Post-compiler Genetic Software Optimization for Reduced Energy Consumption

Eric Schulte[*], Jonathan Dorn[†], Stephen Harding[*], Stephanie Forrest[*], and Westley Weimer[†]

[*] Department of Computer Science , University of New Mexico , Albuquerque, NM 87131-0001 ,
{eschulte,stharding,forrest}@cs.unm.edu
[†] Department of Computer Science , University of Virginia , Charlottesville, VA 22904-4740,
{dorn,weimer}@cs.virginia.edu

Abstract

Modern compilers typically optimize only executable size and speed, rarely exploring non-functional properties such as power efficiency. Such properties are usually hardware-specific and time-intensive to optimize, with challenging search spaces that may not be amenable to standard dataflow optimizations. We present a post-compilation technique capable of targeting measurable non-functional aspects of software execution for programs that compile to x86 assembly. Our approach combines insights from evolutionary computation, mutational robustness, profile-guided optimization and superoptimization. We iteratively search for program variants that retain required functional behavior while improving non-functional behavior, using selected test cases and predictive modeling to guide the search. The resulting optimizations are validated using real-world performance measurements and a larger held-out test suite. Our experimental results on PARSEC benchmark programs show average energy reductions of 20% for a large AMD system and a small Intel system while maintaining the original program functionality on target workloads.

1. Introduction

From embedded systems to warehouse-scale datacenters, extreme scales of computation continue to grow in importance. At such large scales properties beyond pure performance, such as energy consumption [10] and memory utilization, are of increasing interest. For example, electricity used in data centers in 2010 likely accounted for between 1.1% and 1.5% of total global electricity consumption [27]. Despite this, there is a dearth of program optimizations targeting multiple performance requirements (called *non-functional* in the literature [19] to distinguish them from functional correctness requirements). Those that do exist struggle to understand architectural properties and workload interactions in these domains [59]. Although techniques such as voltage scaling and resource hibernation can be applied at the hardware level to reduce energy consumption [41], software optimizations are largely limited to reducing cycle counts and instruction scheduling [33] for consecutive instructions to lower the switching activity [45, § 4.3.3].

At the same time, notions of "relaxed" program semantics [11, 15, 22, 26] are gaining wider acceptance in several communities. Software engineers are increasingly willing to trade guarantees of semantic preservation for greater reliability and robustness [2, 8, 47, 55], availability and security [48], compile-time algorithmic flexibility [5], performance and quality of service [6, 37, 38] or even "amorphous" systems of varying unreliable components [1]. Interest in such tradeoffs spans domains as diverse as computer graphics [56] and networked servers [48].

In addition, mutation-based improvements to evolve or repair off-the-shelf software are increasingly successful and popular [14, 25, 32, 40]. Analogously to profile-guided optimizations, such techniques typically leverage existing test suites or annotations [62] to ensure that modifications preserve required functionality. Evolutionary computation has already seen significant success at circuit synthesis and optimization tasks (e.g., [28, 60]). We believe that these methods can apply to software as well, leveraging the observation that a wide variety of software is surprisingly "mutationally robust", i.e., robust to simple randomized program transformations which often produce semantically distinct yet functionally neutral program implementations [52].

We propose a post-compilation, software-based "Genetic Optimization Algorithm" (GOA) for discovering tradeoffs and optimizations that improve non-functional software aspects, with a particular focus on reducing power consumption. Our approach generates and maintains a population of variant or mutant programs, selecting those that retain required functionality and also improve a modeled or measured objective function. Our approach thus leverages notions from evolutionary computation (population-based search), mutational robustness (simple mutations yield independent implementations of the same specification), profile-guided optimization (performance measured on workloads) and relaxed notions of program semantics (customizing the software to particular runtime goals and environments provided by a software engineer). Formally, we use evolutionary computation (EC) techniques to search through "neutral spaces" to find alternate program implementations that optimize non-functional aspects of program execution. Evolutionary algorithms such as GOA are particularly good at exploring difficult search spaces [28]. Our approach requires only the assembly code of the program being optimized, a test suite, and a measurable optimization target.

We demonstrate GOA by reducing the energy consumption of PARSEC [9] benchmark programs on both desktop-class Intel and server-class AMD hardware. Our search algorithm optimizes an efficient and accurate architecture-specific linear power model parameterized by hardware counters; our final results are validated using physical wall-socket measurements. We find both hardware- and workload-specific optimizations. Overall, GOA reduced the energy consumption of the PARSEC benchmarks by 20% on average as compared to the best available compiler optimizations.

The contributions of this paper are as follows:

- The GOA technique for optimizing non-functional aspects of post-compilation assembly programs using evolutionary computation and an edit minimization step.
- A specialization of GOA to reduce power consumption, including the use of efficient predictive models.
- An empirical evaluation using PARSEC benchmarks across two microarchitectures. We find that GOA reduces energy consumption by 20% as compared to the best available compiler optimizations, generates optimizations that generalize across different size workloads, and in most cases fully retains program functionality even against held-out test cases.

The next section provides two examples of optimizations found using GOA. We then describe the GOA algorithm more formally (Section 3), report experimental results (Section 4) discuss the relevant background material and related work (Section 5), and discuss the significance of our results and conclude (Sections 6 and 7).

2. Motivating Examples

In this section we motivate GOA by detailing three energy optimizations it found in the PARSEC benchmark suite.

The `blackscholes` program implements a mathematical model of a financial market formalized as a partial differential equation. The benchmark does most of its work in a nested loop. The PARSEC implementation needlessly repeats the main calculations in a way that standard static compiler analyses fail to detect.

GOA generates a number of candidate optimizations, each of which is a slight variation on the original `blackscholes` assembly code. These candidates are then validated dynamically, on an adequate test suite. Those that pass all tests and thus retain required functionality are retained. In addition, we use an efficient model to predict the energy behavior of those candidates based on hardware performance counters. For example, optimizations may change the number of floating point operations or cache accesses. Over time, our approach combines and evaluates the best parts of various candidate optimizations, eventually arriving at a set of changes to different parts of the program. That final optimization is then validated using physical energy measurements.

The final `blackscholes` optimization returned by GOA discovered and removed the redundant calculation inefficiency on both AMD and Intel hardware. Interestingly, the implementation of the optimization found differed between the two architectures. In the Intel case the discovered edit removed a "subl" instruction, preventing multiple executions of a loop, while in the AMD case a similar effect was obtained by inserting a literal address which (due to the density of valid x86 instructions in random data [7]) is interpreted as valid x86 code to jump out of the loop, skipping redundant calculations.

The technique also finds hardware-specific optimizations. The `swaptions` benchmark prices portfolios. On AMD systems our technique reduces the total energy used by 42.5%, compared to the code produced by the least-energy combination of flags to `gcc`. On that particular hardware, our optimization reduces the rate of branch miss-prediction by 42.2%. While it is not practical for general compilers to reason about branch prediction strategies for every possible hardware target, GOA is environment-specific and thus has the potential to find such specialized adaptations.

