Improving Program Acceptability Through Source Code Transformations

Ph.D. Dissertation Proposal

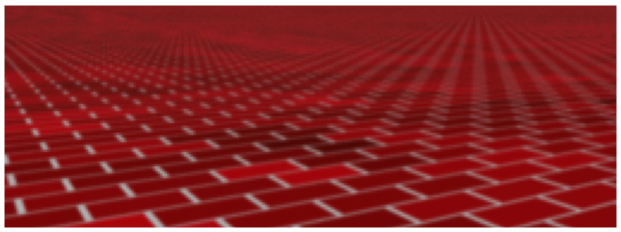
Jonathan Dorn
dorn@virginia.edu

1 Introduction

What does it mean for software to be acceptable? The question has an inherently human element. Unlike functional specifications, security requirements, or performance bounds, which may be documented and measured against that documentation, acceptability is simply a determination of whether the software meets the user's needs [76]. The latter frequently subsumes the former: if software does not function, it is unlikely to be acceptable. However, acceptability also includes "non-functional properties" [41], both documented and undocumented. This convenient phrase has been defined in a number of ways by a number of authors (Glinz [41] and Chung and Prado Leite [25] have collected several different definitions). In the broadest sense, this category contains any desired program properties beyond the functional specification.

The ambiguity of the term belies the very real impact of such properties. For example, safety properties [43] of airline software are responsible for the safety of hundreds of millions of passengers per year in the U.S. alone [9]. Software developers account for 7 out of every 1000 workers in the U.S. [2, 8], each of whom spends an order of magnitude more time reading code than writing it [62]. Data centers may account for over 1% of global energy consumption [58], an amount significantly affected by the efficiency of the applications deployed in those centers [93]. The stunning visuals that contribute to the billions of dollars brought in by 3D movies each year [1] are generated by programs that must balance the competing properties of speed and image quality. Figure 1 demonstrates the need for an acceptable trade-off with a simple program, orders of magnitude smaller than the programs Pixar uses to draw textures [75].

In part because of the range of factors affecting acceptability, researchers have proposed a variety of methods to help evaluate or ensure it. Many of these methods require significant additional effort beyond the challenge of writing acceptable software. For example, one might construct a formal argument for the safety of a system [46] or conduct targeted code reviews for readability [56]. Other approaches increase the effort of developing the software, requiring a particular design to allow performance optimization [39] or


(a) Visually acceptable, but unacceptable run time.

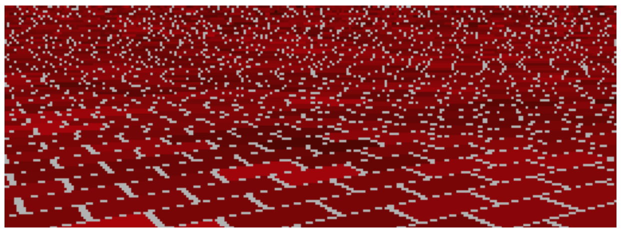

(b) Acceptable run time, but visually unacceptable.

Figure 1: Renderings of a brick wall. The left image has acceptable visual quality, but required significantly more time to render. The right image was produced much more quickly, but shows low visual quality. Both images were produced at the same resolution; the pixels are enlarged to show detail.

evaluating (or approximating) challenging integrals by hand [13].

Our research aims to provide *automatic* assistance to developers using *existing artifacts* (i.e., the programs they would have written anyway). We leverage the hypothesis that competent programmers produce software that is close to acceptable [34]. That is, in cases where the software is not yet acceptable, they tend to have written a program with many of the desired properties and with a structure that can support the rest. Our key technical insight is that local source code transformations—operations that change one part, possibly just one expression, of a program, producing a slightly different program—can bridge the gap to acceptability. We consider two classes of transformations: those that impart the desired properties by construction (Sections 2 and 4) and those that may improve properties in some cases (Sections 2 and 3). For the latter class, we observe that acceptability properties can often be approximated by domain-specific metrics. The combination of a mechanism for generating candidate programs via source code transformations and a mechanism for approximating acceptability suggest interpreting the creation through the lens of search-based software engineering (SBSE) [45]. That is, we will attempt to improve acceptability by searching a set of programs to find one that is highly acceptable. Our set of programs—the *search space*—consists of all programs we can generate by repeatedly applying source code transformations to the human-written original.

We apply this insight to three problem domains. In the first research thrust (Section 2), we use program transformations and search-based techniques to balance run time and visual quality in graphics programs. Second, we employ transformations and search to reduce the energy used while running large applications in our second research thrust (Section 3). Finally, our third research thrust (Section 4) investigates the impact of transformations of coding style on the productivity of programmers working with the transformed code.

2 Visual and Performance Acceptability in Graphics Programs

In our first research thrust, we apply program transformations to improve the visual and runtime quality of graphics programs. We derive provably correct transformations for simple cases and apply search-based techniques to identify transformations that provide acceptable approximations when the derivation is intractable.

Computer graphics technologies underlie multi-billion dollar movie and video game industries. The use of 3D visualization technologies is growing in fields as diverse as medical imaging [38], seismic exploration [36], and aerospace engineering [53]. In all of these scenarios, the software must maintain an internal representation of a virtual environment and determine for each image (or *frame*) the portion of the environment to draw. Many of these applications are *interactive*, updating the image in response to user inputs, which requires that the computations to do so happen quickly enough to maintain the user's productivity [35]. The speed of rendering is important in non-interactive applications as well, such as rendering computer animated movies, since validating an image within an hour is much more efficient than waiting 10 or more hours [99].

This performance must be achieved while maintaining a sufficient level of visual quality. Despite sophisticated physical simulations of the real world [77], rendered images are still susceptible to a form of visual error known as *aliasing*, caused by sampling a complicated image too coarsely (for example, at the regularly spaced pixels on a screen) [28]. Aliasing may appear in many different forms, including as jagged lines in place of smooth ones, small details that seem to appear and disappear, or regular features that appear to cluster instead of being evenly distributed. Such effects may be merely distracting to viewers, or they may make it difficult to distinguish the intended information from the aliasing.

We propose to focus on improving the run time and image quality resulting from one aspect of the rendering process, specifically *texture mapping*. The *texture* refers to the pattern of color on a surface. The rendering system derives the color of each pixel by determining which surfaces are visible through that pixel and calculating the amount of light reflected from those surfaces through the pixel. This is formalized in the *rendering equation* [54]. We present a simplified form here:

$$\widehat{f}(w) = color = \int_P I(p) f(p) k(p,w)\, dp, \tag{1}$$

where I, f, and k respectively compute the incoming light intensity, texture, and contribution to the pixel while w indicates the pixel shape and location and $p \in P$ is a vector concatenating points on the surface, light direction, and other parameters.[1] Note that exact solutions to this integral do not exist for all functions I, f, and k [13]; in these cases, approximate solutions are the only option.

There are two common approaches to evaluating these integrals, *supersampling* and *prefiltering*, which each have different tradeoffs between image quality and run time. We discuss supersampling here, followed by prefiltering in subsequent paragraphs. Supersampling evaluates the function at several locations, approximating the integral as a piecewise combination of the samples. Although this process is simple to implement, sampling rapidly-changing functions (such as a black grid on a white background, which rapidly changes from white to black to white again) may result in large errors unless a large number of samples or sophisticated sampling techniques are employed. Furthermore, evaluating a large number of samples increases the computational load on the system, which in turn reduces the resources available for other computation, decreases the frame rate, or both. Thus, computing enough samples to produce a high quality image may be too slow for many applications.

In prefiltering, the integral is partially evaluated offline and this intermediate result is stored in a form that allows rapid runtime approximation of the final value. The specific techniques used to prefilter depend on the way in which the function is defined. In practice, the surface color is often defined via a lookup table, such as a photograph or artistic rendering, or as a program known as a *procedural shader* [26]. We discuss lookup tables next, followed by procedural shaders.

