Stochastic Superoptimization

Eric Schkufza
Stanford University
eschkufz@cs.stanford.edu

Rahul Sharma
Stanford University
sharmar@cs.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Abstract

We formulate the loop-free binary superoptimization task as a stochastic search problem. The competing constraints of transformation correctness and performance improvement are encoded as terms in a cost function, and a Markov Chain Monte Carlo sampler is used to rapidly explore the space of all possible programs to find one that is an optimization of a given target program. Although our method sacrifices completeness, the scope of programs we are able to consider, and the resulting quality of the programs that we produce, far exceed those of existing superoptimizers. Beginning from binaries compiled by `llvm -O0` for 64-bit x86, our prototype implementation, STOKE, is able to produce programs which either match or outperform the code produced by `gcc -O3`, `icc -O3`, and in some cases, expert handwritten assembly.

Categories and Subject Descriptors D.1.2 [*Automatic Programming*]: Program Synthesis; D.1.2 [*Automatic Programming*]: Program Transformation; D.1.2 [*Automatic Programming*]: Program Verification; D.3.4 [*Processors*]: Optimization

General Terms Performance, Verification

Keywords 64-bit; x86; x86-64; Binary; Markov Chain Monte Carlo; MCMC; Stochastic Search; Superoptimization; SMT

1. Introduction

For many application domains there is considerable value in producing the most performant code possible. Unfortunately, the traditional structure of a compiler's optimization phase is often ill-suited to this task. Attempting to factor the optimization problem into a collection of small subproblems that can be solved independently, although suitable for generating consistently good code, leads to the well-known phase ordering problem. 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 programs. In contrast to a traditional compiler, which uses performance constraints to drive the generation of a single program, these systems consider multiple programs and then select the one that is best able to satisfy those constraints. Solutions range from the explicit enumeration of a class of programs that can be formed using a large executable hardware instruction set [2] to implicit enumeration through symbolic theorem proving techniques of programs over some restricted register transaction language [9, 11, 14].

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

```
# gcc -O3

.L0:
  movq rsi, r9
  movl ecx, ecx
  shrq 32, rsi
  andl 0xffffffff, r9d
  movq rcx, rax
  movl edx, edx
  imulq r9, rax
  imulq rdx, r9
  imulq rsi, rdx
  imulq rsi, rcx
  addq rdx, rax
  jae .L2
  movabsq 0x100000000, rdx
  addq rdx, rcx
.L2:
  movq rax, rsi
  movq rax, rdx
  shrq 32, rsi
  salq 32, rdx
  addq rsi, rcx
  addq r9, rdx
  adcq 0, rcx
  addq r8, rdx
  adcq 0, rcx
  addq rdi, rdx
  adcq 0, rcx
  movq rcx, r8
  movq rdx, rdi
```

```
# STOKE

.L0:
  shlq 32, rcx
  movl edx, edx
  xorq rdx, rcx
  movq rcx, rax
  mulq rsi
  addq r8, rdi
  adcq 0, rdx
  addq rdi, rax
  adcq 0, rdx
  movq rdx, r8
  movq rax, rdi
```

Figure 1. Montgomery multiplication kernel from the OpenSSL big number library, compiled by `gcc -O3` (left) and STOKE (right). The STOKE code is 16 lines shorter, 1.6x faster, and slightly faster than expert handwritten assembly.

An attractive feature of these systems is completeness: If a program exists that meets the desired constraints, that program will be found. Unfortunately, completeness also places limitations on the space of programs that can realistically be considered. Because of the huge number of programs involved, explicit enumeration-based techniques are limited to programs of up to some fixed length which is currently well below the threshold at which many interesting optimizations take place. Implicit enumeration techniques can overcome this limitation, but at the cost of expert-written rules for shrinking the search space. The resulting optimizations are as good, but no better, than the quality of the rules written by an expert.

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

We overcome these limitations by taking a different approach based on incomplete search: the competing requirements of correctness and speed are defined as terms in a cost function over the complex search space of all loop-free executable hardware instruction sequences, and the program optimization task is formulated as a cost minimization problem. Although the resulting search space is highly irregular and not amenable to exact optimization techniques, 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 programs.

Although our technique sacrifices completeness by trading systematic enumeration for stochastic search, we nonetheless dramatically increase the space of programs that our system is able to consider while simultaneously improving the quality of the resulting code. Consider the example shown in Figure 1, the Montgomery multiplication kernel from the OpenSSL big number library for arbitrary precision integer arithmetic. Beginning from a binary compiled by `llvm -O0` (116 lines, not shown), we produce a code sequence which is 16 lines shorter and 1.6 times faster than the one produced by `gcc -O3`, and even slightly faster than the expert handwritten assembly included in the OpenSSL repository. The performance improvement results primarily from the use of a different assembly level algorithm than the one given in the original code which is beyond the dataflow preserving algebraic transformations of a traditional compiler. The code is both automatically discovered and verified to be equivalent to the original. To the best of our knowledge, it is truly optimal: it is the fastest program for this function written in the 64-bit x86 (x86-64) instruction set.

Our work makes a number of contributions that have not previously been demonstrated. The remainder of this paper explores each in turn. Section 2 summarizes previous work in superoptimization and discusses its limitations. Section 3 presents a mathematical formalism for transforming the program optimization task into a stochastic cost minimization problem. Section 4 discusses how that theory is applied in a system for optimizing the runtime performance of x86-64 binaries, and Section 5 describes our prototype implementation, STOKE. Finally, Section 6 evaluates STOKE on a set of benchmarks drawn from cryptography, linear algebra, and low-level programming, and shows that STOKE is able to produce code that either matches or outperforms the code produced by production compilers.

2. Related Work

Previous approaches to superoptimization have focused on the exploration of all possibilities within some restricted class of programs. Although these systems have been demonstrated to be quite effective within certain domains, their general applicability remains limited. We discuss these limitations in the context of the Montgomery multiplication kernel shown in Figure 1.

The structure of the code is as follows: Two 32-bit values, ecx and edx, are concatenated and multiplied by the 64-bit value rsi to produce a 128-bit product. Two more 64-bit values, rdi and r8 are added to that product, and the result is split between two registers, r8 and rdi. The primary source of optimization is best highlighted by comparison. The code produced by `gcc -O3`, Figure 1 (left), performs the 128-bit multiplication as four 64-bit multiplications and then combines the results; the rewrite produced by STOKE, Figure 1 (right), uses a hardware intrinsic which requires that the inputs first be permuted and then moved to distinguished register locations so that the multiplication may be performed in a single step. The odd looking move on line 5 produces the non-obvious but necessary side effect of zeroing the upper 32 bits of `rdx`.

Massalin's original paper on superoptimization [14] describes a system that explicitly enumerates sequences of code of increasing length and selects the first such code identical to the input program on a set of testcases. Massalin reports being able to optimize instruction sequences of up to length 12. However to do so, it is necessary to restrict the set of enumerable opcodes to between 10 and 15. In contrast, STOKE used 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 magnitude.

Denali [11], and the more recent Equality Saturation technique [18], attempt to gain scalability by only considering programs that are known to be equal to the input program. Candidate programs are explored through successive application of equality preserving transformation axioms. Because these techniques are goal-directed, they dramatically improve both the number of primitive instructions and the length of programs that can realistically be considered. However, both also rely heavily on expert knowledge. It is unclear whether an expert would know a priori to encode an equality axiom defining the multiplication transformation shown in Figure 1, or more generally, whether a set of expert written rules could ever cover the set of all interesting program optimizations. It is nonetheless worth noting that these techniques can to a certain extent deal with loop optimizations, while other techniques, including our own, are limited to loop-free code.