Finally, our technique finds unintuitive optimizations. In the `vips` image processing application on Intel systems, our technique is able to decrease the total energy used by 20.3%. This is accomplished by an optimization which increases cache misses by 20× while decreasing instructions by 30%, in effect trading increased off-chip communication for decreased computation.

Note that in each of the cases discussed above, no subset of the PARSEC-provided compiler flags and/or the usual `gcc` "–O*x*" flags produced these optimizations in practice. These examples show that beneficial energy optimizations exist which are not exploited by common compilers, suggesting the need for a technique that can automatically find such optimizations.

3. Genetic Optimization Algorithm (GOA)

This section decribes our technique for optimizing non-functional aspects of off-the-shelf assembly programs, which we call GOA (Genetic Optimization Algorithm). An overview of GOA is shown in Figure 1. GOA takes as input the original program source, an adequate test suite capable of exercising program executables, and a scalar-valued objective (or fitness) function indicating the relative desirability of any mutant program that passes the test suite. At a high level, the process repeatedly mutates the program to create candidate optimizations, ultimately returning the one with the best observed fitness value that also passes the test suite.

During compilation and linking, the intermediate assembly code representation of the program is extracted from the build process (Section 3.1) and is used to seed a population of mutated copies of the program (Section 3.3). An EC algorithm (Section 3.2) searches for optimized versions of the original program. Every iteration of the main search loop (1) selects a candidate optimization from the population, (2) transforms it (Section 3.3), (3) links the result into an executable, (4) runs the resulting executable against the supplied test suite, (5) collects performance information for programs

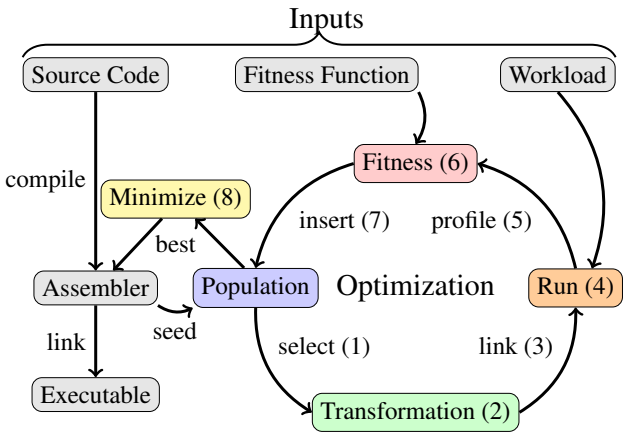

Figure 1: Overview of the program optimization process.

that pass all tests, (6) combines the profiling information into a scalar fitness score using the fitness function (Section 3.4), and (7) reinserts the optimization and its fitness score into the population. The process continues until a runtime budget is exceeded or a desired optimization target is reached. When the algorithm completes, a post-processing step (8) takes the best individual found in the search and minimizes it with respect to the original program (Section 3.5). Finally, the result is linked into an optimized executable.

The remainder of this section details the particulars of this process.

3.1. GOA Inputs

Our algorithm operates on a single assembly file encoding the program, which can either be extracted automatically from the build process, or provided directly. The complete assembly source associated with a C program may be obtained via gcc's "–combine" flag. For C++, manual concatenation may be required. In practice this was straightforward for the PARSEC programs used as benchmarks in this work. Special care may be required to ensure performance-critical library functions are included in the resulting executable, because our technique optimizes only visible assembly code (i.e., not the contents of external libraries).

The algorithm also takes as input a test suite or indicative workload that serves as an implicit specification: an optimized program implementation that passes the test suite is assumed to retain all required functionality [32] but may have improved non-functional aspects. There are two traditional costs associated with adequate test suites, both of which are well-studied in the software engineering literature: the cost of creating the test suite, and the cost of running it. Both are costs that our method shares with profile-guided optimization techniques [43], and we view the task of efficient testing as orthogonal to this work. For test suite construction, we note that many techniques for automated test input and test suite generation are available (e.g., [16]) and that our scenario admits using the original program as an oracle, which dramatically reduces costs. For

Input: Original Program, P : *Program*
Input: Workload, Run : $Program \rightarrow ExecutionMetrics$
Input: Fitness Function, Fitness : $ExecutionMetrics \rightarrow \mathbb{R}$
Parameters: *PopSize*, *CrossRate*, *TournamentSize*, *MaxEvals*
Output: Program that optimizes Fitness

1: **let** $Pop \leftarrow PopSize$ copies of $\langle \mathsf{P}, \mathsf{Fitness}(\mathsf{Run}(P)) \rangle$
2: **let** $EvalCounter \leftarrow 0$
3: **repeat in every thread**
4: **let** $p \leftarrow \mathsf{null}$
5: **if** $\mathsf{Random}() < CrossRate$ **then**
6: **let** $p_1 \leftarrow \mathsf{Tournament}(Pop, TournamentSize, +)$
7: **let** $p_2 \leftarrow \mathsf{Tournament}(Pop, TournamentSize, +)$
8: $p \leftarrow \mathsf{Crossover}(p_1, p_2)$
9: **else**
10: $p \leftarrow \mathsf{Tournament}(Pop, TournamentSize, +)$
11: **end if**
12: **let** $p' \leftarrow \mathsf{Mutate}(p)$
13: $\mathsf{AddTo}(Pop, \langle p', \mathsf{Fitness}(\mathsf{Run}(p')) \rangle)$
14: $\mathsf{EvictFrom}(Pop, \mathsf{Tournament}(Pop, TournamentSize, -))$
15: **until** $EvalCounter \geq MaxEvals$
16: **return** $\mathsf{Minimize}(\mathsf{Best}(Pop))$

Figure 2: High-level pseudocode for the main loop of our genetic optimization algorithm.

the cost of running the test suite, we note that our approach is amenable to test suite reduction and prioritization (e.g., [57]).

Finally, our approach also requires a fitness function (e.g., energy consumption), which we detail in Section 3.4.

3.2. Genetic Optimization Algorithm

Unlike recent approaches that apply evolutionary computation to software engineering (e.g,. [32]), we use a *steady state* EC algorithm [34]. This means that the population is not replaced completely in discrete steps (generations). Instead, individual program variants (candidate optimizations) are selected from the population for additional transformations, and then reinserted. The steady state method simplifies our algorithm, reduces the maximum memory overhead (steady state algorithms require half the memory because the population is never duplicated), and is more readily parallelized.

The pseudocode for GOA is shown in Figure 2. The main evolutionary loop can be run in parallel across multiple threads of execution. Each thread requires access to the population *Pop* and evaluation counter *EvalCounter*.