Lookup tables store precomputed texture outputs for discrete values of the function parameters; an interpolation algorithm is used to look up outputs for intermediate parameter values. The standard approaches to prefilter lookup tables use mipmaps [102] (which store precomputed definite integrals at discrete scales, allowing runtime interpolation of the desired scale) or summed area tables [29] (which store precomputed indefinite integrals, allowing quick computation of the definite integral at runtime). These approaches offer exact solutions in many cases with a constant computational cost. However, they also scale exponentially in the number of dimensions; it is typically not practical to precompute integrals of functions that depend on more than two or three variables. For example, in three-dimensional space, Equation (1) may incoporate two coordinates for the surface location as well as two each for incoming and outgoing light directions, such that f is a at least a six-dimensional function. Thus, while these approaches are effective in some cases, for more complex texture functions, they increase storage requirements exponentially and can replace fast computations with relatively slow memory accesses.

In contrast to lookup tables, a procedural shader requires no additional memory tables and allows for a larger number of function parameters, limited only by the programming language. For procedural shaders, prefiltering consists of transforming the source code such that the shader computes the integral instead of the integrand. For a few specialized types of procedural shaders, exact analytic prefiltering is a natural side-effect of their construction [59]. In most cases, however, the shader developer must manually calculate the appropriate integral. Even when a solution exists, this is usually complicated and time consuming for realistic shaders, so it is rarely done in practice [13]. The development of techniques to prefilter arbitrary procedural shaders remains an open problem.

We hypothesize that we can construct fast, high quality prefiltered procedural shaders using source code transformations and search-based software engineering. We exploit the insight that in many cases, the (e.g., linear) combination of prefiltered functions produces another prefiltered function. Thus, we propose a compiler-based approach to transform an existing shader program into a prefiltered version using its recursively-prefiltered subcomponents. We handle the base case of this recursion by prefiltering the primitives of the shader language offline. Even in cases where composition of prefiltered subcomponents does not result in the correct solution to the integral, we observe that a combination of prefiltered and non-prefiltered subcomponents may produce a reasonable approximation. We therefore propose to use search techniques to identify the subset of prefiltered components that generates a high quality approximate solution. We

[1] Some researchers have included terms and parameters for light emission [54], surface scattering [67], wavelength and time [98]. Taken together, p may have upwards of 10 dimensions. This complexity further motivates the need for effective approximation.

hypothesize that this approach will produce shaders that show less aliasing than the original shader without the run time cost of supersampling.

We will evaluate our technique by comparing the runtimes and the images produced using the original shader and our generated shader to a target image. We will succeed on runtime if the time to produce an image with the generated shader is less than the time to produce an image by supersampling the original shader. We will succeed on image quality if the difference between the image produced with the generated shader and the target is less than that for the image produced by the original shader.

2.1 Research Thrust 1: Background and Related Work

In this subsection, we present relevant background on prefiltering and code transformations for shader programs.

Norton and Rockwood [68] propose an approximation for prefiltering shaders that can be decomposed as a sum of sines. Our approach is similar in spirit but more general, addressing a larger class of both shader and kernel functions. Heitz et al. [48] describe a technique using precomputed lookup tables to compute pixel colors of static or procedural color map textures for which the mean and standard deviation can be efficiently computed. Their technique assumes that a shader to calculate the mean already exists; our approach aims to generate such shaders.

Several researchers have investigated techniques for accelerating procedural shaders in contexts with reduced requirements for level-of-detail. Olano et al. [69] present a compiler technique for applying local transformations to a procedural shader, replacing memory accesses with constant colors. Pellacini's [74] compiler technique locally simplifies computational logic in the shader as well as removing texture accesses. His approach uses a hill-climbing search to generate a sequence of progressively simpler shaders with increasing error relative to the original. Sitthi-amorn et al. [91] use a genetic algorithm that applies local syntactic simplifications to optimize the Pareto frontier between rendering time and image error. Wang et al. [100] also employ a genetic algorithm to search through code transformations to optimize a Pareto frontier over time, error, and memory consumption. Rather than addressing rendering time while tolerating a certain amount of infidelity in the resulting image, our approach explicitly addresses prefiltering, which previous techniques achieve accidentally, allowing them to tolerate error in shaders at lower levels of detail. We apply local transformations to the shader program, but produce a single shader with the desired properties.

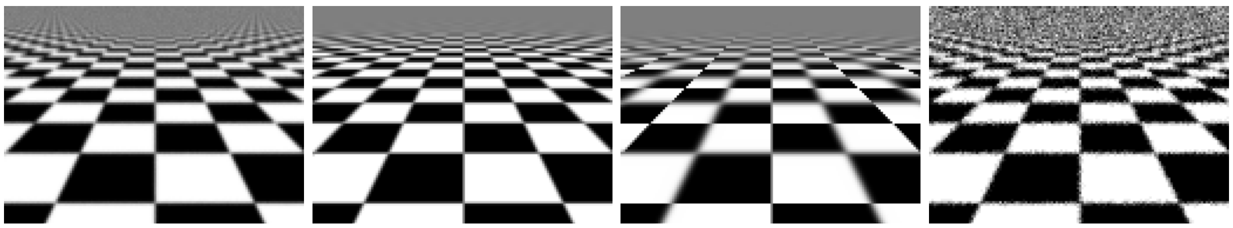
Heidrich et al. [47] and Velázquez-Armendáriz et al. [97] describe compiler-based transformations that automatically augment shaders to also compute approximate bounds on their output value as a function of the bounds of their inputs. These bounds allow renderers to apply techniques such as importance sampling to more efficiently converge. The transformations applied by our technique are similar in spirit. However, they are designed to prefilter the output function instead of quantifying its bounds.

2.2 Research Thrust 1: Proposed Research

In this research thrust, we consider the problem of analytically computing the prefiltered version of a procedural shader function, such that Equation (1) may be computed directly at run time. Perhaps unsurprisingly, there is currently no known way of analytically computing these prefiltered functions in general. We approach the problem by analyzing the conditions under which exact solutions are possible. We then proceed to investigate several approximation strategies for when they are not.

We assume the incident lighting is locally constant, i.e., it does not change rapidly across a single pixel. For k, we make the common choice of the Gaussian function [89] with mean and covariance given by w. With these definitions in place, we wish to derive $\widehat{f}(w)$ as given in Equation (1).

For many built-in functions that are commonly found in procedural shader languages, such as $\lfloor x \rfloor$ or $\mathit{saturate}(x)$ [82], this integral can be computed directly. Additionally, because integration is a linear operator, the prefiltered version of any linear combination of expressions is straightforward to compute. Specifically, for any functions $f_1 \ldots f_n$ and constants $c_0, c_1 \ldots c_n$, such that $g(p) = c_0 + c_1 f_1(p) + \cdots + c_n f_n(p)$, it is the case that $\widehat{g}(w) = c_0 + c_1 \widehat{f_1}(w) + \cdots + c_n \widehat{f_n}(w)$. Similarly, in the special case where the shader function and



(a) Target (supersampled) (b) Separable (c) Non-separable (d) 16 samples

Figure 2: Renderings of a checkerboard procedural shader. The images were rendered using (a) supersampling for a slow but high quality image, (b) a prefiltered, multiplicatively separable shader, (c) a prefiltered shader that is not multiplicatively separable, and (d) 16 supersamples to represent the capabilities of real-time supersampling. The pixels in all images have been enlarged to show detail.

the pixel function are multiplicatively separable functions of multiple parameters (e.g., texture coordinates, direction to the viewer, etc.), we can compute the prefiltered version in terms of its components. That is, if $f(p) = g(x)h(y)$, where x and y represent disjoint elements of p, $\widehat{f}(w) = \widehat{g}(w_x)\widehat{h}(w_y)$. Figure 2 demonstrates this visually. The leftmost image was generated using supersampling, producing a high quality image with a long render time. The second image (2b) was generated using a prefiltered, multiplicatively separable shader, producing a high quality image with significantly shorter render time. (The slight error in the upper left and right corners is due to the pixel function not being multiplicatively separable in those regions.)

Not all shaders are conveniently linear combinations of terms of mathematically separable functions, however. To handle such shaders, we propose to use search-based software engineering techniques [45], exploring the space of similar shaders to find an implementation that approximates the prefiltered shader. To motivate why this might be possible, consider Figure 2c, which was produced by naively applying our proposed technique to a non-separable shader formulation. We constructed the prefiltered shader as if it were separable, by prefiltering the components and combining them, to produce an image with reasonable quality. In particular, rendering that image required an order of magnitude less time than rendering 2d with only 16% more visual error.

