use vector intrinsics. To expose this property, our integer implementation is unrolled four times by hand, as shown in Figure 10. Despite annotation to indicate that *x* and *y* are aligned and do not alias, neither production compiler is able to produce vectorized code. STOKE on the other hand, is able to discover the ideal implementation.

Figure 8. Speedups over `llvm -O0` versus STOKE runtimes. Benchmarks for which an algorithmically distinct rewrite was discovered are shown in bold; synthesis timeouts are annotated with a –.

	Speedup (×100%)		Runtime (s)	
	gcc/icc -O3	STOKE	Synth.	Opt.
p01	1.60	1.60	0.15	3.05
p02	1.60	1.60	0.16	3.14
p03	1.60	1.60	0.34	3.45
p04	1.60	1.60	2.33	3.55
p05	1.60	1.60	0.47	3.24
p06	1.60	1.60	1.57	6.26
p07	2.00	2.00	1.34	3.10
p08	2.20	2.20	0.63	3.24
p09	1.20	1.20	0.26	3.21
p10	1.80	1.80	7.49	3.61
p11	1.50	1.50	0.87	3.05
p12	1.50	1.50	5.29	3.34
p13	3.25	3.25	0.22	3.08
p14	1.86	1.86	1.43	3.07
p15	2.14	2.14	2.83	3.17
p16	1.80	1.80	6.86	4.62
p17	2.60	2.60	10.65	4.45
p18	2.44	2.50	0.30	4.04
p19	1.93	1.97	-	18.37
p20	1.78	1.78	-	36.72
p21	1.62	1.65	6.97	4.96
p22	3.38	3.41	0.02	4.02
p23	5.53	6.32	0.13	4.36
p24	4.67	4.47	-	48.90
p25	2.17	2.34	3.29	4.43
mont mul	2.84	4.54	319.03	111.64
linked list	1.10	1.09	3.94	8.08
SAXPY	1.82	2.46	10.35	6.66

Figure 9. Cycling Through 3 Values benchmark.

```
int p21(int x, int a, int b, int c) {
  return ((-(x == c)) & (a ^ c)) ^
         ((-(x == a)) & (b ^ c)) ^ c;
}
```

```
 1 # gcc -O3
 2
 3 movl edx, eax
 4 xorl edx, edx
 5 xorl ecx, eax
 6 cmpl esi, edi
 7 sete dl
 8 negl edx
 9 andl edx, eax
10 xorl edx, edx
11 xorl ecx, eax
12 cmpl ecx, edi
13 sete dl
14 xorl ecx, esi
15 negl edx
16 andl esi, edx
17 xorl edx, eax
```

```
1 # STOKE
2
3 cmpl edi, ecx
4 cmovel esi, ecx
5 xorl edi, esi
6 cmovel edx, ecx
7 movq rcx, rax
```

In closing, we note that STOKE is not without its limitations. Figure 11 shows the Linked List Traversal benchmark of Bansal and Aiken.[2] The code iterates over a list of integers and doubles each element. Because STOKE is only able to

Figure 10. SAXPY benchmark.

```
void SAXPY(int* x, int* y, int a) {
  x[i]   = a * x[i]   + y[i];
  x[i+1] = a * x[i+1] + y[i+1];
  x[i+2] = a * x[i+2] + y[i+2];
  x[i+3] = a * x[i+3] + y[i+3];
}
```

```
 1 # gcc -O3                  1 # STOKE
 2                            2
 3 movslq ecx,rcx             3 movd edi,xmm0
 4 leaq (rsi,rcx,4),r8        4 shufps 0,xmm0,xmm0
 5 leaq 1(rcx),r9             5 movups (rsi,rcx,4),xmm1
 6 movl (r8),eax              6 pmullw xmm1,xmm0
 7 imull edi,eax              7 movups (rdx,rcx,4),xmm1
 8 addl (rdx,rcx,4),eax       8 paddw xmm1,xmm0
 9 movl eax,(r8)              9 movups xmm0,(rsi,rcx,4)
10 leaq (rsi,r9,4),r8
11 movl (r8),eax
12 imull edi,eax
13 addl (rdx,r9,4),eax
14 leaq 2(rcx),r9
15 addq 3,rcx
16 movl eax,(r8)
17 leaq (rsi,r9,4),r8
18 movl (r8),eax
19 imull edi,eax
20 addl (rdx,r9,4),eax
21 movl eax,(r8)
22 leaq (rsi,rcx,4),rax
23 imull (rax),edi
24 addl (rdx,rcx,4),edi
25 movl edi,(rax)
```