Bansal [2] describes a system that automatically enumerates 32-bit x86 superoptimizations and stores the results in a database for later use. By exploiting symmetries between programs that are equivalent up to register renaming, Bansal is able to scale this method to optimizations that take input code sequences of at most length 6 and produce code sequences of at most length 3. This approach has the dual benefit of hiding the high cost of superoptimization by performing a search once and for all offline and eliminating the dependence on expert knowledge. To some extent, the low cost of performing a database query 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. However, the multiplication kernel shown in Figure 1 has the interesting property shared by many real world programs that no sequence of short superoptimizations will transform the code produced by `gcc -O3` into the code produced by STOKE. We follow Bansal's approach in overall system architecture by using testcases to help classify programs as promising or not and submitting the most promising candidates to a verification engine to prove or refute their correctness.

Sketching [17] and Brahma [9] address the closely related component-based program synthesis problem. These systems rely on either a declarative program specification, or a user-specified partial program, and operate on statements in bit-vector calculi rather than directly on hardware instructions. Liang [12] considers the task of learning programs from testcases alone, but at a similarly high level of abstraction. Although useful for synthesizing non-trivial programs, the internal representations used by these systems precludes them from reasoning directly about the runtime performance of the code that they produce.

STOKE differs from previous approaches to superoptimization by relying on incomplete stochastic search and making heavy use of MCMC sampling to explore the extremely high dimensional, irregular search space of loop-free assembly programs. For many optimization problems of this form, MCMC sampling is the only known general solution method which is also tractable. Successful applications are many, and include protein alignment [16], code breaking [6], and scene modeling and rendering in computer graphics [5, 19].

3. Cost Minimization

To formulate the program optimization task as a cost minimization problem, we first define a cost function with terms that balance the hard constraints of correctness preservation and the soft constraints of performance improvement. The primary advantage of this approach is that it removes the burden of reasoning directly about the mutually-dependent optimization issues faced by a traditional compiler. For instance, rather than considering the trade-offs between register allocation and instruction selection, we simply define a term which reflects the primary consequence of the decision: expected runtime. We then utilize a cost minimization search procedure to produce a program that balances those trade-offs as effectively as possible. We run the procedure for as long as is feasible, and select the lowest-cost result which satisfies all of the hard constraints.

In formalizing this idea we use the following notation. We refer to an input program as the *target* ($\mathcal{T}$) and a candidate compilation as a *rewrite* ($\mathcal{R}$), we say that a function $f(X;Y)$ takes inputs X and is parameterized by Y, and finally, we define the indicator function for boolean variables:

$$\mathbf{1}\{\phi\} = \begin{cases} 1 & \phi = \text{true} \\ 0 & \phi = \text{false} \end{cases} \tag{1}$$

3.1 Cost Function

At the highest level, a cost function should include both a correctness term $\text{eq}(\cdot)$ and a performance term, $\text{perf}(\cdot)$. An optimization, $\mathcal{R}'$, is any rewrite for which the cost function obtains a minimum value and the correctness term is zero.

$$c(\mathcal{R};\mathcal{T}) = \text{eq}(\mathcal{R};\mathcal{T}) + \text{perf}(\mathcal{R};\mathcal{T}) \tag{2}$$

$$\mathcal{R}' = \arg\min_{r} \Big(\text{perf}(r;\mathcal{T}) \;\Big|\; \text{eq}(r;\mathcal{T}) = 0\Big) \tag{3}$$

The transformation correctness term, $\text{eq}(\cdot)$, measures the similarity of two functions. The term is zero if and only if the two functions are equal. For our purposes, two code sequences are regarded as functions of registers and memory contents, and are equal if for all machine states that agree on the live inputs with the respect to the target, the two codes produce identical side effects on the live outputs with respect to the target. Because program optimization is undefined for ill-formed programs, it is unnecessary that $\text{eq}(\cdot)$ be defined for a target or rewrite that produce some undefined behavior. However nothing prevents us from doing so, and it would be a straightforward extension to produce a definition of $\text{eq}(\cdot)$ which preserved hardware exception behavior as well.

The performance improvement term, $\text{perf}(\cdot)$, quantifies the performance improvement of a rewrite with respect to the target. Depending on the application, this term could reflect code size, expected runtime, number of disk accesses, power consumption, or any other measure of resource usage. Crucially, the extent to which this term accurately reflects the performance improvement of a rewrite directly affects the quality of the results discovered by a search procedure.

3.2 MCMC Sampling

In general, we expect cost functions of the form described above to be highly irregular and not amenable to exact optimization techniques. The common approach to solving this problem is to employ the use of an MCMC sampler. Although a complete discussion of MCMC sampling techniques is beyond the scope of this paper, 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 from more often than regions of low probability. When applied to cost minimization, this technique has the attractive property that in the limit the most frequently occuring sample will be taken from the minimum (optimal) value of the function. In practice, well before this limiting behavior is observed, MCMC sampling functions as an intelligent hill climbing method which is robust against irregular functions that are dense with local minima. A common method (described by [8]) for transforming an arbitrary cost function, $c(\cdot)$, into a probability density function is the following, where β is a constant and Z is a partition function that normalizes the distribution:

$$p(\mathcal{R};\mathcal{T}) = \frac{1}{Z}\exp\Big(-\beta\cdot c(\mathcal{R};\mathcal{T})\Big) \tag{4}$$

Although computing Z is in general intractable, the Metropolis-Hastings algorithm for generating Markov chains is designed to explore density functions such as $p(\cdot)$ without the need to compute the partition function [10, 15]. The basic idea is simple. The algorithm maintains a current rewrite $\mathcal{R}$ and proposes a modified rewrite $\mathcal{R}^*$ as the next step in the chain. The *proposal* $\mathcal{R}^*$ is either accepted or rejected. If the proposal 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 so long as the proposals are *ergodic* (capable of transforming any point in the space to any other through some sequence of applications) the algorithm will in the limit produce a sequence of samples with the properties described above (i.e., in proportion to their cost). This global property depends on the local acceptance criteria of a proposal $\mathcal{R} \to \mathcal{R}^*$, which is governed by the Metropolis-Hastings acceptance probability, where $q(\mathcal{R}^*|\mathcal{R})$ is the proposal distribution from which a new rewrite $\mathcal{R}^*$ is sampled given the current rewrite, $\mathcal{R}$:

$$\alpha(\mathcal{R}\to\mathcal{R}^*;\mathcal{T}) = \min\left(1, \frac{p(\mathcal{R}^*;\mathcal{T})q(\mathcal{R}|\mathcal{R}^*)}{p(\mathcal{R};\mathcal{T})q(\mathcal{R}^*|\mathcal{R})}\right) \tag{5}$$

This proposal distribution is key to a successful application of the algorithm. Empirically, the best results are obtained by a distribution which makes both local proposals that make minor modifications to $\mathcal{R}$ and global proposals that induce major changes. In the event that the proposal distributions are symmetric, $q(\mathcal{R}^*|\mathcal{R}) = q(\mathcal{R}|\mathcal{R}^*)$, the acceptance probability can be reduced to the much simpler Metropolis ratio, which can be computed directly from $c(\cdot)$:

$$\begin{aligned} \alpha(\mathcal{R}\to\mathcal{R}^*;\mathcal{T}) &= \min\left(1, \frac{p(\mathcal{R}^*;\mathcal{T})}{p(\mathcal{R};\mathcal{T})}\right) \\ &= \min\left(1, \exp\left(-\beta\cdot\frac{c(\mathcal{R}^*;\mathcal{T})}{c(\mathcal{R};\mathcal{T})}\right)\right) \end{aligned} \tag{6}$$

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 in value between $\mathcal{R}^*$ and $\mathcal{R}$. This property prevents the search from becoming trapped in local minima while remaining less likely to accept a move that is much worse than available alternatives.