We define the search space with respect to the abstract syntax tree [10] (AST) of the shader programs. To produce a new shader from an existing shader, we select a non-prefiltered node of the AST and replace it with the corresponding prefiltered subtree. Each replacement subtree takes the same inputs and returns the same output type as the node it replaces, ensuring that the resulting program remains valid.

As this transformation may not always result in the correct prefiltered shader, we define the following measurement of the quality or fitness of the new shader. We consider the similarity between the images the shader produces for a set of representative scenes and the corresponding target images. Note that our technique is independent of the particular similarity metric used; for our experiments we will measure the average per-pixel L^2 distance in RGB [57]. Our search space consists of all programs reachable via a finite sequence of node replacements. Since the search space is exponential in the size of the shader, for real-world programs it is infeasible to evaluate every such shader. Instead, we propose to use genetic search to to guide exploration of this search space.

2.3 Research Thrust 1: Preliminary Results

We have constructed a prototype implementation of our technique and evaluated it on the benchmarks listed in Table 1. These benchmarks were drawn from shaders used in previous work on antialiasing [103]. As shown in the table, the shaders generated by our approach always run in less time than is required for modest 16x supersampling. Additionally, our shaders produce images with less error than the original shader in all cases except the `perlin` shader, and often less error than the slower supersampling approach. We therefore conclude that this approach is a promising way to improve the acceptability of procedural shaders along both time and quality dimensions.

		Normalized L^2 error		Normalized Runtime		
Shader	Nodes	Ours	16 SS	Ours	16 SS	Description
`step`	1	**0.084**	0.349	**0.67**	18.39	Black and white plane
`ridges`	1	**0.075**	0.377	**1.37**	18.28	$\mathit{fract}(x)$
`pulse`	2	**0.070**	0.296	**2.31**	18.17	Black and white stripes
`noise1`	3	**0.362**	0.378	**2.87**	17.62	Super-imposed noise
`checker`	4	**0.173**	0.299	**4.26**	18.30	Checkerboard
`circles1`	5	**0.244**	0.280	**0.92**	18.32	Tiled circles
`wood`	18	0.943	**0.327**	**3.37**	16.83	Wood grain
`brick`	26	**0.132**	0.322	**2.89**	17.01	Brick wall
`noise2`	28	0.297	**0.291**	**3.01**	17.61	Color mapped noise
`circles2`	74	0.755	**0.318**	**0.93**	18.98	Overlapping circles
`perlin`	244	1.000	**0.464**	**1.00**	17.62	Improved Perlin noise

Table 1: Shaders used for evaluation. "Nodes" lists the number of replaceable nodes in the AST. "L^2 error" and "Runtime" indicate the performance of our prefiltered shaders versus 16 supersamples. Performance numbers are normalized to the error and runtime of the original shader. Our shaders are often an order of magnitude faster than 16x multi-sampling while maintaining comparable or better image quality. The entries corresponding to lower error or lower runtime are in bold.

3 Energy Usage Acceptability in Data Center Applications

In our second proposed research thrust, we use search-based software engineering to apply program transformations to the problem of optimizing energy usage in data center applications. We develop techniques to scale a search for lower energy programs up to the size and complexity of modern data center software.

In recent years, the number and size of data centers have grown to the point where they are estimated to be responsible for over 1% of global energy consumption [58]. The mechanical and electrical systems (such as lighting, cooling, air circulation, and uninterruptible power supplies) required to support the warehouse scale computation may double, triple, or even quadruple [51] the power required by the computation itself. Since the load on many of these support systems grows with the computational load—increased computation leads to increased waste heat, which must be removed—computational efficiency remains a significant determinant of the economic and environmental costs of data centers.

We approach energy reduction as a program optimization task, formalizing it as the problem of selecting a program from the set of all programs such that it meets a functional specification and simultaneously optimizes an objective function. Since determining whether a program meets a functional specification is undecidable in general [81], we restrict ourselves to an approximation. This formalization suggests an interpretation of program optimization as a search problem [45]. We focus on energy usage because of its growing economic and environmental impact and because the hardware and software complexity of modern data centers make it difficult to reason about the impact of optimization decisions [93, 95].

Recently, we introduced Genetic Optimization Algorithm (GOA) [86], a technique for non-functional optimization that operates on assembly files and achieved an average 20% energy reduction on the PARSEC [16] benchmark suite in our experiments. Our previous work investigated the expressive power of GOA; in this proposal, we turn to the question of scalability. Although these benchmarks were designed in part so that their behavior would "mimic large-scale multi-threaded commercial programs" [17], the programs themselves are significantly smaller than real-world programs. Kanev et al. note that at Google, "binaries often reach 100s of MBs in size" [55]; this is two orders of magnitude larger than the largest PARSEC benchmarks.

This has a profound impact on the expected time required for GOA to optimize a program, since the size of GOA's search space is superlinear in the size of the program being optimized. GOA may apply any of $\mathcal{O}(N^2)$ possible individual transformations to a program with N lines; when we consider programs that are two orders of magnitude larger, the search space grows by four orders. Conversely, if we were to consider only a subset of the lines in a program, we could reduce the search space substantially. For example, we observe that GOA uses a characteristic workload to evaluate whether a transformed program uses more or less energy. Although changes to code that is not exercized by that workload will likely not affect the energy usage seen by GOA, it nevertheless considers transformations to every line in the program equally. This suggests an opportunity to use different workloads for separate searches to find distinct optimizations that may then be combined.

Figure 3 shows the results of a preliminary investigation into the effect of reducing the search space for the `swaptions` benchmark. We partitioned the functions executed by the benchmark into two sets, and for each set we ran a GOA-like search that was restricted to make modifications only to the functions in that set. As shown in the figure, both searches separately found sets of transformations that reduce the energy used by the benchmark. Merging both sets of transformations reduced energy usage even further. In contrast, although GOA can in principle find the same sets of transformations, when we ran it on the same benchmark for the same duration, it failed to find any transformations that reduced the energy used. This suggests that reducing the search space and targeting relevant regions of the program may allow larger energy reductions to be found more quickly.

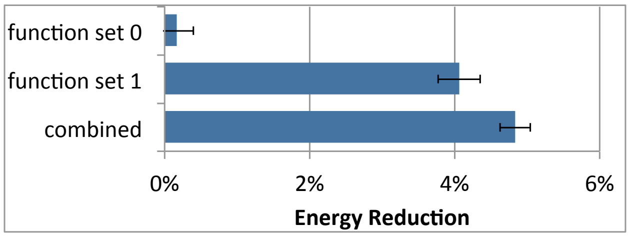


Figure 3: Reduction in energy used at the wall socket after optimizing `swaptions` benchmark.

We hypothesize that we can develop a technique to improve non-functional properties, such as power, of data center programs. To do this, we propose to investigate two high-level approaches to reduce the space searched by a GOA-like optimization algorithm. First, we propose to merge the transformations found by optimizing a program with respect to different workloads. Second, we propose to introduce a combination of program analysis and and profiling techniques to provide important guidance to prioritize a GOA-like search for optimizations. The search space we consider includes programs with different execution *profiles* than the original program we are optimizing. At a high level, a profile is a mapping from lines of a program to performance metrics (e.g., an estimate of execution counts). To account for the non-uniformity of profiles in the search space, we propose to investigate techniques for efficiently computing a new approximate profile for a program based on a known profile for a different program and the search path between the two programs.

We will compare the performance of our search using prioritization and merging techniques to the performance of GOA, both in terms of the final optimizations achieved and in terms of how quickly those optimizations are found. We will succeed if our algorithm is able to find optimizations on large programs more quickly (or at all, given finite search times) than GOA.

3.1 Research Thrust 2: Background and Related Work

In this subsection, we present the most relevant background including optimization techniques, relaxed program semantics, and evolutionary computation.

Existing optimization techniques vary in the ways they implicitly or explicitly restrict the search space and the algorithms they use to search that space. GOA uses a genetic search to find optimized variants based on an original program. The fitness function runs each candidate program on an indicative workload; if the result does not match the original, the program is assigned pessimal fitness, otherwise, its fitness is its non-functional performance. The search space consists of the original program as well as programs that may be generated by a finite number of insertions of lines copied from elsewhere in the program as well as deletions of assembly lines. Using this search space leverages *mutational robustness*, the observation that small random program transformations can produce semantically different, but functionally equivalent, programs [87]. While GOA selects transformations uniformly at random, we propose to adjust the probabilities of different transformations using performance information to guide the search away from transformations that are unlikely to have a significant effect on the non-functional performance.