The population is initialized with a number of copies of the original program (line 1). In every iteration of the main loop (lines 3–15) the search space of possible optimizations is explored by transforming the program using random mutation and crossover operations (described in the next subsection). The probability *CrossRate* controls the application of the crossover operator (lines 6–8). If a crossover is to be performed, two high-fitness parents are chosen from the population via *tournament selection* [44, § 2.3] and combined to form one new optimization (line 8). Otherwise, a single high-fitness optimization is selected. In either case, the candidate optimization is mutated (line 12), its fitness is calculated

(i.e., by linking it and running it on the test suite, see Section 3.4), and it is reinserted into the population (line 13). The steady state algorithm immediately follows, selecting a member of the population for eviction using a "negative" tournament to choose a low-fitness optimization and keep the population size constant (line 14). Note that Fitness returns a maximally poor fitness value for variants that fail any test case, so optimizations that do not retain required functionality are quickly purged from the population. Eventually, the fittest optimization is returned, after a minimization step that removes unnecessary or redundant changes that may have accumulatd (Section 3.5).

We report results using a population of size $MaxPop = 2^9$, a crossover probability of $CrossRate = \frac{2}{3}$, a tournament size of $TournamentSize = 2$ both for selection and eviction, and we run for $MaxEvals = 2^{18}$ fitness evaluations, typically resulting in runs runtimes of 16 hours or less (using 12 threads on a server-class AMD machine) allowing for "overnight" optimization.

3.3. Program Representation and Operations

Candidate optimizations are represented as a linear array of assembly statements [50, 52]. Programs are transformed by mutation and crossover operators defined over the arrays of instructions. The instructions themselves are treated as atomic: arguments to individual instructions are never changed, but instructions can be copied wholesale. This potentially limits the search, but in practice most useful instructions are available to be copied from somewhere else in the program. This decision allows us to avoid the "problem of argumented instructions" (cf. [58]) when modifying assembly code.

The mutation step selects one of the three mutation operations (*Copy*, *Delete* or *Swap*) at random and applies it to locations in the program, selected uniformly at random, with replacement. When crossover is performed, two program locations are selected from within the length of the shorter program to be crossed. Two-point crossover is then applied at these points to generate a single output program variant from the two input program variants, as shown in Figure 3.

The mutation operations are not language or domain specific. They do not create entirely new code. Instead, they produce new arrangements of the argumented assembly instructions present in the original program. Our operators are reasonable extensions of the standard EC mutation and crossover operations, which are typically defined over bitstrings. In our case, the "terminal set" consists of the unique instructions in the original program. Mutation operators explore the search space through small local changes, while crossover can escape local optima, either by combining the best aspects of partial solutions or by taking long jumps in the search space.

3.4. Evaluation and Fitness

The goal of GOA is to optimize a given fitness function. In our energy-optimizing implementation, the fitness function uses

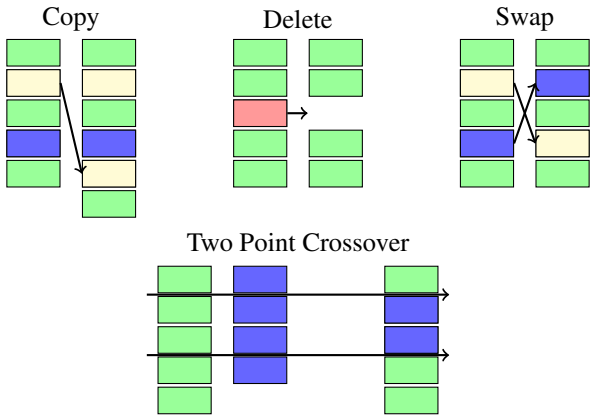

Figure 3: Mutation and Crossover operations on programs represented as linear arrays of argumented assembly instructions.

hardware performance counters [18] captured during test suite execution. In particular, our experiments use a linear energy model (Section 4.3) over the values of those counters.

Test input data were selected to fulfill two requirements: (1) minimize the runtime of each evaluation, and (2) return sufficiently stable hardware performance counter values to allow reliable fitness assessment. The full test suite, which validates that candidate optimizations retain required functionality, need not be the same as the abbreviated test data, which measures non-functional fitness.

3.5. Minimization

The randomized nature of EC algorithms sometimes produces irrelevant or redundant transformations. For example, crossover may produce the best optimization result, but that result may contain a superfluous "no-op" when compared to the original. We prefer optimizations that attain the largest fitness improvement with the fewest changes to the original program. To accomplish this, we include a final minimization step to remove changes that do not improve the fitness. Minimization allows us both to focus on mutations that produce a measurable improvement (Section 4.4) and avoid altering functionality that is not exercised during the fitness evaluation. We only customize the software to the runtime goals and environment provided by the software engineer (Section 5.3).

Our minimization algorithm is based on Delta Debugging [64]. Here, Delta Debugging takes a set of deltas (edit operations) between two versions of a program and determines a 1-minimal subset of those deltas that cause the first program to act like the second. We reduce the best optimization found by the evolutionary search to a set of single-line insertions and deletions against the original (e.g., as generated with the `diff` Unix utility). We then use Delta Debugging to minimize that set with respect to the fitness function. If the application of a particular delta has no measurable effect on the fitness function, we do not consider it to be a part of the optimization. Eliminating such superfluous deltas helps to ensure that we

Program	C/C++ Lines of Code	ASM Lines of Code	Description
blackscholes	510	7,932	Finance modeling
bodytrack	14,513	955,888	Human video tracking
ferret	15,188	288,981	Image search engine
fluidanimate	11,424	44,681	Fluid dynamics animation
freqmine	2,710	104,722	Frequent itemset mining
swaptions	1,649	61,134	Portfolio pricing
vips	142,019	132,012	Image transformation
x264	37,454	111,718	MPEG-4 video encoder
total	225,467	1,707,068	

Table 1: Selected PARSEC benchmark applications.

do not alter untested program functionality. We apply the minimal set of changes identified by Delta Debugging to the original program to produce the final optimized program.

Note that minimization takes place with respect to the fitness function and therefore uses modeled energy (see Section 4.3) rather than physically measured energy. This means that the minimization algorithm maintains an improvement in modeled energy, but may alter the improvement in physically measured energy, depending on the accuracy of the model.

3.6. Algorithm Summary

GOA maintains a population of linear arrays of assembly instructions, mutating and combining them with the goal of optimizing a given objective function while retaining all required functionality as indicated by a test suite. In the next section we evaluate the effectiveness of GOA by applying it to the problem of reducing energy consumption.

4. Experimental Evaluation

Our experiments address three research questions:

- Does GOA reduce energy consumption? (RQ1)
- Do our results generalize to multiple architectures and held-out workloads? (RQ2)
- Do GOA optimizations retain required functionality? (RQ3)

Recall that GOA requires three inputs: a program to be optimized, a fitness function, and a workload or test suite. We evaluate effectiveness and usability against the PARSEC benchmark suite (Section 4.1). For a fitness function, we develop an efficient energy consumption model, based on hardware counter values (Section 4.3). For the test suite we select from the PARSEC suite the smallest inputs that generate a runtime of at least one second on each hardware platform. After optimization completes, we evaluate our results using physical wall-plug measurements of energy. The software optimization tools described in this work including installation and usage instructions are publicly available.[1]

4.1. Benchmark Programs and Systems

We use the popular PARSEC [9] benchmark suite of programs representing "emerging workloads". We evaluate GOA on all of the PARSEC applications that produce testable output and include more than one input set. Testable output is required to ensure that the optimizations retain required functionality. More than one input set is required for our experimental design, which involves both training and held-out data. The eight applications satisfying these requirements are shown with sizes and brief descriptions in Table 1. Of the two PARSEC applications which do not meet these requirements, the `raytrace` program does not produce any testable output and the `facesim` program does not provide multiple input sets; both were excluded from this study.