4. x86-64 Binary Optimization

Having discussed cost minimization in the abstract, we now turn to the practical details of implementing cost minimization for optimizing the runtime performance of x86-64 binaries. As x86-64 is one of the most complex ISAs currently available, we expect that the discussion in this section should generalize well to other architectures.

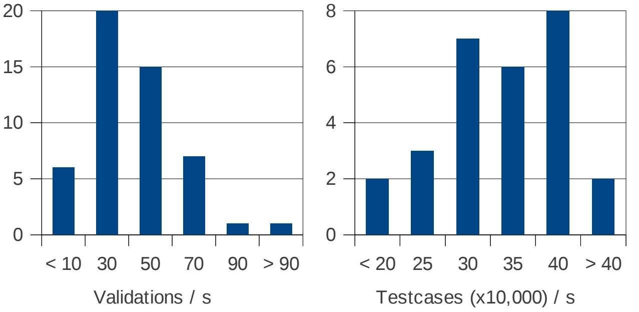

Figure 2. Histograms of validations per second (left), and testcase evaluations per second (right), for the benchmarks discussed in Section 6, The low validation throughput is insufficient for MCMC sampling and motivates an approach based on testcases.

4.1 Transformation Correctness

For loop-free sequences of x86-64 assembly code, a natural choice for implementing the transformation correctness term is a symbolic validator such as the one used in [4]. For a candidate rewrite, the term may be defined in terms of an invocation of the validator as:

$$\mathrm{eq}(\mathcal{R};\mathcal{T}) = 1 - \Big(\mathbf{1}\{\mathrm{VALIDATE}(\mathcal{T},\mathcal{R})\}\Big) \tag{7}$$

Unfortunately, the total number of validations that can currently be performed per second, even for modestly sized codes, is low. Figure 2 (left) suggests that for the benchmarks discussed in Section 6 the number is well below 100. Because MCMC sampling is effective only insofar as it is able to explore sufficiently large numbers of proposals, the repeated computation of Equation 7 in its inner-most loop would almost certainly drive that number well below a useful threshold.

This observation motivates the definition of an approximation to $\mathrm{eq}(\cdot)$ based on testcases, τ. Intuitively, we execute the proposal $\mathcal{R}^*$ on a set of inputs and measure "how close" the output is to the output of the target on those same inputs. For a given input, we use the number of bits difference in live outputs (i.e., the Hamming distance) to measure correctness. Besides being much faster than using a theorem prover, this approximation of program 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 the search.

$$\begin{aligned}\mathrm{eq}'(\mathcal{R};\mathcal{T},\tau) = &\sum_{t\in\tau}\mathrm{reg}(\mathcal{R};\mathcal{T},t) + \mathrm{mem}(\mathcal{R};\mathcal{T},t)\\ &+\sum_{t\in\tau}\mathrm{err}(\mathcal{R};\mathcal{T},t)\end{aligned} \tag{8}$$

In the above formula, $\mathrm{reg}(\cdot)$ compares the side effects, $\mathrm{val}(\cdot)$, that both functions produce on live register outputs, ρ, with respect to the target, and counts the number of bits that the results differ by. These outputs can include general purpose, SSE, and condition registers. $\mathrm{mem}(\cdot)$ is defined analogously for live memory outputs, μ. We use the population count function, $\mathrm{POP}(\cdot)$, to count the number of 1-bits in the 64-bit representation of an integer.

$$\mathrm{reg}(\mathcal{R};\mathcal{T},t) = \sum_{r\in\rho}\mathrm{POP}\Big(\mathrm{val}(\mathcal{T},r)\oplus\mathrm{val}(\mathcal{R},r)\Big) \tag{9}$$

$$\mathrm{mem}(\mathcal{R};\mathcal{T},t) = \sum_{m\in\mu}\mathrm{POP}\Big(\mathrm{val}(\mathcal{T},m)\oplus\mathrm{val}(\mathcal{R},m)\Big) \tag{10}$$

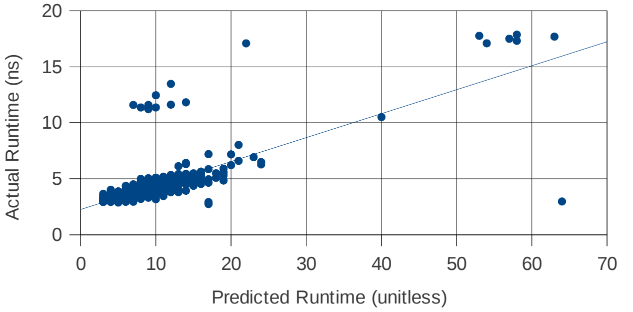

Figure 3. Comparison of predicted and actual runtimes for the benchmarks described in Section 6, along with rewrites generated in the course of our experiments. The points are well correlated but distinguished by outliers characterized by instruction level parallelism and memory traffic at the micro-op level. The approximation is sufficient for the benchmarks we consider.

$\mathrm{err}(\cdot)$ is used to distinguish programs which exhibit undefined behavior, by counting and then penalizing the number of segfaults, $\mathrm{sigsegv}(\cdot)$, floating point exceptions, $\mathrm{sigfloat}(\cdot)$, and reads from undefined memory or registers, $\mathrm{undef}(\cdot)$, which occur during execution of a rewrite. Note that $\mathrm{sigsegv}(\cdot)$ is defined in terms of the target, which determines the set of addresses which may be successfully dereferenced by a rewrite for a particular testcase. Rewrites are run in a sandbox to ensure that undefined behavior can be detected safely. The extension to additional exceptional behavior would be straightforward.

$$\begin{aligned}\mathrm{err}(\mathcal{R};\mathcal{T},t) = &\; w_{sf}\cdot\mathrm{sigsegv}(\mathcal{R};\mathcal{T},t)\\ &+ w_{fp}\cdot\mathrm{sigfloat}(\mathcal{R};t)\\ &+ w_{ur}\cdot\mathrm{undef}(\mathcal{R};t)\end{aligned} \tag{11}$$

The evaluation of $\mathrm{eq}'(\cdot)$ may be accomplished either by JIT compilation, or the use of a hardware emulator. In our experiments we have chosen the latter. Figure 2 (right) shows the number of testcase evaluations that our emulator is able to perform per second: just under 500,000. This implementation allows us to define an optimized method for computing $\mathrm{eq}(\cdot)$ which achieves sufficient throughput to be useful for MCMC sampling.

$$\mathrm{eq}^*(\mathcal{R};\mathcal{T},\tau) = \begin{cases}\mathrm{eq}(\mathcal{R};\mathcal{T}) & \mathrm{eq}'(\mathcal{R};\mathcal{T},\tau)=0\\ \mathrm{eq}'(\mathcal{R};\mathcal{T},\tau) & \text{otherwise}\end{cases} \tag{12}$$

In addition to performance, Equation 12 has the following desirable properties. First, failed computations of $\mathrm{eq}(\cdot)$ will produce a counterexample testcase that may be used to refine τ as described in [4]. The careful reader will note that refining τ affects the cost function, $c(\cdot)$, and effectively changes the search space that it defines. However in practice, the number of failed validations that are required to produce a robust set of testcases that accurately predict success is quite low. Second, as discussed above, it smooths the search space by allowing the transformation equality metric to incrementally quantify the difference between two programs.