Traditional optimizing compilers [10] restrict the search space by using only provably semantics-preserving transformations. Profile-guided optimization (PGO) techniques construct a profile of program behavior on a set of representative workloads as a pre-compilation step. The optimizing compiler uses this profile to guide the search for an optimized program, for example by preferring transformations in "hot paths" [24]. In contrast, we propose to evaluate the program's non-functional performance repeatedly to assess the impact of candidate transformations.

Although auto-tuning techniques profile several different candidate transformations before deciding the best one to apply, the candidate transformations are selected from a set of user-provided implementations. This requires that the application be written in a modular style to allow the substitutions. For example, the FFTW [39] source code is written in "codelets" (small optimized sub-steps) which are combined into hardware dependent "plans" at runtime. Our proposed technique, on the other hand, makes no assumptions about the way in which the code was written.

Superoptimization techniques employ a broad set of search algorithms, such as brute-force [63] or stochastic search [84]. These techniques search over the space of short sequences of instructions (typically about 10) to select an optimal sequence that matches a target sequence's functionality. Superoptimization techniques are limited in their scalability due to the exponential explosion in the number of programs containing N instructions and by the difficulty of establishing program equivalence. Our proposed approach is similar in spirit, but achieves greater scalability by focusing the search in the neighborhood of a known working implementation and allowing for relaxed program semantics instead of requiring proofs of program equivalence.

We propose to construct a practical algorithm for the optimization of non-functional properties in data center scale applications that combines the expressivity of evolutionary computation with the targetability of profile-guided optimization, using novel algorithms to efficiently maintain an accurate profile throughout the search.

3.2 Research Thrust 2: Proposed Research

We have seen that reducing the energy requirements of large data center applications can have significant economic and environmental benefits. However, as the size of a program increases, so too does the size of the potential search space, meaning that the fraction of that space that may be explored in a fixed amount of time decreases. Thus, to scale to large programs, we propose to develop techniques to reduce the search space and guide the search toward the portion of the remaining space most likely to contain optimal programs.

We propose to enable the reduction of energy requirements in data center applications by applying an optimization algorithm employing evolutionary search guided by partitioned execution profiles. Intuitively, transformations to code that is never executed will have little effect on the cost of running the program. Conversely, if the search targets only the instructions exercised by a small workload, it may miss opportunities for optimizations that would affect other workloads. We hypothesize that optimizations found for different workloads may be profitably combined to produce an executable with reduced cost under a variety of workloads. Within a single workload, and as with profile-guided optimization, we hypothesize that some regions of a program have a larger effect on the overall cost of running the program than others, such that optimizing those regions will have a proportionally larger effect than optimizing others.

Although using different workloads may allow us to find optimizations for code that is not always executed, separate searches may discover distinct optimizations for the same code (e.g., initialization routines). Merging these optimizations may result in conflicts. That is, one set of transformations may negate the performance benefit of another. Worse, the two sets of transformations may retain functionality individually, but degrade the program's functionality when applied together. We will investigate techniques for combining separate optimizations to retain the benefits without the negative side effects. For example, Delta Debugging [104] is a technique to find a minimal set of fault-inducing edits. We can invert this to find a maximal subset of transformations that retain tested functionality. Alternatively, may conduct a second search over the combined transformations to minimize energy use while retaining functionality.

```
Input:  Original Program, P : $Program$
Input:  Non-Functional Eval, Fitness : $Program \to \mathbb{R}$
Input:  Profile Function, Profile : $Program \to (Instruction \to \mathbb{R})$
Parameters:  $PopSize$, $CrossRate$, $TournamentSize$, $MaxEvals$
Output:  Program that optimizes Fitness
 1: $Pop \leftarrow PopSize$ copies of $\langle \mathsf{P}, \mathsf{Fitness}(\mathsf{P}), \mathsf{Profile}(\mathsf{P}) \rangle$
 2: $EvalCounter \leftarrow 0$
 3: repeat
 4:     if Random() $< CrossRate$ then
 5:         $p_1, profile_1 \leftarrow$ Tournament($Pop, TournamentSize, +$)
 6:         $p_2, profile_2 \leftarrow$ Tournament($Pop, TournamentSize, +$)
 7:         $p \leftarrow$ Crossover($p_1, p_2$)
 8:         $profile \leftarrow$ UpdateProfile($p, \langle p_1, p_2 \rangle, \langle profile_1, profile_2 \rangle$)
 9:     else
10:         $p, profile \leftarrow$ Tournament($Pop, TournamentSize, +$)
11:     end if
12:     $p' \leftarrow$ Mutate($p, profile$)
13:     $profile' \leftarrow$ UpdateProfile($p', \langle p \rangle, \langle profile \rangle$)
14:     AddTo($Pop, \langle p', \mathsf{Fitness}(p'), profile' \rangle$)
15:     EvictFrom($Pop$, Tournament($Pop, TournamentSize, -$))
16:     $EvalCounter \leftarrow EvalCounter + 1$
17: until $EvalCounter \geq MaxEvals$
18: return Minimize(Best($Pop$))
```

Figure 4: Pseudocode for the main loop of proposed algorithm.

Figure 4 shows a high-level search algorithm for program optimization, based on GOA and updated to incorporate profile information. The execution profile is used to guide the search during mutation on line 12, which can use the information while selecting the transformation to apply and thus determining which program to inspect next. Since the search may encounter programs that have different runtime behavior, it is not feasible to maintain a single global profile. Instead, the algorithm must associate with each program an estimate of that program's execution profile, for use in the mutation operator. However, it is impractical to simply profile the execution of every program the search investigates. Such an approach would either drastically reduce the number of programs that could be evaluated in the same amount of time, or drastically increase the time required to process the same number of programs. For example, naively following the performance sampling approach suggested by Schulte et al. [85] would allow an order of magnitude fewer variants to be evaluated, given a constant search budget. The key research challenge, then, is to develop techniques for effectively approximating the profile of new variants without incurring the full cost of profile acquisition for every variant.

We envision a profile approximation function (UpdateProfile) that takes as arguments a program p, its parent programs and their profiles, and returns a new approximate profile for the child program. Recall that a profile is a mapping from program instructions to performance metrics. We will investigate several algorithms for assigning performance values to the instructions of the new program.

1. One trivial approach is to map each instruction in the child to the value for the corresponding instruction in the parent. This would result in a new profile that is almost identical to the parent's. Deleting an instruction simply removes that instruction from the profile while the value for a newly inserted line is the same as for its source in the original program. Using this approach, it is straightforward to construct a profile for the result of crossover by simply tracking which parent was the source of each line in the child.

2. A second technique computes a very approximate profile that maps each line to its loop nesting depth. Static analyses for identifying loops in programs are well studied [10]. Intuitively, inner loops are likely to execute more frequently than outer loops, so preferring mutations in more deeply nested loops may be sufficient to guide the search toward more energy-expensive regions.

3. Our third approach will adapt static path frequency analysis techniques [21, 94], which have primarily been aimed at higher-level languages like Java, for use on assembly programs. These techniques train a machine learning model using static features of the source code to predict the runtime execution frequency of different paths. That is, they are static techniques explicitly designed to approximate runtime profiles. Many of the features of such models have assembly language analogs, such as local variable use on the stack, common control flow constructs including if statements and method calls, and boolean operators.

4. Fourth, we will construct an algorithm that returns a short sequence of instructions based on the difference between the child program and its parents. We will use forward program slicing techniques [52] to determine the set of statements that may be affected by a new instruction. Instead of profiling the whole program, we will profile the slice and use the result to construct a new profile from a copy of the parent's profile. However, since profiling sufficiently large slices may be no faster than profiling the entire program, we will prefer the more accurate approach of recomputing the entire profile in such cases.

We note that these proposed investigations depend on, and leverage, experience we have developed with genetic programming [37, 86] and program analysis [37].

3.3 Research Thrust 2: Proposed Experiments

In this research thrust, we consider three hypotheses. First, we hypothesize that profile information can effectively guide a genetic search to efficiently identify program optimizations in large, real-world programs. Second, we hypothesize that it is possible to approximate profile information, so as to reduce the time taken to evaluate each variant while continuing to effectively guide the search for optimizations, resulting in a more efficient search overall. Third, we hypothesize that optimizations found using different workloads (and, by implication, different profiles) can be effectively merged.