We evaluate on both an Intel Core i7 system and an AMD Opteron system. The Intel system has 4 physical cores, Hyper-Threading, and 8 GB of memory, and is indicative of desktop or personal developer hardware. The AMD system has 48 cores and 120 GB of memory, and is representative of more powerful server-class machines.

We compare the performance of GOA's optimized executables against the original executable compiled using the PARSEC tool with its built-in optimization flags.[2] We further improve the baseline by selecting "gcc –O3", "gcc –O1", or the PARSEC flags on a per-benchmark basis, choosing whichever alternative has the lowest energy consumption.

4.2. Held-Out Test Suite

We use a large held-out test suite to evaluate the degree to which the optimizations found by GOA customize the program semantics to the training workload. For each benchmark besides `blackscholes`, we randomly generated 100 sets of command-line arguments (and argument values, as appropriate) from the valid flags accepted by the program. Since `blackscholes` operates on a well-structured input file but takes a fixed sequence of arguments, we generated its tests using 100 randomly-generated input files.

Each test was run using the original program and its output as an oracle to validate the output of the optimized program. If the original program rejected the input or arguments (e.g., because some flags cannot be used together), we rejected that test and generated a new one. We also rejected tests for which the original program did not generate the same output when run a second time, or for which the original program took more than 30 seconds to run.

The optimized programs were evaluated by running them with the same inputs and test data, comparing the output against the oracle. In most cases, we used a binary comparison between the output files. However, for `x264` tests producing

[1] https://github.com/eschulte/optimization

[2] PARSEC includes a version of `x264` with non-portable hand-written assembly. To fairly compare against both Intel and AMD, we report `x264` numbers using the portable C implementation.

video output, we used manual visual comparison to determine output correctness.

4.3. Energy Model

Our fitness funtion uses a linear energy model based on process-specific hardware counters similar to that developed by Shen *et al.* [54]. We simplify their model in two ways:

- We do not build workload-specific power models. Instead, we develop one power model per machine trained to fit multiple workloads and use this single model for every benchmark on that machine.
- We do not consider shared resources.

Incorporating these simplifications gives the following model:

$$power = C_{const} + C_{ins} \times \frac{ins}{cycle} + C_{flops} \times \frac{flops}{cycle} + C_{tca} \times \frac{tca}{cycle} + C_{mem} \times \frac{mem}{cycle} \quad (1)$$

$$energy = seconds \times power \quad (2)$$

Total energy (Equation 2) is given by the predicted power (Equation 1) multiplied by the runtime. The values for the constant coefficients are given in Table 2; they were obtained empirically for each target architecture, using data collected for each PARSEC benchmark, the SPEC CPU benchmark suite, and the `sleep` UNIX utility. For each program, we collected the performance counters as well as the average watts consumed, measured by a Watts up? PRO meter. We combined these data in a linear regression to determine the coefficients in Table 2.

Coefficient	Description	Intel (4-core)	AMD (48-core)
C_{const}	constant power draw	31.530	394.74
C_{ins}	instructions	20.490	-83.68
C_{flops}	floating point ops.	9.838	60.23
C_{tca}	cache accesses	-4.102	-16.38
C_{mem}	cache misses	2962.678	-4209.09

Table 2: Power model coefficients.

The disparity between the AMD and Intel coefficients is likely explained by significant differences in the size and class of the two machines. For example, the 13× increase in idle power of the AMD machine as compared to the Intel machine may be explained by the presence of 12 times as many cores, and 15 times as much memory.

Even without our simplifications, the predictive power of linear models is rarely perfect. McCullough *et al.* note that on a simple multi-core system, CPU-prediction error is often 10–14% with 150% worst case error prediction [36]. We checked for the presence of overfitting using 10-fold cross-validation and found a 4–6% difference in the average absolute error, which is adequate for our application. Since we ultimately evaluate energy reduction using physical wall-socket measurements, our energy model is required only to be accurate and efficient enough to guide the evolutionary search.

We find that our models have an average of 7% absolute error relative to the wall-socket measurements. Collecting the counter values and computing the total power increases the test suite runtime by a negligible amount. Thus, our power model is both sufficiently efficient and accurate to serve as our fitness function.

The Intel Performance Counter Monitor (PCM) counter may be used to estimate energy consumption. We did not use this counter both because the model used to estimate energy is not public, and because it estimates energy consumption for an entire socket and does not provide per-process energy consumption. Relying on the PCM would reduce the parallelism available to our GOA implementation.

4.4. RQ1 — Reduce Energy Consumption

Table 3 gives the main experimental results. The "Energy Reduction" columns report energy reduction while executing the tests by the optimized program, as measured physically and compared to the original. For example, if the original program requires 100 units and the optimized version requires 20, that corresponds to an 80% reduction.

GOA found optimizations that resulted in large reductions in energy consumption in many cases, with the overall reduction on the supplied workloads averaging 20%. Although in some cases—such as in `bodytrack` on AMD or `bodytrack`, `ferret`, `fluidanimate`, `freqmine` and `x264` on Intel—GOA failed to find optimizations that provided significant improvement in energy consumption, it did discover optimizations that reduced energy consumption by an order of magnitude for `blackscholes` and over one-third for `swaptions` on both systems. We find that CPU-bound programs are more amenable to our technique than those that perform large amounts of disk IO. This result suggests that GOA is likely better at generating efficient sequences of executing assembly instructions than at improving patterns of memory access. Overall, the average energy reduction is 20% for all programs and architectures, or 39% considering only those programs with non-zero improvement.

While some optimizations are easily analyzed through inspection of assembly patches (e.g., the deletion of "`call im_region_black`" from `vips` skipping unnecessary zeroing of a region of data), many optimizations result in unintuitive assembly changes which are most easily analyzed using profiling tools. Such inspection shows that discovered optimizations run the gamut from removing explicit semantic inefficiencies in `blackscholes` to re-organizing assembly instructions in `swaptions` and `vips` in such a way as to decrease the rate of branch mispredictions. The AMD versions of `fluidanimate` and `x264` seem to improve performance by reducing idle cycles spent waiting for off-chip resources. See Section 2 for a detailed discussion of some optimizations.