4.2 Performance Improvement

A straightforward method for computing the performance improvement term is to JIT compile both target and rewrite and compare their runtimes. Unfortunately, as with the transformation correctness term, the amount of time required to both compile a function and execute it sufficiently many times to eliminate transient performance effects is prohibitively expensive to be used in the inner-most loop of MCMC sampling. In our experiments, we adopt a simple heuristic for approximating the runtime performance of a function, which is based on a static approximation of the average latencies of its instructions.

$$\begin{aligned} \mathrm{perf}(\mathcal{R};\mathcal{T}) &= H(\mathcal{R}) - H(\mathcal{T}) \\ H(f) &= \sum_{i \in \mathrm{inst}(f)} \mathrm{LATENCY}(i) \end{aligned} \tag{13}$$

Figure 3 shows a reasonably high correlation between the heuristic and the actual runtimes of the benchmarks described in Section 6, along with rewrites for those benchmarks which were generated in the course of our experiments. Outliers are characterized by disproportionately high instruction level parallelism at the micro-op level and the performance effects of inconsistent memory access times. A more accurate model of the second order performance effects introduced by a modern CISC processor is straightforward if tedious to construct and we expect would be necessary for more complex programs. Nonetheless, the approximation is largely sufficient for the benchmarks that we consider. Whatever errors stem from this imprecision can be addressed by recomputing $\mathrm{perf}(\cdot)$ using the slower JIT compilation method as a postprocessing step. In our experiments we record the top-n lowest cost samples produced by MCMC sampling, rerank each based on their actual runtimes, and return the best result.

4.3 MCMC Sampling

For x86-64 binary optimization, we represent candidate rewrites as finite loop-free sequences of instructions, of length ℓ, where a distinguished token, UNUSED, allows for the representation of programs that contain fewer than ℓ instructions. This simplifying assumption is essential to the formulation of MCMC sampling discussed in Section 3.2, as it places a constant value on the dimensionality of the search space. The interested reader may consult [1] for a thorough treatment of why this is necessary. Our definition of the proposal distribution, $q(\cdot)$, chooses among four possible moves: the first two minor, and the last two major:

Opcode. With probability p_c, an instruction is selected at random, and its opcode is replaced by a random opcode. The new opcode is drawn from an equivalence class of opcodes which require the same number and type of operands as the old opcode. We construct these classes from the set of arithmetic and fixed point SSE opcodes.

Operand. With probability p_o, an instruction is selected at random and one of its operands is replaced by a random operand drawn from an equivalence class of operands with equivalent types to the old operand. If the operand is an immediate, its value is drawn from a set of predefined constants. We construct this set using the range -16 to 16 and all subsequent powers of 2.

Swap. With probability p_s, two lines of code are selected at random and interchanged. Each line may correspond to either an instruction or the UNUSED token.

Instruction. With probability p_i, an instruction is selected at random and replaced either by an unconstrained random instruction or the UNUSED token. A random instruction is constructed by first selecting an opcode at random and then choosing random operands of the appropriate types. The UNUSED token is proposed with probability p_u.

These definitions satisfy the ergodicity property described in Section 3.2. Any program can be transformed into any other through repeated application of Instruction moves. These definitions also satisfy the symmetry property, and thus allow the computation of acceptance probability using Equation 6. To see why, note that the probabilities of performing all four move types 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. Swap and instruction moves are similarly unconstrained in both directions.

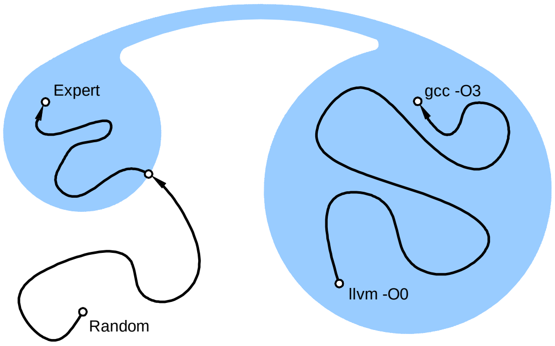

Figure 4. Abstract depiction of the search space for the Montgomery multiplication benchmark. O0 and O3 optimized codes occupy a densely connected part of the space which is easily traversed. Expert code occupies an entirely different region of the space which is reachable only by way of an extremely low probability path.

4.4 Separating Synthesis from Optimization

An early implementation of STOKE based on the above principles, was able to consistently transform `llvm -O0` code into the equivalent of `gcc -O3` code. Unfortunately, it was unable to produce results which were competitive with expert hand-written code. The reason is suggested by Figure 4, which gives an abstract depiction of the search space for the Montgomery multiplication benchmark. For loop-free sequences of code, `llvm -O0` and `gcc -O3` codes differ primarily with respect to efficient use of the stack and choices of individual instructions. Beyond these differences, the resulting codes are algorithmically quite similar. This is because compiler optimizers are generally designed to compose many small local transformations: dead code elimination deletes one instruction, constant propagation changes one register to an immediate, and strength reduction replaces a multiplication with an add. With respect to the search space, such sequences of local optimizations define a region of equivalent programs that are densely connected by very short sequences of moves (often just one) that are easily traversed by a local search method. Beginning from `llvm -O0` code, MCMC sampling will quickly identify local inefficiencies one by one, improve each in turn, and hill climb its way to a `gcc -O3` code.

The expert code discovered by STOKE occupies an entirely different region of the search space. As noted earlier, it has the property that no sequence of small equality preserving transformations connect it to either the `llvm -O0` or the `gcc -O3` code. It represents a completely distinct algorithm for implementing the Montgomery multiplication kernel at the assembly level, one which requires that its input values be permuted and relocated to distinguished register locations to permit the use of hardware intrinsics. The only method we know of for a local search procedure to transform either code into the expert code is to traverse the extremely low probability path that builds the expert 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.

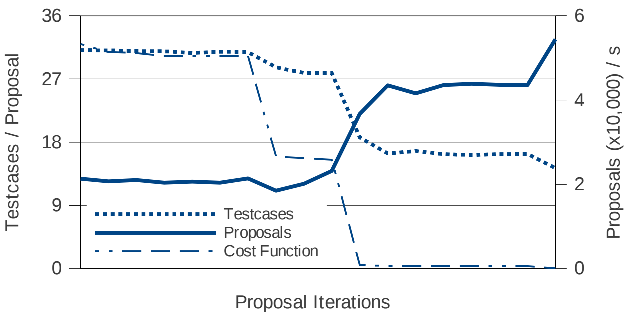

Figure 5. Proposals evaluated per second versus testcases evaluated prior to early termination, during synthesis for the Montgomery multiplication benchmark. Reducing the number of evaluated testcases produces an almost 3x improvement in proposal throughput. Cost function shown unitless for reference.

This observation motivates the division of cost minimization into two phases:

- **Synthesis** A synthesis phase focused solely on correctness, which attempts to locate regions of equivalent programs which are distinct from the region occupied by the target.
- **Optimization** An optimization phase focused on speed, which searches for the fastest program within each of those regions.