To evaluate the first two hypotheses, we will develop our search algorithm to make it modular with respect to the algorithm used to compute the profile. This will allow us to easily run experiments that vary only in the choice of profiling algorithm. For each algorithm described above, we will run the optimization algorithm on a set of real-world open-source projects. We will compare against two baseline profiling algorithms: one that collects a full profile for every variant and another that returns a profile that maps every instruction to the same value. We will keep track of the time required to find the best optimization, the number of variants considered before the best optimization, and the magnitude of the improvement in performance for each algorithm on each benchmark.

Note that the two baselines represent different bounds on the performance of our approximation algorithm. The baseline algorithm that collects a profile for every variant provides the most accurate possible profile information at the cost of significant run time. The baseline algorithm that maps every instruction to the same value is trivial to compute but is equivalent to the GOA approach that does not use a profile, and so provides no useful information to the search. If the search using the first baseline is able to find better optimizations after evaluating fewer variants, we will validate our first hypothesis, that profile information can effectively guide the genetic search. To validate our second hypothesis, that approximate profile information can lead to a more efficient search, we must find comparable or better optimizations in less run time using our profile approximation than either baseline.

To evaluate our third hypothesis, we will optimize each benchmark separately on two different workloads, using the best algorithm identified in the previous experiments. Note that if we fail to validate our first two hypotheses, the best algorithm may be one of the previous baselines. We will then combine the results using our candidate merging algorithms to produce a single optimized version of each benchmark. In this experiment, we have three baselines to compare against, each of which will have a wall time budget equal to the total run time of the two separate searches. The baselines use the same search algorithm but differ in the workload used in the fitness function, with one baseline each for the separate workloads and a third that uses both workloads. If the merged optimizations out-perform the first two baselines on held-out workloads, we will have shown that merging can produce more general optimizations. If the merged optimizations out-perform the third baseline, we will have demonstrated the value of reducing the search space.

We will measure the performance of the optimization algorithm on a set of open source programs. To be included in our evaluation, programs must be compilable to assembly code, such that the assembly can then be used to generate the program's primary executable. In addition, the project must include a test suite or set of example inputs, which we will partition into training and testing input sets. To be eligible for inclusion in the training set, the test case or example input must be small enough to evaluate quickly in the main loop of the genetic algorithm. We will only consider projects for which we can construct disjoint, non-empty training and testing sets. We will collect at least 10 projects meeting these criteria for our evaluation.

4 Coding Style Acceptability for Programmers

In our third proposed research thrust, we investigate the impact of transforming the coding style in which a program is written on its acceptability to programmers. Specifically, we propose to develop a model of coding style that permits measuring the similarity of styles, which we will use to assess how well developers are able to understand code in different styles.

Understanding the program "plays a role in nearly all software tasks" [19]. It is necessary to understand the program to write it, to review it, to test it, to debug it, and to extend it. With the exception of the very first time a program is written or its requirements are documented, the programmer must understand an existing artifact. This understanding consists of constructing a mental model and validating it against the source code or other documentation [19]. This has led to the common observation that programmers spend far more—as much as 10 times more [62]—time reading code than writing it [44, 80]. Reading code is a skill that must be acquired [33] because understanding programs is hard [15, 92]

Given the importance and difficulty of reading programs, it is unsurprising that significant effort has been directed toward making it easier. Practitioners from open-source communities [3, 4] and industry [5] have produced coding standards documents designed, at least in part, to "reduce cognitive friction" [7] and "improve the readability of code" [6]. According to Google's own style guide, "every major open-source project has its own style guide" [5]. Researchers have investigated the impact of coding style on program comprehension [14, 72] and have produced metrics for software readability [78]. These metrics have been shown to correlate with defect density [20], suggesting that they do capture aspects of how well programmers understand the code. Recent work has demonstrated that these metrics can do more than measure readability; they can guide automatic code generators in producing more readable code [31].

Brooks has proposed that programmers reading to understand source code look for *beacons*, syntactic and semantic aspects of code that indicate particular constructs or functionality [19]. For example, many coding style guides recommend placing closing braces on their own lines [3, 7], which may serve as a syntactic beacon indicating changes in control flow [72]. The presence of either the procedure call `swap(x,y)` or the sequence `{tmp := x; x := y; y := tmp}` inside a loop may serve as a semantic beacon indicating sorting functionality [101]. As this second example shows, there may be many different beacons for the same construct. Indeed, many constructs in actual programs may be marked by only a subset of the possible beacons: few sorting implementations will include both a call to `swap` and the three assignments above. Furthermore, there are situations other than sorting algorithms in which `swap` might appear inside a loop. The same beacon may indicate different features with different probabilities. A programmer may increase their confidence in their recognition of a particular construct by combining evidence from multiple beacons.

In this paradigm, reading code written in a different style entails a mismatch between the reader's expected beacon interpretations and the writer's intention. In a survey of integrators, team members responsible for accepting and integrating pull requests on GitHub, Gousios et al. found that conforming to the project's coding style was one of the primary factors in assessing the request's quality and whether it will be accepted [42]. Indeed, the goals of many coding style guides are directed more toward consistency than toward ideal formatting [4–7]. Nonetheless, there is evidence that an individual's coding style is distinctive enough to allow effective determination of authorship based on style alone [22]. Since the recognition of beacons may vary from programmer to programmer, impacting their ability to understand source code, and may change with experience [27], the possibility of style-based "cognitive friction" remains. We are interested in the extent to which an individual's coding style affects their interaction with code written in a different style.

Coding style covers a range of programmer decisions from textual formatting to high level modularity decisions [71]. In this work, we focus on the typographical aspects of coding style [70] that can be automatically quantified and admit rule-based pretty-printers and reformatters. For example, we consider placement of comments, order of declarations, use of whitespace and indentation, and similar formatting guidelines. Although we may include simple aspects of identifier naming conventions, such as capitalization or the use of common abbreviations, we do not include guidelines for factoring functions, algorithmic choices, or interface design. Several tools exist to check that a program adheres to these typographical rules or to reformat code so that it does [88]. We propose to represent arbitrary coding styles in a way that supports rule checking and reformatting tools.

We hypothesize that programmers internalize a coding style that they are most familiar with and that they will perform maintenance tasks most efficiently on code that matches their internalized style. That is, given a particular piece of source code, a programmer will understand it more quickly and more completely if it is presented in a familiar rather than unfamiliar style. We propose to develop a measure of the similarity between coding styles that correlates with human perceptions of stylistic similarity. We hypothesize that such a metric will also correlate with maintenance performance. That is, given two identical programs with different styles, our metric should rate them as similar when a programmer performs well on both of them and as dissimilar if a programmer performs well on only one. Note that under our hypothesis, if a programmer has difficulty understanding both programs, they may or may not be similar to each other, but they should both be dissimilar to a program in the programmer's preferred style.

We will evaluate our similarity metric along two dimensions. First, we will be successful if our metric assigns small similarity values to source code that humans perceive as having different styles and large values to source code perceived as having similar styles. Second, we succeed if our metric accurately predicts the performance of the participants of a human study on common code maintenance tasks.

4.1 Research Thrust 3: Background and Related Work

In this section, we present background on coding styles and program comprehension.

Many researchers have investigated the effects of formatting on program comprehension. For example, indentation is commonly understood to significantly improve program understanding [64, 70], although Love [61] found no significant improvement due to indentation. Oman and Cook [72] particularly argue that typographical style—the textual layout of the program—can significantly affect comprehension even when structural style—the choices of programming constructs—remains the same. Other researchers have focused on identifier names, finding evidence for the importance of meaningful identifiers [18], particularly the use of full words in identifiers [60].

Buse and Weimer [20] applied machine learning techniques to develop a metric of software readability. Their metric used measures of both typographic style, such as line length and the fraction of blank lines, and structural style, including the use of control structure keywords. The researchers showed that low modeled readability scores were correlated with code that was likely to change soon. In contrast to the large number of features (25) used in the Buse and Weimer metric, Posnett et al. [78] demonstrated a similar metric using machine learning with only three features. Their reduced feature set includes features pertaining to both typographic and structural style. Recently, we used a similar readability metric to guide an automatic test generator to produce more readable tests [31].