	Program Changes				Energy Reduction				Runtime Reduction		Functionality	
	Code Edits		Binary Size		Training		Held-Out		Held-Out		Held-Out	
Program	AMD	Intel	AMD	Intel	AMD	Intel	AMD	Intel	AMD	Intel	AMD	Intel
blackscholes	120	3	-8.2%	0%	92.1%	85.5%	91.7%	83.3%	91.7%	81.3%	100%	100%
bodytrack	19656	3	-38.7%	0%	0%	0%	0.6%	0%	0.3%	0.2%	92%	100%
ferret	11	1	84.8%	0%	1.6%	0%	5.9%	0%	-7.9%	-0.1%	100%	100%
fluidanimate	27	51	-3.3%	11.4%	10.2%	0%	—	—	—	—	6%	31%
freqmine	14	54	18.7%	34.9%	3.2%	0%	3.3%	-1.6%	3.2%	0.1%	100%	100%
swaptions	141	6	27.0%	18.5%	42.5%	34.4%	41.6%	36.9%	42.0%	36.6%	100%	100%
vips	57	66	-52.8%	0%	21.7%	20.3%	21.3%	—	29.8%	—	100%	100%
x264	34	2	0%	0%	8.3%	0%	9.2%	0%	9.8%	0%	27%	100%
average	57.7	23.3	3.4%	8.1%	22.5%	17.5%	24.8%	19.8%	24.1%	19.7%	78.1%	91.4%

Table 3: Results of optimizing PARSEC applications to reduce energy consumption. The "Code Edits" column shows the number of unified diffs between the original and optimized program assembler. The "Binary Size" column shows the change in size of the compiled executable. The "Energy Reduction" columns report the physically measured energy reduction compared to the original required to run the tests in the fitness function ("Training Workload") or to run all other PARSEC workloads for that benchmark ("Held-Out Workloads"). The "Runtime Reduction" columns report the decrease in runtime compared to the original. In some cases, the measured energy reduction is statistically indistinguishable from zero ($p > 0.05$). Note that for some benchmarks (e.g., `bodytrack`), although there is no measured improvement, the minimization algorithm maintains modeled improvement, resulting in a new binary. We do not report energy reduction for workloads for which the optimized variant did not pass the associated tests (indicated by dashes). The "Functionality on" columns report the accuracy on the held-out test suite.

4.5. RQ2 — Generality

We evaluate the generality of our results on the two architectures and using the held-out test cases described earlier.

The "AMD" and "Intel" columns in Table 3 show results for the two architectures (described in Section 4.1). On both architectures, optimizations fix branch mispredictions in `swaptions` and `vips`; rely on relaxed notions of program semantics, such as by eliminating unnecessary loops in `blackscholes`; or remove redundant zeroing behavior in `vips`.

The GOA technique appears to find more optimizations on AMD than Intel. We hypothesize that the more significant energy draws of the larger server-class AMD system provided more room for improvement. It is also worth noting that the two programs which do admit more improvement on the AMD system than on the Intel system (`fluidanimate` and `x264`) are also the only two programs for which we find significant failures on held-out test data, which may indicate that workload-specific improvements or customizations are more easily accomplished (or more readily rewarded) on the AMD than the Intel system. Despite these differences, GOA substantially reduces energy consumption on both AMD-based server-class and Intel-based workstation-class systems.

We also evaluate generality in terms of held-out workloads. GOA only has access to the supplied workload (Section 3.2). The PARSEC benchmark includes multiple workloads of varying size for each of our benchmark applications. The "Energy Reduction on Held-Out Workloads" columns in Table 3 show how optimizations learned on the smaller workloads apply to the larger held-out workloads.

Overall, we find that performance gains on the training workload generalize well to workloads of other sizes. The energy reduction of 20% on the training workloads is similar to the 22% we see on held-out workloads. (However, when the optimizations customized to the training workload change the semantics on the held-out workload and produce different answers, the energy consumption varies dramatically. On Intel, `fluidanimate` and `vips` consume significantly more energy, while on AMD `fluidanimate` consumes significantly less.) We attribute this improvement against held-out workloads to their increased size, which leads to a larger fraction of runtime spent in the inner loops where most optimizations are located. This generalization is important for the practical application of this technique. The training workload must execute quickly, as it is run as part of the inner loop of the GOA algorithm.

Although high-quality training data remains an essential input to our approach, the energy reductions found by GOA in this experiment generalized beyond particular training data used, and no special care was required in selecting training workloads, as the defaults supplied as part of the PARSEC benchmark proved sufficient.

4.6. RQ3 — Functionality and Relaxed Semantics

A strength of our technique is its ability to tailor optimizations to the workload and architecture available [17, 43]. In many situations (e.g., if the deployment scenario is controlled, as in a datacenter, or if the available test suite is extensive) this is advantageous, and "relaxed semantics" may be an acceptable tradeoff for an improvement to non-functional properties. However, in some situations it may be more important to replicate original program behavior even in untested scenarios. We constructed a set of held-out test cases (see 4.2) and evaluated untested behavior by measuring the accuracy of our

optimizations on these tests (the final columns in Table 3).

Recall the optimizations are required to pass all available held-in test cases. The "Functionality on" columns show that the optimizations we find generally behave just as the original, even on held-out tests not seen during the optimization process. The exceptions are `x264` (in which the AMD optimization works across every held-out input, but does not appear to work at all with some option flags) and `fluidanimate` (where the optimizations appeared to be brittle to many changes to the input, including workloads of different sizes). Notably, the minimization step (Section 3.5) ensures that changes to parts of the program not exercised by the training set are likely to be dropped (because removing those changes does not influence the energy consumption on the training set). Anecdotally we note that the un-minimized optimizations typically showed worse performance on the held-out tests than did the minimized optimizations. Although not a proof, this gives confidence that our technique produces optimizations that retain, or only slightly relax, the original program semantics.

Developers concerned with retaining the exact behavior of the original program could produce additional test inputs via a number of off-the-shelf random testing techniques (e.g., [16, 20, 31, 53]) and use the original program's behavior as the oracle. This would restrict the search to considering only optimizations that are even more similar to the original. It would also increase the time cost for running GOA, but does not limit its applicability. We note the increasing willingness to trade guarantees of exact semantic preservation for other desired system properties [1, 2, 5, 6, 8, 37, 38, 48, 55, 56]. Taken together, the growing popularity of acceptability-oriented computing [46, 47, 48] and the difficulty of manually optimizing for energy [17, 33, 45], suggest that automatically reducing energy and manually focusing on semantics is profitable tradeoff compared to the traditional converse.

4.7. Threats to Validity

Although the experimental results suggest that GOA can optimize energy, the results may not generalize to other non-functional properties or to other programs. One threat to generality is that energy utilization and modeling is machine- or architecture-specific, and our optimizations are only individually valid for a particular target architecture. We attempt to mitigate this threat by considering two disparate architectures. We note that this is a general issue for performance optimization [39] and not specific to our approach. A second threat is that energy consumption can differ dramatically by workload—optimizations on training test cases may not be as effective when deployed. We mitigate this threat by physically measuring energy reduction on held-out workloads. A third threat is that our approach requires high-quality test cases (or specifications, etc.). If the test suite does not ensure that the implementation adheres to its specification, the resulting optimizations may over-customize the program to the environment and specific workload.

The performance of this post-compiler optimization technique may be dependent upon the compiler used to generate the assembly code used as input to the technique. Our only evaluation was limited to GCC, and thus it is possible that our results will not generalize to assembler produced by other compilers (e.g., our improvement of branch prediction may not be found in a compiler which produces code with better branch prediction behavior).