Figure 11. Linked List Traversal benchmark.

```
while (head != 0) {
  head->val *= 2;
  head = head->next;
}
```

```
1 # gcc -O3               1 # STOKE
2                         2
3 movq -8(rsp), rdi       3 .L1:
4 .L1:                    4 movq -8(rsp), rdi
5 sall (rdi)              5 sall (rdi)
6 movq 8(rdi), rdi        6 movq 8(rdi), rdi
7 .L2:                    7 movq rdi, -8(rsp)
8 testq rdi, rdi          8 .L2:
9 jne .L1                 9 movq -8(rsp), rdi
                         10 testq rdi, rdi
                         11 jne .L1
```

reason about loop-free code—recent work has explored solutions to this problem[15]—it fails to eliminate the stack movement at the beginning of each iteration. STOKE is also unable to synthesize a rewrite for three of the Hacker's Delight benchmarks. Nonetheless, using its optimization phase alone it is able to discover rewrites that perform comparably to the production compiler code.

9. CONCLUSION

We have shown a new approach to program optimization based on stochastic search. Compared to a traditional compiler, which factors optimization into a sequence of small independently solvable subproblems, our approach uses cost minimization and considers the competing constraints of transformation correctness and performance improvement simultaneously. Although the method sacrifices completeness, it is competitive with production compilers and has been demonstrated capable of producing code that can out-perform expert handwritten assembly.

This article is based on work that was originally published in 2013. Since then, STOKE has undergone substantial improvement; the updated results that appear here were produced using the current implementation and improve on the original by over an order of magnitude. Interested readers are encouraged to consult the original text *Stochastic Superoptimization*[13] and those that followed.

Data-Driven Equivalence Checking[15] describes extensions to STOKE that enable the optimization of code sequences with non-trivial control flow. It defines a sound method for guaranteeing that the optimizations produced by STOKE are correct for all possible inputs even in the presence of loops. The method is based on a data-driven algorithm that observes test case executions and automatically infers invariants for producing inductive proofs of equivalence. The prototype implementation is the first sound equivalence checker for loops written in x86_64 assembly.

Stochastic Optimization of Floating-Point Programs with Tunable Precision[14] describes extensions to STOKE that enable the optimization of floating-point code sequences. By modifying the definition of the eq(·) term to account for relaxed constraints on floating-point equality STOKE is able to generate reduced precision implementations of Intel's handwritten C numeric library that are up to six times faster than the original, and achieve end-to-end speedups of over 30% on high-performance applications that can tolerate a loss of precision while still remaining correct. Because these optimizations are mostly not amenable to formal verification using the current state of the art, the paper describes a search technique for characterizing maximum error. **C**