Statistical language models based on n-gram models have used to model a variety of aspects of source code, supporting improved error reporting [23], code completion [50, 79, 96], and even automated translation between programming languages [66]. N-gram models learn a conditional probability distribution for the nth token in a sequence, given the preceding $n-1$ tokens. Hellendoorn et al. used an n-gram model to measure the similarity between projects on GitHub and pull requests for those projects [49]. They found that the more similar the code in a pull request is to the project, the more likely it is to be accepted. They also found that developers' pull requests became more similar to a project as they gained experience with it. Allamanis et al. [11] used an n-gram model to model coding style.

In this research thrust, we focus on the low-level typographic elements of coding style [70], rather than those aspects dealing with software design and structure. We are primarily interested in the effect that the formatting of the code has on programmers' abilities to understand and maintain the code. We hypothesize that programmers have a preferred style in which they are most effective at maintenance tasks.

4.2 Research Thrust 3: Proposed Research

In this research thrust, we will investigate the interaction between the style in which code is written and the speed and accuracy with which programmers maintain that code. This research goal suggests two primary sub-problems. First, can we model the style in which a particular sample of code is written in a way that admits similarity comparisons? Second, do programmers perform similarly on maintenance tasks on code samples that are written in similar styles, all other things being equal?

Answering the first question requires us to represent the style of arbitrary source code. We propose to extend the NATURALIZE framework presented by Allamanis et al. [11]. This framework learns an n-gram language model from a training set of source code, which they use to generate configurations for a rule-based style-checking tool. The grams in their model represent typological style features, including, but not limited to, indentation, placement of newlines, use of whitespace within lines, as well as the tokens of the Java language. The checking tool can check whether a new file is written in the style of the training code; alternatively, the n-gram model can be used to estimate how likely it is that the new file follows the same style. The authors also discuss using the n-gram model to construct a pretty-printer for reformatting new code into the learned style. They have made the source code for these tools available online via GitHub. We propose to leverage these tools as a reference for constructing a measure of coding style similarity.

We will investigate a few different approaches to build this similarity measure:

1. One approach is to leverage a configurable rule-based style-checking tool. We can learn language models for a number of open-source projects and generate configurations for the tool as described by Allamanis et al. [11] In this approach, we could define the similarity measure based on counts of rule violations. For example, to measure the similarity of style between two pieces of code, we might first identify the configurations that result in the fewest number of violations for each piece. Then we could define the similarity as a function of the violations resulting from evaluating each piece of code against the other configuration.

2. A second approach is to use the n-gram probability model more directly. For example, we might define the similarity between the styles of two pieces of code to be the similarity, (e.g., using the earth mover's distance [83]) of n-gram probability distributions learned from each. This approach has the advantage that it does not rely on collecting a sufficiently representative sample of open-source projects.

3. Alternatively, we can apply machine-learning algorithms to develop the similarity measure. For example, we might define the features of our model to be the parameters of the style-checking tool's configuration. More specifically, to incorporate both styles we could define the features to be the difference or ratio between pairs of parameter values. To measure the similarity between the styles of two pieces of code, we could learn n-gram probabilities, generate configurations for each, and apply the learned model to the result.

4. In a combination of the previous two approaches, we will consider using machine learning to determine a subset of n-grams that are diagnostic and compare only the probabilities of the diagnostic n-grams.

Answering the second question requires presenting programmers with code samples of similar complexity but different styles. While we discus this in further detail in Section 4.3, we discuss possible research implications here. One simple way to acquire pairs of programs with the same complexity and different styles is to pretty-print a single program in both styles. This implicitly requires a typological view of coding style, since structural changes are outside the scope of pretty-printers. This restriction to typological style seems reasonable, since several researchers have observed that elements typological style affect program understanding [64, 72]. A relatively minor extension is to extend our definition of style to incorporate naming decisions. This might be supported in a modified pretty-printer by incorporating identifier name generation algorithms [12, 32].

However, Caliskan et al. note that an individual's coding style includes choices of syntactic constructs, such as using `while` instead of `for`, and higher-level program organization [22]. Oman and Cook refer to this as the programmer's structural style [71]. Some researchers have proposed using models that view programs as trees or graphs rather than linear sequences of tokens [22,65,73] to capture some of this higher-level style. Although we prefer to target typographic style to simplify pretty-printing according to a style encoded in a learned language model, we may need to consider incorporating some of these elements of structural style.

4.3 Research Thrust 3: Proposed Experiments

In this research thrust, we consider two hypotheses. Our first hypothesis is that we can use n-gram language models of coding style to develop a measure of coding style similarity that correlates with human perception. Our second hypothesis is that programmers have a preferred style such that they perform maintenance tasks more effectively on code that is in that style. Note that while both hypotheses depend on human perception of coding style, they do not strictly depend on each other. However, an automatic measure of coding style similarity will simplify evaluation of the second hypothesis. We therefore plan to evaluate these hypotheses in the order listed.

To evaluate these hypotheses, we propose to conduct IRB-approved human studies. We note that we have successfully submitted an IRB protocol in the past, leading to publication [30,31]. In the first proposed human study, we will present humans with two samples of source code and ask them to rate the styles as similar or dissimilar. If we decide to use one of the machine-learning approaches described above, this study will provide the training data needed to learn the model. We have previously used a similar presentation style to collect data to train a machine learning model to predict readability [31]. We will use 10-fold cross-validation to evaluate the potential for overfitting the trained model. However, we may optionally repeat this study to collect a distinct testing set.

Our second proposed human study will consist of two sets of questions. In the first set, we will ask participants to write short samples of code to accomplish simple tasks. For example, we may ask them to check that a list is sorted or to implement binary search. We will use these samples to represent the participant's natural coding style. In the second set of questions, we will present participants with several examples of source code in different styles and ask them to complete maintenance tasks [90] on the code. We will measure both the time required to complete the tasks and the correctness of their response.

We propose to use one or both of the following techniques to control the complexity of the code samples. This will allow us to attribute differences in performance to differences in style instead of differences in the code. The first approach is to use pretty-printers to reformat the same code into different styles. In this case, the inherent complexity of the code will be same by construction. The second approach is to select our code examples from programming textbooks [40]. We can ensure that the complexity of the code is consistent by selecting the same algorithm, such as quicksort, from different textbooks. Since different textbook authors often give examples written in styles that are different from other authors, this provides us with samples of similar complexity but different styles.

Note that we will not present the same participant with the same algorithm more than once, regardless of style. This is necessary to minimize the risk that an improvement in performance is due to greater experience with the algorithm instead of a change in coding style. While it is possible that participants may already be familiar with the algorithms presented, especially for examples drawn from textbooks, we plan to randomize which algorithms are presented in each style for each participant. This procedure should help eliminate any systematic bias in performance due to unintended correlations between style and algorithm.

We will evaluate the participants' performance on the maintenance questions as a function of the similarity between the code they were asked to maintain and the code samples they provided at the beginning. If there is a statistically significant correlation between similarity and performance, we will be able to confirm our second hypothesis.

5 Schedule

The proposed research comprises three high-level thrusts: improving shader acceptability through prefiltering, improving acceptability of data center applications with scalable energy optimization, and an investiga-

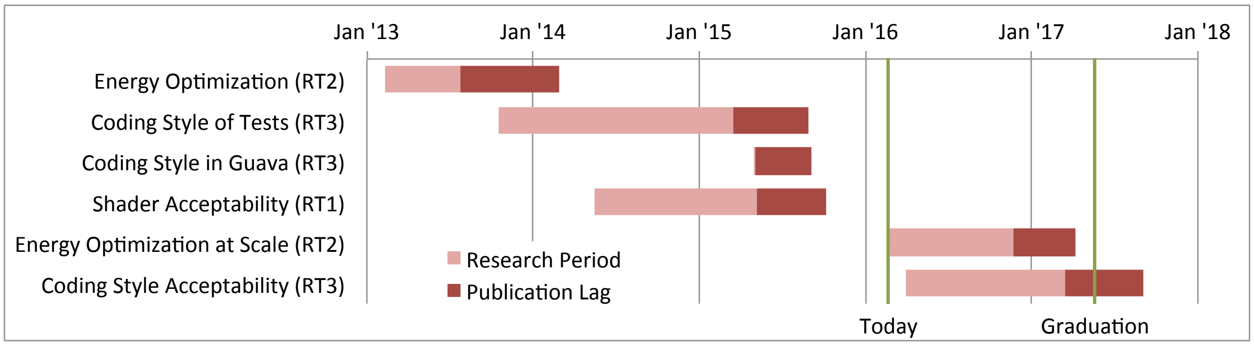


Figure 5: Proposed research schedule. Parentheticals note the research thrust aligned with each publication.

tion of acceptability of coding styles for programmers. In the past, we have targeted conferences related to computer architecture (the *International Conference on Architectural Support for Programming Languages and Operating Systems*), computer graphics (the *Pacific Conference on Computer Graphics and Applications*), and software engineering and maintenance (the *ACM SIGSOFT Symposium on the Foundations of Software Engineering*). For the proposed dissertation, we will again consider these architecture and software engineering venues, in addition to the *International Conference on Automated Software Engineering* and the *ACM SIGPLAN Conference on Systems, Programming, Languages and Applications: Software for Humanity.*

Figure 5 outlines the proposed schedule for the work outlined in this document. Uncertainty in the schedule includes the possibility of additional collaborations on software readability for automatically generated tests that includes conducting additional human studies. We also allow some slack in the schedule to accomodate a possible internship in the summer of 2016.