The two phases share the same search implementation; only the starting point and the acceptance functions are different. Synthesis begins with a random starting point (a sequence of randomly chosen instructions), while optimization begins with a code sequence known to be equivalent to the target. For proposals, synthesis ignores the performance improvement term altogether and simply uses Equation 12 as its cost function. Optimization uses both terms, which allows it to measure improvement while also allowing it to experiment with "shortcuts" that (temporarily) violate transformation correctness.

4.5 Optimized Acceptance Computation

The optimized method for computing $\mathrm{eq}^*(\cdot)$ given in Equation 12 is sufficiently fast for MCMC sampling. However, its performance can be further improved. As described so far, $\mathrm{eq}^*(\cdot)$ is computed by first running a proposal on testcases, summing the results, noting the ratio in total cost with that of the current rewrite, and then sampling a random variable to decide whether or not to accept the proposal. Instead, by first sampling the random variable p, and then computing the maximum value of the ratio that the algorithm will accept given p, it is possible to terminate the evaluation of testcases as soon as that bound is exceeded.

Specifically, because the formulation of the proposal distribution $q(\cdot)$ is symmetric we may compute the acceptance probability $\alpha(\cdot)$ of a proposal directly from $c(\cdot)$ as shown in Equation 6. By first sampling p we can invert $\alpha(\cdot)$ to solve for the maximum cost rewrite $c(\cdot)$ that the algorithm will accept.

$$\begin{aligned} p &< \alpha(\mathcal{R}\rightarrow\mathcal{R}^*;\mathcal{T}) \\ &< \min\left(1, \exp\left(-\beta\cdot\frac{c(\mathcal{R}^*;\mathcal{T})}{c(\mathcal{R};\mathcal{T})}\right)\right) \qquad (14) \\ c(\mathcal{R}^*;\mathcal{T},\tau) &< c(\mathcal{R};\mathcal{T},\tau) - \frac{\log(p)}{\beta} \end{aligned}$$

Because the computation of $\mathrm{eq}'(\cdot)$ is based on the iterative evaluation of testcases, it is only necessary to do so for as long as the running sum does not exceed this upper bound. Once it does, we know that the proposal is guaranteed to be rejected, and no further computation is necessary. 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 do the average number of testcases which must be evaluated prior to early termination. This in turn produces a considerable increase in the number of proposals evaluated per second, which at peak exceeds 50,000.

	al	bl	cl	dl
Target	1111	0000	0000	0000
Rewrite	0000	1000	1100	1111

$$\begin{array}{r|c|c|c|c} & \textbf{al} & \textbf{bl} & \textbf{cl} & \textbf{dl} \\ \mathrm{val}(\mathcal{T},\mathrm{al})\oplus\mathrm{val}(\mathcal{R},\cdot) & 1111 & 0111 & 0011 & 0000 \\ \mathrm{POP}\Big(\mathrm{val}(\mathcal{T},\mathrm{al})\oplus\mathrm{val}(\mathcal{R},\cdot)\Big) & 4 & 3 & 2 & 0 \\ w_m\cdot\mathbf{1}\{\mathrm{al}\neq\cdot\} & 0 & 1 & 1 & 1 \end{array}$$

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

Figure 6. Strict versus improved equality functions for a machine state in which ax is live out. Strict assigns the maximum possible cost to a rewrite which produces the correct value in the wrong location. Improved assigns a cost of almost zero.

4.6 Improved Equality Metric

A second and even more important improvement stems from the observation that the definition of $\mathrm{reg}(\cdot)$ given in Equation 9 is unnecessarily strict. An illustrative example is shown in Figure 6. Consider a simplified machine with four 4-bit registers, and a target function that produces side effects in register al. The final machine states produced by running the target and a candidate rewrite are shown at the top of the figure. Because the rewrite produces the inverse of the desired value in al, it is assigned the maximum possible cost. This is unfortunate however, as the rewrite does produce the correct value, only in the wrong place: dl. A substantial improvement in performance is obtained by rewarding rewrites that produce correct (or nearly correct) values in the wrong locations. The improved cost function examines all registers of equivalent bitwidth, $\mathrm{bw}(\cdot)$, selects the one that most closely matches the value of the target register, and assigns an additional small penalty, w_m, if the selected register differs from the original.

$$\begin{aligned} \mathrm{reg}'(\mathcal{R};\mathcal{T},\tau) &= \sum_{r\in\rho}\min_{r'\in\mathrm{bw}(r)}\mathrm{R}(r,r';\tau) \\ \mathrm{R}(r,r';\tau) &= \mathrm{POP}\Big(\mathrm{val}(\mathcal{T},r)\oplus\mathrm{val}(\mathcal{R},r')\Big) \qquad (15) \\ &\quad + w_m\cdot\mathbf{1}\{r\neq r'\} \end{aligned}$$

For brevity, we note that it is possible to improve the definition of memory equality analogously. Although we do so in our experiments, the time required to compute this term does grow quadrat-

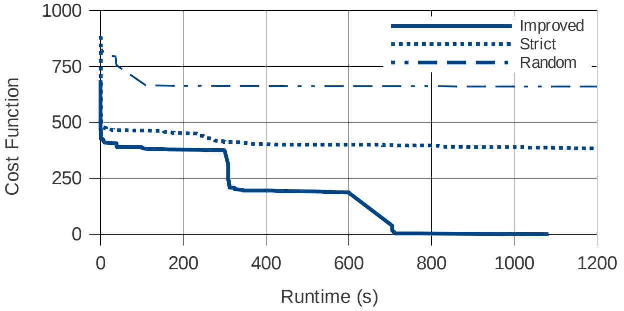

Figure 7. Strict versus improved synthesis cost functions for the Montgomery multiplication benchmark. In the amount of time (s) required for the improved function to converge, the strict function produces a result comparable to that of a purely random search.

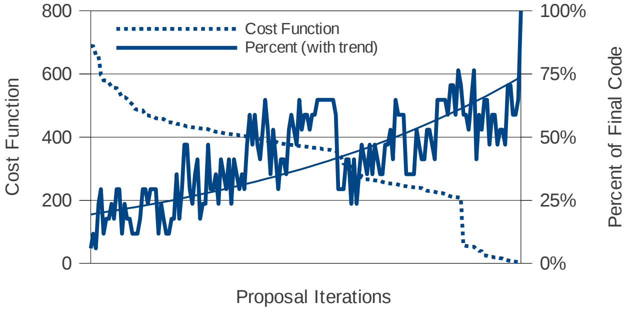

Figure 8. Cost function versus percentage of instructions which appear in the final zero-cost rewrite. Random search is an effective method for performing synthesis insofar as it is able to discover partially correct rewrites incrementally.

ically with the size of the target function's memory footprint. We expect that for more complex programs, an alternate implementation will be necessary.

Figure 7 shows the results of using the improved definitions of register and memory equality during synthesis for the Montgomery multiplication benchmark. In the amount of time required for the improved cost function to converge to a zero-cost rewrite, the strict version obtains a minimum cost which was only slightly superior to that obtained by a purely random search. The dramatic increase in performance can be explained as an implicit parallelization of the search procedure. By allowing a rewrite to place a correct value in an arbitrary location, the improved cost function allows candidates to simultaneously explore up to as many alternate computations as can be fit within an instruction sequence of length ℓ.