References

1. Andrieu, C., de Freitas, N., Doucet, A., Jordan, M.I. An introduction to MCMC for machine learning. *Machine Learning 50*, 1–2 (2003), 5–43.
2. Bansal, S., Aiken, A. Binary translation using peephole superoptimizers. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008.* R. Draves and R. van Renesse, eds. (San Diego, CA, USA, December 8–10, 2008). USENIX Association, 177–192.
3. Blackford, L.S., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K., Whaley, R.C. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw. 28*, 2 (2002), 135–151.
4. Cadar, C., Dunbar, D., Engler, D.R. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008.* R. Draves and R. van Renesse, eds. (San Diego, CA, USA, December 8–10, 2008). USENIX Association, 209–224.
5. Ganesh, V., Dill, D.L. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV) 2007.* W. Damm and H. Hermanns, eds. Volume of 4590 *Lecture Notes in Computer Science* (Berlin, Germany, July 3–7, 2007), Springer, 519–531.
6. Gilks, W.R. *Markov Chain Monte Carlo in Practice.* Chapman and Hall/CRC, 1999.
7. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN*

Conference on Programming Language Design and Implementation (PLDI) 2011. M. W. Hall and D. A. Padua, eds. (San Jose, CA, USA, June 4–8, 2011). ACM, 62–73.

8. Hastings, W.K. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika 57*, 1 (1970), 97–109.
9. Joshi, R., Nelson, G., Randall, K.H. Denali: A goal-directed superoptimizer. *In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany, 2002). ACM, New York, NY, USA, 304–314.
10. Liang, P., Jordan, M.I., Klein, D. Learning programs: A hierarchical Bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. J. Fürnkranz and T. Joachims, eds. (Haifa, Israel, June 21–24, 2010). Omnipress, 639–646.
11. Luk, C.-K., Cohn, R.S., Muth, R., Patil, H., Klauser, A., Lowney, P.G., Wallace, S., Reddi, V.J., Hazelwood, K.M. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA, 2005). ACM, New York, NY, USA, 190–200.
12. Massalin, H. Superoptimizer – A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II).* R. H. Katz, ed. (Palo Alto, CA, USA, October 5–8, 1987). ACM Press, 122–126.
13. Schkufza, E., Sharma, R., Aiken, A. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS'13.* V. Sarkar and R. Bodík, eds. (Houston, TX, USA, March 16–20, 2013). ACM, 305–316.
14. Schkufza, E., Sharma, R., Aiken, A. Stochastic optimization of floating-point programs with tunable precision. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14.* M. F. P. O'Boyle and K. Pingali, eds. (Edinburgh, United Kingdom, June 09–11, 2014). ACM.
15. Sharma, R., Schkufza, E., Churchill, B.R., Aiken, A. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, Part of SPLASH 2013.* A. L. Hosking, P. Th. Eugster, and C. V. Lopes, eds. (Indianapolis, IN, USA, October 26–31, 2013). ACM, 391–406.
16. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006.* J. P. Shen and M. Martonosi, eds. (San Jose, CA, USA, October 21–25, 2006). ACM, 404–415.
17. Tate, R., Stepp, M., Tatlock, Z., Lerner, S. Equality saturation: A new approach to optimization. *Logical Methods Comput. Sci. 7*, 1 (2011).
18. Warren, H.S. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

Eric Schkufza, Rahul Sharma, and Alex Aiken ({eschkufz, sharmar, aiken}@cs.stanford.edu), Stanford University, Stanford, CA.

© 2016 ACM 0001-0782/16/02 $15.00

ACM Transactions on Parallel Computing

Solutions to Complex Issues in Parallelism

Editor-in-Chief: Phillip B. Gibbons, Intel Labs, Pittsburgh, USA

ACM Transactions on Parallel Computing (TOPC) is a forum for novel and innovative work on all aspects of parallel computing, including foundational and theoretical aspects, systems, languages, architectures, tools, and applications. It will address all classes of parallel-processing platforms including concurrent, multithreaded, multicore, accelerated, multiprocessor, clusters, and supercomputers.

Subject Areas

- Parallel Programming Languages and Models
- Parallel System Software
- Parallel Architectures
- Parallel Algorithms and Theory
- Parallel Applications
- Tools for Parallel Computing

Association for Computing Machinery

Advancing Computing as a Science & Profession

For further information or to submit your manuscript, visit topc.acm.org

Subscribe at www.acm.org/subscribe