6 Summary

The long-term goal of our work is to reduce the cost of transforming a functioning program into an acceptable one. We target three aspects of acceptability from diverse application domains and aim to develop automated techniques to assist programmers in improving the acceptability of their existing program artifacts. Specifically, we will investigate

- Visual quality and run time acceptability for graphics shader programs (Section 2.2),
- The acceptability of energy requirements on data center applications (Section 3.2), and
- The implicit acceptability of coding style for programmers (Section 4.2).

We will explore both correct-by-construction transformations and search-based techniques that that optimize an objective designed to correlate with acceptability. We plan to evaluate the effectiveness against of these techniques both against proxies that approximate specific aspects of acceptability and, when practical, using human judgments of acceptability. We hope that our research will help simplify the task of producing acceptable software by demonstrating practical automated techniques for improving non-functional properties that directly impact acceptability.

References

[1] Animation - computer movies at the box office - box office mojo. `http://www.boxofficemojo.com/genres/chart/?id=computeranimation.htm`. Accessed: 2015-12-17.

[2] Civilian labor force (seasonally adjusted). `http://data.bls.gov/pdq/SurveyOutputServlet?request_action=wh&graph_name=LN_cpsbref1`. Accessed: 2016-02-02.

[3] Coding style. `https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Coding_Style`. Accessed: 2016-01-20.

[4] GNU coding standards. `https://www.gnu.org/prep/standards/standards.html`. Accessed: 2016-01-20.

[5] Google style guides. `https://github.com/google/styleguide/`. Accessed: 2016-01-20.

[6] Pep 0008 – style guide for python code. `https://www.python.org/dev/peps/pep-0008/`. Accessed: 2016-01-20.

[7] PSR-2: Coding style guide. `http://www.php-fig.org/psr/psr-2/`. Accessed: 2016-01-20.

[8] Software developers : Occupational handbook. `http://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm`. Accessed: 2016-02-02.

[9] Total passengers on u.s airlines and foreign airlines u.s. flights increased 1.3 `http://www.rita.dot.gov/bts/press_releases/bts016_13`. Accessed: 2016-02-05.

[10] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[11] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Foundations of Software Engineering*, FSE 2014, pages 281–293, 2014.

[12] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, 2015.

[13] A. A. Apodaca and L. Gritz. *Advanced RenderMan: Creating CGI for Motion Picture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.

[14] M. Arab. Enhancing program comprehension: Formatting and documenting. *SIGPLAN Notices*, 27(2):37–46, Feb. 1992.

[15] J. Atwood. When understanding means rewriting. `http://blog.codinghorror.com/when-understanding-means-rewriting/`. Accessed: 2016-02-01.

[16] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[17] C. Bienia, S. Kumar, J. 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.

[18] S. Blinman and A. Cockburn. Program comprehension: Investigating the effects of naming style and documentation. In *Australasian Conference on User Interface*, AUIC '05, pages 73–78, 2005.

[19] R. Brooks. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, 18(6):543–554, 1983.

[20] R. P. Buse and W. Weimer. Learning a metric for code readability. *IEEE Trans. Software Eng.*, Nov. 2009.

[21] R. P. L. Buse and W. Weimer. The road not taken: Estimating path execution frequency statically. In *International Conference on Software Engineering*, 2009.

[22] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt. De-anonymizing programmers via code stylometry. In *USENIX Security Symposium*, pages 255–270, 2015.

[23] J. C. Campbell, A. Hindle, and J. N. Amaral. Syntax errors just aren't natural: Improving error reporting with language models. In *Mining Software Repositories*, MSR 2014, pages 252–261, 2014.

[24] P. P. Chang, S. A. Mahlke, and W.-M. W. Hwu. Using profile information to assist classic code optimizations. *Software: Practice and Experience*, 21(12):1301–1321, 1991.

[25] L. Chung and J. C. Prado Leite. On non-functional requirements in software engineering. In A. T. Borgida, V. K. Chaudhri, P. Giorgini, and E. S. Yu, editors, *Conceptual Modeling: Foundations and Applications*, pages 363–379. Springer-Verlag, Berlin, Heidelberg, 2009.

[26] R. L. Cook. Shade trees. *SIGGRAPH Computer Graphics*, 18(3):223–231, 1984.

[27] M. E. Crosby, J. Scholtz, and S. Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In *14th Workshop of the Psychology of Programming Interest Group*, pages 58–73, 2002.

[28] F. C. Crow. The aliasing problem in computer-generated shaded images. *Communications of the ACM*, 20(11):799–805, Nov 1977.

[29] F. C. Crow. Summed-area tables for texture mapping. *SIGGRAPH Computer Graphics*, 18(3):207–212, Jan 1984.

[30] E. Daka, J. Campos, J. Dorn, G. Fraser, and W. Weimer. Generating readable unit tests for Guava. In *Symposium on Search Based Software Engineering*, pages 235–241, 2015.

[31] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer. Modeling readability to improve unit tests. In *Foundations of Software Engineering*, 2015.

[32] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Using IR methods for labeling source code artifacts: Is it worthwhile? In *International Conference on Program Comprehension*, pages 193–202, June 2012.

[33] L. E. Deimel Jr. The uses of program reading. *SIGCSE Bulletin*, 17(2):5–14, 1985.

[34] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer Magazine*, 11(4):34–41, 1978.

[35] W. J. Doherty and A. J. Thadhani. The economic value of rapid response time. *IBM Report*, 1982.

[36] G. A. Dorn, M. J. Cole, and K. M. Tubman. Visualization in 3-d seismic interpretation. *The Leading Edge*, 14(10):1045–1050, 1995.

[37] J. Dorn, C. Barnes, J. Lawrence, and W. Weimer. Towards automatic band-limited procedural shaders. *Computer Graphics Forum*, 34(7):77–87, 2015.

[38] M. W. Engelhorn. Interactive 3d computer graphics in medical imaging. In *Aachener Symposium für Signaltheorie*, pages 16–27. Springer, 1987.

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

[40] Z. P. Fry and W. Weimer. A human study of fault localization accuracy. In *International Conference on Software Maintenance*, pages 1–10, 2010.

[41] M. Glinz. On non-functional requirements. In *Requirements Engineering*, pages 21–26, 2007.

[42] G. Gousios, A. Zaidman, M. D. Storey, and A. van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *International Conference on Software Engineering*, pages 358–368, 2015.

[43] P. Graydon, I. Habli, R. Hawkins, T. Kelly, and J. Knight. Arguing conformance. *Software, IEEE*, 29(3):50–57, May 2012.

[44] P. Hallam. What do programmers really do anyway? (aka part 2 of the yardstick saga). `http://blogs.msdn.com/b/peterhal/archive/2006/01/04/509302.aspx`. Accessed: 2016-02-01.

[45] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11, 2012.

[46] R. Hawkins, T. Kelly, J. Knight, and P. Graydon. A new approach to creating clear safety arguments. In *Advances in Systems Safety*, pages 3–23. Springer, 2011.

[47] W. Heidrich, P. Slusallek, and H.-P. Seidel. Sampling procedural shaders using affine arithmetic. *ACM Transactions on Graphics*, 17(3):158–176, July 1998.