4.7 Why and When Synthesis Works

It is not intuitive that a randomized search procedure should synthesize a correct rewrite from such an enormous search space in such a short amount of time. In our experience, the reason why is that synthesis is effective precisely when it is possible to discover portions of a correct rewrite incrementally, as opposed to all at once. Figure 8 plots the current best cost obtained during synthesis against the percentage of instructions appearing in both that rewrite and the final rewrite for the Montgomery multiplication benchmark. As search proceeds, the percentage of correct code increases in inverse proportion to the value of the cost function. While this is very encouraging and there are many programs that can be synthesized in pieces, each of which increases the average number of correct bits in the output, there are certainly interesting programs that do not satisfy this property. In the limit, any code which performs a complex computation that is reduced to a single boolean value poses a problem for our technique. The discovery of partially correct computations is useful as a guide for random search only insofar as it in turn produces a partially correct result, which can be detected by a cost function.

This observation motivates the desire for a cost function which maximizes the signal produced by a partially correct rewrite. We discussed a successful application of this principle in Section 4.6. Nonetheless, there remains room for improvement. Consider a program which rounds its inputs up to the next highest power of two. This program has the interesting property of differing from the program which simply returns zero in just one bit per testcase. The improved cost function discussed above assigns a very low cost to this constant zero function, which although nearly perfect is completely wrong, and exhibits no partially correct computations that can be hill-climbed to a correct rewrite.

Fortunately, we note that even when synthesis fails, optimization is still possible. It must simply proceed only from the region occupied by the target as a starting point.

5. STOKE

STOKE is a prototype implementation of the concepts described above. Its high-level design is shown in Figure 9. A user provides a target binary which is created using a standard compiler (in our experiments, `llvm -O0`); in the event that the target contains loops, STOKE identifies loop-free subsequences of the code which it will attempt to optimize instead. The user also provides an annotated driver in which the target is called in an appropriate context. Based on the user's annotations, STOKE automatically generates random inputs to the target, compiles the driver, and runs the code under instrumentation to produce testcases. The target and testcases are broadcast to a set of synthesis threads which after a fixed amount of time report back validated candidate rewrites. In like fashion, multiple threads perform optimization on both the target and those rewrites. Finally, the set of rewrites with a final cost that is within 20% of the minimum result are re-ranked based on actual runtime, and the best result is returned to the user.

5.1 Test Case Generation and Evaluation

STOKE automatically generates testcases using annotations provided by a user. Because STOKE operates on x86-64 assembly, those inputs are limited to fixed-width bit strings, which unless otherwise specified, are sampled uniformly at random. If the target uses an input to form a memory address, the user must annotate that input with a range of values that guarantee that the resulting addresses are legal given the context in which the target is invoked. The compiled program is executed under instrumentation using Intel's PinTool [13]. As each instruction is executed, the tool records the state of all general purpose, SSE, and condition registers, as well as dereferenced memory. The initial state of the registers, along with the first values dereferenced from each memory address are used to form testcase inputs. Outputs are formed analogously. By default, STOKE generates 32 testcases for each target.

For each testcase, The set of addresses dereferenced by the target are used to define the sandbox in which candidate rewrites are executed. Attempts to dereference invalid addresses are trapped and replaced by instructions which produce a constant zero value. Attempts to read from registers in an undefined state and computations which produce floating point exceptions are handled analogously.

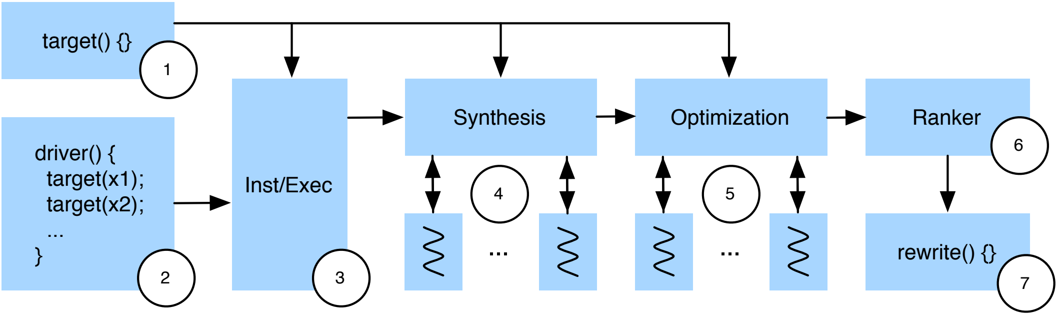

Figure 9. The high-level design of STOKE. A target binary created by a production compiler (1) and driver code (2) are run under instrumentation (3) using automatically generated inputs to produce testcases. Synthesis threads (4) use the target and testcases to generate candidate rewrites, which along with the target are refined by optimization threads (5). The results are ranked (6) and the rewrite with the lowest cost is returned to the user (7).

w_{sf}	1	p_c	0.16	p_u	0.16
w_{fp}	1	p_o	0.5	β	0.1
w_{ur}	2	p_s	0.16	ℓ	50
w_m	3	p_i	0.16		

Figure 10. MCMC sampling parameters used by STOKE during both synthesis and optimization phases.

5.2 Validation

STOKE uses a sound procedure for validating the equality of two sequences of loop-free assembly which is similar to the one described in [2]. Code sequences are converted into SMT formulae in the quantifier free theory of bit-vector arithmetic used by the STP [7] theorem prover, and used to produce a query which asks whether both sequences produce the same side effects on live outputs when executed from the same initial machine state. For our purposes, a machine state consists of general purpose, SSE, and condition registers, and memory. Depending on type, registers are modeled as between 8- and 128-bit vectors. Memory is modeled as two vectors: a 64-bit address and an 8-bit value (x86-64 is byte addressable).

STOKE first asserts the constraint that both sequences agree on the initial machine state of the live inputs with respect to the target. Next, it iterates over the instructions in the target, and for each instruction asserts a constraint which encodes the transformation it produces on the machine state. These constraints are chained together to produce a constraint on the final machine state of the live outputs with respect to the target. Analogous constraints are asserted for the rewrite. Finally, for all pairs of memory accesses at addresses $\texttt{addr}_1$ and $\texttt{addr}_2$, STOKE asserts an additional constraint which relates their values: $\texttt{addr}_1 = \texttt{addr}_2 \Rightarrow \texttt{val}_1 = \texttt{val}_2$. Using these constraints, STOKE performs an STP query which asks whether there does not exist an initial machine state which causes the two sequences to produce different values for the live outputs with respect to the target. If the answer is "yes", then the sequences are determined to be equal. If the answer is "no", then the prover produces a counter example which is used to produce a new testcase.

STOKE makes two simplifying assumptions which are necessary to keep validator runtimes tractable. First, it assumes that stack addresses are represented exclusively as constant offsets from the stack pointer. This allows STOKE to treat stack addresses as nameable locations, and minimizes the number of expensive memory constraints which must be asserted. This is essential for validating against `llvm -O0` code, which exhibits heavy stack traffic. Second, it treats 64-bit multiplication and division as uninterpreted functions by asserting the constraint that the instructions produce identical arbitrary values when executed on identical inputs. This assumption is necessary irrespective of the compiler toolchain used to produce the target. Whereas STP diverges when reasoning explicitly about two or more such operations, our benchmarks contain as many as four per program.

5.3 Parallel Synthesis and Optimization

Synthesis and optimization are executed in parallel on a small cluster consisting of 40 dual-core 1.8 GHz AMD Opterons. Both are allocated computational budgets of 30 minutes. The MCMC sampling parameters used by both phases are summarized in Figure 10.