Finally, by embracing relaxed program semantics we acknowledge that our optimizations may change the behavior of the program. We view this as an advantage, especially if it allows us to improve program behavior (cf. program repair [25, 32], which also necessarily changes implementation semantics) while still handling supplied workloads correctly. Our experimental use of held-out test suites provides one way to assess the magnitude of this issue for cases where relaxed semantics may not be appropriate.

5. Background and Related Work

At a high level, our approach combines (1) fundamental software engineering tools such as compilers and profilers, (2) mutation operations and algorithms from evolutionary computation, (3) relaxed notions of program semantics to customize software to a specified environment, and (4) software mutational robustness and neutral spaces. In the remainder of this section we review briefly the most relevant related work across these diverse fields.

5.1. Compilers and Profilers

Compilers. Optimizing compilers are well-established in both research and industrial practice [3]. Our work is orthogonal to traditional compilation tools and compiler optimizations, providing *post*-compilation optimizations that refine compiler-generated assembly code.

Traditional compiler optimizations consider only those transformations that are provably semantics-preserving (e.g., as discovered by dataflow analysis). In practice, the C language includes undefined behavior which may lead to compiler- and architecture-dependent runtime behavior. Compilers guarantee the semantics only of programs that do not use such undefined behaviors (often with surprising effects, such as compilers eliding null checks in programs which rely on undefined behavior [61, §3.3.4]).

The great effort required to prove transformations to be semantics-preserving, combined with the great differences in energy consumption across machines, means that providing semantics-preserving compiler optimizations to reduce energy consumption remains unlikely in the near future. For example, a feature request that the popular LLVM compiler add an "–Oe" flag to optimize for energy was rejected with the counter-suggestion to use the established optimizations for speed instead.[3]

[3] http://llvm.org/bugs/show_bug.cgi?id=6210

Profiling. Profiling techniques have long been used to guide both automated and manual optimizations [21]. Our technique uses hardware counters, allowing for fine-grained measurements of hardware events without requiring virtualization or instrumentation overheads. This allows test programs to operate at native speeds, which is critical in a search that explores hundreds of thousands of mutants. Our prototype implementation uses the Linux Perf [18] framework to collect hardware counters on a per-process basis.

Profile Guided Optimization. Profile guided optimization (PGO) techniques combine test runs on data characteristic of a projected workload with instrumented programs (or, more recently, with hardware counters) to profile program runtime behavior. These profiles then guide the application of standard compiler optimizations to improve runtime performance, especially to reposition code to reduce instruction loads [43].

PGO can be used to tailor optimizations to a particular workload and architecture. However, the actual transformations provide the same guarantees and limitations of traditional compiler passes. That is, PGO typically narrows its search space by focusing on particular parts of the program (e.g., hot paths). By contrast, our work broadens the scope of possible optimizations to generic mutations that are applied randomly, and it admits a relaxed notion of program semantics.

Auto-tuning. Auto-tuning extends PGO, providing additional performance enhancements and the ability to more aggressively customize an application to any given hardware. It often requires that applications be written in a distinct modular style. For example, the exceptional performance gains found by the popular FFTW [17] required that the fast Fourier transform (FFT) source code be written in "codelets" (small optimized sub-steps) which were then combined into hardware dependent "plans."

Auto-tuning addresses the same problem as this work, namely that "computer architectures have become so complex that manually optimizing software is difficult to the point of impracticality." [17, § 5] Unlike auto-tuning, however, our approach does not require that the software be written in any special manner and applies to arbitrary assembly code.

Superoptimization. Massalin's classic work introducing superoptimization [35] exhaustively tested all possible combinations of instructions in Motorola's 68020 assembly language to find the fastest possible implementation of simple programs up to 14 instructions in length. The more recent MCMC project [49] leverages increased computing power and heuristic search to find optimizations on the same scale of ~10 instructions in the much larger x86 instruction set.

Despite its impressive results, the MCMC technique is not directly comparable to this work. They focus on sequences of assembly instructions of ~10 lines in length while we operate on entire programs (hundreds of thousands of lines of assembly on average). Because of the extremely small size and simple functionality of their benchmark code, they are able to begin their heuristic search using a *random* sequence of x86 assembly instructions (using all instructions, including vector operations), and work back from these random sequences to functional code. It is not feasible to transform random sequences of assembly instructions into whole programs with hundreds of thousands of lines of assembly code and sophisticated functionality.

Both approaches to superoptimization use simple tests of random sequences of assembly instructions to find faster assembly sequences than those accessible through traditional compiler optimizations. Although GOA addresses an entirely different scale in terms of functional and size complexity of the programs analyzed, and thus does not use random sequences of instructions or make use of the entire available x86 instruction set, we exploit the same insights that optimal instruction sequences are often not directly accessible through semantics-preserving operations and that tests can restrict attention to desired modifications. Ultimately, we see MCMC as an orthogonal optimization, and one that could even be used in conjunction with our technique (e.g., as an alternating phase targeting the hottest profiled paths).

5.2. Evolutionary Computation

Program Repair. Evolutionary computation is one of several techniques that has been used to automatically repair program defects by searching for patches (sequences of edits) [14, 25, 32, 40]. This approach has been applied at the assembly level [50, 51] but differs from the current work in that program repair addresses a discrete single-dimensional objective (a successful repair passes all test cases) rather than a continuous refinement (reducing power consumption).

Evolutionary Improvement. Previous work on the evolutionary improvement of software has been limited to the mutation of abstract syntax tree representations of simple ~10-line C functions [63]. While optimizations were found which were not possible using standard compiler transformations, the small program size and simplified build environment mean the results may not generalize to legacy software.

Exploring Tradeoffs. Evolutionary computation techniques have recently been used to explore tradeoffs between execution time and visual fidelity in graphics shader programs [56]. This work is similar in spirit to ours but lacks an implicit specification through test cases: instead, all mutants are valid and a Pareto-optimal frontier of non-dominated options with respect to execution time and visual fidelity is produced.

5.3. Relaxed Program Semantics

Software engineers are increasingly willing to trade full semantic preservation for non-functional properties of software (examples are given in Section 1). For example "loop perforation" may be used to remove loop iterations to reduce program resource requirements while maintaining an acceptable Quality of Service (QoS) [37, 38]. Similarly adaptive systems may

change program behavior by switching function implementations [5] at runtime in response to QoS monitoring [23]. A formal system has even been developed which can be used to prove "acceptability properties" of some applications of the "relaxed" program transformations [11].

We view our technique as partially customizing software to particular runtime goals and environments provided by a software engineer. While we do change explicitly programmed behavior (e.g., removing loop iterations, dropping calculation, etc.), we do not use the notion of QoS, rather we we always give the *exact* right answer on tested inputs, but we we might break the program for some untested inputs.

5.4. Mutational Robustness and Neutral Spaces

Recent work [52] showed that software functionality is suprisingly robust to random mutations similar to those used in this work, with over 30% of mutations producing *neutral* program variants that still pass an original test suite. These results were obtained over a wide variety of software (including both large open source projects and extremely well tested benchmarks from the software testing community [24]). Many of these neutral mutations introduced non-trivial algorithmic changes to the program, yielding new implementations of the program's implicit specification.