[48] E. Heitz, D. Nowrouzezahrai, P. Poulin, and F. Neyret. Filtering color mapped textures and surfaces. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 129–136, 2013.

[49] V. J. Hellendoorn, P. T. Devanbu, and A. Bacchelli. Will they like this?: Evaluating code contributions with language models. In *Mining Software Repositories*, MSR '15, pages 157–167, 2015.

[50] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *International Conference on Software Engineering*, pages 837–847, 2012.

[51] U. Hoelzle and L. A. Barroso. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[52] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[53] P. K. Jha, C. Shekhar, and L. S. Kumar. Visual simulation application for hardware in-loop simulation (HILS) of aerospace vehicles. In *Emerging ICT for Bridging the Future-Proceedings of the 49th Annual Convention of the Computer Society of India (CSI) Volume 1*, pages 473–480. Springer, 2015.

[54] J. T. Kajiya. The rendering equation. In *SIGGRAPH Computer Graphics*, volume 20, pages 143–150, 1986.

[55] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *International Symposium on Computer Architecture*, pages 158–169, 2015.

[56] J. C. Knight and E. A. Myers. Phased inspections and their implementation. *SIGSOFT Softw. Eng. Notes*, 16(3):29–35, 1991.

[57] M. Kokare, B. Chatterji, and P. Biswas. Comparison of similarity metrics for texture image retrieval. In *Conference on Convergent Technologies for the Asia-Pacific Region*, volume 2, pages 571–575, Oct 2003.

[58] J. Koomey. *Growth in data center electricity use 2005 to 2010.* Analytics Press, Oakland, CA, 2011.

[59] A. Lagae, S. Lefebvre, G. Drettakis, and P. Dutré. Procedural noise using sparse gabor convolution. *ACM Transactions on Graphics*, 28(3):54:1–54:10, Jul 2009.

[60] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *International Conference on Program Comprehension*, pages 3–12, 2006.

[61] T. Love. An experimental investigation of the effect of program structure on program understanding. *ACM SIGPLAN Notices*, 12(3):105–113, Mar. 1977.

[62] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Robert C. Martin series. Prentice Hall, 2009.

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

[64] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman. Program indentation and comprehensibility. *Communications of the ACM*, 26(11):861–867, Nov. 1983.

[65] A. T. Nguyen and T. N. Nguyen. Graph-based statistical language model for code. In *International Conference on Software Engineering*, pages 858–868, 2015.

[66] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Lexical statistical machine translation for language migration. In *Foundations of Software Engineering*, ESEC/FSE 2013, pages 651–654, New York, NY, USA, 2013. ACM.

[67] F. Nicodemus, J. Richmond, J. Hsia, I. Ginsberg, and T. Limperis. Geometric considerations and nomenclature for reflectance, volume 161 of monograph. *National Bureau of Standards (US)*, 1977.

[68] A. Norton, A. P. Rockwood, and P. T. Skolmoski. Clamping: A method of antialiasing textured surfaces by bandwidth limiting in object space. *SIGGRAPH Computer Graphics*, 16(3), Jul 1982.

[69] M. Olano, B. Kuehne, and M. Simmons. Automatic shader level of detail. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '03, pages 7–14, 2003.

[70] P. W. Oman and C. R. Cook. A paradigm for programming style research. *ACM Sigplan Notices*, 23(12):69–78, 1988.

[71] P. W. Oman and C. R. Cook. A taxonomy for programming style. In *Conference on Cooperation*, CSC '90, pages 244–250, 1990.

[72] P. W. Oman and C. R. Cook. Typographic style is more than cosmetic. *Commun. ACM*, 33(5):506–520, May 1990.

[73] C. Omar. Structured statistical syntax tree prediction. In *Systems, Programming, and Applications: Software for Humanity*, SPLASH '13, pages 113–114, 2013.

[74] F. Pellacini. User-configurable automatic shader simplification. *ACM Transactions on Graphics*, 24(3):445–452, Jul 2005.

[75] F. Pellacini, K. Vidimče, A. Lefohn, A. Mohr, M. Leone, and J. Warren. Lpics: A hybrid hardware-accelerated relighting engine for computer cinematography. *ACM Transactions on Graphics*, 24(3), July 2005.

[76] S. L. Pfleeger. *Software Engineering: Theory and Practice.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2001.

[77] M. Pharr and G. Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2004.

[78] D. Posnett, A. Hindle, and P. T. Devanbu. A simpler model of software readability. In *Mining Software Repositories*, pages 73–82, 2011.

[79] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Programming Language Design and Implementation*, PLDI '14, pages 419–428, 2014.

[80] D. R. Raymond. Reading source code. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 3–16, 1991.

[81] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathemathical Society*, 74:358–366, 1953.

[82] R. J. Rost and B. Licea-Kane. *OpenGL Shading Language*. Addison-Wesley Professional, 3rd edition, 2009.

[83] Y. Rubner, C. Tomasi, and L. J. Guibas. The earth mover's distance as a metric for image retrieval. *International journal of computer vision*, 40(2):99–121, 2000.

[84] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems*, 2013.

[85] E. Schulte, J. DiLorenzo, S. Forrest, and W. Weimer. Automated repair of binary and assembly programs for cooperating embedded devices. In *Architectural Support for Programming Languages and Operating Systems*, 2013.

[86] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer. Post-compiler software optimization for reducing energy. In *Architectural Support for Programming Languages and Operating Systems*, 2014.

[87] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, pages 1–32, 2013.

[88] T. Schweitzer. UniversalIndentGUI. `http://universalindent.sourceforge.net/features.php`. Accessed: 2016-02-22.

[89] L. G. Shapiro and G. C. Stockman. *Computer Vision*. Prentice Hall, January 2001.

[90] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Foundations of Software Engineering*, pages 23–34, 2006.

[91] P. Sitthi-Amorn, N. Modly, W. Weimer, and J. Lawrence. Genetic programming for shader simplification. *ACM Transactions on Graphics*, 30(6), Dec 2011.

[92] J. Spolsky. Things you should never do, part I. `http://www.joelonsoftware.com/articles/fog0000000069.html`. Accessed: 2016-02-01.

[93] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune. Optimizing Google's warehouse scale computers: The NUMA experience. In *High Performance Computer Architecture*, pages 188–197, 2013.

[94] D. Tetzlaff and S. Glesner. Static prediction of loop iteration counts using machine learning to enable hot spot optimizations. In *Software Engineering and Advanced Applications*, pages 300–307, Sept 2013.

[95] A. Tiwari, M. A. Laurenzano, L. Carrington, and A. Snavely. Auto-tuning for energy usage in scientific applications. In *Proceedings of the 2011 International Conference on Parallel Processing*, Euro-Par'11, pages 178–187, 2012.

[96] Z. Tu, Z. Su, and P. Devanbu. On the localness of software. In *Foundations of Software Engineering*, FSE 2014, pages 269–280, 2014.

[97] E. Velázquez-Armendáriz, S. Zhao, M. Hašan, B. Walter, and K. Bala. Automatic bounding of programmable shaders for efficient global illumination. *ACM Transactions on Graphics*, 28(5):142:1–142:9, Dec. 2009.

[98] W. H. Venable Jr and J. J. Hsia. Optical radiation measurements: describing spectrophotometric measurements. *Final Technical Note National Bureau of Standards, Washington, DC. DC Heat Div.*, 1, 1974.

[99] R. Villemin, C. Hery, S. Konishi, T. Tejima, R. Villemin, and D. G. Yu. Art and technology at pixar, from toy story to today. In *SIGGRAPH Asia 2015 Courses*, SA '15, pages 5:1–5:89, 2015.

[100] R. Wang, X. Yang, Y. Yuan, W. Chen, K. Bala, and H. Bao. Automatic shader simplification using surface signal approximation. *ACM Transactions on Graphics*, 33(6), Nov 2014.

[101] S. Wiedenbeck. The initial stage of program comprehension. *International Journal of Man-Machine Studies*, 35(4):517–540, 1991.

[102] L. Williams. Pyramidal parametrics. *SIGGRAPH Computer Graphics*, 17(3):1–11, Jul 1983.

[103] L. Yang, D. Nehab, P. V. Sander, P. Sitthi-amorn, J. Lawrence, and H. Hoppe. Amortized supersampling. *ACM Transactions on Graphics*, 28(5), Dec 2009.

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