6. Evaluation

In addition to the Montgomery multiplication kernel discussed previously, STOKE was evaluated on benchmarks drawn both from the literature and real-world high-performance codes. The performance improvements obtained for those kernels are summarized in Figure 11 (top), while corresponding STOKE runtimes are shown in Figure 11 (bottom). Beginning from binaries compiled using `llvm -O0`, STOKE consistently discovers rewrites which match the performance of the code produced by `gcc -O3` and `icc -O3`. In several cases, the performance exceeds both and is comparable to expert handwritten assembly. As we explain below, the improvement often results from the discovery of a completely distinct assembly level algorithm for implementing the target code. We close with a discussion of the benchmarks which highlight STOKE's limitations.

6.1 Hacker's Delight

Hacker's Delight [20], commonly referred to as "the bible of bit-twiddling hacks", is a collection of techniques for encoding otherwise complex algorithms as small loop-free sequences of bit-manipulating instructions. Gulwani [9] notes this as a source of benchmarks for program synthesis and superoptimization, and identifies a 25 program benchmark which ranges in complexity from turning off the right-most bit in a word, to rounding up to the next highest power of 2, or selecting the upper 32 bits from a 64-bit multiplication. Our implementation of the benchmark uses the C code found in the original text. For brevity, we discuss only the programs for which STOKE discovers an algorithmically distinct rewrite.

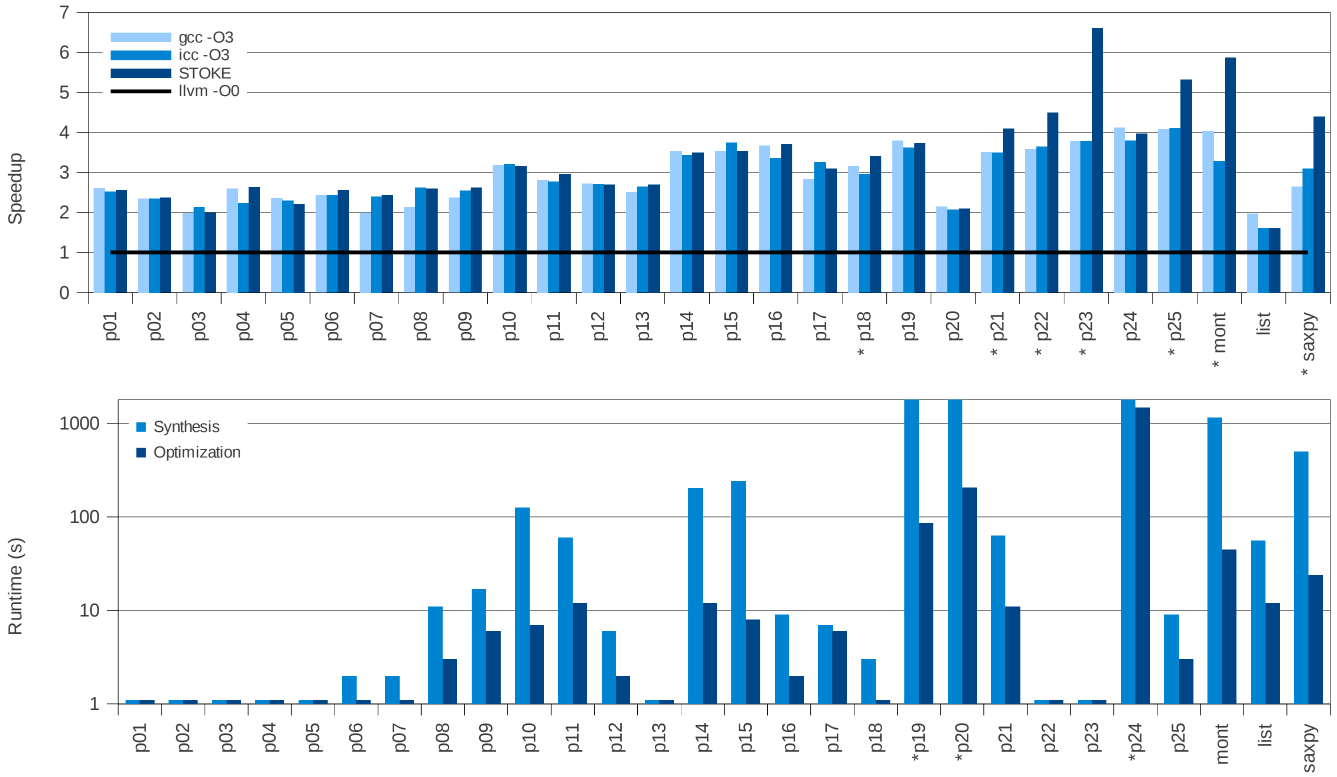

Figure 11. (Top) Average speedup over `llvm -O0` for benchmark kernels. Beginning from code produced by `llvm -O0`, STOKE discovers rewrites which are comparable to code produced by `gcc -O3` and `icc -O3`. In some cases, the rewrite outperforms both, and are comparable to expert handwritten assembly. Kernels for which STOKE discovered an algorithmically distinct rewrite are annotated with a star. (Bottom) Synthesis and optimization runtimes. Kernels for which synthesis timed out are annotated with a star.

Figure 12 shows the "Cycle Through 3 Values" benchmark, which takes an input, x, and transforms it to the next value in the sequence $\langle \mathrm{a}, \mathrm{b}, \mathrm{c} \rangle$: a becomes b, b becomes c, and c becomes a. Hacker's Delight points out that the most natural implementation of this function is a sequence of conditional assignments, but notes that for an ISA without conditional move intrinsics the implementation shown is cheaper than one which uses branch instructions. For x86-64, which has conditional move intrinsics, this is an instance of premature optimization. Unfortunately, neither `gcc` nor `icc` are able to detect this, and are forced to transcribe the code as written. There are no sub-optimal subsequences in the resulting code. Both are simply unable to reason about the semantics of the function as a whole. We expect that equality-preserving superoptimizers would exhibit similar behavior for the same reason. STOKE on the other hand, rediscovers the natural implementation from the 41 line `llvm -O0` compilation. We note that although this rewrite is only five lines long, it remains beyond the reach of superoptimizers based on brute force enumeration.

In similar fashion, for machines without 64-bit instructions, the implementation that Hacker's Delight recommends for the "Compute the Higher Order Half of a 64-bit Product" multiplies two 32-bit inputs in four parts and aggregates the results. The computation resembles the Montgomery multiplication benchmark, and STOKE discovers a rewrite which requires a single multiplication using the appropriate 64-bit intrinsic.

STOKE additionally discovers a number of typical superoptimizer rewrites. These include using the popcnt intrinsic, which counts the number of 1-bits in an integer, as an intermediate step in the "Compute Parity" and "Determine if an Integer is a Power of 2" benchmarks.

6.2 SAXPY

SAXPY (Single-precision Alpha X Plus Y) is a level 1 vector operation in the Basic Linear Algebra Subsystems Library [3]. The function makes heavy use of heap accesses and presents the opportunity for optimization using vector intrinsics. To allow for STOKE to discover this possibility, our implementation is unrolled four times by hand, as shown in Figure 13. Despite heavy annotation to indicate that the arrays pointed to by x and y are aligned and do not alias each other, the production compilers either cannot detect the possibility of a compilation using vector intrinsics, or are precluded from doing so by some internal heuristic.

STOKE on the other hand, when given identical information, is able to discover the natural implementation: the constant a is broadcast four ways from a general purpose register into an SSE register, and then multiplied by and added to the contents of x and y, which are loaded into SSE registers four elements at a time. The four way broadcast does not appear anywhere in either the `gcc -O3` code, or in the original 61 line `llvm -O0` code. As noted above, despite the simplicity, the length of the resulting code is well beyond the reach of existing superoptimizers.