These results help explain how our optimization technique can modify programs without "breaking" them by violating their implicit specifications, and provides a rational for why this approach is able to build "smart" optimizations from such "dumb" program transformations.

6. Discussion

The results of our experiment support the claim that GOA can significantly reduce energy consumption on PARSEC benchmark programs in a way that generalizes to multiple architectures and to held-out workloads and tests. In this section, we discuss the relationship between our work and traditional biological notions of trait selection (optimization), and we outline promising directions for future work.

6.1. Heritability of Desired Traits

Predating Darwin's Origin of Species [13], breeders used a mathematical model termed the *Breeder's Equation* to measure the effectiveness with which they could select for particular traits. We use a modern formulation of the breeder's equation to investigate both heritability as well as the dependencies between traits. The *Multivariate Breeder's Equation* (Equation 3), in which $\Delta\hat{Z}$ is a vector representing the change in phenotypic means, G is a matrix of the additive variance-covariance between traits and β is a vector representing the strength of selection, provides insight not only into heritability, but also into likely side effects of optimizing a given phenotypic trait in terms of other uncontrolled traits.

$$\Delta\hat{Z} = G\beta \tag{3}$$

Biologists use the multivariate breeder's equation to dissect the effect of natural selection into measurable phenotypic traits by regressing phenotypic traits β against fitness [12, Chpt. 4]. In our work, fitness is directly measurable in kilowatt hours, and hardware counters measure the phenotypic traits, thus the biological analysis leads directly to our simple energy model of hardware counters (Section 4.3), which is used as a fitness function creating a selection gradient analogous to β in Equation 3 and is derived in the same way.

Insights from biological work in this area also informed our choice of much larger populations and significantly higher rates of crossover than previous applications of evolutionary techniques to software engineering [32, 56].

6.2. Future Work

Other Architectures. To date, we have applied GOA only to x86 assembly code. Our program representations and mutation operations quite general, and we believe that GOA would apply equally well to other instruction sets, such as ARM [50] or Java bytecode [42]. We note, however, that the high density of x86 instructions in random data [7] may have played a role in our results (Section 2),

Co-evolutionary Model Improvement. Our approach could be extended to iteratively refine models which predict measurable values from hardware performance counters (such as the energy model used in this work). At a high level:

1. Build an initial model from hardware counters and empirical measurements across multiple benchmark programs.
2. Evolve benchmark variants that maximize the difference between the model and reality.
3. Re-train the model using the evolved versions of benchmark programs.

Assuming that the measurable quantity predicted by the model (e.g., energy) can be changed only a limited amount, the evolutionary process will find and exploit errors in the model. Adding individuals evolved to exploit these errors to the model training data train the model would improve subsequent versions of the model. Over multiple iterations, this *competitive co-evolution* [4] between the model and the candidate optimizations should improve both the model and the search.

Mathematical Analysis. Our energy reduction optimizations of `vips` on both systems produce optimized versions that (despite running for fewer cycles) generate significantly more page faults than the original. It would be desirable to predict unintuitive results of optimization such as these, i.e. program features that are not included in the fitness function or energy model.

The concept of *Indirect selection* [12, Chpt. 6] suggests an approach for predicting some side effects of the search.

Indirect selection is defined as the impact of selection on properties (traits) that are not themselves targets of selection but are strongly correlated with the selected traits. Variance-covariance matrices (G in Equation 3) could be used to predict the effects of optimization on program characteristics that are not included directly in the fitness function but likely affected by the optimization process. To do this would require constructing and analyzing the variance-covariance matrix of traits of neutral mutations before the optimization run on a per-program basis.

Compiler Flags. Finding effective combinations and orderings of compiler passes has long been an open research question, and it is known that no single sequence of compiler passes is optimal for all programs [29] or even for all methods in a single program [30]. Our technique could be extended to include multiple sub-populations, each generated using unique combinations of compiler optimizations. By allowing each sub-population to search independently for optimizations and occasionally exchanging high-fitness individuals among the populations, it may be possible to mitigate this problem.

7. Conclusion

We present an automated post-compilation technique for optimizing non-functional properties of software, such as energy consumption. Our Genetic Optimization Algorithm (GOA) combines insights from evolutionary computation, mutational robustness, profile-guided optimization, and superoptimization. GOA is a steady-state evolutionary algorithm, which maintains a population of candidate optimizations (assembly programs), using randomized operators to generate variations, and selecting those that improve an objective function (the power model) while retaining all required functionality expressed in a test suite.

We describe experiments that optimize the PARSEC benchmarks to reduce energy consumption, evaluated via physical wall-socket power measurements. The use case is an embedded deployment or datacenter where the program will be run multiple times. Our technique successfully reduces energy consumption by 20% on average. Our results show that GOA is effective on multiple architectures, is able to find hardware-specific optimizations and to correct inefficient program semantics, that the optimizations found generalize across held-out workloads, and that in most cases the optimizations retain correctness on held-out test cases.

The Genetic Optimization Algorithm is: *powerful*; significantly reducing energy consumption beyond the best available compiler optimizations and capable of customizing software to a target execution environment, *simple*; leveraging widely available tools such as compilers and profilers and requiring no code annotation or technical expertise and *general*; using general program transformations from the genetic algorithm community, able to target multiple measurable objective functions, and applicable to any program which compiles to x86 assembly code.

8. Acknowledgments

{FIXME ...add acknowledgments...}

References

[1] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F Knight Jr, Radhika Nagpal, Erik Rauch, Gerald Jay Sussman, and Ron Weiss. Amorphous computing. *Communications of the ACM*, 43(5):74–82, 2000.

[2] David H. Ackley and Daniel C. Cannon. Pursue robust indefinite scalability. *Proc. HotOS XIII, Napa Valley, California, USA*, 2011.

[3] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[4] Peter J Angeline and Jordan B Pollack. Competitive environments evolve better solutions for complex tasks. In *ICGA*, pages 264–270, 1993.

[5] J. Ansel, C. Chan, Y.L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. *PetaBricks: a language and compiler for algorithmic choice*, volume 44. ACM, 2009.

[6] Woongki Baek and Trishul M Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, volume 45, pages 198–209. ACM, 2010.

[7] E.G. Barrantes, D.H. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. *ACM Transactions on Information and System Security (TISSEC)*, 8(1):3–40, 2005.

[8] J. Beal and Gerald Jay Sussman. Engineered robustness by controlled hallucination. In *AAAI 2008 Fall Symposium "Naturally-Inspired Artificial Intelligence"*, November 2008.

[9] C. Bienia, S. Kumar, J.P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[10] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 83–94, 2000.

[11] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *Programming Language Design and Implementation*, pages 169–180, 2012.

[12] Jeffrey K Conner and Daniel L Hartl. *A primer of ecological genetics*. Sinauer Associates Incorporated, 2004.

[13] Charles Darwin. *On the origin of species*, volume 484. John Murray, London, 1859.