6.3 Limitations

Bansal [2] identifies the Linked List Traversal Benchmark for superoptimizers shown in Figure 14. The code iterates over a list of integers and multiplies each of the elements by two. The code is unique with respect to the benchmarks discussed so far, as it con-

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

```
# gcc -O3

.L0:
  movl edx, eax
  xorl edx, edx
  xorl ecx, eax
  cmpl esi, edi
  sete dl
  negl edx
  andl edx, eax
  xorl edx, edx
  xorl ecx, eax
  cmpl ecx, edi
  sete dl
  xorl ecx, esi
  negl edx
  andl esi, edx
  xorl edx, eax
```

```
# STOKE

.L0:
  cmpl edi, ecx
  cmovel esi, ecx
  xorl edi, esi
  cmovel edx, ecx
  movq rcx, rax
```

Figure 12. Cycling Through 3 Values benchmark. STOKE sees through the esoteric implementation which `gcc -O3` translates literally (left) and rediscovers the intuitive algorithm using conditional move intrinsics (right).

tains a loop. As a result, STOKE is unable to optimize the function as a whole, but rather only its inner-most loop-free fragment. STOKE discovers the same optimizations as Bansal's superoptimizer, the elimination of stack traffic and a strength reduction from multiplication to bit shifting. However it fails in like fashion to eliminate the instructions which copy the head pointer from and back to the stack on every iteration of the loop. The production compilers on the other hand, are able to eliminate the memory traffic by caching the pointer in a register prior to entering the loop. As a result, the rewrite discovered by STOKE is slower than the code produced by `gcc -O3` (surprisingly, `icc` does not perform strength reduction, and produces code which performs similarly). This shortcoming could be addressed by extending our framework to validate and propose modifications to code containing loops.

As shown in Figure 11 (bottom), STOKE is unable to synthesize a rewrite for three of the Hacker's Delight Benchmarks. All three benchmarks, despite being quite complex, have the interesting property that they produce results which differ by only a single bit from a simple yet completely incorrect alternative. The "Round Up to the Next Highest Power of 2" benchmark is nearly indistinguishable from the function which always returns zero. The same is true of the "Next Highest with Same Number of 1-bits", and a small transformation to the "Exchanging Two Fields" benchmark with respect to the identity function. Nonetheless, for these three benchmarks, using its optimization phase alone STOKE is still able to discover rewrites which perform comparably to the production compiler code, which we believe to be optimal. Unfortunately, in general we do not expect this to be the case. A more sophisticated cost function, as described in section 4.7, is surely necessary.

```
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];
}
```

```
# gcc -O3

.L0:
movslq ecx,rcx
leaq (rsi,rcx,4),r8
leaq 1(rcx),r9
movl (r8),eax
imull edi,eax
addl (rdx,rcx,4),eax
movl eax,(r8)
leaq (rsi,r9,4),r8
movl (r8),eax
imull edi,eax
addl (rdx,r9,4),eax
leaq 2(rcx),r9
addq 3,rcx
movl eax,(r8)
leaq (rsi,r9,4),r8
movl (r8),eax
imull edi,eax
addl (rdx,r9,4),eax
movl eax,(r8)
leaq (rsi,rcx,4),rax
imull (rax),edi
addl (rdx,rcx,4),edi
movl edi,(rax)
```

```
# STOKE

.L0:
movd edi,xmm0
shufps 0,xmm0,xmm0
movups (rsi,rcx,4),xmm1
pmullw xmm1,xmm0
movups (rdx,rcx,4),xmm1
paddw xmm1,xmm0
movups xmm0,(rsi,rcx,4)
```

Figure 13. SAXPY benchmark. Unlike `gcc -O3` (top), STOKE discovers a rewrite which uses SSE vector instructions (bottom).

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

```
# gcc -O3

  movq -8(rsp), rdi
.L4:
  sall (rdi)
  movq 8(rdi), rdi
.L6:
  testq rdi, rdi
  jne .L4
```

```
# STOKE

.L4:
  movq -8(rsp), rdi
  sall (rdi)
  movq 8(rdi), rdi
  movq rdi, -8(rsp)
.L6:
  movq -8(rsp), rdi
  testq rdi, rdi
  jne .L4
```

Figure 14. Linked List Traversal benchmark. STOKE discovers the same rewrite (right) as Bansal's superoptimizer, but fails to cache the head pointer in a register, as in the `gcc -O3` code (left).

7. Conclusion and Future Work

We have shown a new approach to the loop-free binary superoptimization task which formulates program optimization as a stochastic search problem. Compared to a traditional compiler, which factors optimization into a sequence of small independently solvable subproblems, our framework is based on cost minimization and considers the competing constraints of transformation correctness and performance improvement simultaneously. We show that an MCMC sampler can be used to rapidly explore cost functions of this form and produce low cost samples which correspond to high quality optimizations. Although our method sacrifices completeness, the scope of programs which we are able to consider, and the quality of the rewrites we produce, far exceed those of existing superoptimizers.

Although our prototype implementation, STOKE, is in many cases able to produce rewrites which are competitive with or outperform the code produced by production compilers, there remains substantial room for improvement. In future work, we intend to pursue both a validation and proposal mechanism for code containing loops and a synthesis cost function which is robust against targets with numerous deceptively attractive, albeit completely incorrect, synthesis alternatives.

Acknowledgments

The authors would like to thank Peter Johnston, Juan Manuel Tamayo, and Kushal Tayal for their assistance in the implementation of STOKE, Ankur Taly and David Ramos for their advice regarding the semantics of x86-64 opcodes and the use of STP, Jake Herczeg for his time spent designing Figure 4, and Martin Rinard for suggesting a technique which focuses more on testcases than validation. This work was supported by NSF grant CCF-0915766 and the Army High Performance Computing Research Center.

References

[1] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan. An introduction to MCMC for machine learning. *Machine Learning*, 50(1-2):5–43, 2003.

[2] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, pages 394–403, 2006.

[3] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28:135–151, 2001.

[4] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[5] S. Chenney and D. A. Forsyth. Sampling plausible solutions to multi-body constraint problems. In *SIGGRAPH*, pages 219–228, 2000.

[6] P. Diaconis. The markov chain monte carlo revolution. *Bulletin of the American Mathematical Society*, 46(2):179–205, Nov. 2008.

[7] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, pages 519–531, 2007.

[8] W. R. Gilks. *Markov Chain Monte Carlo in Practice*. Chapman and Hall/CRC, 1999.

[9] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.

[10] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, Apr. 1970.

[11] R. Joshi, G. Nelson, and K. H. Randall. Denali: A goal-directed superoptimizer. In *PLDI*, pages 304–314, 2002.

[12] P. Liang, M. I. Jordan, and D. Klein. Learning programs: A hierarchical bayesian approach. In *ICML*, pages 639–646, 2010.

[13] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[14] H. Massalin. Superoptimizer - a look at the smallest program. In *ASPLOS*, pages 122–126, 1987.

[15] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[16] A. F. Neuwald, J. S. Liu, D. J. Lipman, and C. E. Lawrence. Extracting protein alignment models from the sequence database. *Nucleic Acids Research*, 25:1665–1677, 1997.

[17] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[18] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *POPL*, pages 264–276, 2009.

[19] E. Veach and L. J. Guibas. Metropolis light transport. In *SIGGRAPH*, pages 65–76, 1997.

[20] H. S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.