[14] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *International Conference on Software Testing, Verification, and Validation*, pages 65–74, 2010.

[15] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *International Symposium on Microarchitecture*, pages 449–460, 2012. DOI=10.1109/MICRO.2012.48.

[16] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *Transactions on Software Engineering*, 38(2):278–292, 2012.

[17] M. Frigo and S.G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384, 1998.

[18] Thomas Gleixner. Performance counters for Linux, 2008. http://lwn.net/Articles/310176/.

[19] Martin Glinz. On non-functional requirements. In *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International*, pages 21–26. IEEE, 2007.

[20] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Programming Language Design and Implementation*, pages 213–223, 2005.

[21] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler (with retrospective). In *Programming Languages Design and Implementation*, pages 49–57, 1982.

[22] Mark Harman, William B. Langdon, Yue Jia, David R. White, and Andrea Arcuriand John A. Clark. The gismoe challenge: constructing the pareto program surface using genetic programming to find better programs (keynote paper). In *Automated Software Engineering*, pages 1–14, 2012. DOI=10.1145/2351676.2351678.

[23] H. Hoffmann, J. Eastep, M.D. Santambrogio, J.E. Miller, and A. Agarwal. Application heartbeats for software performance and health. In *ACM SIGPLAN Notices*, volume 45, pages 347–348. ACM, 2010.

[24] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow-and control flow-based test adequacy criteria. In *International Conference on Software Engineering*, pages 191–200, 1994.

[25] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *International Conference on Sofware Engineering*, 2013.

[26] Christoph M. Kirsch and Hannes Payer. Incorrect systems: it's not the problem, it's the solution. In *Design Automation Conference*, pages 913–917, 2012. DOI=10.1145/2228360.2228523.

[27] Jonathan Koomey. Growth in data center electricity use 2005 to 2010. *Oakland, CA: Analytics Press. August*, 1:2010, 2011.

[28] John R. Koza, Forrest H. Bennett III, David Andrew, and Martin A. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann, 1999.

[29] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *ACM SIGPLAN Notices*, volume 38, pages 12–23. ACM, 2003.

[30] Prasad A Kulkarni, David B Whalley, Gary S Tyson, and Jack W Davidson. Exhaustive optimization phase order space exploration. In *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*, pages 13–pp. IEEE, 2006.

[31] K. Lakhotia, M. Harman, and P. McMinn. A multi-objective approach to search-based test data generation. In *Genetic and Evolutionary Computation Conference*, pages 1098–1105, 2007.

[32] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, pages 3–13, 2012.

[33] M.T.-C. Lee, V. Tiwari, S. Malik, and M. Fujita. Power analysis and minimization techniques for embedded dsp software. *Very Large Scale Integration Systems*, 5(1):123–135, 1997.

[34] Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. http://cs.gmu.edu/~sean/book/metaheuristics/.

[35] H. Massalin. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News*, 15(5):122–126, 1987.

[36] John C McCullough, Yuvraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C Snoeren, and Rajesh K Gupta. Evaluating the effectiveness of model-based power characterization. In *USENIX Annual Technical Conf*, 2011.

[37] S. Misailovic, D.M. Roy, and Martin Rinard. Probabilistically accurate program transformations. *Static Analysis*, pages 316–333, 2011.

[38] S. Misailovic, S. Sidiroglou, H. Hoffmann, and Martin Rinard. Quality of service profiling. In *International Conference on Software Engineering*, pages 25–34, 2010.

[39] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Architectural support for programming languages and operating systems*, pages 265–276, 2009.

[40] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. In *International Conference on Sofware Engineering*, pages 772–781, 2013.

[41] Kevin J Nowka, Gary D Carpenter, Eric W MacDonald, Hung C Ngo, Bishop C Brock, Koji I Ishii, Tuyet Y Nguyen, and Jeffrey L Burns. A 32-bit powerpc system-on-a-chip with support for dynamic voltage scaling and dynamic frequency scaling. *Solid-State Circuits, IEEE Journal of*, 37(11):1441–1447, 2002.

[42] Michael Orlov and Moshe Sipper. Flight of the FINCH through the Java wilderness. *Transactions on Evolutionary Computation*, 15(2):166–192, 2011.

[43] Karl Pettis and Robert C Hansen. Profile guided code positioning. In *ACM SIGPLAN Notices*, volume 25, pages 16–27. ACM, 1990.

[44] R. Poli, W.B. Langdon, and N.F. McPhee. *A field guide to genetic programming*. Lulu Enterprises Uk Ltd, 2008.

[45] Sherief Reda and Abdullah N. Nowroz. Power modeling and characterization of computing devices: a survey. *Electronic Design Automation*, 6(2):121–216, 2012.

[46] Martin Rinard. Acceptability-oriented computing. In *Object-oriented programming, systems, languages, and applications*, pages 221–239, 2003.

[47] Martin Rinard. Survival strategies for synthesized hardware systems. In *Formal Methods and Models for Co-Design*, pages 116–120, 2009.

[48] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and Jr. William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Operating Systems Design and Implementation*, pages 21–21, 2004.

[49] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems*, 2013.

[50] Eric Schulte, Jonathan DiLorenzo, Stephanie Forrest, and Westley Weimer. Automated repair of binary and assembly programs for cooperating embedded devices. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[51] Eric Schulte, Stephanie Forrest, and Westley Weimer. Automatic program repair through the evolution of assembly code. In *Automated Software Engineering*, pages 33–36, 2010.

[52] Eric Schulte, Zachiary Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, To Appear, 2013.

[53] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Foundations of Software Engineering*, pages 263–272, 2005.

[54] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. Power containers: An OS facility for fine-grained power and energy management on multicore servers. In *Architectural support for programming languages and operating systems*, pages 65–76, 2013.

[55] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: automatic software self-healing using rescue points. In *Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2009.

[56] Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic programming for shader simplification. *ACM Transactions on Graphics*, 30(5), 2011.

[57] Adam M. Smith and Gregory M. Kapfhammer. An empirical study of incorporating cost into test suite reduction and prioritization. In *Symposium on Applied Computing*, pages 461–467, 2009.

[58] Thomas Sperl. Taking the redpill: Artificial evolution in native x86 systems. *CoRR*, abs/1105.1534, 2011.

[59] Lingjia Tang, Jason Mars, Xiao Zhang, Robert Hagmann, Robert Hundt, and Eric Tune. Optimizing Google's warehouse scale computers: The NUMA experience. In *High Performance Computer Architecture*, pages 188–197, 2013.

[60] Zdenek Vasícek and Lukás Sekanina. A global postsynthesis optimization method for combinational circuits. In *Design, Automation and Test in Europe*, pages 1525–1528, 2011.

[61] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Improving integer security for systems with kint. In *Operating Systems Design and Implementation*, pages 163–177, 2012.

[62] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis*, pages 61–72, 2010.

[63] David R. White, Andrea Arcuri, and John A. Clark. Evolutionary improvement of programs. *Transactions on Evolutionary Computation*, 15(4):515–538, 2011.

[64] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *Foundations of Software Engineering*, pages 253–267, 1999.